AN12383

在 LPC5500 上使用 PowerQuad 和 CMSIS-DSP 计算 FFT

Rev. 0 — 11/2019 Application Note

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 使用私有内存.....3
3 在演示项目中测量时间.....4
4 在示例工程中计算案例.....4
4.1 [输入].....4
4.2 [输出].....5
5 使用 CMSIS-DSP 软件计算 FFT.....6
5.1 复数 FFT 转换.....6
5.2 实数 FFT 转换.....11
6 使用 PowerQuad 硬件计算 FFT.....15
6.1 定点复数 FFT 变换.....15
6.2 定点实数 FFT 变换.....19
6.3 浮点 FFT 变换.....22
7 总结和结论.....29

1 介绍

快速傅里叶变换（FFT）几乎是数字信号处理（DSP）应用中使用最广泛的计算，它可以完成时域到频域的转换。当信号的样本阵列从时域转换到频域时，一些有用和有趣的属性将会出现，并且可以轻松地用于发现信号的模式。凭借这种特点，FFT 被广泛用于提取语音识别、信号检测和其他机器学习应用程序中的功能，以及对定时采样信号的分析。

Arm® CMSIS-DSP 软件库提供了一组 API，可满足在 Cortex®-M MCU 上计算 FFT 的要求。然而 CMSIS-DSP 中的函数完全由软件实现，即使对其进行了优化。这意味着计算时间主要取决于编译器的优化条件和 CPU 的性能。同样，单纯通过软件进行的复杂过程（如 FFT）的计算时间通常也不短，因此在实时应用中应谨慎考虑。

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）。由于计算完全由 Arm 内核以外的特定硬件执行，因此它运行速度快并节省了 CPU 时间。PowerQuad 可以被视为简化的 DSP 硬件，但功耗更低，并且可以很好地集成到 Arm 生态系统中，因此基于它的开发非常友好。

关于定点 FFT 和浮点 FFT 的用法，它们在不同领域有各自的特定实现和应用。定点 FFT 主要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，而这些条件的原始直接采样值是定点。对于浮点 FFT，通常用于导航系统中以高精度和高分辨率处理经度和纬度。因此，定点 FFT 和浮点 FFT 将在本文中一起讨论。

2 PowerQuad 硬件 FFT 引擎

PowerQuad 提供离散傅立叶变换和离散余弦变换，它们使用 Radix-8 蝶形结构快速傅立叶变换引擎，使用定点算法以 24 位分辨率实现。

图 1 为引擎的 Radix-8 蝶形结构。实现减少了内存访问，并充分利用了 PowerQuad 中可用的四个乘法器。

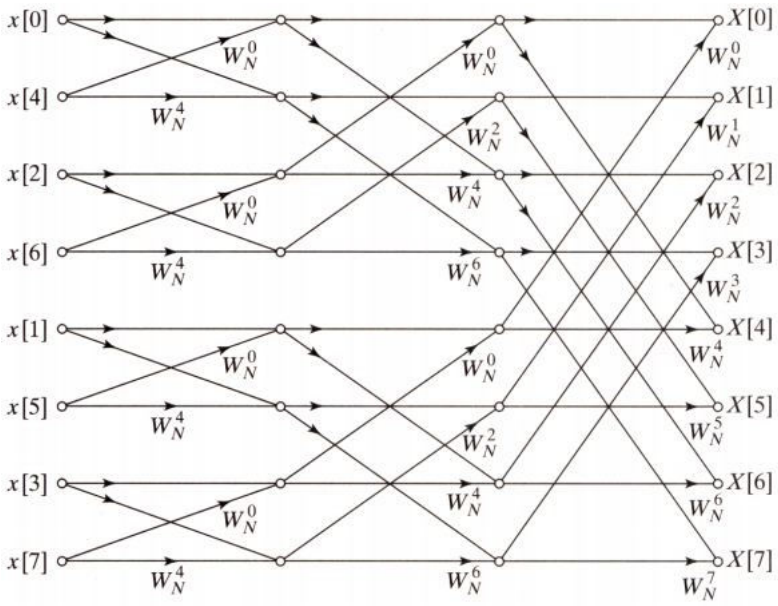

图 1. PowerQuad FFT 引擎的 Radix-8 蝶形结构

2.1 计算方程

离散傅立叶变换可将一个含有 **N** 个复数的序列：

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_{N-1}$$

转换为另一个含有 **N** 个复数的序列：

$$X_0,\ X_1,\ X_2,\ \ldots,\ X_{N-1}$$

定义为：

$$\begin{aligned} \mathrm{X}_k &= \textstyle\sum_{n=0}^{N-1}(x_n \cdot e^{-\mathrm{i}\cdot\frac{2\pi}{\mathrm{N}}\cdot k\cdot n}) \\ &= \textstyle\sum_{n=1}^{N-1}(x_n \cdot \left[\cos\left(\frac{2\pi}{N}\cdot k\cdot n\right) - i\cdot\sin\left(\frac{2\pi}{N}\cdot k\cdot n\right)\right]) \end{aligned}$$

逆变换由下式给出：

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

在大多数实际应用中， **x0**, **x1**, **x2**, ..., **xN-1** 是纯实数，则 DFT 遵循对称性：

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

因此，X_0 和 $X_{N/2}$ 是读取值，DFT 的其余部分仅由 **N/2-1** 个复数完全指定。

注

尽管 PowerQuad 的 FFT 计算引擎也可以通过硬件完成离散余弦变换，但它不如 FFT 常用，并且可以通过 Matrix 方式以更简单的方式进行计算，PowerQuad Matrix 计算引擎也支持这种方式。与 FFT 计算引擎相比，使用 Matrix 计算引擎来计算 DCT 更容易、更灵活。 因此，本应用笔记不会详细描述 DCT，因为其用法与 FFT 几乎相同。

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 引擎只能用定点数作为输入和输出，甚至可以将临时数据保留在 TEMP 区域中。

注

FFT 引擎仅查看输入 32-bit 数据的低 27 位，因此任何预缩放都不得超过该值，以避免饱和。

如果应用中需要浮点数的 FFT，则用户必须将浮点输入数转换为定点数，启动计算，然后将输出的定点数转换为浮点数。幸运的是，PowerQuad 的 Matrix 引擎提供了矩阵缩放功能，可以通过混合格式计算来加快转换速度。

2.2.2 存储器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）需要输入数据序列按如下方式排列在内存中。

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

  $$(x_{0\_real} + i^* x_{0\_im}),\ (x_{1\_real} + i^* x_{1\_im}),\ \ldots\ (x_{N-1\_real} + i^* x_{N-1\_im})$$

  则内存中的输入数组必须组织为：

  $$\{x_{0\_real},\ x_{0\_im},\ x_{1\_real},\ x_{1\_im},\ \ldots,\ x_{N-1\_real},\ x_{N-1\_im}\}$$

- 如果输入序列 $x_0, x_1 \ldots x_{N-1}$ 是实数，则内存中的输入数组必须组织为：

  $$\{x_0,\ x_1,\ \ldots x_{N-1}\}$$

输出序列将始终以复数数组形式存储在内存中，其中实值输出数据的虚部为零。

PowerQuad FFTs/DCTs 支持的序列长度为 N = 16、32、64、128、256、512。

2.2.3 默认硬件预缩放单元

PowerQuad FFT 引擎在硬件默认情况下计算 FFT 之前，将输入数据的值缩放 1/N（除以 N），以使这些值在 DFT 和逆 DFT 的计算期间都不会产生溢出。如果需要不按比例缩放的结果，则必须先将输入数据乘以 N，再将其放入 INPUT A 区域，或者为 INPUT A 区域设置硬件预缩放单元。

FFT 逆变换也按 1/N 缩放，但按照逆 DFT 公式是正确的，因此不需要缩放处理。

如果应用愿意替换现有项目中已使用的 CMSIS-DSP 的 FFT API，以保持输入和输出数据对齐，则应手动添加预缩放单元。但如果是新设计的应用，则可以考虑省略此步骤，因为输出之间的比例关系仍然相同，这是 FFT 计算中最重要的信息。

以下部分显示了使用和不使用手动预缩放单元的不同结果。

2.3 使用私有内存

专用 RAM 是专门用于 PowerQuad 的内存区域。PowerQuad 可以专门访问这部分内存，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。当 PowerQuad 以交错方式同时访问带有 32 位总线的四个存储区时，它可以实现等效的 128 位总线带宽。鼓励使用专用 RAM，因为这意味着 PowerQuad 可以更快地访问数据，因此可以同时从 RAM 和系统访问一个操作数，从而提高了性能。

LPC5500 上专用 RAM 的空间为 16KB，地址在 0xE000_0000 和 0xE000_3FFF 之间。专用 RAM 仅支持 32 位寻址，因为它用于浮点数据（这是 PowerQuad 的本机形式）。通常情况下专用 RAM 中的所有地址空间都可以用于四个内存处理程序，即 **INPUT A**、**INPUT B**、**TEMP**、**OUTPUT**。而且，当数据进出专用 RAM 时，选择内存处理程序的格式无效。

然而，FFT 是一种特殊情况，因为它的引擎是定点引擎，而所有其他功能本身就是浮点数。FFT 引擎被设计为使用 AHB 作为输入（**INPUT A**）和最终输出（**OUTPUT**），它们的内存位于常规内存空间中。专用内存只能用作 **TEMP** 内存处理程序的临时存储。 启动 FFT 引擎时，允许私有 RAM 进行中间（TEMP）存储。由于 FFT 在定点上运行，因此它也将其临时数据以定点形式存储，并以定点形式取回它。

实际上，TEMP 区域仅用于 FFT（用于中间计算）和矩阵求逆。对于其他功能，仅用到内存指针如下：**INPUT A**、**INPUT B**、**OUTPUT**。另一个重要的注意事项是内存处理程序的内存地址对齐。由于 PowerQuad 一次用 4 个字（128 位）读取输入并写入输出，因此为 PowerQuad 内存处理程序分配的内存地址应对齐 4 个字（或 16 字节）。FFT 在这里也是一种特殊情况，对于 TEMP 内存处理程序，它需要根据其空间大小进行对齐。例如，512 个点意味着 512 个复数对，那么它需要对齐 1024 个存储字。

由于 FFT 是使用私有 RAM 的唯一真正较大的操作，因此它是唯一具有如此大的对齐要求的操作。因此，建议始终为其 TEMP 存储器处理程序使用 0xE000_0000，以允许硬件 FFT 引擎占用 FFT 所需的空间。

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演示案例。但这里有个提示：在一些专门用于测量的测试工程中，仍然可以通过基于中断的时间测量方法来测量目标函数的多次运行，然后获得一次执行的平均时间。

在本文的演示代码中，选择 SysTick 定时器作为硬件定时器，这样代码可以很好地移植到其他 Arm Cortex-M MCU 中，然后使用 24 位计数器值直接进行计时。对于 SysTick 计时器的时钟源，运行在 96 MHz 的 LPC5500，最大定时周期可达 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);
```

本文将在不同的条件下对每个功能案例的运行时间进行测量，并总结测量时间以显示计算性能。

4 在示例工程中计算案例

本文档对所有演示计算案例使用通用计算过程。它运行从给定阵列到预期输出阵列的 512 点 FFT 转换。

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}。

- 对于实数定点数，它们是整数 **1** 或 **2**。
- 对于实数浮点数，它们是浮点数 **1.0f** 或 **2.0f**。

- 对于复数定点数，它们是复数**(1，0)** 或 **(2，0)**。
- 对于复数浮点数，它们是复数**(1.0f，0.0f)** 或 **(2.0f，0.0f)**。

总而言之，对于不同的计算情况，输入的值是相同的。

4.2 [输出]

输出数组的值将全部为零，除了：

- 第 0 个数是 765。
- 第 256 个数是 -256。

此输出很有意义。从原始输入阵列可以看出，输入数的平均值为 1.5，简单切换波形的幅度为 0.5，这意味着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。没有其他频率因素。

在频域中，用于 512 点 FFT 变换的步长为 1/512。然后，只有第一项和 1/2（第 256 个）的位置非零。第一项是直流因子，第 256 项是简单的开关波形。非零位置的值应为振幅：result [0] = 1.5, result [256] = -0.5。

但是，使用通用数学计算器（如 Matlab）输出结果时可以简化 1/N 这一步。这意味着直接输出将是最终结果的 N 倍。在本文的情况下，实际结果应为：result [0]= 768, result [256] = -256。

使用以下脚本通过 FreeMat 软件（类似于 MabLab 的数学计算器的开源版本，http://freemat.sourceforge.net/）进行计算，也可以证明结果。

```
--> 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
```

运算结果显示在终端。

```
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
```

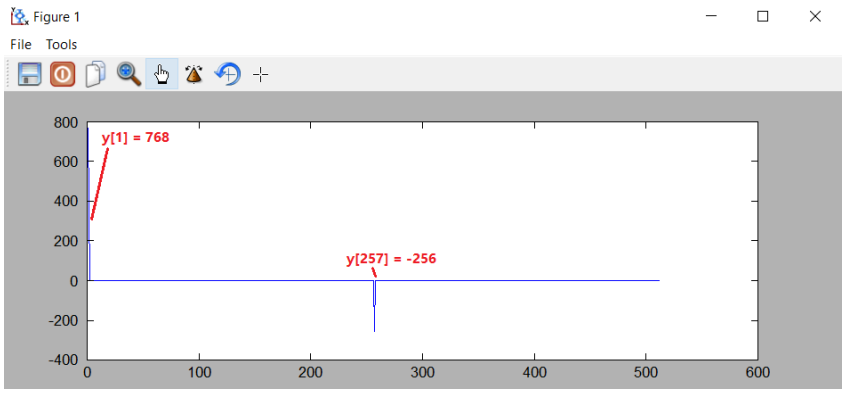

图 2. 使用 FreeMat 进行 FFT 运算

5 使用 CMSIS-DSP 软件计算 FFT

在展示 PowerQuad FFT 引擎的用法之前，这里先介绍 CMSIS-DSP FFT API 的用法，这些 API 已被基于 MCU 的 DSP 开发人员所熟知。 CMSIS-DSP FFT API 通过优化的软件实现。

快速傅立叶变换（FFT）是一种有效的算法，用于计算离散傅立叶变换（DFT）。FFT 的速度可能比 DFT 快几个数量级，尤其是对于较长的长度。有单独的算法可以处理浮点、Q15 和 Q31 数据类型。

FFT 函数在本地运行。也就是说，保存输入数据的数组也将用于保存相应的结果。输入数据很复杂，并且包含 2*fftLen 交错值，如下所示。

$$\{real[0], imag[0], real[1], imag[1], . . \}$$

FFT 结果将包含在同一阵列中，并且频域值将具有相同的交织。 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()` and `arm_rff_fast_f32()`(`arm_rfft_f32()`不再使用)
- `arm_rfft_q31()`
- `arm_rfft_q15()`

有关这些功能的详细信息，请参阅 http://www.keil.com/pack/doc/CMSIS/DSP/html/group__groupTransforms.html。

下面介绍各种格式的 API 的用法。所有情况都可以在启用 Arm Armtex-M33 内核、FPU 和 DSP 指令的 LPC5500 平台上运行。

5.1 复数 FFT 转换

5.1.1 F32 类型的复数 FFT 计算

浮点复数 FFT 使用混合基数算法。根据需要，可以执行多个基数为 8 的阶段以及单个基数为 2 或基数为 4 的阶段。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，并且在计算前向变换时，输出值可能会**增加 fftLen 倍**。逆变换包括 1/fftLen 的缩放，作为计算的一部分，这与逆 FFT 的教科书定义匹配。

在源文件 arm_const_structs.h 中提供并定义了包含旋转因子和位反转表的预初始化数据结构。在函数中包含此头文件，然后将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)
```

这项任务的代码为：

```
/* 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 显示了运算结果。

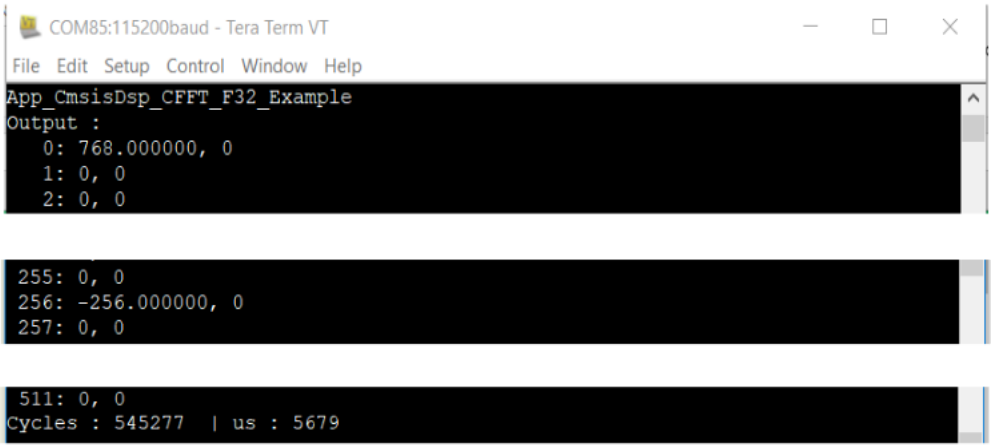

图 3. App_CmsisDsp_CFFT_F32_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看到：

- 事实证明，inputF32[]的内存实际上是由计算函数修改的，而输入数值则被输出数值覆盖。输出数值使用两项作为一个复数的实部和虚部。
- 事实证明，CMSIS-DSP 函数会忽略结果的 1/fftLen 缩放。以下所有情况都将不使用 1/fftLen 缩放的结果作为通用数据。
- 无编译优化情况下的运行耗时。表 5 总结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。另外，定点数的范围可能会使您感到困惑，因为 Q31 数应在（-1，1）范围内。 但是，在这种情况下的应用程序层面上，它们仅用作纯 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，因为除非将整个应用程序完全设计为在内存中使用所有特殊格式的定点数，否则 FFT 的输出在大多数情况下被当作普通值被后续处理流程使用。

这项任务的代码是：

```
/* 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 显示了运算结果。

图 4. App_CmsisDsp_CFFT_Q31_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 定点版本的 FFT 在函数中的缩放比例为 1/fftLen。这种方式可以保留更多的高位数据，并防止计算期间的溢出。但是，由于我们需要实现作为浮点数的通用目标，因此在代码中，会通过软件方式手动使用一个预缩放单元。

实际上，定点 FFT 函数会根据计算长度自动转换输入。在内部，每级输入均按比例缩小 2，以避免 CFFT/CIFFT 过程内部出现饱和。因此，输出格式随 FFT 大小而不同。表 1 和 表 2 描述了不同 FFT 大小和要放大的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	要放大的位数
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

CIFFT 大小	输入格式	输出格式	要放大的位数
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 预计会花费更少的内存和时间，但高位数据更少，也适合处理原始格式为 16 位的数据。其用法与 Q31 版本相同。而且，我们仍然可以像在 Q31 版本的情况中那样，使用具有适当移位的纯 16 位整数。

这项任务的代码是：

```
/* 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 显示了运算结果。

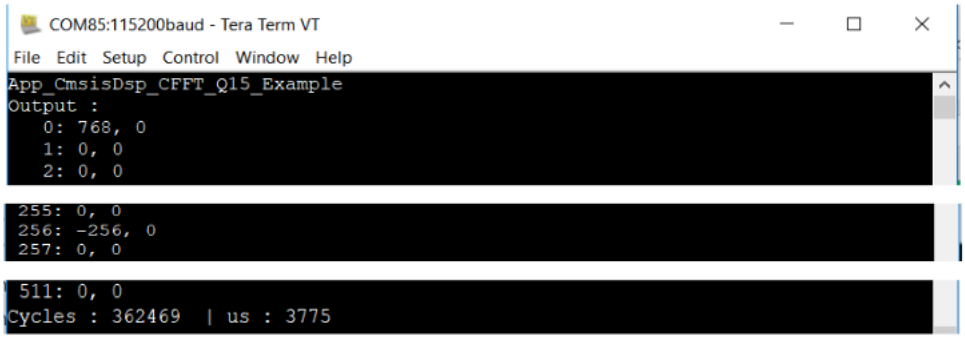

图 5. App_CmsisDsp_CFFT_Q15_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- Q15 版本的 FFT 与 Q31 版本一样，在函数中的缩放比例为 1/fftLen。为了得到通用数据，在代码中，会通过软件方式手动使用预缩放单元。

表 3 和 表 4 描述了 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	要放大的位数
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

CIFFT 大小	输入格式	输出格式	要放大的位数
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 FFT 转换

实数 N 点序列的 FFT 在频域中具有对称性。数据的后半部分等于频率翻转的前半部分的共轭。因此，仅使用 N/2 个复数就可以唯一表示结果。这些以实部和虚部交替打包到输出数组中。

$$X = \{real[0], imag[0], real[1], imag[1], real[2], imag[2] \ldots real[(N/2)-1], imag[(N/2)-1\}$$

碰巧第一个复数（`real[0]`,`imag[0]`）实际上是纯实数，而 `real[0]`代表 DC 偏移，而 `imag[0]`应该为 0。因此 `imag[0]`的位置可用于恢复 `real[N/2]`，这是另一个纯实数。（`real[1]`, `imag[1]`）是基频，（`real[2]`, `imag[2]`）是一次谐波，依此类推。

实际的 FFT 功能以这种方式打包频域数据。正向变换以这种形式输出数据，而逆向变换则期望以这种形式输入数据。该功能始终执行所需的位反转，以便输入和输出数据始终处于正常顺序。**该函数支持长度为[32、64、128，...，4096]点的采样。**

CMSIS DSP 库包括用于计算实数序列 FFT 的专用算法。 FFT 是在复数上定义的，但是在许多应用中，输入数据是实数。实际的 FFT 算法利用了 FFT 的对称性，并且比相同长度的复数计算具有速度优势。

快速 RFFT 算法是基于混合基数 CFFT 实现的，从而降低了处理器载荷。图 6 展示了计算一个实序列的 N 点正向 FFT 的步骤。

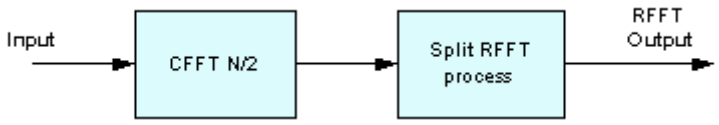

图 6. 实数 FFT

实数序列被当作一个复数序列做 CFFT 运算。随后，处理阶段对数据进行整形以复数形式得到一半的频谱。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，第一个复数样本包含两个打包的实数值。

逆向 RFFT 的输入应与正向 RFFT 的输出保持相同的格式。 第一处理阶段对数据进行预处理，以稍后执行逆 CFFT。

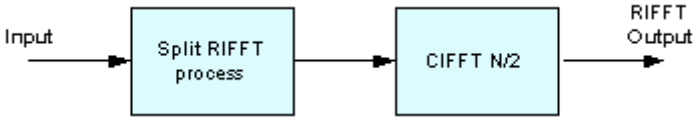

图 7. 实数逆 FFT

使用 N 点实数 FFT 的总结：

- 输入数组的长度为 N，即 N 个实数。
- 对于频谱的前半部分，输出数组的长度也为 N，具有 N/2 个复数，因为数据的后半部分等于频率上移的前半部分的共轭。
- 实际上，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 提供了一种新的 API，可以**快速**替换旧的 API，以计算实数浮点 FFT。现在 `arm_rfft_fast_init_f32()/ arm_rfft_fast_f32` 是唯一推荐的计算方法。并且，输入输出对应的内存空间不能像复数 FFT 函数一样在同一块内存当中存放，输入数据内存和输出数据内存在用户代码中是分开的。且输出数据的方式有些不同，这需要更多注意。

这项任务的代码是：

```
/* 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 显示了运行结果。

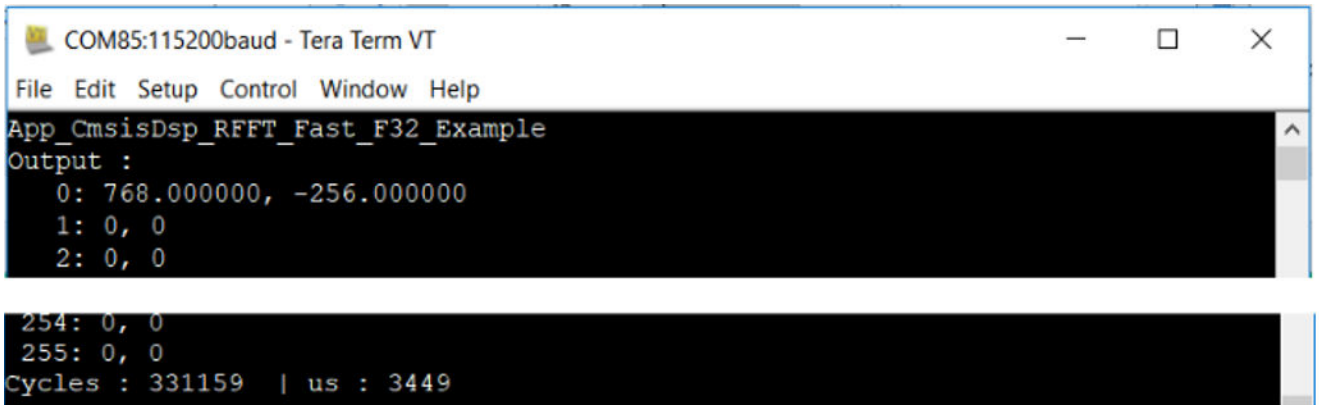

图 8. App_CmsisDsp_RFFT_Fast_F32_Example 的终端日志

根据这种情况的代码和终端日志，我们可以看出：

- 关于输出数值。这些项目仍适用于复数，但是具有输入项目的一半长度（输入在 512 个存储项目中具有 512 个实数，而输出在 512 个存储项目中具有 256 个复数）。输出数组的第一项与其他项不同。第一个复数（`real[0]`, `imag[0]`）实际上都是实数。`real[0]`代表直流偏置，`imag[0]`应该为 0。（`real[1]`,`imag[1]`）是基频，（`real[2]`,`imag[2]`）是一次谐波，以此类推。

5.2.2 Q31 类型的实数 FFT 计算

Q31 的实数 FFT 与浮点版本完全不同，它使用了**快速**方法。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，输出数组用于复数而长度不减少。这意味着输出数组的内存将是输入数组内存的两倍。

这项任务的代码是：

```
/* 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 显示了运行结果。

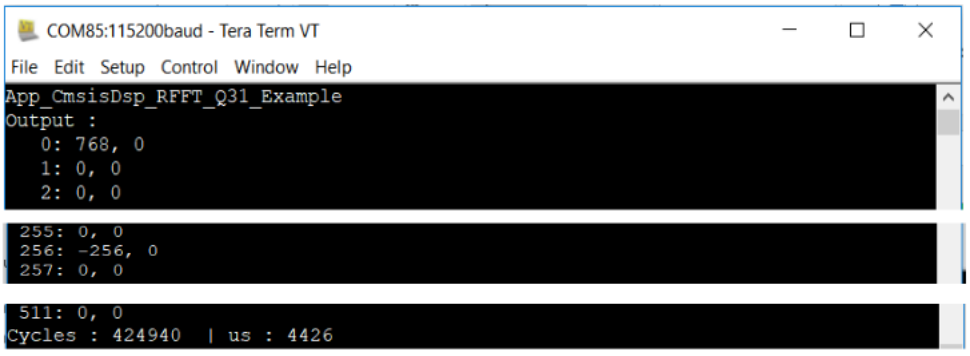

图 9. App_CmsisDsp_RFFT_Q31_Example 的终端日志

根据这种情况的代码和终端日志，我们可以看出：

- 预缩放单元用于得到通用数据。
- 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 继承了 Q31 版本的字符。

这项任务的代码是：

```
/* 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 显示了运行结果。

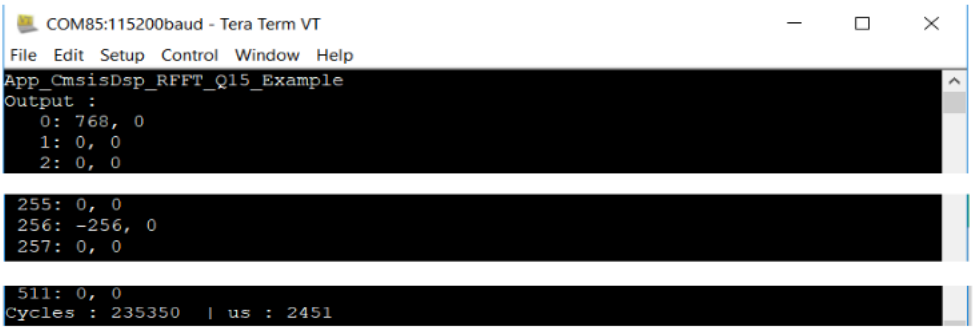

图 10. App_CmsisDsp_RFFT_Q15_Example 的终端日志

根据这种情况的代码和终端日志，我们可以看出：

- 它看起来和 Q31 版本一样。
- 它比 Q31 版本运行得稍快一点。

6 使用 PowerQuad 硬件计算 FFT

然而，CMSIS-DSP API 的纯软件实现仍然受到 Arm 内核的体系结构（狭窄的内存总线）和编译器的性能（不同级别的优化条件）的限制。但另一方面，PowerQuad 的计算引擎（包括 FFT 引擎）是通过硬件实现和优化的，通过比较 CMSIS-DSP 的使用，可以节省大量 CPU 负载和代码大小，并显著提高性能。而且，作为一个协处理器集成在一起，PowerQuad 还可根据需要与 Arm 内核并行运行，以满足实时系统的要求。

NXP MCUXpresso SDK 软件库已经支持 PowerQuad 模块。在 PowerQuad 驱动程序中，有一组用于计算 FFT 的 API：

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

PowerQuad 硬件最初不支持浮点 FFT。但创建了基于现有 PowerQuad 硬件的软件解决方案来解锁此功能，因此它可以涵盖申请 CMSIS-DSP FFT API 的相同领域。

下面将讨论 API 的用法。

6.1 定点复数 FFT 变换

PowerQuad FFT 引擎硬件仅支持定点 FFT 转换，因此 PowerQuad 硬件可以直接处理定点 FFT 任务。

6.1.1 Q31 类型的复数 FFT 计算

在以前的 CMSIS-DSP 情况下，为了实现通用目标输出，一个软件实现的预缩放单元用于处理输入数据。对于 PowerQuad，硬件提供了一个新选项，可以通过硬件预缩放单元设置来完成。输入和输出数据都有其所对应的硬件预缩放单元设置。在这种情况下，对于 512 点 FFT，预缩放数应为 512，`pq_cfg.inputAPrescale` 的响应设置值为 **9**，因为输入值将与 512 相乘时左移 9 位。

关于配置 PowerQuad 硬件的输入和输出格式。由于 FFT 引擎使用 Input A、Temp 和 Output 内存处理程序，而硬件仅支持定点 FFT，因此 `pq_cfg.inputAFormat`, `pq_cfg.tmpFormat`, `pq_cfg.outputFormat` 中这些内存处理程序的格式设置适用于定点，例如 **kPQ_32Bit** 或 **kPQ_16Bit**。在这种情况下，它们是 **kPQ_32Bit**。对于 FFT 引擎，将忽略输出存储器处理程序的设置。同样，输入和输出数组必须是 32 位的数。

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

临时 RAM 处理程序使用从 `0xE000_0000` 开始的专用 RAM，以在计算过程中保留中间数据。对于 512 点 FFT，要使用 1 K 32 位字保留 512 个复数，应在私有 RAM 中实际保留总共 4 KB 的内存空间。

在这种情况下，关键函数是 `PQ_TransformRFFT()`，但将 Q31 数值作为输入和输出，而输入数值是复数。

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 显示了运行结果。

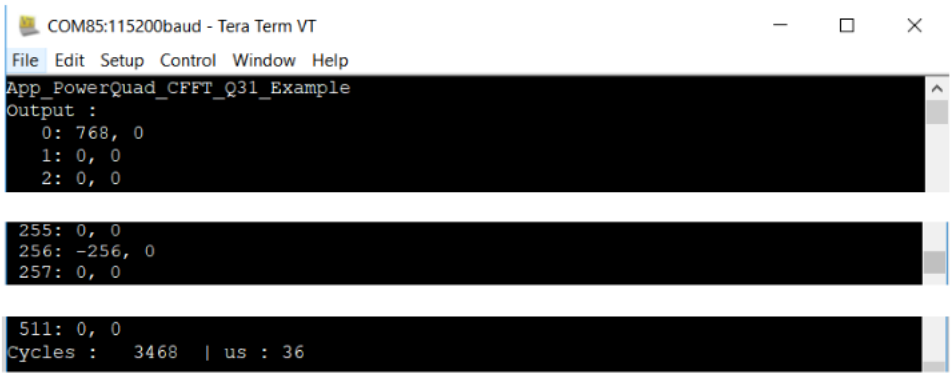

图 11. App_PowerQuad_CFFT_Q31_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 硬件预缩放单元与软件缩放单元一样生效。
- 预期结果（通用数据）由 PowerQuad 硬件产生。
- 确实比 CMSIS-DSP 复数 Q31 定点 FFT 函数更快。

实际上，此处关于预缩放单元对定点输出数值的用法可以复用 CMSIS-DSP 定点 FFT 输出的数据表。

6.1.2 Q15 类型的复数 FFT 计算

使用 PowerQuad FFT 引擎，复数 Q15 任务与复数 Q31 任务几乎相同，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 显示了运算结果。

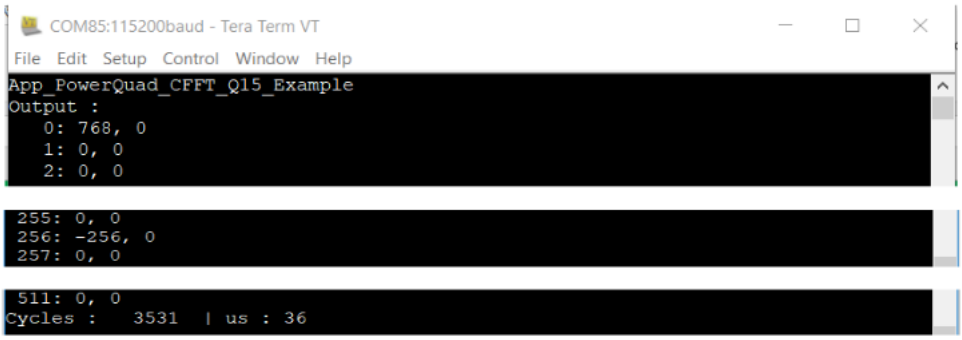

图 12. App_PowerQuad_CFFT_Q15_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 硬件预缩放单元生效。
- 预期结果（通用数据）由 PowerQuad 硬件产生。
- 它不比 Q31 的复数 FFT 快，甚至在实际运行中也慢一点。 因此，数量较少的位不会减少 PowerQuad 硬件的工作量。

6.2 定点实数 FFT 变换

PowerQuad 硬件的纯实数 FFT 压缩了虚部，并且仅将实数保留在输入数组中。它比复数的 FFT 节省了一半的内存，而 PowerQuad 硬件也可以识别这种方式。但是，PowerQuad 始终将输出保留为复数（CMSIS-DSP API 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()`，但以 Q31 数值作为输入和输出，且输入数值是纯实数。

这项任务的代码是：

```
/* 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 显示了运行结果。

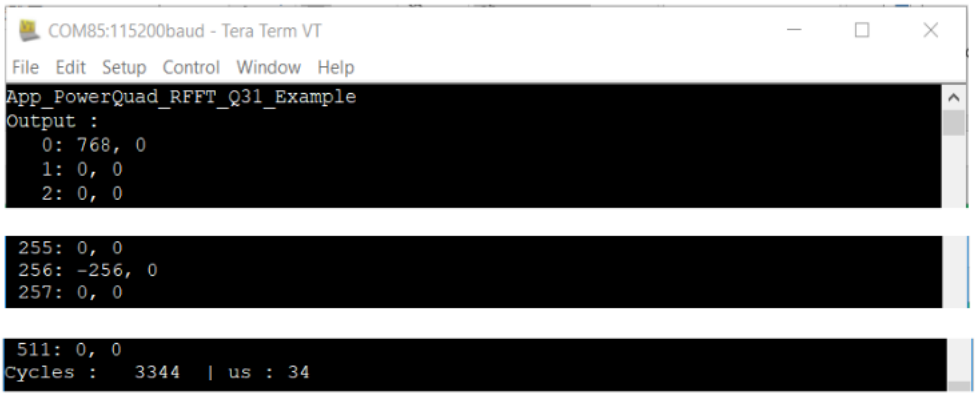

图 13. App_PowerQuad_RFFT_Q31_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 硬件预缩放单元生效。
- 预期结果（通用数据）由 PowerQuad 硬件产生。
- 由于减少了存储器操作，因此它比 Q31 的复数 FFT 快一点。
- 输出数据的长度不会像 CMSIS-DSP 功能那样减少一半。 对于用户而言将更为简单，因此无需使用特殊格式来应对复杂的 FFT 计算。

6.2.2 Q15 类型的实数 FFT 计算

使用 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` 的数据结构设置为 **kPQ_16Bit**。

这项任务的代码是：

```
/* 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 显示了运算结果。

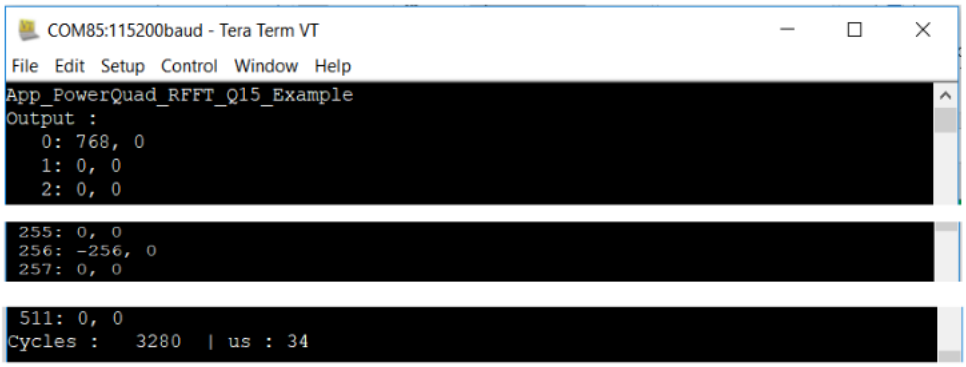

图 14. App_PowerQuad_RFFT_Q15_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 硬件预缩放单元生效。
- 预期结果（通用数据）由 PowerQuad 硬件产生。
- 由于减少了存储器操作，因此它比 Q31 的复数 FFT 快一点。
- 输出数据的长度不会像 CMSIS-DSP 功能那样减少一半。对于用户来说更简单，因此无需使用特殊格式来应对复杂的 FFT 计算。

6.3 浮点 FFT 变换

PowerQuad 硬件不直接支持浮点 FFT。 但是在某些应用中，为了从 PowerQuad 硬件计算引擎的强大加速中获得好处，而又无需进行代码更改，用户可能只想通过用 PowerQuad 的实现替换现有的用于浮点 FFT 的 CMSIS-DSP API 来更新其项目，然后将需要在浮点和定点之间进行数据格式转换。

幸运的是，PowerQuad 的矩阵缩放函数可以帮助处理硬件的格式转换，并且它比 `arm_float_to_q31()`/`arm_q31_to_float()` 的 ARM-CMSIS DSP API 运行得更快。因此，只需将浮点输入数转换为定点执行定点 FFT 和将定点输出转换为浮点的操作连接起来，我们就可以创建一个基于 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

在 CMSIS-DSP 中，存在有关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，当设置具有不同值格式的输入和输出并执行 sclaer 为 1.0f 的矩阵缩放操作，这意味着该值不会从输入和输出中更改，而是在从输入缓冲区到输出缓冲区的值移动过程中自动完成转换。

浮点值和定点值之间的格式转换示例代码为：

```
/* 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 显示了运行结果。

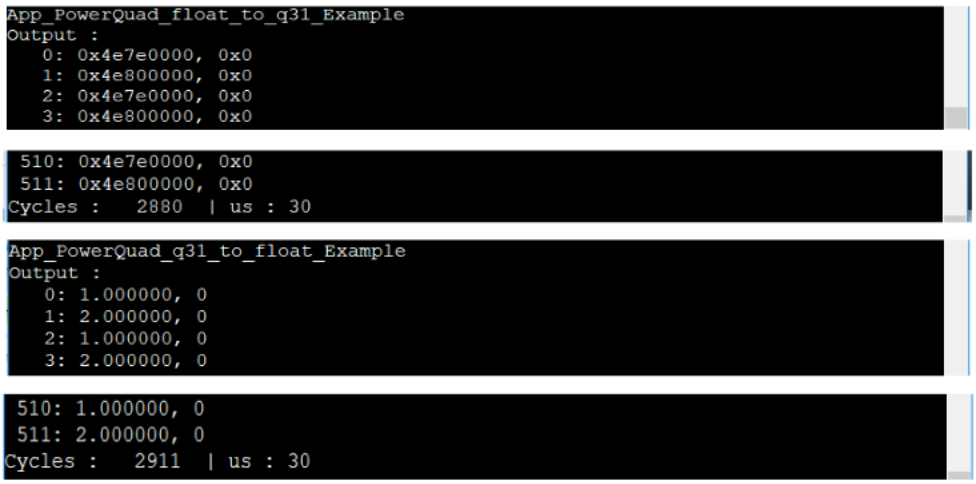

图 15. 格式切换函数的终端日志

实际上，相同的测试用例也与 ARM-CMSIS DSP API 一起运行。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。但是，使用转换函数时有一些限制：

- 对于 CMSIS-DSP API，定点数应该遵循标准 q31 格式，其范围应在（-1,1）之间。
- 对于 PowerQuad API，数组的最大长度为 256。如果需要处理更长的数组，则应多次调用 Matrix Scale 函数。

6.3.2 F32 类型的复数 FFT 计算

在这种情况下，通过四次调用 256 点 Matrix Scale 函数，将 512 个浮点输入复数（数组中的 1024 个数字）转换为定点输入数。运行硬件 FFT 以获取输出定点数后，将调用另外 4 次的 256 点 Matrix Scale 函数以获取浮点输出数。

这项任务的代码是：

```
/* 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 显示了运行结果。

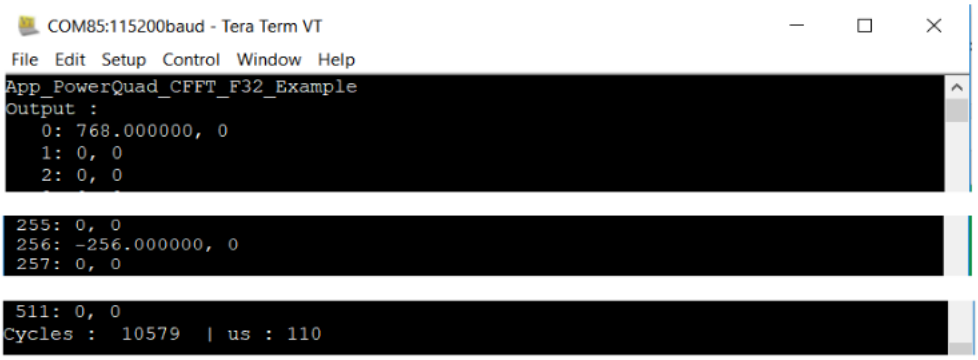

图 16. App_PowerQuad_CFFT_F32_Example 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 结果正确，与 CMSIS-DSP 的浮点复数 FFT 结果相同。
- 硬件转换功能运行良好。
- 它运行的时间几乎等于 2 × PowerQuad 矩阵缩放 + 1 × PowerQuad CFFT 的时间。 它看起来比 CMSIS-DSP 中的 `arm_cfft_f32()` 函数快。

6.3.3 F32 类型的实数 FFT 计算

在这种情况下，将打包浮点实数的输入数组转换为 Q31 数值，然后由 PowerQuad 的 FFT 引擎使用 `PQ_TransformRFFT()` 函数进行计算，以获取 Q31 数值的输出，最后使用 PowerQuad 的硬件矩阵缩放函数将其转换为浮点格式。

这项任务的代码是：

```
/* 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 显示了运行结果。

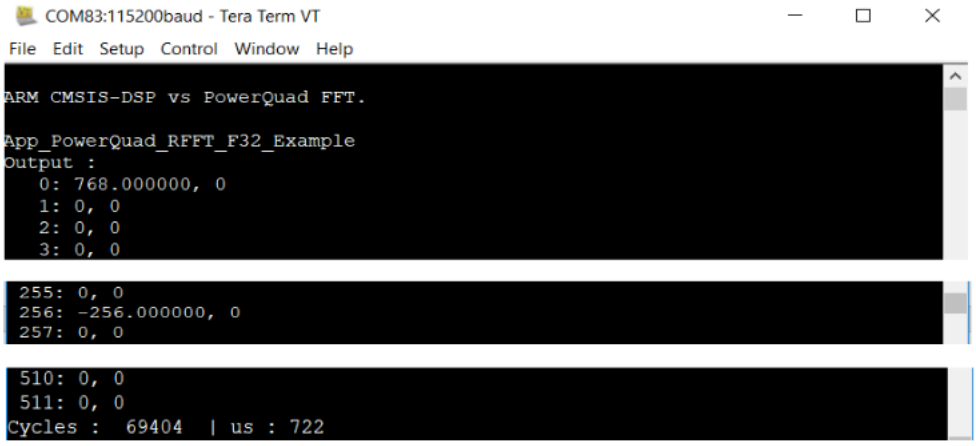

图 17. 使用 F32 数字计算实数 FFT 的终端日志

根据这种情况下的代码和终端日志，我们可以看出：

- 根据 Arm CMSIS-DSP 的数据转换的使用，应将输入数据缩小到（-1，1）范围。 还有一点，转换数的输出是严格的 q31 数，而我们实际上使用了类似整数的定点数（**q0** 格式）。因此，完成了对输入浮点数的额外缩放。然后，我们可以像其他演示案例一样获得通用数据。
- 由于该解决方法，`arm_float_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更快。 关于时序比较，将在本文后面的部分中进行讨论。

7 总结和结论

到目前为止，本文介绍了在同一计算案例中使用 CMSIS-DSP 软件和 PowerQuad 硬件计算 FFT 的用法。 因此，在为相同格式的输入和输出计算 FFT 时，可以使用 PowerQuad 硬件来代替 CMSIS-DSP 软件。尽管如此，演示案例表明 PowerQuad 的运行速度比 CMSIS-DSP 快得多。

在本文的最后一部分中，这里总结了一些演示案例的时序属性表，以显示 PowerQuad 性能。 在 IAR IDE 的**工程选项**对话框中设置不同的编译优化条件，如 图 18 所示。

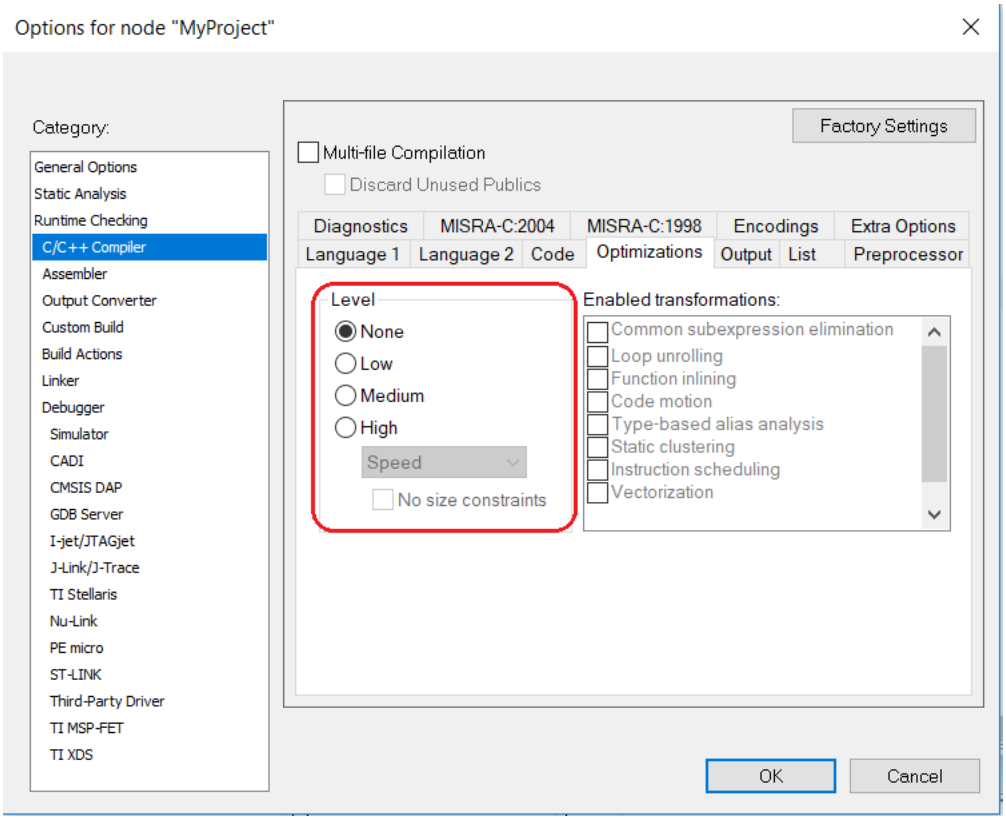

图 18. IAR 项目中编译器的最佳选项

表 5 显示了优化测量时间。

表 5. 在最佳条件下测量时

间 演示案例	无		低		中		高（速度）		无（FPU 不可用）	
	周期	us	周期	us	周期	us	周期	us	周期	us
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
App_CmsisDsp_RFFT_Fast_F32_Example	331456	3452	232862	2425	165032	1719	155098	1615	2293419	23889

下页继续...

表 5. 在最佳条件下测量时间（续）

演示案例	无		低		中		高（速度）		无（FPU 不可用）	
	周期	us	周期	us	周期	us	周期	us	周期	us
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

通过 表 5，我们可以看出：

- PowerQuad 的计算速度比 CMSIS-DSP 函数快得多。 测量值快约 100 倍。
- 对于不同格式数据，不同编译优化条件的 FFT 计算，PowerQuad 的计时性能稳定。 但是 CMSIS-DSP 软件的性能会因编译优化条件而有很大差异。 对于 CMSIS-DSP 软件实现函数，较高级别的优化并不总是使代码运行更快（对于 App_CmsisDsp_CFFT_Q15_Example，较低级别的优化运行 1876 us，而中等级别的优化运行 1978 us。
- 定点运算并不总是比浮点运算快。禁用硬件 FPU 时，使用常规定点指令，浮点计算需要更多的 CPU 周期。在这种情况下，定点算法将运行得更加平稳。但是，当编译器使能 FPU 后，浮点计算指令可以节省更多时间，并且可以在一条指令中直接计算浮点数，而定点计算需要更多的指令才能将大数据的计算转换为多个步骤并花费更多时间。因此，这就是为什么在为编译器启用 FPU 时，App_CmsisDsp_CFFT_F32_Example 演示案例的运行速度比 App_CmsisDsp_CFFT_Q31_Example 快，但是在禁用 FPU 时，运行情况却慢得多的原因。
- 对于 CMSIS-DSP 软件和 PowerQuad 硬件，浮点数和定点数之间的格式转换花费大量时间，几乎处于同一水平。
- 对于 App_PowerQuad_RFFT_F32_Example 演示案例，即使使用了有关格式转换问题的软件解决方法，并已替换为 ARM CMSIS-DSP 的部分实现，它仍比纯软件方法快约 x3 倍。但是，更建议使用复数的浮点 FFT，因为它运行速度快得多，但需要少量额外的内存。或在应用程序中将原始数据格式修改为定点数，即可达到最佳性能。

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测量时间

演示案例	无		低		中		高（速度）		无（FPU 不可用）	
	周期	us	周期	us	周期	us	周期	us	周期	us
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
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

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: 11/2019

Document identifier: AN12383

arm