AN13496

Computing FFT with PowerQuad and CMSIS-DSP on LPC5500

Rev. 0 — 31 December 2021 Application Note

Contents

1 Introduction.....1
2 PowerQuad hardware FFT engine 1
2.1 Computing equations.....2
2.2 Input and output details.....3
2.3 Using private RAM.....4
3 Measuring time in demo project.....4
4 Computing cases in demo project...5
4.1 [INPUT].....5
4.2 [OUTPUT].....5
5 Computing FFT with CMSIS-DSP software.....6
5.1 Complex FFT transforms.....7
5.2 Real FFT transforms.....13
6 Computing FFT with PowerQuad hardware.....18
6.1 Fixed-point complex FFT transforms.....18
6.2 Fixed-point real FFT transforms.22
6.3 Float-point FFT transform.....26
7 Summary and conclusion.....35
8 Revision history.....39

1 Introduction

Fast Frontier Transform (FFT) is the most popular computation in Digital Signal Processing (DSP) application. It can transform between timing field and frequency field. When the sample array of signal is transformed from timing field to frequency field, some useful and interesting attributes appear. The attributes are used to find the pattern of signals. With this feature, FFT extracts the features in voice recognition, signal detection, and other machine-learning application with the analysis of timing-sampling signals.

The Arm® CMSIS-DSP Software Library provides a group of APIs to fulfill the requirement of computing FFT on Cortex®-M MCUs. However, even well-optimized, the software implements the functions in CMSIS-DSP. The computing time depends on the optimization condition of the compiler and CPU performance. Also, the computing time of the complex process, like FFT purely by software, is long. Consider the process in the real-time application.

The PowerQuad hardware module is designed to accelerate some general DSP computing tasks, including math functions, matrix functions, filter functions, and transform functions (including FFT). The specific hardware, other than Arm core, executes the computing. It runs fast and saves CPU time. The PowerQuad is a simplified DSP hardware with less power consumption. It is integrated inside the Arm ecosystem. The development based on PowerQuad is very friendly.

The fixed-point FFT and floating-point FFT have their own implementation and application in different field. The fixed-point FFT processes the audio, video, and other data captured from hardware sensor modules like ADC, while the original direct sample value for these conditions is fixed-point. The floating-point FFT processes the longitude and latitude with high accuracy and high resolution in navigation system. This application note discusses fixed-point FFT and floating-point FFT.

2 PowerQuad hardware FFT engine

The PowerQuad provides Discrete Fourier Transform (DFT) and Discrete Cosine Transform (DCT). They are implemented with a Radix-8 butterfly structure FFT engine. The engine uses fixed-point arithmetic at a resolution of 24 bits.

Figure 1 shows the Radix-8 butterfly structure of the engine. This implementation reduces memory accesses and uses the four multipliers available in PowerQuad.

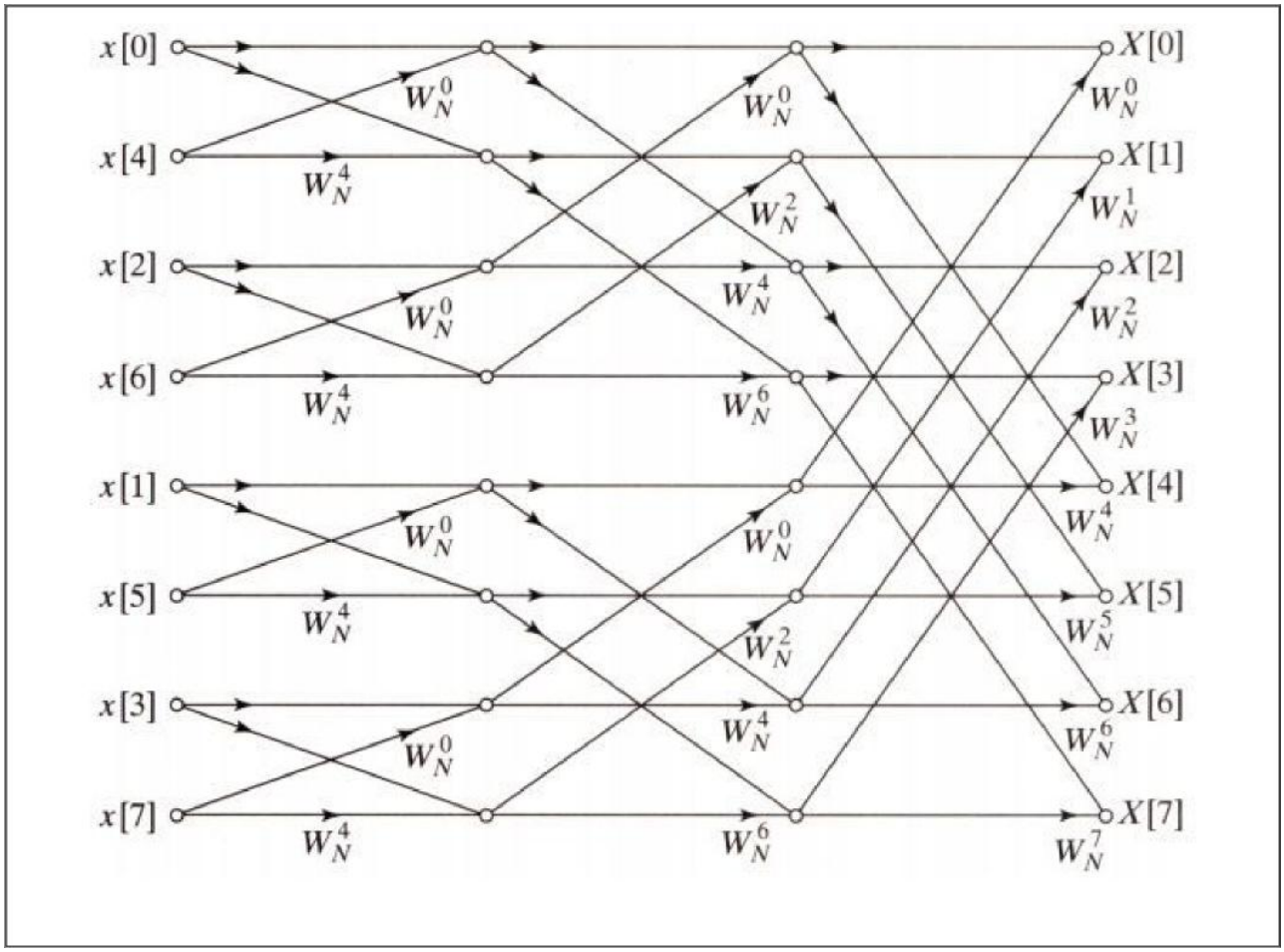

Figure 1. Radix-8 butterfly structure of PowerQuad FFT engine

2.1 Computing equations

DFT transforms a sequence of N complex numbers:

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

into another sequence of N complex numbers:

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

which is defined by:

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

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

The inverse transform is given by:

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

In most practical applications, the x_0, x_1, x_2, ..., x_{N-1} are the pure real numbers. DFT obeys the symmetry:

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

It follows that X_0 and $X_{N/2}$ are read values and **N/2 -1** specifies complex numbers the remainder of the DFT.

NOTE

FFT computing engine of PowerQuad supports DCT. It is not as popular as FFT. FFT computing engine computes in the matrix way, which is simpler. The matrix computing engine of PowerQuad also supports DST. Compared with FFT computing engine, the matrix computing engine computes DCT much easier and is more flexible. DCT usage is almost same with FFT, so this application note does not describe DCT in details.

2.2 Input and output details

2.2.1 Fixed-point numbers only for FFT engine

PowerQuad FFT engine uses fixed-point number as input and output, even to keep the temporary data in the TEMP region.

NOTE

FFT engine looks at the bottom 27 bits of the input word. To avoid saturation, any pre-scaling must not exceed.

If FFT of the floating-point numbers is required, convert the floating-point input numbers to the fixed-point numbers. Launch the computing and convert the output fixed-point numbers to the floating-point ones. Fortunately, the matrix engine of PowerQuad provides a function of matrix scale. It accelerates the conversion by mixed format computing.

2.2.2 Input and output sequence in memory

The pure real numbers (prefixed by *r*) and the complex flavors of the functions (prefixed by *c*) expect to arrange the input data sequences in memory as below:

- If the input sequence x_0, x_1, ... x_{N-1} are complex numbers of the form, while N is the length of the array:

  $(X_{0\_real} + i * X_{0\_im}), (X_{1\_real} + i * X_{1\_im}), ...(X_{N-1\_real} + i * X_{N-1\_im})$

  the input array in memory is organized as:

  $\{X_{0\_real}, X_{0\_im}, X_{1\_real}, X_{1\_im}, ..., X_{N-1\_real}, X_{N-1\_im}\}$

- If the input sequence x_0, x_1, ... x_{N-1} are real numbers, the input array in memory is organized as:

  $\{X_0, X_1, ... X_{N-1}$

  The output sequence is stored in memory, organized as an array of complex numbers. The imaginary parts are all zero for real-valued output data.

  The supported lengths for PowerQuad FFTs/DCTs are N = 16, 32, 64, 128, 256, and 512 points.

2.2.3 Default hardware prescaler

The PowerQuad FFT engine scales the value of input data by 1/N (divide N) before computing the FFT by hardware default. The value is not overflown during the computing of both DFT and inverse DFT. If an unscaled result is necessary, before placing the input data in the INPUT A region, multiply it by N or set up the hardware prescaler for the INPUT A region.

The inverse FFT is scaled by 1/N, and it is correct as per the inverse DFT formula. No scaling treatment is needed.

To replace the FFT API of CMSIS-DSP, which is used in the existing project, manually add the prescaler to keep the input and output data. If the application is newly designed, this step is not necessary. The proportional relation among the outputs is still the same, which is the most important information of FFT computing.

Below shows the different results with and without the manual prescaler.

2.3 Using private RAM

The private RAM is an area of memory specifically for PowerQuad. To accelerate the whole process of computing as fast as possible, PowerQuad accesses this part of memory exclusively without any arbitration delay. PowerQuad accesses the four banks of memory with 32-bit bus simultaneously in an interleave way. It can achieve equivalent 128-bit bus band wide. If using the private RAM, PowerQuad can access the data quicker. PowerQuad can access one operand from RAM and the other from system at the same time. The performance is improved.

The space for private RAM on LPC5500 is 16 kB with the address between `0xE000_0000` and `0xE000_3FFF`. The private RAM supports only 32-bit addressing. It is meant for floating point data (which is the native form of PowerQuad). All address spaces in the private RAM are used for the four memory handlers, INPUT A, INPUT B, TEMP, and OUTPUT. When data is traveling in and out of the private RAM, it takes no effect to choose the format of a memory handler.

FFT is a special case because its engine is a fixed-point engine, while all other functions are natively floating points. FFT engine is designed to operate with AHB as input (INPUT A) and final output (OUTPUT). The memory is located at general memory space. Private memory is used as temporary storage for TEMP memory handler. When launching the FFT engine, the private RAM is allowed intermediate (TEMP) storage. FFT is operating at fixed point. It deposits its temporary data at fixed point and gets it back at fixed point.

TEMP area is only used for FFT (for intermediate calculations) and matrix inversion. For other functions, the only useful memory handlers are the INPUT A, INPUT B, and OUTPUT.

Another important notice is the alignment of memory address for memory handlers. Since the PowerQuad reads the input and writes the output with four words (128-bit) a time, the allocated memory address for the PowerQuad memory handler must be 4-word (or 16-byte) aligned. FFT is a special case. For TEMP memory handler, it needs the alignment to its space size. For example, **512 points** means 512 complex pairs and it must align with 1024 words.

As FFT is the only really big operation using private RAM, it is the only one that has such large alignment requirements. So, use `0xE000_0000` for its TEMP memory handler, allowing the hardware FFT engine to consume space for FFT.

3 Measuring time in demo project

The functions run fast, so interrupt-based timing method is not suitable in the demo case.

NOTE

In some test projects specially for measuring, interrupt-based timing method is still available. The method is to get the average time for one execution, measure plenty times of the target function.

For the demo code in this document, **SysTick timer** is chosen as the hardware timer. The code is portable for other Arm Cortex-M MCUs. Then use the 24-bit counter value directly for timing. LPC5500 runs at 96 MHz for the SysTick timer's clock source, so the maximum timing period can be 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)
```

The usage is:

```
uint32_t cycles;

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

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

In this document, the running time of each functional case is measured in different condition. The measuring time is summarized to show the computing performance.

4 Computing cases in demo project

This document uses a general computing process for all the demo computing cases. It runs the 512-point FFT transform from a given array to the expected output array.

4.1 [INPUT]

The input array includes pure real numbers {1, 2, 1, 2, 1, 2, ..., 1, 2} with the length of 512.

- For the real fixed-point numbers, they are the integer number 1 or 2.
- For the real floating-point numbers, they are the floating number 1.0f or 2.0f.
- For the complex fixed-point numbers, they are the complex number (1, 0) or (2, 0).
- For the complex floating-point numbers, they are the complex number (1.0f, 0.0f) or (2.0f, 0.0f).

The values of input are same for different computing cases.

4.2 [OUTPUT]

The output array of values is all zero except for:

- The 0^{th} number is 765.
- The 256^{th} number is -256.

This output makes sense. As seen in the original input array, the average value of the input number is 1.5 and the amplitude of the simple switching waveform is 0.5. It means that the original input can be represented as 1.5-0.5, 1.5+0.5, 1.5-0.5, 1.5+0.5, ... The switching period is 2, with the frequency of ½. The phase is negative. No other frequency factors.

In the frequency field, the step for 512-point FFT is 1/512. Only the first item and the position for 1/2 (the 256^{th}) are non-zero. The first item is for DC factor and the 256^{th} is for the simple switching waveform. The value for the non-zero position is the amplitude: result [0] = 1.5, result [256] = -0.5.

When outputting the result, use the general mathematics calculator (like Matlab) to simplify the step of 1/N. It means that the direct output multiples N from the final result. In the cases in this document, the actual result is: result [0] = 768, result [256] = -256.

To get the result, the calculation with FreeMat software (an open-source version of MabLab-like mathematics calculator, FreeMat) with the following script is also supported.

```
--> 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
```

The result is shown at the terminal.

```
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
```

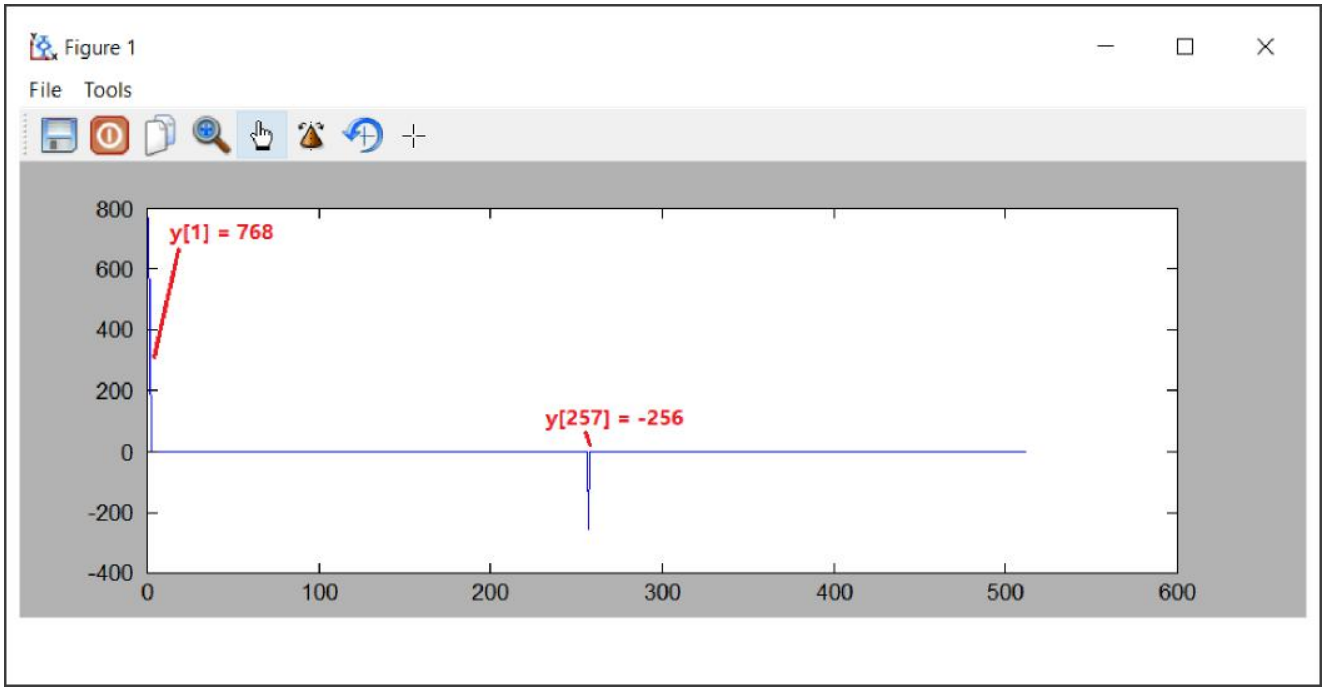

Figure 2. FFT calculation by FreeMat

5 Computing FFT with CMSIS-DSP software

Before introducing the usage of PowerQuad FFT engine, this chapter tells the usage of CMSIS-DSP FFT APIs, well known by DSP developers with MCU knowledge. The optimized software implement CMSIS-DSP FFT APIs.

FFT is an efficient algorithm for computing DFT. FFT can be orders of magnitude faster than the DFT, especially for long lengths. There are separate algorithms for handling floating-point, Q15, and Q31 data types.

The FFT functions operate in-place. That is, to hold the corresponding result, use the array holding the input data. The input data is complex and contains 2*`fftLen` interleaved values as below.

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

FFT result is contained in the same array. The frequency domain values contain the same interleaving. CMSIS-DSP provides a group of APIs for computing FFT:

- `arm_cfft_f32()`
- `arm_cfft_q31()`
- `arm_cfft_q15()`
- `arm_rfft_fast_f32_init()` and `arm_rff_fast_f32()` (`arm_rfft_f32()` is not used any more)
- `arm_rfft_q31()`
- `arm_rfft_q15()`

For detailed information about these functions, see CMSIS-DSP.

Below describes the usage of APIs for various formats. All the cases run well on the LPC5500 platform with Arm Cortex-M33 core, FPU, and DSP instructions enabled.

5.1 Complex FFT transforms

5.1.1 Computing FFT with complex F32 numbers

The floating-point complex FFT uses a mixed-radix algorithm. Multiple radix-8 stages are performed along with a single radix-2 or radix-4 stage, as needed. The algorithm supports lengths of `[16, 32, 64, ..., 4096]` and every length uses a different twiddle factor table.

The function uses the standard FFT definition. When computing the forward transform, output values grow with a factor of `fftLen`. The inverse transform includes a scale of `1/fftLen` as part of the calculation. The transform matches the textbook definition of the inverse FFT.

The source file, `arm_const_structs.h`, provides and defines pre-initialized data structures. The structure contains twiddle factors and bit reversal tables. Include this header in your function and pass one of the constant structures as an argument to `arm_cfft_f32`. For example:

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

The code for the task is:

```
/* 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. */
```

Figure 3 shows the result.

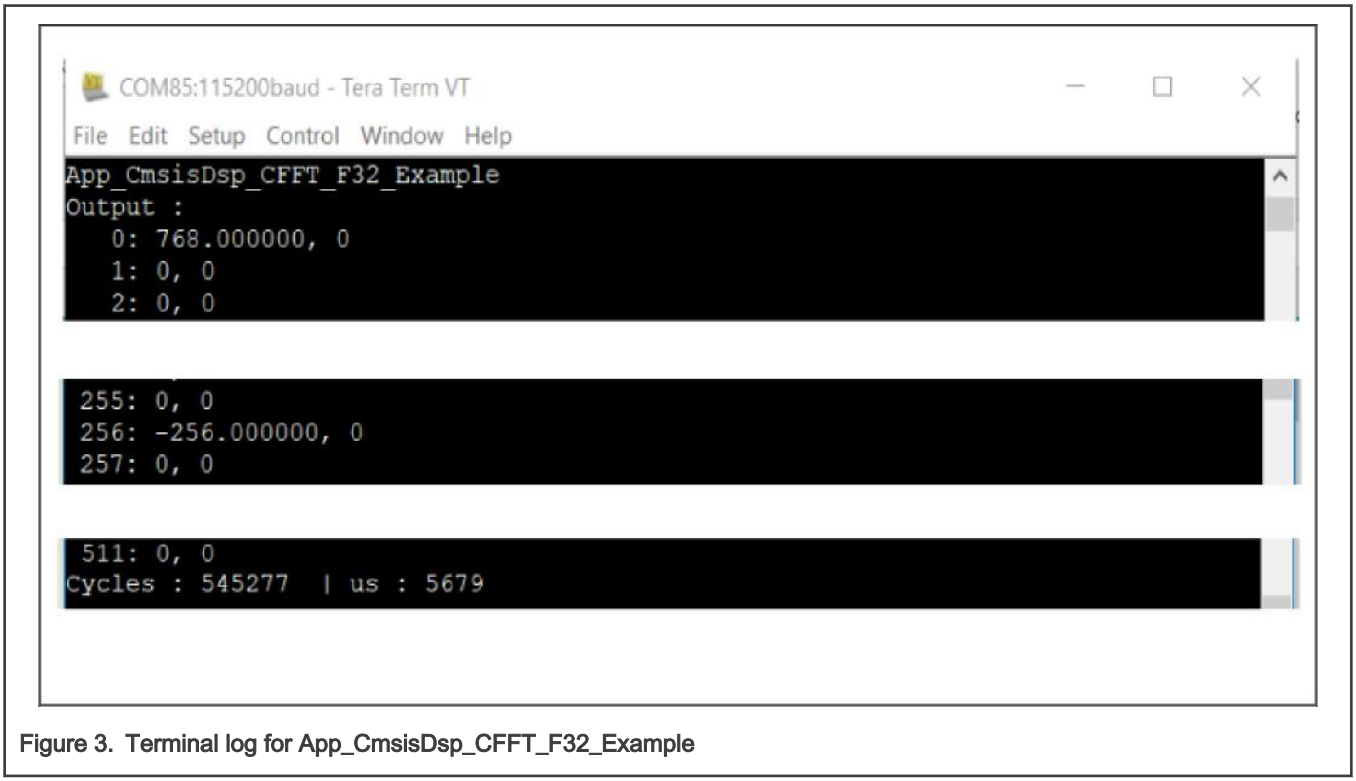

Figure 3. Terminal log for App_CmsisDsp_CFFT_F32_Example

Per the code and terminal log in this case, we can see:

- The computing function modifies the memory of `inputF32[]`. The output numbers cover the input numbers. The output number uses two items as the real part and complex part for a complex value.
- The CMSIS-DSP function ignores the `1/fftLen` scale for the result. All following cases use the result without `1/fftLen` scale as the common target.
- The running time goes with no compiling optimization. Table 5 summarizes all computing time in different optimal conditions.

5.1.2 Computing FFT with complex Q31 numbers

FFT of Q31 version is implemented differently from the floating-point one. Q31 number is in the range of (-1, 1), so the range of the fixed-point number is confusing. However, in the application level of this case, they are used as the pure 32-bit integers, or can be seen as a **Q0** in fixed-point format. This consideration makes sense. The output of FFT is used as normal values to complete the following procedure, unless the whole application is totally designed with all special formatted fixed-point numbers in memory.

The code for the task is:

```
/* 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. */
```

Figure 4 shows the result.

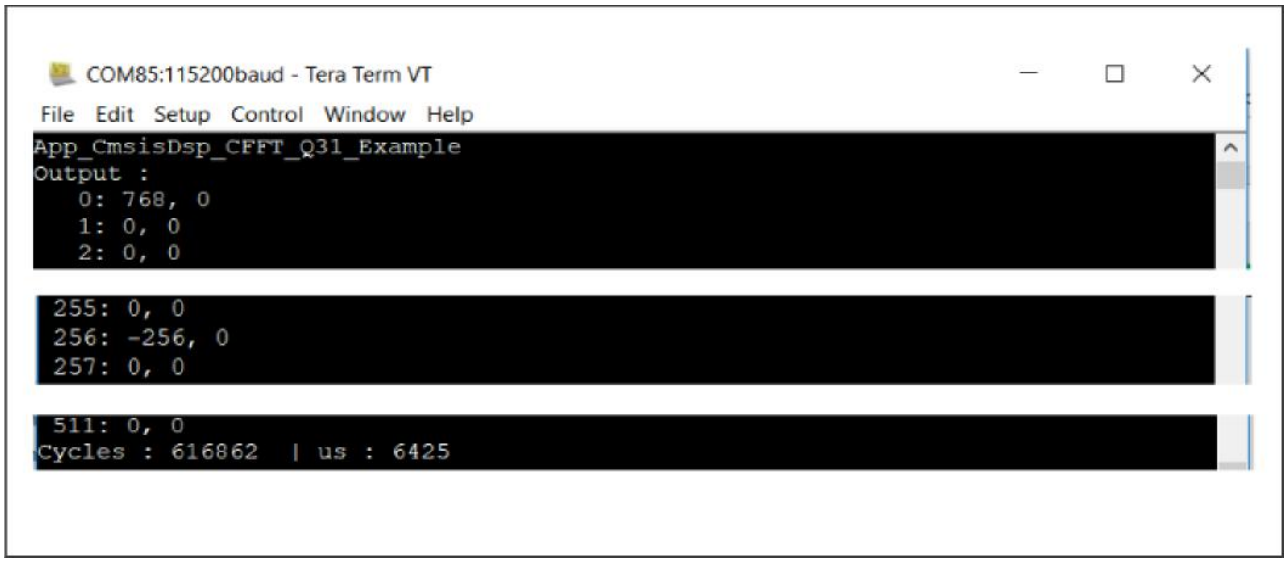

Figure 4. Terminal log for App_CmsisDsp_CFFT_Q31_Example

Per the code and terminal log shown for this case, we can see:

- FFT of the fixed-point version does the scale of `1/fftLen` inside the function. This way saves more significant figures and prevents the overflow during computing. To achieve the common target, in the code, manually use a prescaler on the software.

The fixed-point FFT functions shift the input automatically according to the computing length. To avoid saturations inside CFFT/CIFFT process, the input is downscaled by **2** for every stage. The output format is different with FFT size. Table 1 and Table 2 describe the input and output formats for different FFT sizes and number of bits to upscale.

Table 1. Input/Output format of Q31 CFFT in CMSIS-DSP

CFFT size	Input format	Output format	Number of bits to upscale
16	1.31	5.27	4
64	1.31	7.25	6
256	1.31	9.23	8
1024	1.31	11.21	10

Table 2. Input/Output format of Q31 CIFFT in CMSIS-DSP

CFFT size	Input format	Output format	Number of bits to upscale
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 Computing FFT with complex Q15 numbers

FFT of Q15 version in CMSIS-DSP costs less memory and time, but with less significant figures. It is suitable to process the data whose original format is 16-bit. Its usage is the same as the Q31 version. The pure 16-bit integer numbers with suitable shift can also be used, as we did in Q31 version's case.

The code for the task is:

```
/* 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. */
```

Figure 5 shows the result.

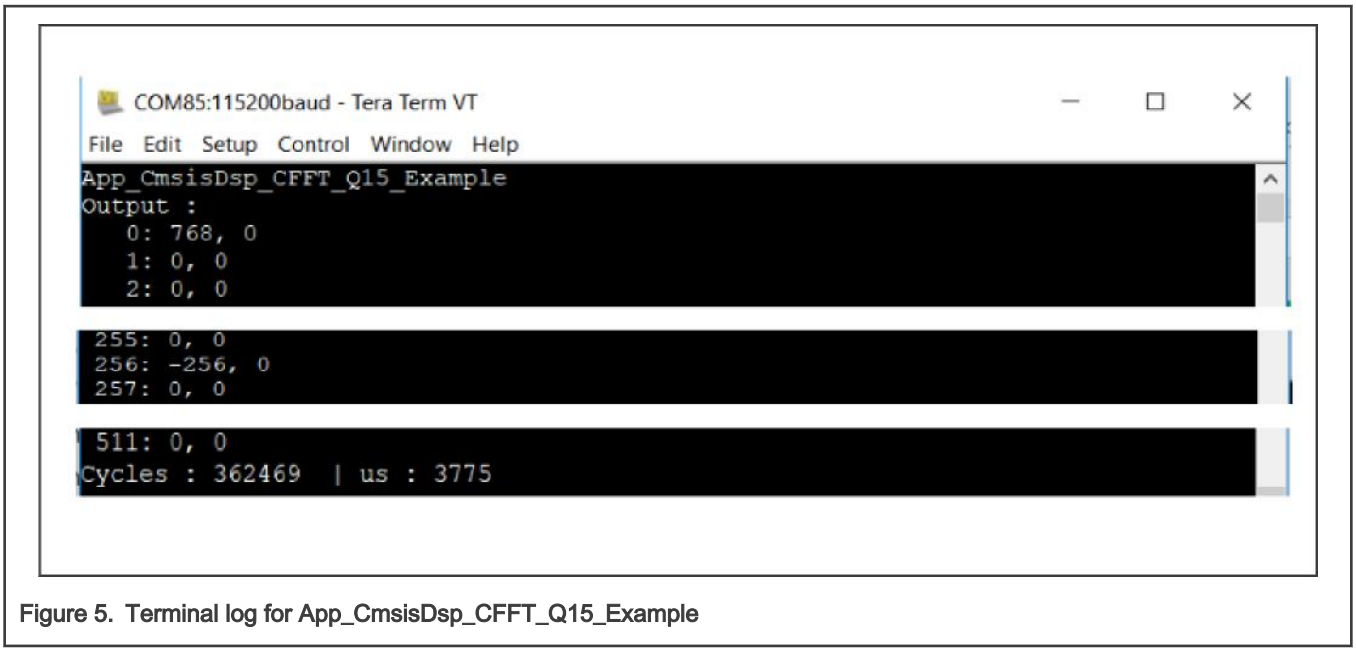

Figure 5. Terminal log for App_CmsisDsp_CFFT_Q15_Example

Per the code and terminal log shown for this case, we can see:

- FFT of Q15 version does the scale of `1/fftLen` inside the function, like Q31 version. To achieve the common target, in the code, manually use a prescaler on the software.

Table 3 and Table 4 describe the input and output format for the Q15 FFT.

Table 3. Input/Output format of Q15 CFFT in CMSIS-DSP

CFFT size	Input format	Output format	Number of bits to upscale
16	1.15	5.11	4
64	1.15	7.9	6
256	1.15	9.7	8
1024	1.151	11.5	10

Table 4. Input/Output format of Q15 CIFFT in CMSIS-DSP

CFFT size	Input format	Output format	Number of bits to upscale
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 Real FFT transforms

The FFT of a real N-point sequence is even symmetric in the frequency domain. The second half of the data equals the conjugate of the first half flipped in frequency. Only N/2 complex numbers represent the result uniquely. They are packed into the output array in alternating real and imaginary components.

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

It happens that the first complex number (`real[0], imag[0]`) is pure real number. `real[0]` represents the DC offset and `imag[0]` must be 0. Use the position of `imag[0]` to restore the `real[N/2]`. It is another pure real number. (`real[1]`, `imag[1]`) is the fundamental frequency, (`real[2]`, `imag[2]`) is the first harmonic, and so on.

The real FFT functions pack the frequency domain data in this fashion. The forward transform outputs the data in this form. The inverse transform expects input data in this form. The function performs the needed bit-reversal so that the input and output data is in normal order. The functions support lengths of **`[32, 64, 128, ..., 4096]`** samples.

The CMSIS DSP library includes specialized algorithms for computing the FFT of real data sequences. FFT is defined over complex data but in many applications, the input numbers are real. Real FFT algorithms take advantage of the symmetry properties of FFT and have a speed advantage over complex algorithms of the same length.

The Fast RFFT algorithm relays on the mixed radix CFFT that save processor usage. Figure 6 shows the steps of computing the real length N forward FFT of a sequence.

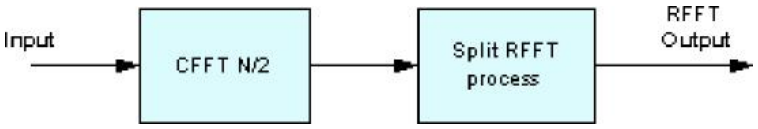

Figure 6. Real FFT

The real sequence was thought to be complex to perform a CFFT. Later, a processing stage reshapes the data to obtain half of the frequency spectrum in complex format. Except for the first complex number that contains the two real numbers **`x[0]`** and **`x[N/2]`**, all the data is complex. In other words, the first complex sample contains two real values packed.

Keep the input for the inverse RFFT the same format as the output of the forward RFFT. A first processing stage pre-processes the data to later perform an inverse CFFT.

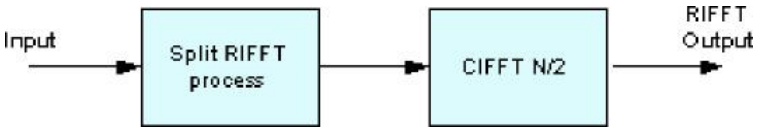

Figure 7. Real inverse FFT

As a summary for using the N point real FFT:

- The length of input array is N, with N real numbers.
- The length of output array is also N, with N/2 complex number, for the first half of the frequency spectrum. The second half of the data equals the conjugate of the first half flipped in frequency.
- The first complex number of the output array is packed with the two real number, `real[0]` and `real[N/2]`.

5.2.1 Computing FFT with real F32 numbers

To replace the old one for computing the real floating-point FFT, CMSIS-DSP provides a new API with fast. Now, only the APIs of `arm_rfft_fast_init_f32()/arm_rfft_fast_f32` are recommended for computing. The input and output memory are in-place as the complex FFT functions. The input and output memory are separated in user code. The way of outputting numbers is a little different.

The code for the task is:

```
/* 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. */
```

Figure 8 shows the result.

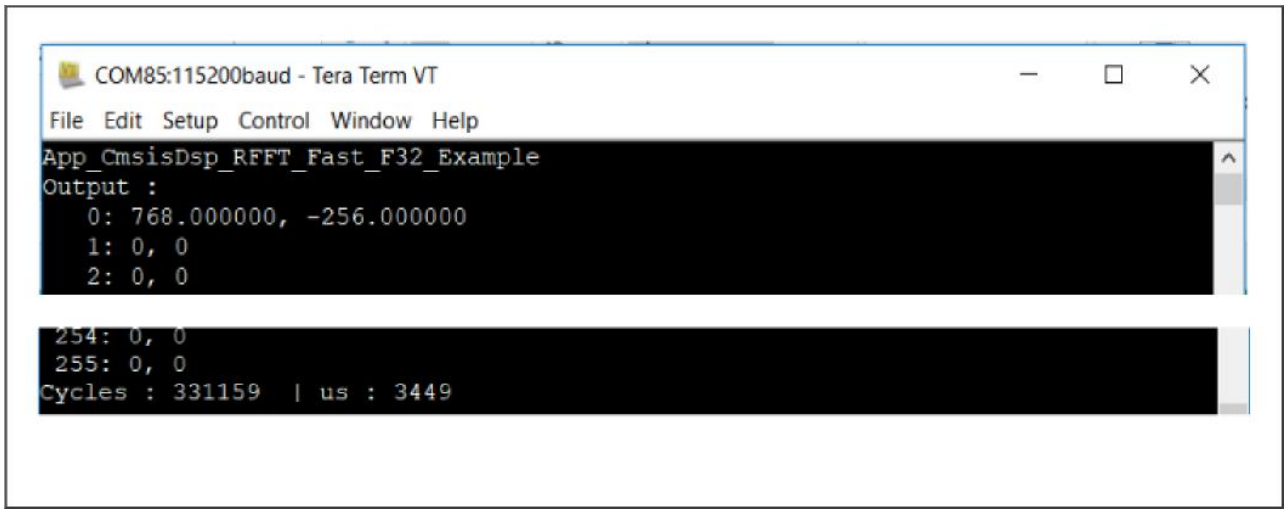

Figure 8. Terminal log for App_CmsisDsp_RFFT_Fast_F32_Example

Per the code and terminal log shown for this case, we can see:

- The output numbers are for the complex numbers but with the half-length of input items (the input is with 512 real numbers in 512 memory items, the output is with 256 complex numbers in 512 memory items).

  The first item of output array is different from others.

  — The first complex number (`real[0]`, `imag[0]`) is all real.

  — `real[0]` represents the DC offset.

  — `imag[0]` must be 0.

  — (`real[1]`, `imag[1]`) is the fundamental frequency, (`real[2]`, `imag[2]`) is the first harmonic, and so on.

5.2.2 Computing FFT with real Q31 numbers

The real FFT of Q31 is different from the floating-point version using a fast way. It uses the old format, like in complex FFT function. The input array is packed with all the real numbers. The output array is for the complex numbers without length reduced. It means that the memory for the output array is twice in size of the memory for the input array.

The code for the task is:

```
/* 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. */
```

Figure 9 shows the result.

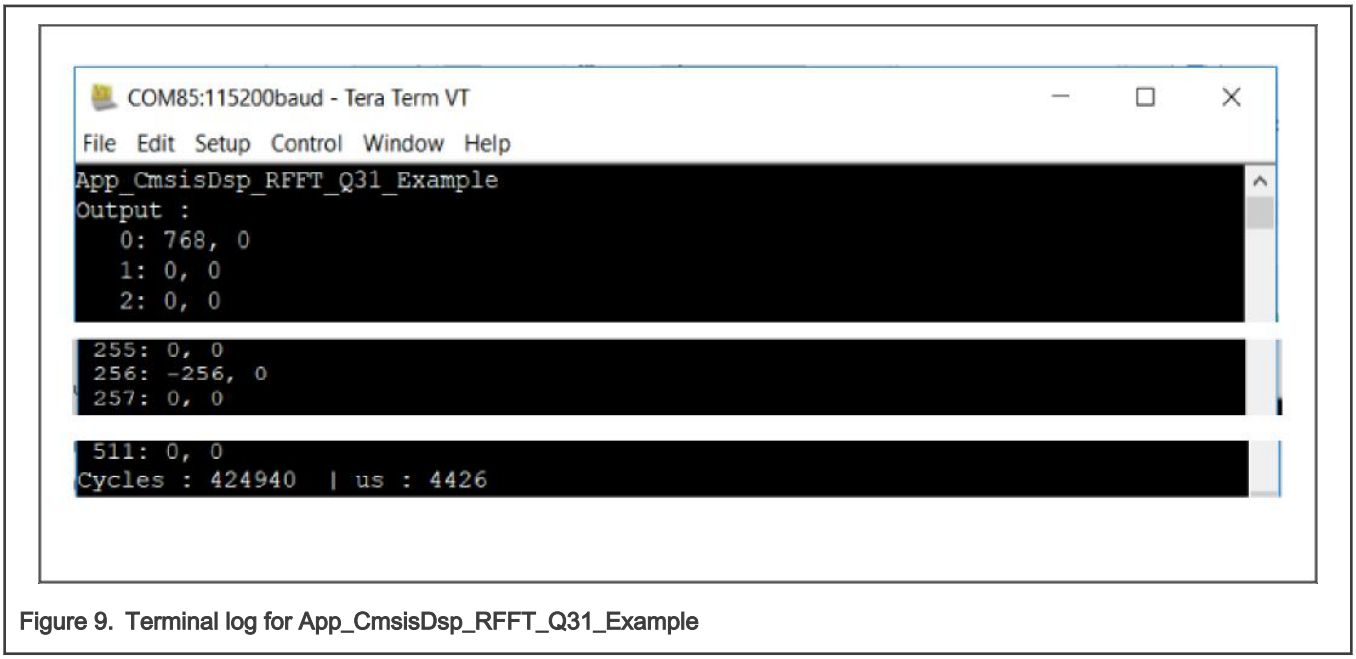

Figure 9. Terminal log for App_CmsisDsp_RFFT_Q31_Example

Per the code and terminal log shown for this case, we can see:

- The prescaler is used to achieve the common target.
- The length of available input array is 512 for the 512 real numbers. The length of available output array is 1024 for the 512 complex numbers.
- The output array is with the same format as for the traditional complex functions. The first number is not special as the fast floating-point real FFT.

5.2.3 Computing FFT with real Q15 numbers

The real FFT of Q15 version inherits the characters of Q31 version.

The code for the task is:

```
/* 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. */
```

Figure 10 shows the result.

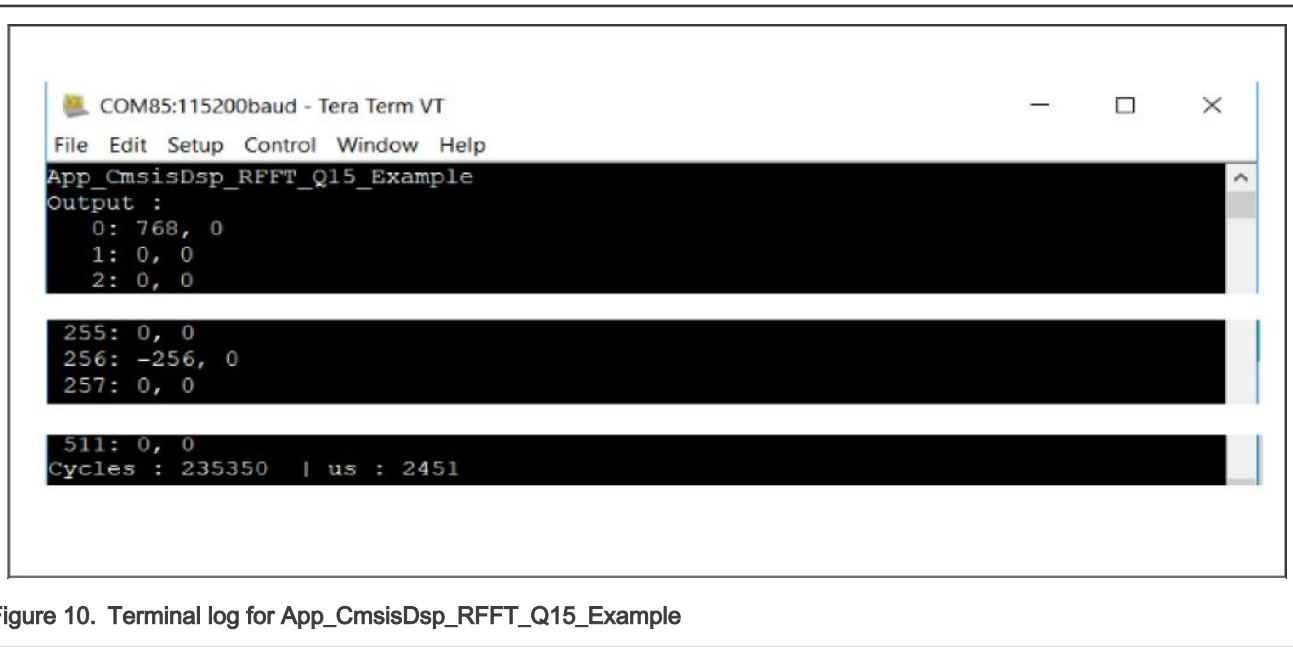

Figure 10. Terminal log for App_CmsisDsp_RFFT_Q15_Example

Per the code and terminal log shown for this case, we can see:

- It looks same as the Q31 version.
- It runs a little faster than the Q31 version.

6 Computing FFT with PowerQuad hardware

The pure software implementation of CMSIS-DSP APIs is limited by:

- the architecture of Arm core (the narrow memory bus)
- the performance of the compiler (the optimizing condition of different level)

The hardware implements and optimizes the computing engines (including FFT engine) of PowerQuad. Comparing the usage of CMSIS-DSP, it saves a lot of CPU load and code size with significant performance improvement. PowerQuad is integrated as a coprocessor. To meet the requirement in the real-time system, PowerQuad runs with Arm core in parallel.

NXP MCUXpresso SDK software library supports the PowerQuad module. Within the PowerQuad driver, there are a group of APIs for computing FFT:

- `PQ_TransformCFFT()`
- `PQ_TransformRFFT()`
- `PQ_SetConfig ()` sets the format of various fixed-point.

PowerQuad hardware does not support the floating-point FFT. To unlock this feature, use a software solution based on the existing PowerQuad hardware. The solution covers the same field applying for CMSIS-DSP FFT APIs.

Below discusses the usage of APIs.

6.1 Fixed-point complex FFT transforms

PowerQuad FFT engine supports only fixed-point FFT transform, so process the fixed-point FFT task directly on the PowerQuad hardware.

6.1.1 Computing FFT with complex Q31 numbers

In previous CMSIS-DSP cases, to achieve the common target output, a software prescaler is used for the input numbers. PowerQuad hardware provides a new option with hardware prescaler settings. Both input and output number have their owner hardware prescaler setting. In this case for the 512-point FFT, the prescaler number is 512 and the responding setting value for `pq_cfg.inputAPrescale` is 9. The input value is left shift 9 bits as the multiplication with 512.

When configuring the input and output format for PowerQuad hardware:

- Use Input A, Temp, and Output memory handlers for FFT engine.
- The hardware only supports fixed-point FFT.
- The format settings for these memory handler, in `pq_cfg.inputAFormat`, `pq_cfg.tmpFormat`, and `pq_cfg.outputFormat`, are for the fixed-point, like `kPQ_32Bit` or `kPQ_16Bit`. In this case, they are `kPQ_32Bit`.
- Ignore the setting for output memory handler for FFT engine.
- The input and output array must be 32-bit words.

The numbers input array for FFT of the complex number is assembled with the real part and the imaginary part. Each part takes one 32-bit word in memory. The output numbers are the complex numbers.

To keep the intermediate data during computing, temp memory handler uses the private RAM starting from `0xE000_0000`. For the 512-point FFT, to keep the 512 complex numbers with 1 K 32-bit word, reserve 4 kB memory in the private RAM.

The critical function in this case is the `PQ_TransformRFFT()`. With Q31 numbers as input and output, the input numbers are complex ones.

The code for the task is:

```
/* 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. */
```

Figure 11 shows the result.

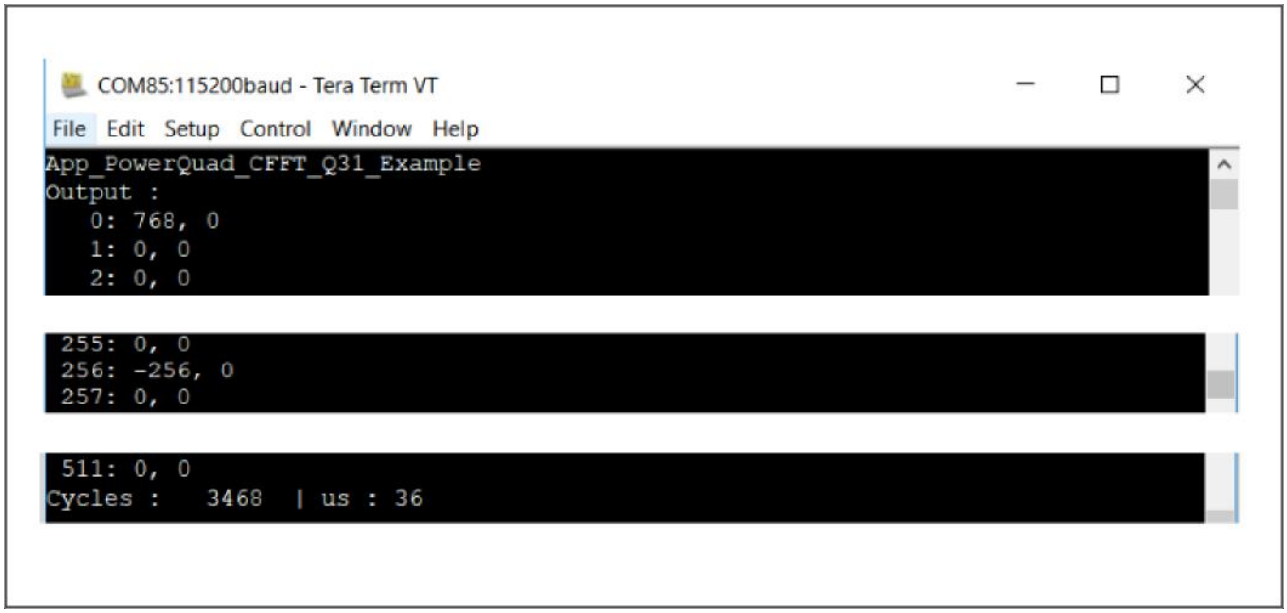

Figure 11. Terminal log for App_PowerQuad_CFFT_Q31_Example

Per the code and terminal log shown for this case, we can see:

- The hardware prescaler takes effect, just like the software scaler.
- PowerQuad hardware creates the expected result (common target).
- It is really faster than the CMSIS-DSP complex Q31 fixed-point FFT function.

The prescaler for output fixed-point numbers can reuse the table for the output of CMSIS-DSP fixed-point FFT.

6.1.2 Computing FFT with complex Q15 numbers

With the PowerQuad FFT engine, the complex Q15 task is the same with the complex Q31 task. The difference is:

- The data format settings for `pq_cfg.inputAFormat`, `pq_cfg.tmpFormat`, and `pq_cfg.outputFormat` are `kPQ_16Bit`.

The code for the task is:

```
/* 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. */
```

Figure 12 shows the result.

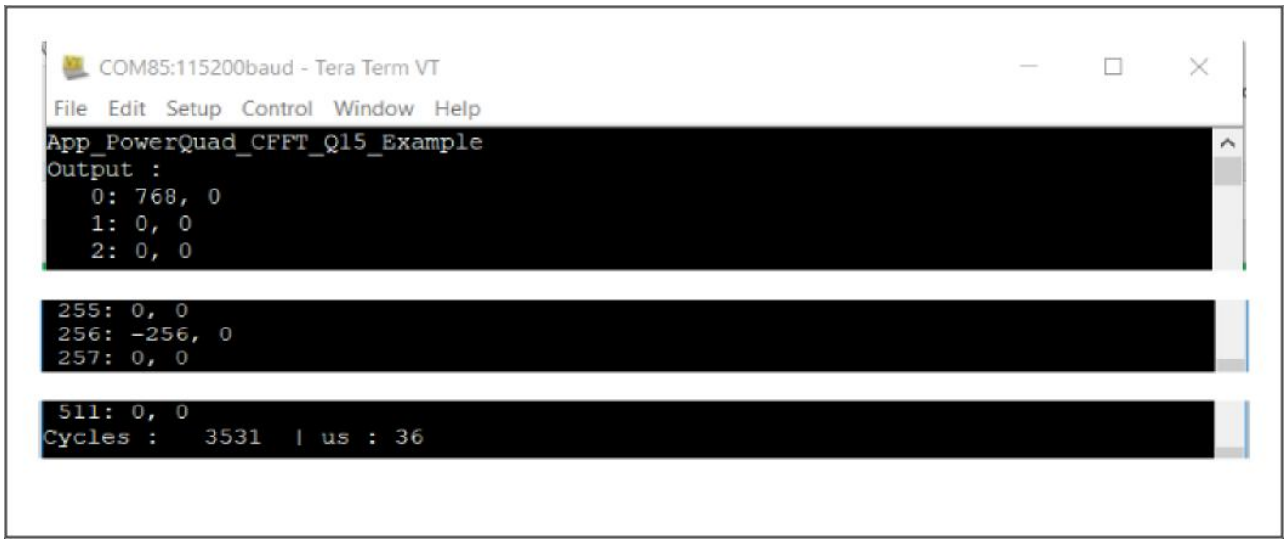

Figure 12. Terminal log for App_PowerQuad_CFFT_Q15_Example

Per the code and terminal log shown for this case, we can see:

- The hardware prescaler takes effect as well.
- PowerQuad hardware creates the expected result (common target).
- It is not faster than the complex Q31 FFT. In actual cases, it runs even more slowly. The fewer bits in the number does not reduce the workload of PowerQuad hardware.

6.2 Fixed-point real FFT transforms

FFT of the pure real number by PowerQuad hardware packs the imaginary part. It keeps only the real part of numbers in the input array. It saves half-length of the memory than FFT of the complex number. The PowerQuad hardware can recognize this way. However, the PowerQuad keeps the output as complex numbers (CMSIS-DSP APIs use the same way).

6.2.1 Computing FFT with real Q31 numbers

The critical function is the `PQ_TransformRFFT()` but with the Q31 numbers as input and output. The input numbers are pure real ones.

The code for the task is:

```
/* 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. */
```

Figure 13 shows the result.

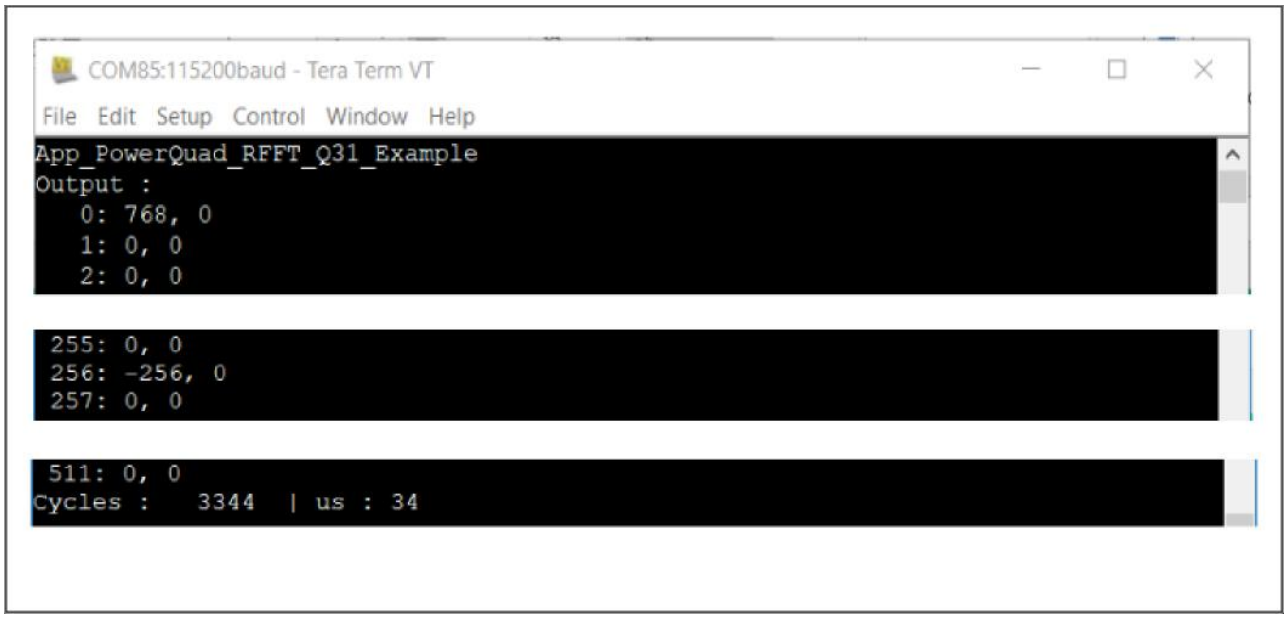

Figure 13. Terminal log for App_PowerQuad_RFFT_Q31_Example

Per the code and terminal log shown for this case, we can see:

- The hardware prescaler takes effect as well.
- The expected result (common target) is created by the PowerQuad hardware.
- Due to the reduced memory operation, it is a little faster than the complex Q31 FFT.
- The length of output numbers does not reduce to half like CMSIS-DSP functions. No special format is necessary against the complex FFT computing.

6.2.2 Computing FFT with real Q15 numbers

With the PowerQuad FFT engine, the read Q15 task is almost same with the real Q31 task. The difference is:

- The data format settings for `pq_cfg.inputAFormat`, `pq_cfg.tmpFormat`, and `pq_cfg.outputFormat` are `kPQ_16Bit`.

The code for the task is:

```
/* 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. */
```

Figure 14 shows the result.

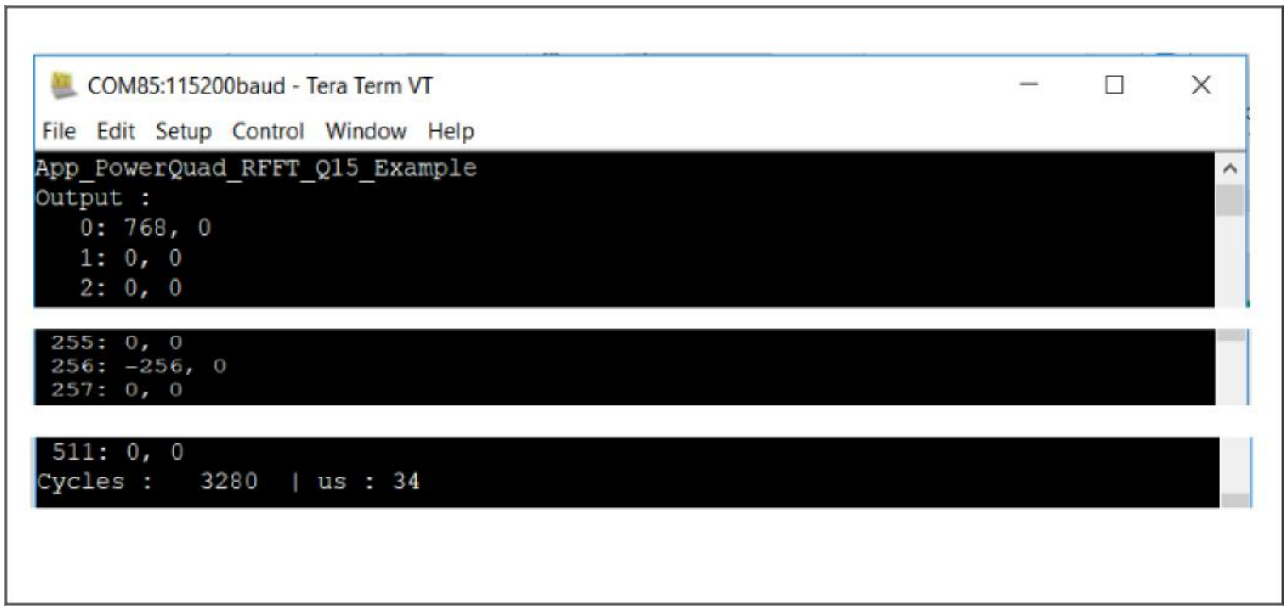

Figure 14. Terminal log for App_PowerQuad_RFFT_Q15_Example

Per the code and terminal log shown for this case, we can see:

- The hardware prescaler takes effect as well.
- The expected result (common target) is created by the PowerQuad hardware.
- Due to the reduced memory operations, it is a little faster than the complex Q31 FFT.
- The length of output numbers does not reduce to half like CMSIS-DSP functions. No special format is necessary against the complex FFT computing.

6.3 Float-point FFT transform

PowerQuad hardware does not support the floating-point FFT directly. In some applications, to get the advantage from the powerful acceleration of PowerQuad hardware computing engine but with less code change, replace the existing CMSIS-DSP APIs for floating-point FFT with the implementation of PowerQuad. A data format conversion between floating-point and fixed-point is necessary.

Fortunately, the matrix scale function of PowerQuad can deal with the format conversion by hardware. It runs faster than ARM-CMSIS DSP APIs of `arm_float_to_q31()`/`arm_q31_to_float()`. To connect the operations of converting floating-point input numbers to fixed-point one, fixed-point FFT and converting fixed-pointed output to floating-point one, create a floating-point FFT function all based on the PowerQuad hardware.

6.3.1 Format conversion using PowerQuad matrix scale function

CMSIS-DSP contains APIs about converting the floating-point numbers to fixed-point numbers, such as, `arm_float_to_q31()` and `arm_q31_to_float()`. In the PowerQuad module, set the input and output with different value format and execute the matrix scale with the scaler of 1.0f. The value is not changed from input and output. Perform the conversion automatically when moving value from input buffer to output buffer.

The example code of format conversion between floating-point value and fixed-point value is:

```
/* 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. */
```

Figure 15 shows the result.

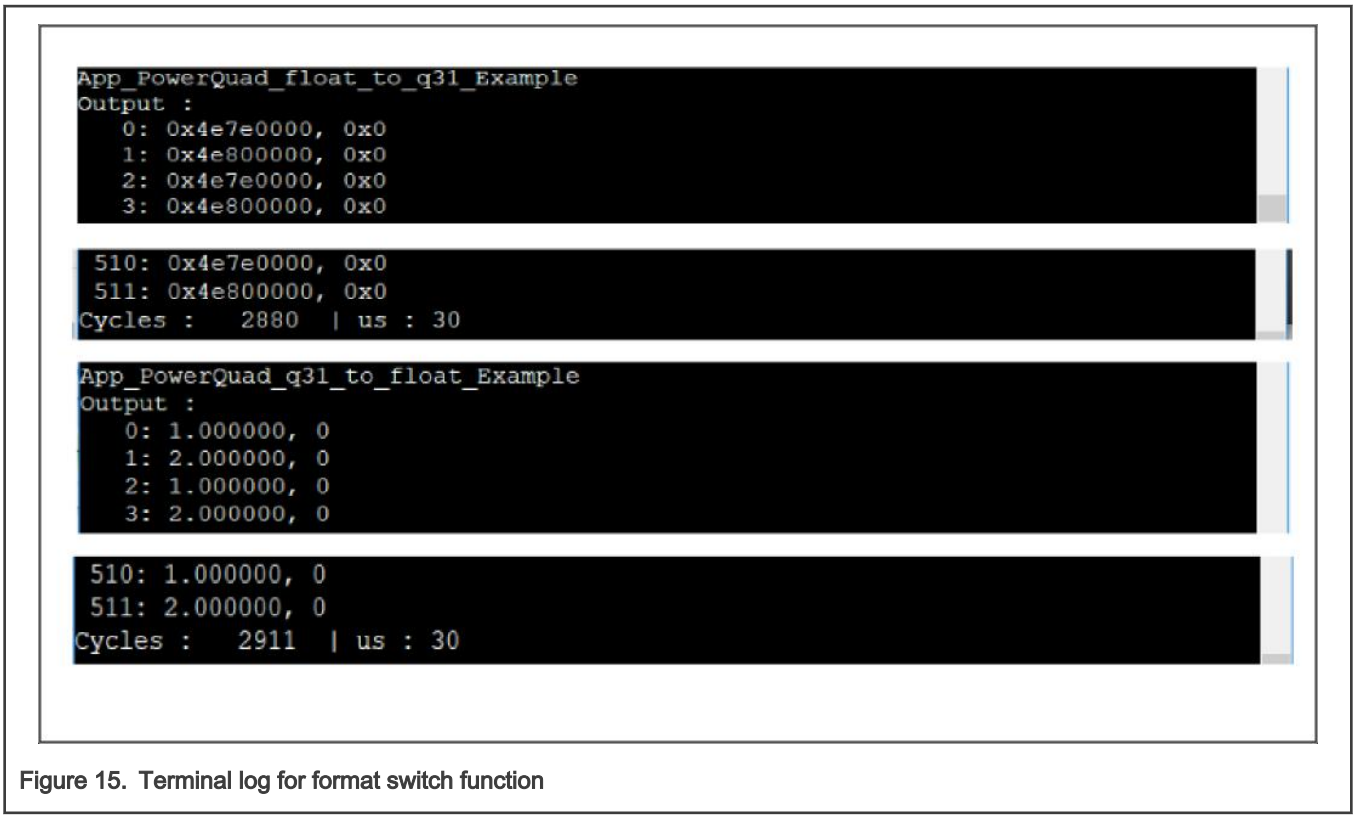

Figure 15. Terminal log for format switch function

Run the same test cases with ARM-CMSIS DSP APIs. Without the compiling optimization, the `arm_float_to_q31()` and `arm_q31_to_float()` are slower than the conversion functions of PowerQuad. Below lists the limitations when using the conversion function:

- For CMSIS-DSP APIs, to follow the standard q31 format, the fixed-point numbers must not be out of the range (-1, 1) .
- For PowerQuad APIs, the maximum length for the array is 256. To process the longer array, call the matrix scale function more times.

6.3.2 Computing FFT with complex F32 numbers

In this case,

1. Call the 256-point matrix scale function for four time.
2. Convert the 512 floating-point input complex numbers (1024 numbers in the array) to fixed-point input numbers.
3. Run the hardware FFT.
4. Get the output fixed-point numbers.
5. Call 256-point Matrix scale functions for four times.
6. Get the floating-point output number.

The code for the task is:

```
/* 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. */
```

Figure 16 shows the result.

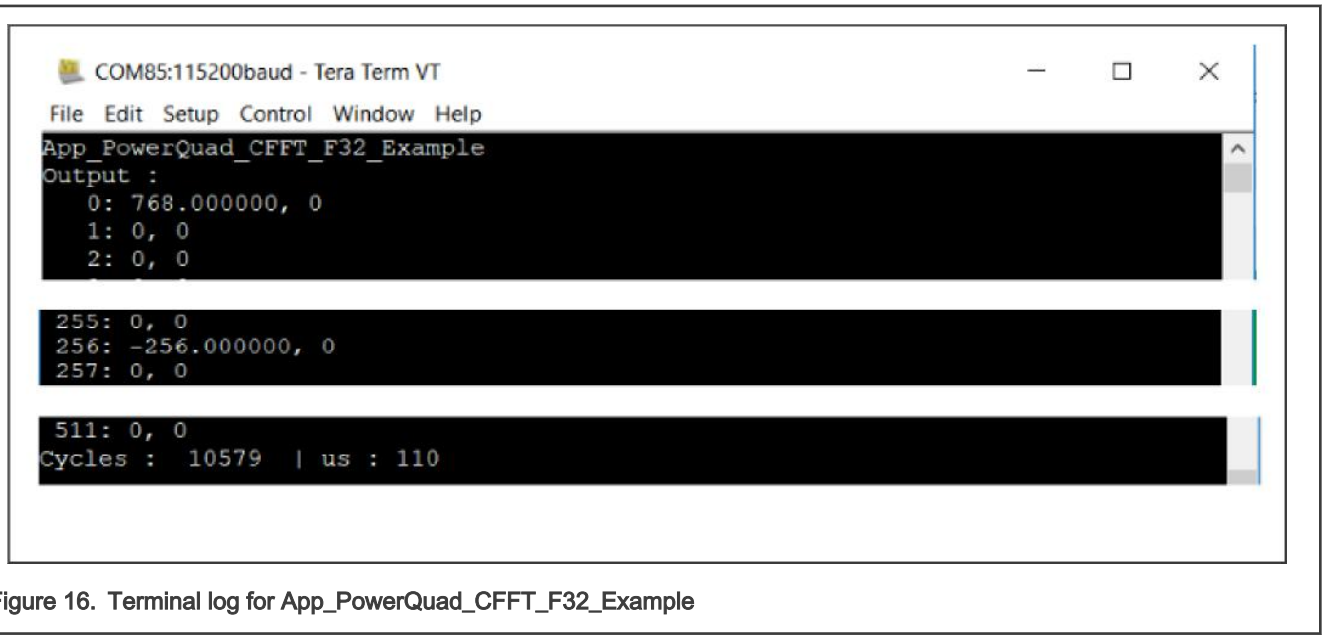

Figure 16. Terminal log for App_PowerQuad_CFFT_F32_Example

Per the code and terminal log shown for this case, we can see:

- The result is correct, same with the floating complex FFT of CMSIS-DSP.
- The hardware conversion functions work well.
- The time it runs almost equals to the time for 2 x PowerQuad Matrix Scale + 1 x PowerQuad CFFT. It looks faster than the `arm_cfft_f32()` function in CMSIS-DSP.

6.3.3 Computing FFT with real F32 numbers

In this case,

1. Convert the input array of packed real floating numbers to Q31 numbers.
2. Use the FFT engine of PowerQuad to compute with `PQ_TransformRFFT()` function.
3. Get the output of Q31 numbers.
4. Use the matrix Scale function of PowerQuad.
5. Convert to the floating-point format.

The code for the task is:

```
/* 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. */
```

Figure 17 shows the result.

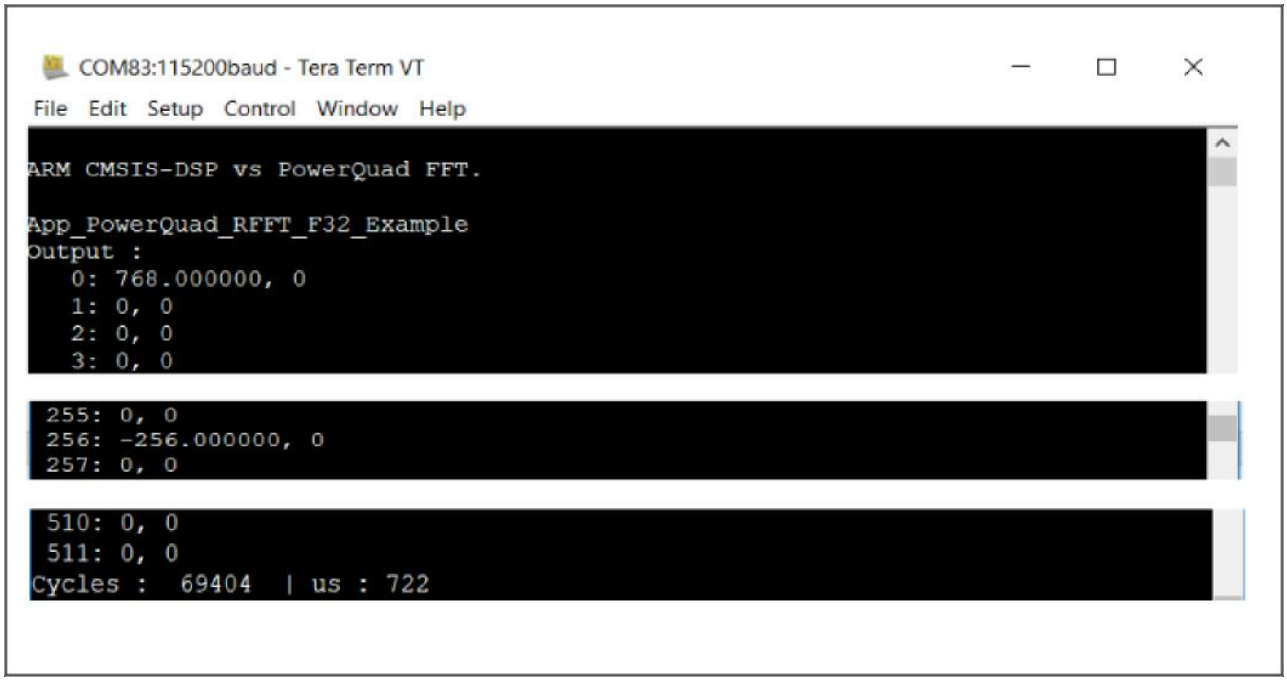

Figure 17. Terminal log for computing FFT with real F32 numbers

Per the code and terminal log shown for this case, we can see:

- Per the conversion number of Arm CMSIS-DSP, the input numbers are zoomed down into the range (-1, 1). The output of the conversion number is the strict Q31 number. We use the integer-like fixed-point number (with `q0` format) and an additional zoom down to the input-floating numbers. Then we can get the common target like other demo cases.
- Due to the workaround, the `arm_flaot_to_q31()` function consumes the most time of the whole process. Even through, it runs faster than the implementation of pure software. For the timing comparison, see Summary and conclusion.

7 Summary and conclusion

This application note tells the usage of computing FFT with CMSIS-DSP software and PowerQuad hardware in the same computing case. PowerQuad hardware can replace CMSIS-DSP software when computing FFT for the same format of input and output. The demo cases show that PowerQuad runs much faster than CMSIS-DSP.

This section lists some tables about the timing characters for the demo cases. The summary shows the capability of PowerQuad' accumulation. The different compiling optimization conditions are set in the **Project Option** dialog box in the IAR IDE, as shown in Figure 18.

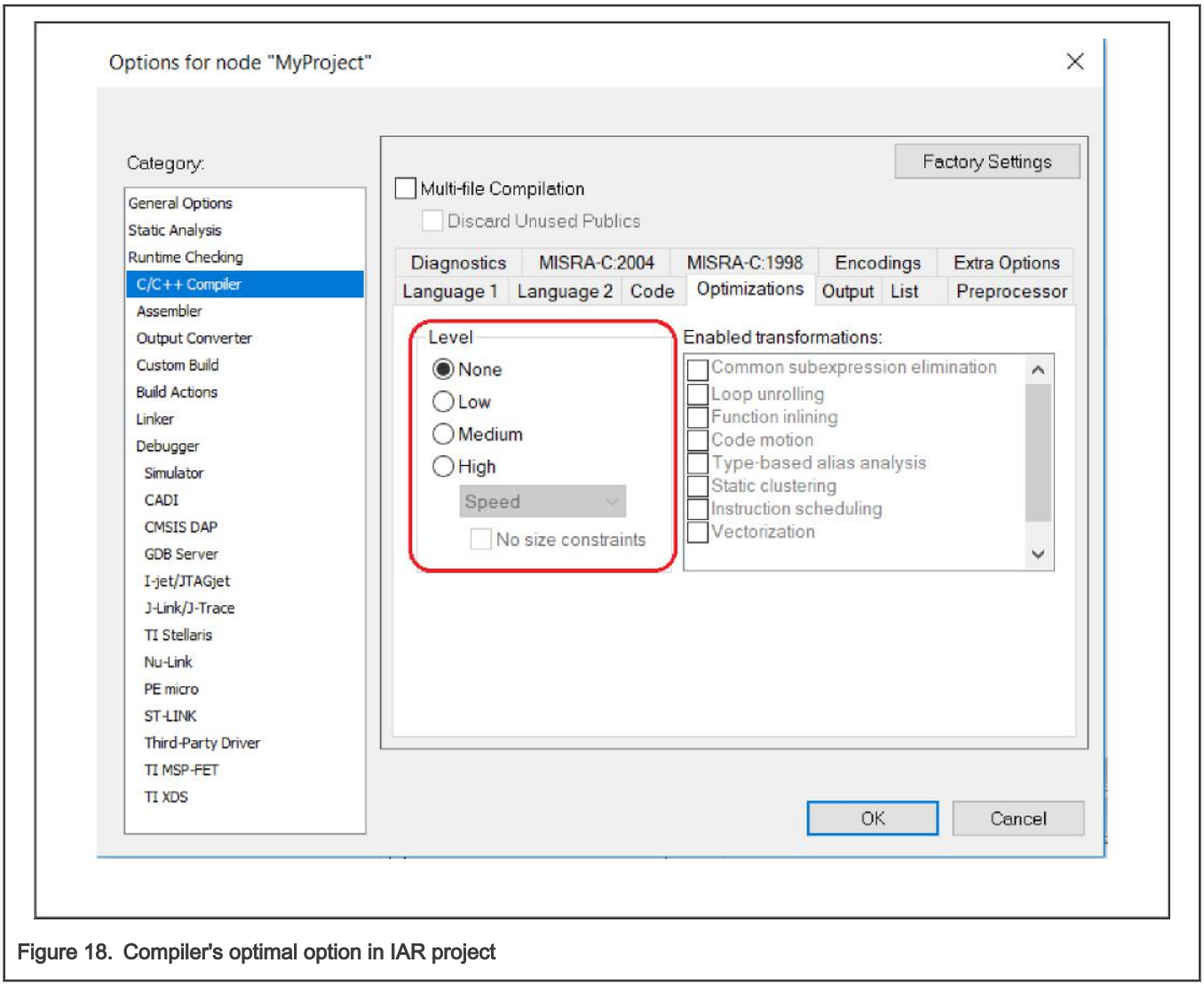

Figure 18. Compiler's optimal option in IAR project

Table 5 summarizes the measuring time.

Table 5. Measuring time on optimal conditions

Demo cases	None		Low		Medium		High (speed)		None (FPU disabled)	
	cycles	µs	cycles	µs	cycles	µs	cycles	µs	cycles	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

Table continues on the next page...

Table 5. Measuring time on optimal conditions (continued)

Demo cases	None		Low		Medium		High (speed)		None (FPU disabled)	
	cycles	µs	cycles	µs	cycles	µs	cycles	µs	cycles	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

In Table 5, we can see:

- The computing of PowerQuad is much faster than CMSIS-DSP functions, about x100 times faster in measuring values.
- The timing performance of PowerQuad is stable for FFT computing with different format numbers and compiling optimization conditions. But the performance of CMSIS-DSP software varies with the compiling optimization condition. When implementing CMSIS-DSP software, the optimization of a higher level does not run the code faster (in `App_CmsisDsp_CFFT_Q15_Example`, the optimization of a low-level optimization runs 1876 µs while that of a medium level runs 1978 µs).
- The computing of fixed-point is not faster than floating-point. When the hardware FPU is disabled, the computing of floating-point needs more CPU cycles with general fixed-point instructions. On this condition, the fixed-point algorithm runs more smoothly. When FPU is enabled for compiler, the computing instruments of floating-point save more time and calculate the floating-point number directly in one instrument, while that of fixed-point needs more instruments to convert a large number of calculations into several steps and cost more time. Therefore, when the FPU is enabled for

compiler, `App_CmsisDsp_CFFT_F32_Example` demo case runs faster than `App_CmsisDsp_CFFT_Q31_Example`, but much slower when FPU is disabled.

- The format conversion between floating-point numbers and fixed-point numbers costs much time. At the same level, both are for CMSIS-DSP software and PowerQuad hardware.
- For `App_PowerQuad_RFFT_F32_Example` demo case, even with the software workaround about format conversion issue and replaced with part of implementation from ARM CMSIS-DSP, it is still about x3 times faster than the pure software way. As the complex floating-point FFT runs faster but with less additional memory, it is more recommended. Or, to achieve the best performance, modify the original data format to fixed-point number in the application.

When running on 150 MHz core clock, the record is as shown in Table 6.

Table 6. Measuring time on various conditions with 150 MHz core clock

Demo cases	None		Low		Medium		High (speed)		None (FPU disabled)	
	cycles	µs	cycles	µs	cycles	µs	cycles	µs	cycles	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

Table continues on the next page...

Table 6. Measuring time on various conditions with 150 MHz core clock (continued)

Demo cases	None		Low		Medium		High (speed)		None (FPU disabled)	
	cycles	µs	cycles	µs	cycles	µs	cycles	µs	cycles	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 Revision history

Rev.	Date	Description
0	31 December 2021	Initial release

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: 31 December 2021
Document identifier: AN13496

arm