AN12383

Computing FFT with PowerQuad and CMSIS-DSP on LPC5500

Rev. 0 — 11/2019 Application Note

Contents

1 Introduction

Fast Frontier Transform (FFT) is almost the most popular computation in Digital Signal Processing (DSP) application. It can make the transform between timing field to frequency field. When the sample array of signal is transformed from timing field to frequency field, some useful and interesting attributes would appear and can be easily used to find out the pattern of signals. With this feature, FFT is widely used to extract 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 provided a group of APIs to fulfill the requirement of computing FFT on Cortex®-M MCUs. However, the functions in CMSIS-DSP are purely implemented by software, even it is well optimized. That means the computing time depends on the compiler's optimization condition and the CPU's performance heavily. Also, the computing time of the complex process like FFT purely by software is usually not short, and this should be considered carefully in the real-time application.

The PowerQuad hardware module is designed to accelerate some general DSP computing tasks, including the Math Functions, Matrix Functions, Filter Functions and the Transform Functions (including FFT). As the computing is totally executed by specific hardware other than the Arm core, it runs fast and saves CPU time. The PowerQuad can be considered as a simplified DSP hardware but with less power consumption and well integrated inside the Arm ecosystem, so the development based on it could be very friendly.

About the usage of the fixed-point FFT and the floating-point FFT, they have their each specific implementation and application in different field. The fixed-point FFT is mostly used to process the audio, video and other data captured from hardware sensor modules like ADC, while the original direct sample value for these condition is fixed-point. For the floating-point FFT, it is commonly used to process the longitude and latitude with high accuracy and high resolution in navigation system. So, both the fixed-point FFT and the floating-point FFT would be discussed together in the paper.

2 PowerQuad hardware FFT engine

The PowerQuad provides Discrete Fourier Transforms and Discrete Cosine Transforms, implemented with a Radix-8 Butterfly structure Fast Fourier Transform engine using fixed-point arithmetic at a resolution of 24 bits.

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

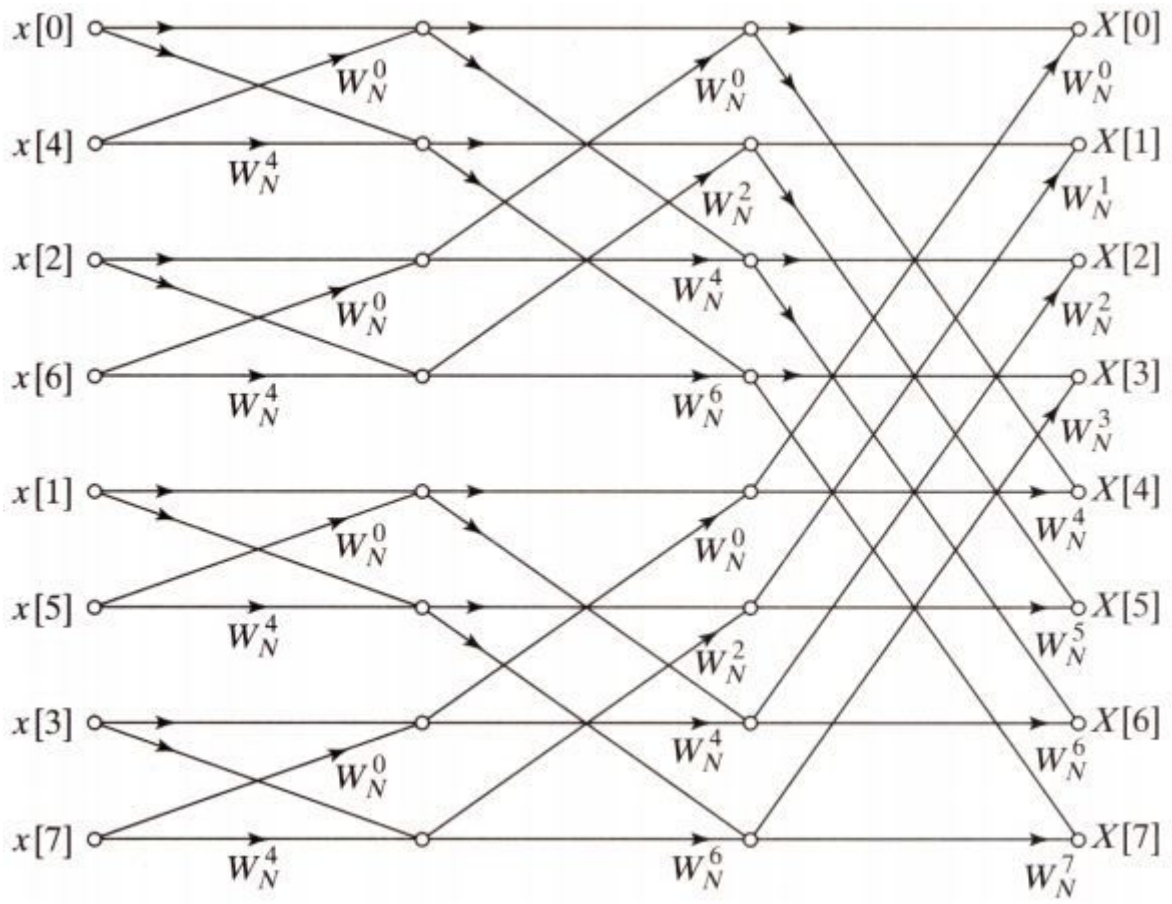

Figure 1. Radix-8 butterfly structure of PowerQuad FFT engine

2.1 Computing equations

The Discrete Fourier Transform 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:

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

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

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

It follows that X_0 and $X_{N/2}$ are read values, and the remainder of the DFT is completely specified by just **N/2 -1** complex numbers.

NOTE

Although PowerQuad's FFT computing engine can support the Discrete Cosine Transform by hardware as well, it is not so popular as the FFT, and it can be computed by the Matrix way in a simpler way, which is also supported by PowerQuad Matrix computing engine. Using Matrix computing engine to compute the DCT is easier and more flexible than FFT computing engine. So, this applinication note will not describe the DCT in detail, as its usage is almost the same with the FFT.

2.2 Input and output details

2.2.1 Fixed-point numbers only for FFT engine

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

NOTE

The FFT engine only looks at the bottom 27 bits of the input word, so any pre-scaling must not exceed this to avoid saturation.

If the floating-point numbers' FFT is required in application, user must convert the floating-point input numbers to the fixed-point numbers, launching the computing and converting the output fixed-point numbers to the floating-point ones. Fortunately, the PowerQuad's Matrix engine provides a function of matrix scale, which would accelerate the conversion by mixed format computing.

2.2.2 Input and output sequence in memory

The purely real (prefixed by **r**), and the complex flavors of the functions (prefixed by **c**) expect the input data sequences to be arranged in memory as follows.

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

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

  then the input array in memory must be organized as:

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

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

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

The output sequence will always be stored in memory organized as an array of complex numbers where the imaginary parts will be 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, so that the values will not be overflow during the computing of both DFT and inverse DFT. If an unscaled result is necessary, the input data before being placed in the INPUT A region must first be multiplied by N, or setup the hardware prescaler for the INPUT A region.

The inverse FFT is also scaled by 1/N, but this is correct as per the inverse DFT formula, so no scaling treatment is needed.

When the application is willing to replace the CMSIS-DSP's FFT API which was already used in the existing project, to keep the input and output data to be aligned, the prescaler should be added manually. However, if the application is newly designed, it can be considered to omit this step, as the proportional relation among the outputs is still the same, which is the most important information of the FFT computing.

The following 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. PowerQuad can access this part of memory exclusively without any arbitration delay, so that to accelerate the whole process of computing as fast as possible. As the PowerQuad access the four banks of memory with 32-bit bus simultaneously in an interleave way, it can achieve equivalent 128-bit bus band wide. Using private RAM is encouraged because it means that PowerQuad can access the data quicker and therefore access one operand from RAM and one from system at same time, which provides performance improvement.

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, because it was meant for floating point data (which is the native form of PowerQuad). Generally, all the address space in the private RAM can be used for the four memory handlers, **INPUT A**, **INPUT B**, **TEMP** and **OUTPUT**. And choosing a memory handler's format has no effect when data is traveling in and out of the private RAM.

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

Actually, the TEMP area is only used for the FFT (for intermediate calculations) and Matrix inversion. For the 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 read the input and write the output with 4 words (128-bit) a time, the allocated memory address for the PowerQuad memory handler should be 4-word (or 16-byte) aligned. The FFT is a special case here as well, for the TEMP memory handler, it needs the alignment to its space size. For example, 512 points means 512 complex pairs, then it needs to align 1024 words.

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

3 Measuring time in demo project

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

In the demo code for this paper, SysTick timer is chosen as the hardware timer, so that the code here could be well portable for the other Arm Cortex-M MCUs. Then use the 24-bit counter value directly for timing. For the LPC5500, which is running at 96 MHz for the SysTick timer's clock source, the max timing period could 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 would be:

```
uint32_t cycles;

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

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

The running time of each functional case in this paper would be measured in different condition, the measuring time would be 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)**.

All in all, the values of inputs are the same for different computing cases.

4.2 [OUTPUT]

The output array of values would be all zero except for:

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

This output makes sense. From the original input array, it can be seen that the average value of the input number is 1.5, and the amplitude of the simple switching waveform is 0.5, that means 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 1/2, and the phase would be negative. No other frequency factors.

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

However, using the general mathematic calculator (like Matlab) would simplify the step of 1/N when outputting the result. That means, the direct output would multiple N from the final result. In the cases for this paper, the actual result should be: result [0] = 768, result [256] = -256.

The result can also be proved by the calculation with FreeMat software (an opensource version of MabLab-like mathematic calculator, http://freemat.sourceforge.net/) using the follow script.

```
--> 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 in 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
```

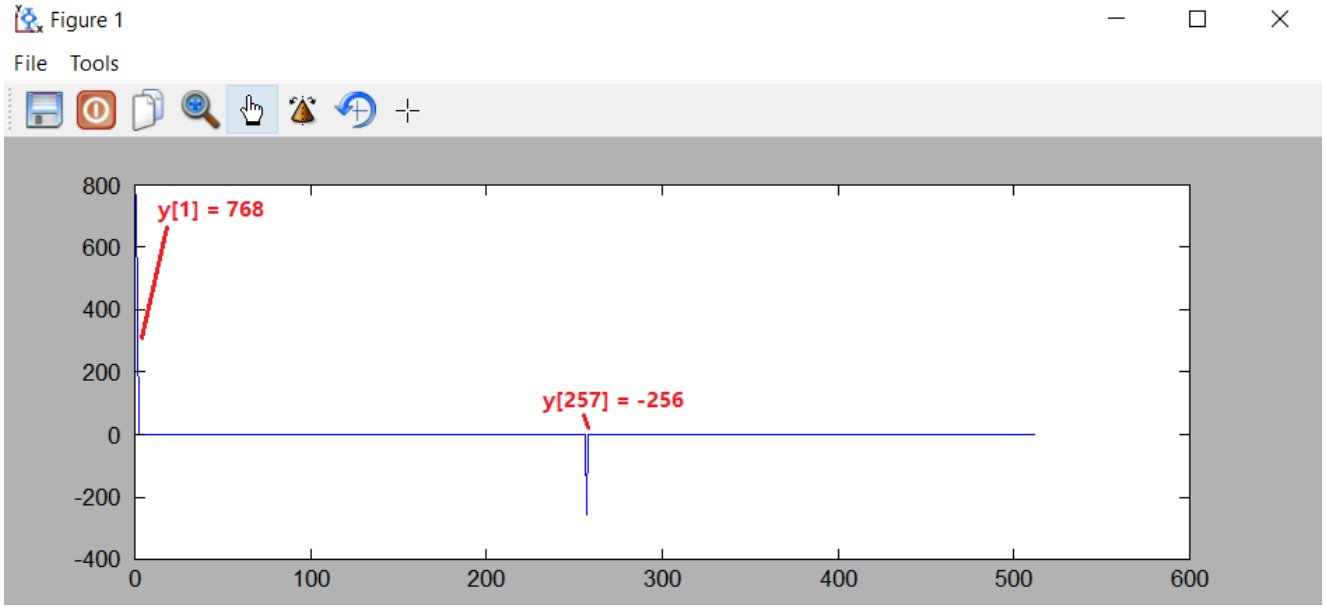

Figure 2. FFT calculation by FreeMat

5 Computing FFT with CMSIS-DSP software

Before showing the usage of PowerQuad FFT engine, here tells the usage of CMSIS-DSP FFT APIs which are already well known by the MCU-based DSP developers. The CMSIS-DSP FFT APIs are implemented by optimized software.

The Fast Fourier Transform (FFT) is an efficient algorithm for computing the Discrete Fourier Transform (DFT). The 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, the array holding the input data will also be used to hold the corresponding result. The input data is complex and contains 2*fftLen interleaved values as shown below.

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

The FFT result will be contained in the same array and the frequency domain values will have 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, please refer to http://www.keil.com/pack/doc/CMSIS/DSP/html/group__groupTransforms.html.

The following describes the usage of APIs for various formats. All the cases are runnable 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 each length uses a different twiddle factor table.

The function uses the standard FFT definition and output values may **grow by a factor of fftLen** when computing the forward transform. The inverse transform includes a scale of 1/fftLen as part of the calculation and this matches the textbook definition of the inverse FFT.

Pre-initialized data structures containing twiddle factors and bit reversal tables are provided and defined in the source file `arm_const_structs.h`. Include this header in your function and then 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.

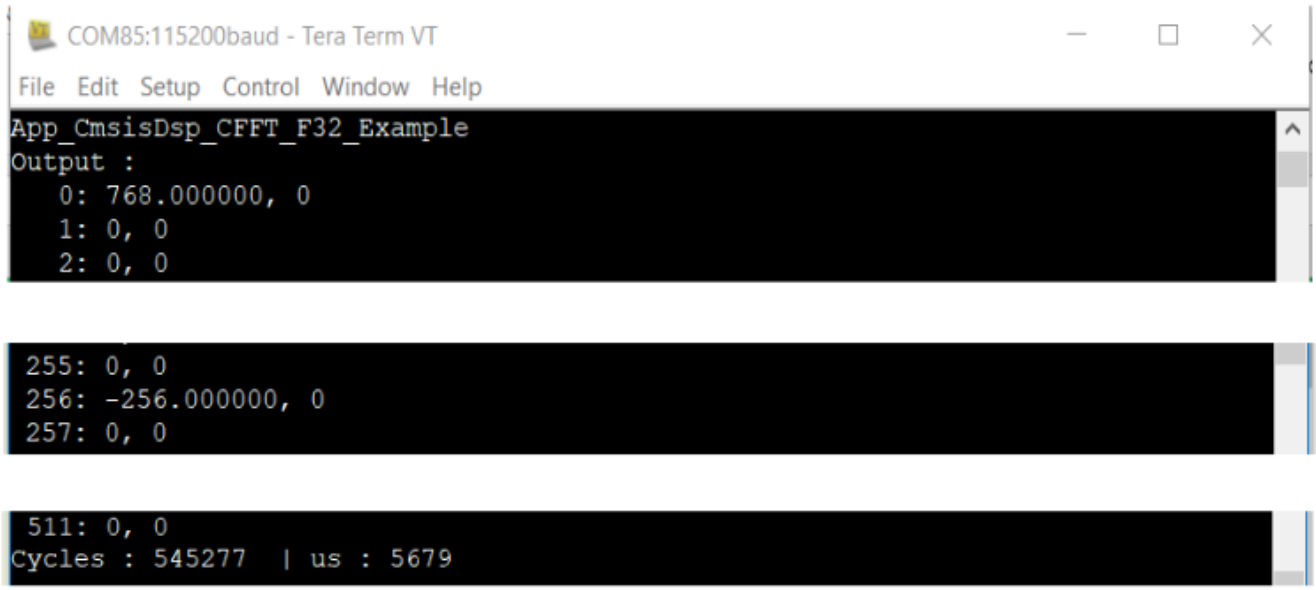

Figure 3. Terminal log for App_CmsisDsp_CFFT_F32_Example

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

- It is proven that the memory of `inputF32[]` is actually modified by the computing function, and the input numbers are covered by the output numbers. The output number is using two items as the real part and complex part for a complex value.
- It is proven that the CMSIS-DSP function ignores the 1/fftLen scale for the result. All the following case would use the result without 1/fftLen scale as the common target.
- The running timegoes with no compiling optimization. Table 5 summarizes all the computing time in different optimal condition.

5.1.2 Computing FFT with complex Q31 numbers

The Q31's version FFT is implemented differently from the floating-point one. Also, maybe the fixed-point number's range would confuse you, since the Q31 number should in the range of (-1, 1). However, in the application level of this case, they are just used as the pure 32-bit integers, or can be seen as a Q0 in fixed-point format. This consideration makes sense, since the output of FFT would be mostly used as normal values to feed 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.

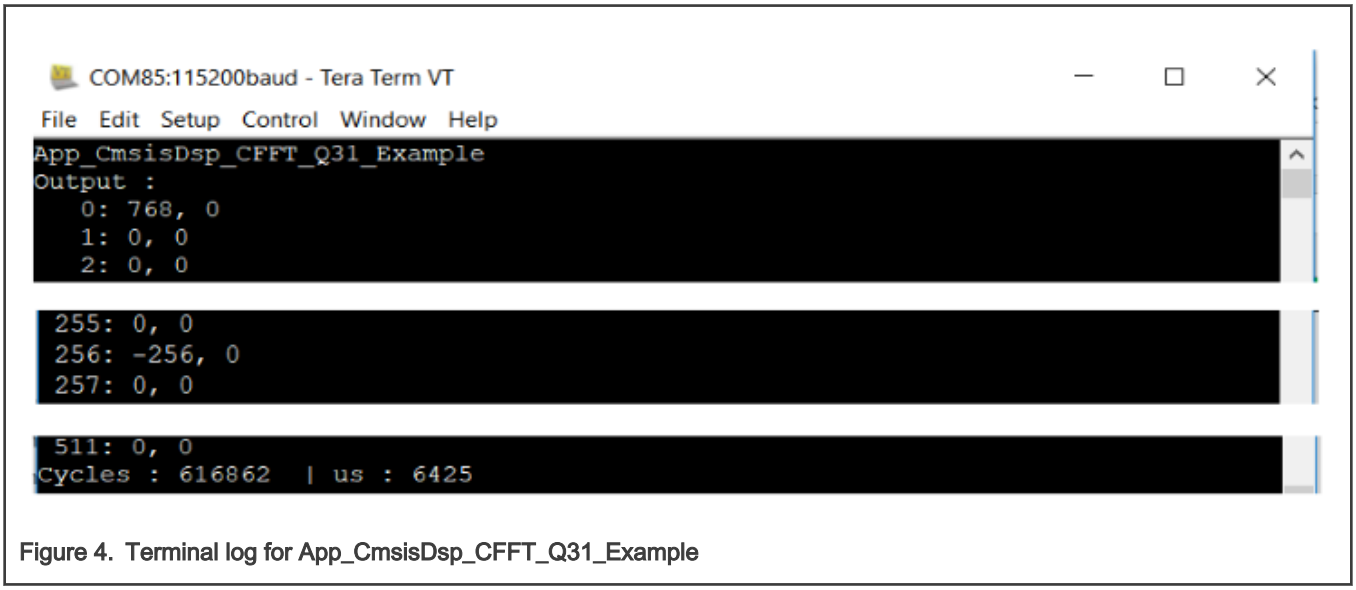

Figure 4. Terminal log for App_CmsisDsp_CFFT_Q31_Example

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

- The fixed-point version's FFT does the scale of 1/fftLen inside the function. This way can save more significant figures and prevent the overflow during computing. However, as we need to achieve the common target as the floating-point one, in the code, a prescaler is used manually by software.

Actually, the fixed-point FFT functions would shift the input automatically according to the computing length. Internally, the input is downscaled by 2 for every stage to avoid saturations inside CFFT/CIFFT process. Hence 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

CIFFT 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

Q15 version's FFT in CMSIS-DSP is expected to cost less memory and time, but with less significant figures. It is also suitable to process the data which original format is 16-bit. Its usage is the same as the Q31 version. Also, we can still use the pure 16-bit integer numbers with suitable shift as we did in Q31 version's case before.

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.

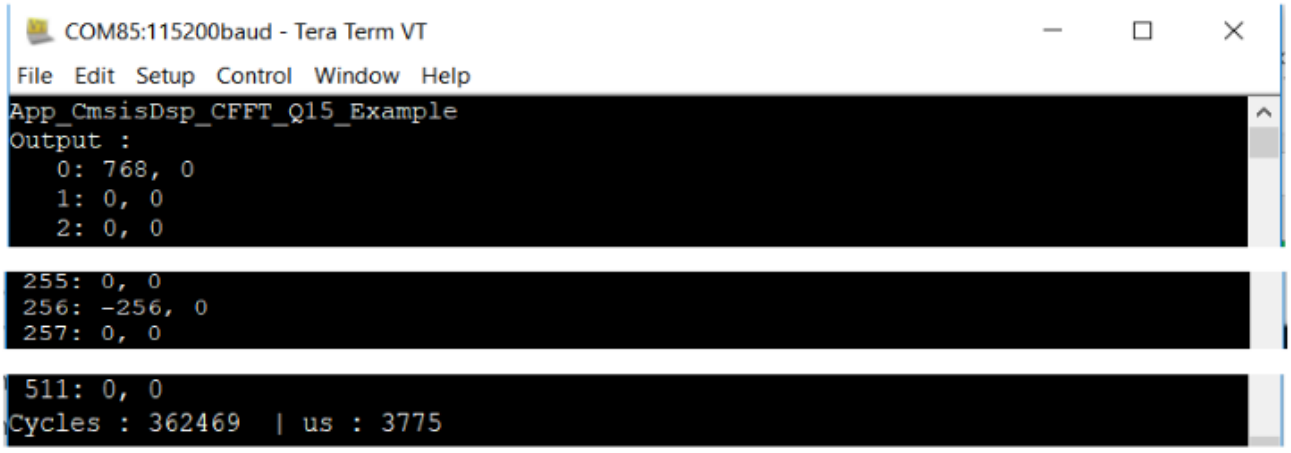

Figure 5. Terminal log for App_CmsisDsp_CFFT_Q15_Example

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

- The Q15 version's FFT does the scale of 1/fftLen inside the function like the Q31 version's. To achieve the common target, in the code, a prescaler is used manually by 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

CIFFT 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 has even symmetry in the frequency domain. The second half of the data equals the conjugate of the first half flipped in frequency. So, the result can be uniquely represented using only N/2 complex numbers. These 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 actually pure real, while the `real[0]` represents the DC offset and `imag[0]` should be **0**. So the position of `imag[0]` can be used to restore the `real[N/2]`, which 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 and the inverse transform expects input data in this form. The function always performs the needed bit-reversal so that the input and output data is always 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. The 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 the 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.

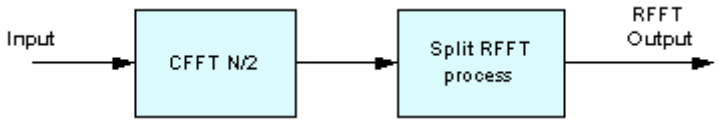

Figure 6. Real fast fourier transform

The real sequence is initially treated as if it were 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.

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

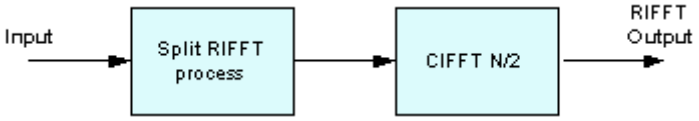

Figure 7. Real inverse fast fourier transform

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, since 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 actually packed with the two real number, `real[0]` and `real[N/2]`.

5.2.1 Computing FFT with real F32 numbers

CMSIS-DSP provides a new API with **fast** to replace the old one for computing the real floating-point FFT. Now, the APIs of `arm_rfft_fast_init_f32()/arm_rfft_fast_f32` are the only recommended way for computing. Also, the input and output

memory would not be **in-place** as the complex FFT functions. The input memory and memory are separated in user code. And the way of outputting numbers is a little different, which needs more attention.

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 8shows the result.

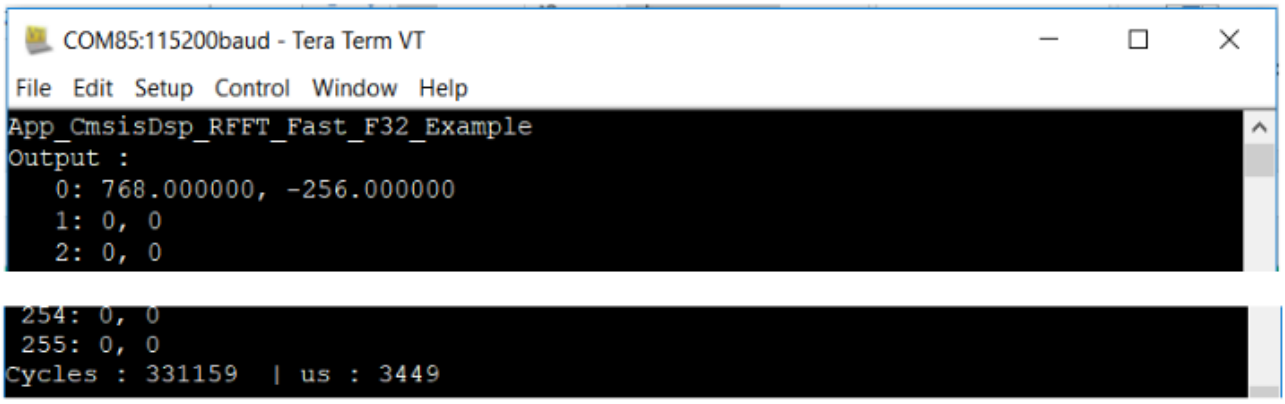

Figure 8. Terminal log for App_CmsisDsp_RFFT_Fast_F32_Example

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

- About the output numbers. The items are still for the complex numbers, but with the half length of the 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 actually all real. `real[0]` represents the DC offset, and `imag[0]` should 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

Q31's real FFT is totally 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, and the output array is for the complex numbers without length reduced. That means the memory for the output array would be twice 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.

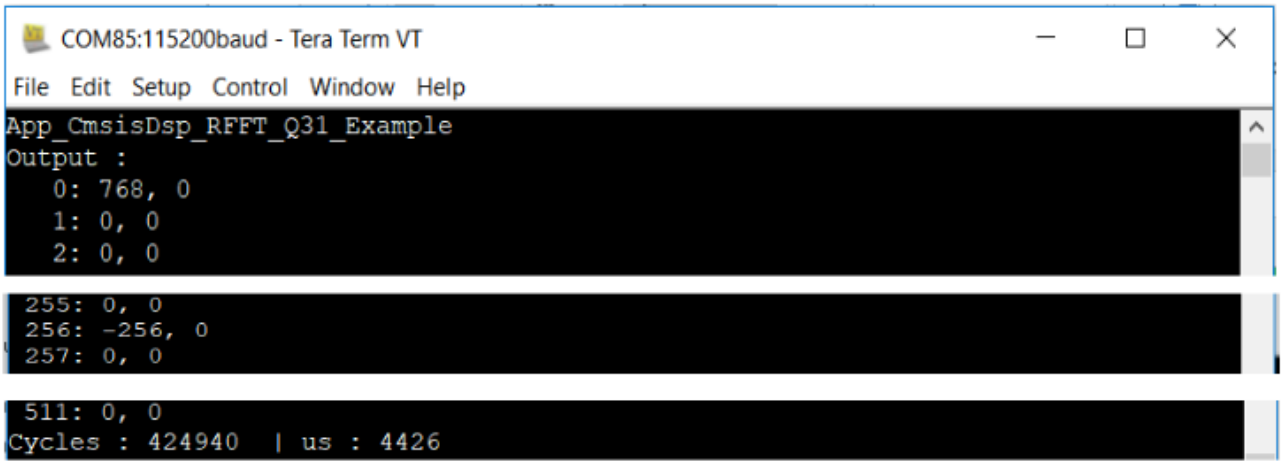

Figure 9. Terminal log for App_CmsisDsp_RFFT_Q31_Example

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

- The prescaler is used to achieve the common target.
- The available input array's length is 512 for the 512 real numbers, the available output array's length 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 did.

5.2.3 Computing FFT with real Q15 numbers

Q15's version real FFT inherits the characters of the Q31's 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.

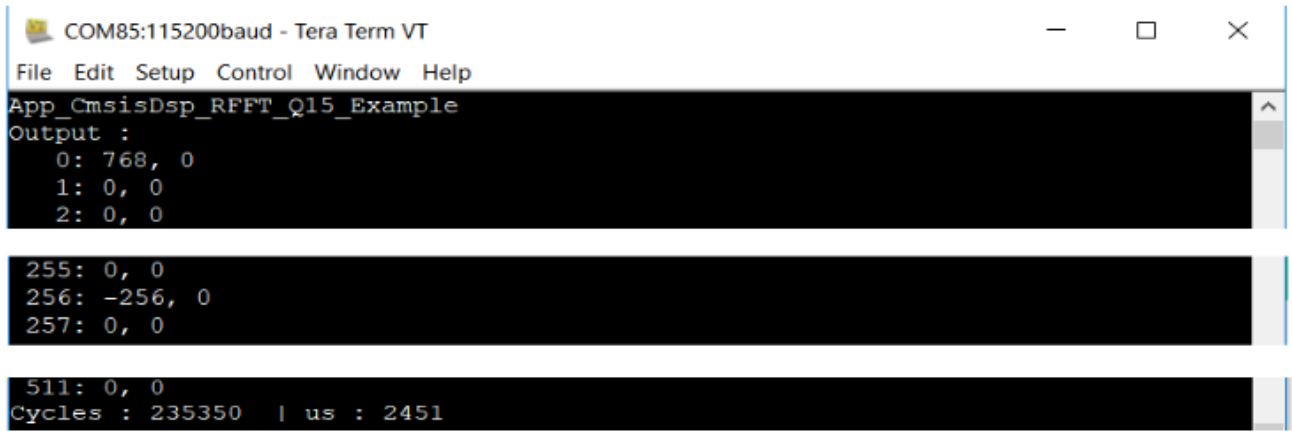

Figure 10. Terminal log for App_CmsisDsp_RFFT_Q15_Example

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

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

6 Computing FFT with PowerQuad hardware

However, the pure software implementation of CMSIS-DSP APIs is still limited by the Arm core's architecture (the narrow memory bus) and compiler's performance (different level's optimizing condition). But on the other side, the PowerQuad's computing engines (including the FFT engine) are implemented and optimized by hardware, comparing the usage of CMSIS-DSP, it saves a

lot of CPU load and code size with significant performance improvement. Also, as integrated as a co-processor, the PowerQuad can also run with Arm core parallelly if necessary, to meet the requirement in the real-time system.

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

- `PQ_TransformCFFT()`
- `PQ_TransformRFFT()`
- `PQ_SetConfig ()` is used to setup the format of various fixed-point.

The floating-point FFT is not originally support by PowerQuad hardware. However, a software solution based on existing PowerQuad hardware is created to unlock this feature, so it can cover the same field applying for the CMSIS-DSP FFT APIs.

The following will discuss the usage of APIs.

6.1 Fixed-point complex FFT transforms

PowerQuad FFT engine hardware support only fixed-point FFT transform, so the fixed-point FFT task can be processed directly by the PowerQuad hardware.

6.1.1 Computing FFT with complex Q31 numbers

In the previous CMSIS-DSP cases, to achieve the common target output, a software prescaler is applied to the input numbers. For the PowerQuad, the hardware provides a new option which can be done by hardware prescaler setting. The both the input and the output number have their owner hardware prescaler setting. In this case for the 512-point FFT, the prescaler number should be 512, the responding setting value for `pq_cfg.inputAPrescale` is **9**, as the input value would left shift 9 bits as the multiplication with 512.

About configuring the input and output format for PowerQuad hardware. As the Input A, Temp and Output memory handlers are used for the FFT engine while the hardware only supports fixed-point FFT, the format settings for these memory hander, 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**. The setting for the Output memory hander will be ignored for FFT engine. Also, the input and output array must be the 32-bit words.

The numbers input array for complex number's FFT, is assembled with the real part and the imaginary part, while each part would take one 32-bit word in memory. The output numbers are always the complex numbers.

Private RAM starts from `0xE000_0000` is used by the Temp memory handler, to keep the intermediate data during computing. For the 512-point FFT, to keep the 512 complex numbers with 1 K 32-bit word, totally 4KB memory should be actually reserved in the Private RAM.

The critical function in this case is the `PQ_TransformRFFT()` but with the Q31 numbers as input and output, while 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.

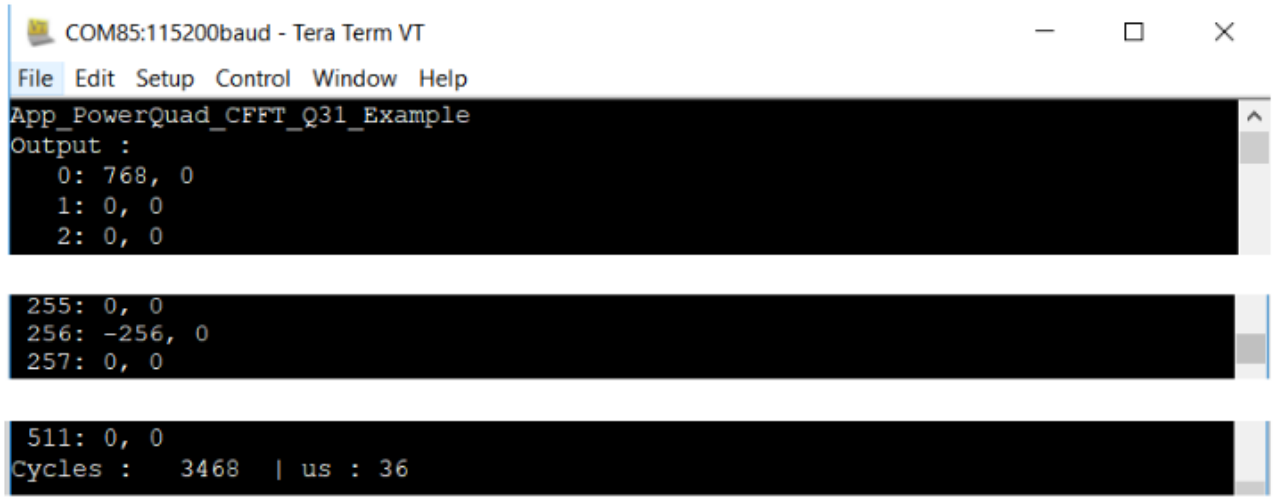

Figure 11. Terminal log for App_PowerQuad_CFFT_Q31_Example

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

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

Actually, about the usage of the prescaler for output fixed-point numbers here can reuse the table for CMSIS-DSP fixed-point FFT's output.

6.1.2 Computing FFT with complex Q15 numbers

With the PowerQuad FFT engine, the complex Q15 task is almost the same with the complex Q31 task while 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.

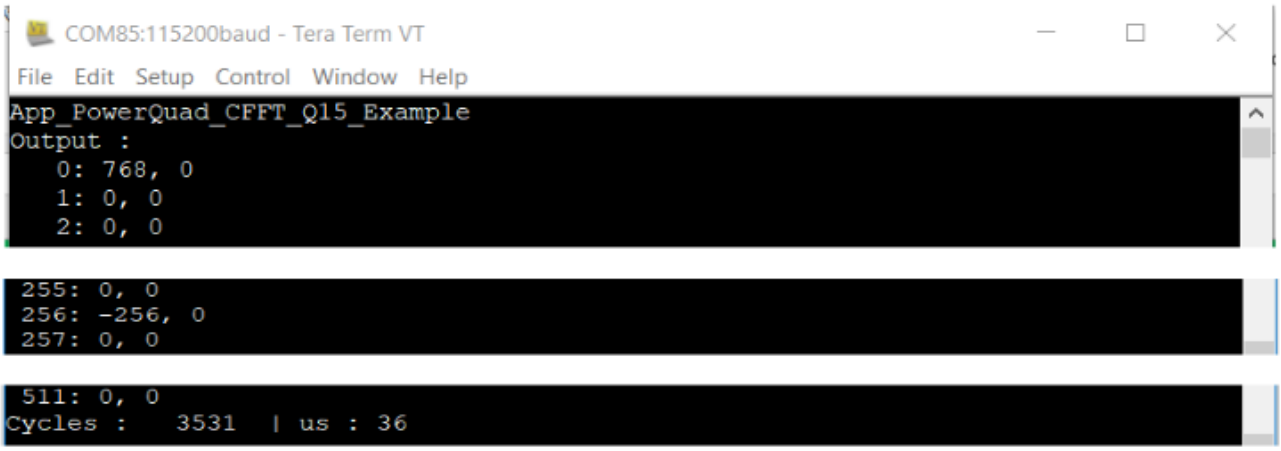

Figure 12. Terminal log for App_PowerQuad_CFFT_Q15_Example

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

- The hardware prescaler takes effect as well.
- The expected result (common target) is created by PowerQuad hardware.
- It is not faster than the complex Q31 FFT, even a little slower in the actual run. Therefore, the lesser bits in the number will not reduce the workload of PowerQuad hardware.

6.2 Fixed-point real FFT transforms

The pure real number's FFT by PowerQuad hardware packs the imaginary part and only keep the real part of numbers in the input array. It saves half length of the memory than the complex number's FFT and the PowerQuad hardware can also recognize this way. However, the PowerQuad always keep the output as complex numbers. (The CMSIS-DSP APIs are using 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, while 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.

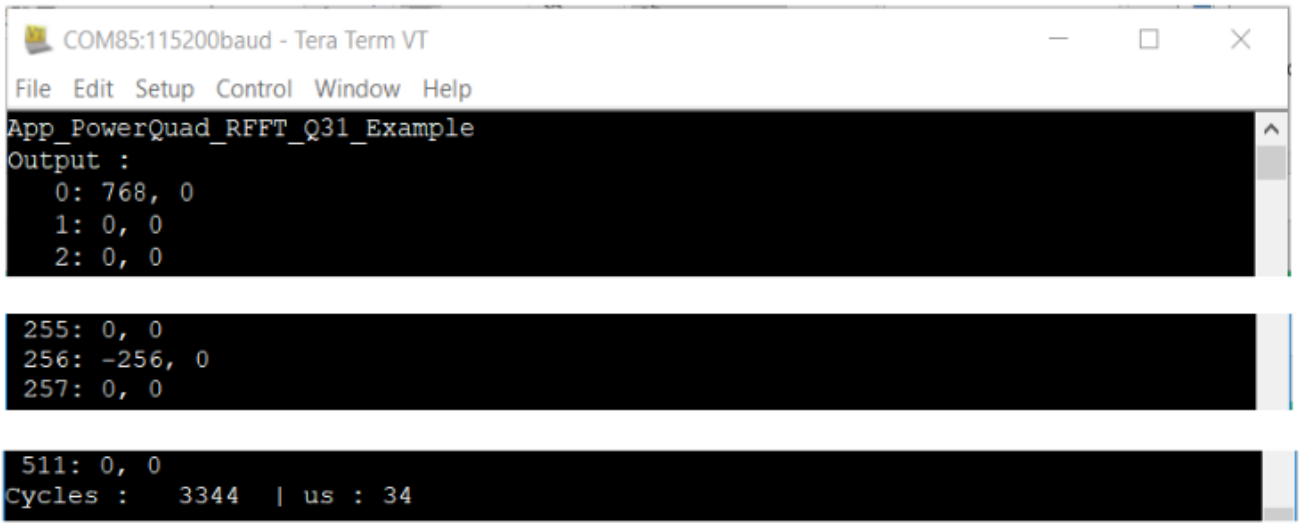

Figure 13. Terminal log for App_PowerQuad_RFFT_Q31_Example

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 PowerQuad hardware.
- It is a little faster than the complex Q31 FFT, caused by the reduced memory operations.
- The length of output numbers does not reduce to half like CMSIS-DSP functions. It would be simpler for user so that no special format is used 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 the same with the real Q31 task while 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.

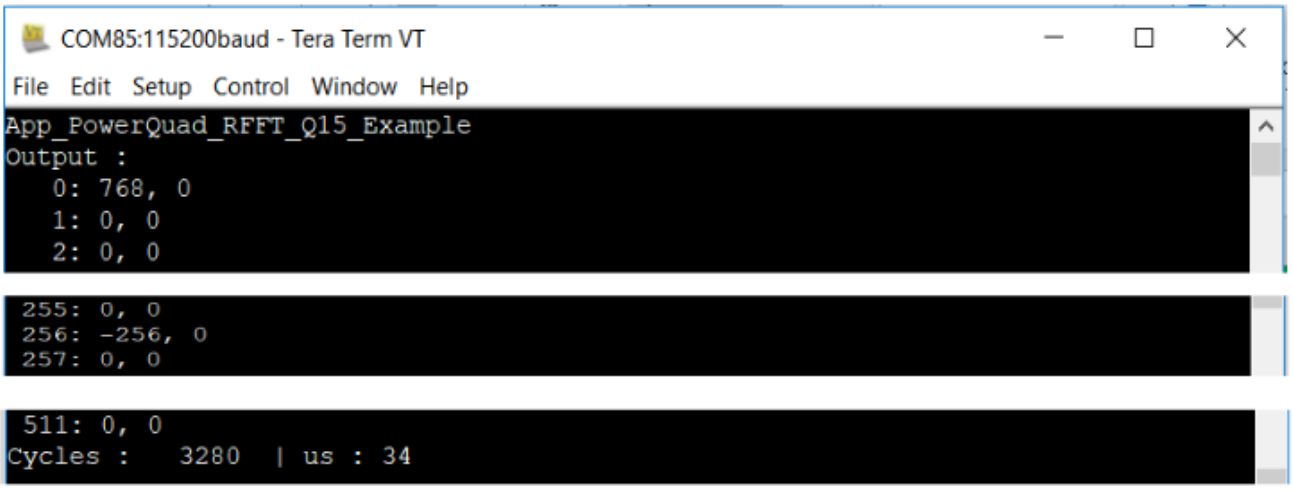

Figure 14. Terminal log for App_PowerQuad_RFFT_Q15_Example

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 PowerQuad hardware.
- It is a little faster than the complex Q31 FFT, caused by the reduced memory operations.
- The length of output numbers does not reduce to half like CMSIS-DSP functions. It is simpler for user so that no special format is used against the complex FFT computing.

6.3 Float-point FFT transform

PowerQuad hardware does not support the floating-point FFT directly. But in some applications, to get the advantage from the powerful acceleration of PowerQuad hardware computing engine but with little code change, users might want to update their project simply by replacing the existing CMSIS-DSP APIs for floating-point FFT with the PowerQuad's implementation. Then a data format conversion between floating-point and fixed-point would be necessary.

Fortunately, the PowerQuad's Matrix Scale function can help to deal with the format conversion by hardware, and it runs faster than the ARM-CMSIS DSP APIs of `arm_float_to_q31()`/`arm_q31_to_float()`. So, just 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, then we can create a floating-point FFT function all based on the PowerQuad hardware.

6.3.1 Format conversion using PowerQuad matrix scale function

In the CMSIS-DSP, there are APIs about converting the floating-point numbers to fixed-point numbers, for example: `arm_float_to_q31()` and `arm_q31_to_float()`. In the PowerQuad module, when setting up the input and output with different value format and executing the Matrix Scale with the scaler is **1.0f**, which means the value would not be changed from input and output, then the conversion can be done automatically during 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.

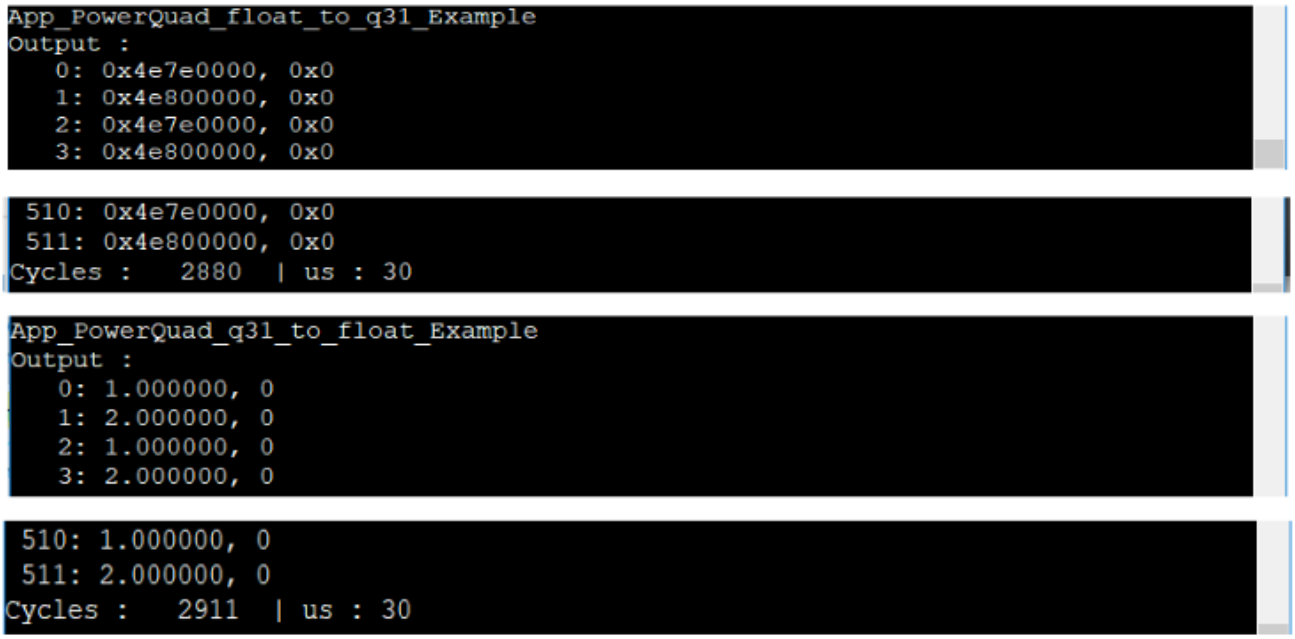

Figure 15. Terminal log for format switch function

Actually, the same test cases were run with ARM-CMSIS DSP APIs as well. Without the compiling optimization, the `arm_float_to_q31()` and `arm_q31_to_float()` are slower than the PowerQuad's conversion functions. However, there are some limitations when using the conversion function:

- For CMSIS-DSP APIs, the fixed-point numbers should not be out of the range (-1, 1) to follow the standard q31 format.
- For PowerQuad APIs, the max length for the array is 256. If longer array needs to be processed, the Matrix Scale function should be called more times.

6.3.2 Computing FFT with complex F32 numbers

In this case, the 512 floating-point input complex numbers (1024 numbers in the array) are converted to fixed-point input numbers by calling the 256-point Matrix Scale function for four time. After running the hardware FFT to get the output fixed-point numbers, another 4-times' 256-point Matrix Scale functions are called to 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.

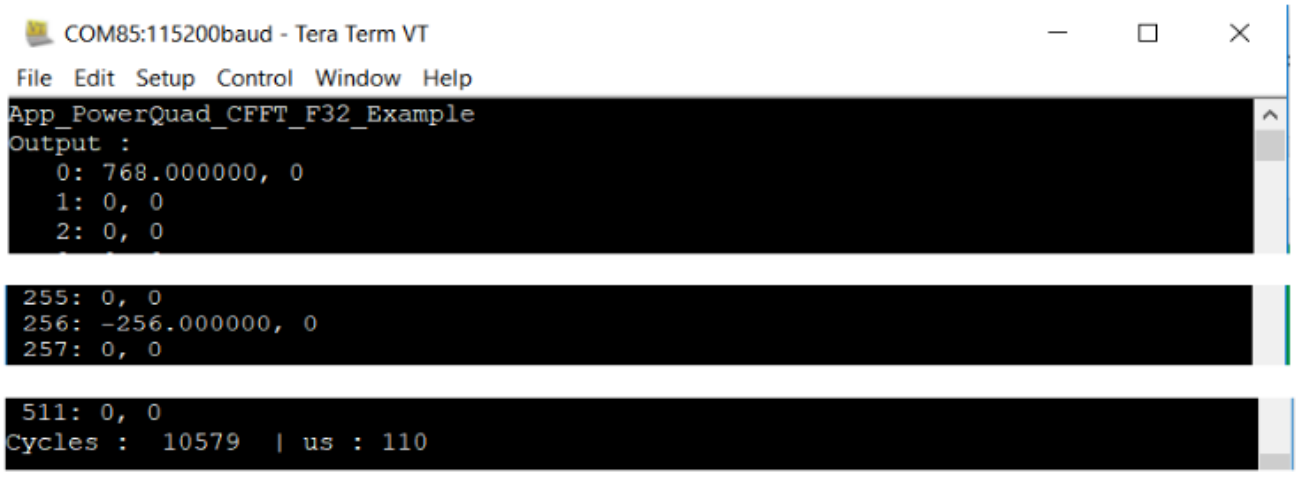

Figure 16. Terminal log for App_PowerQuad_CFFT_F32_Example

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

- The result is correct, the same with CMSIS-DSP's floating complex FFT's.
- The hardware conversion functions are working well.
- The time it runs almost equals to the time for the 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, the input array of packed real floating numbers is converted to the Q31 numbers, then computed by the PowerQuad's FFT engine with `PQ_TransformRFFT()` function to get the output of Q31 numbers, finally converted to the floating-point format with PowerQuad's Matrix Scale function by hardware as well.

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.

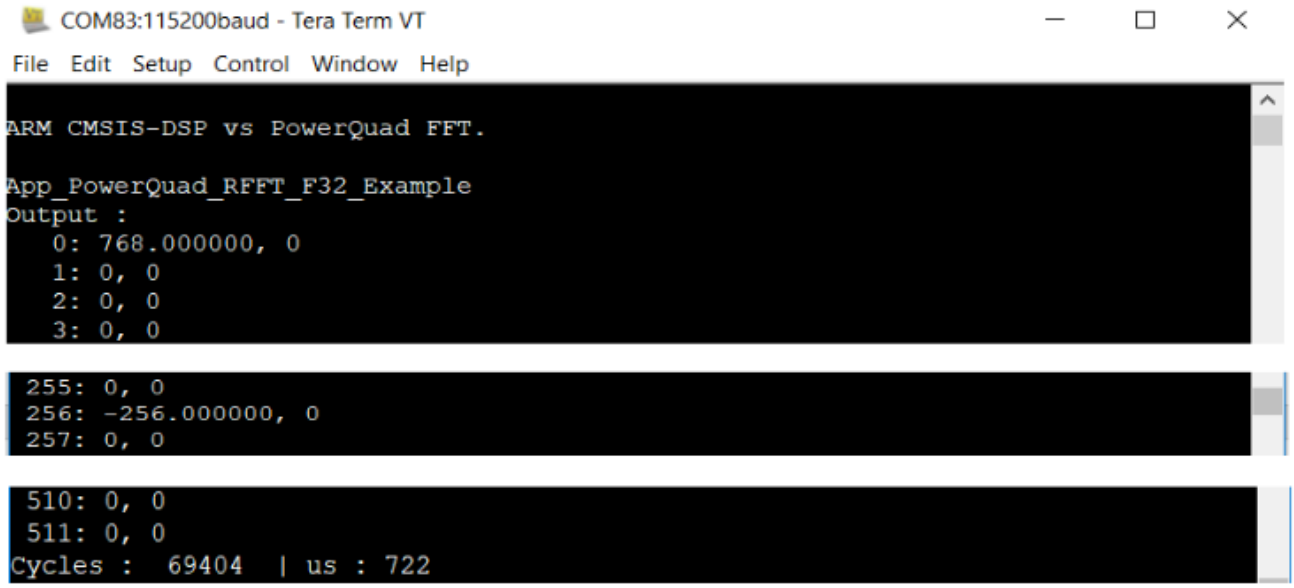

Figure 17. Terminal log for computing FFT with real F32 numbers

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

- Per the usage of Arm CMSIS-DSP's conversion number, the input numbers should be zoomed down into the range (-1, 1). Another point, the output of the conversion number is the strict q31 number, while we actually used the integer-like fixed-point number (with **q0** format). So, an additional zoom down to the input floating numbers were done. 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 is still running faster the pure software's implementation. About comparing the timing, it would be discussed in a later section in this paper.

7 Summary and conclusion

Till now, this paper tells the usage of computing FFT with CMSIS-DSP software and PowerQuad hardware for a same computing case. So, the PowerQuad hardware can be used to replace the CMSIS-DSP software when computing the FFT for the same format of input and output. Never the less, the demo cases showed that the PowerQuad would run much faster than the CMSIS-DSP.

In the last section of this paper, here are some tables about the timing characters for the demo cases would be summarized, to show 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.

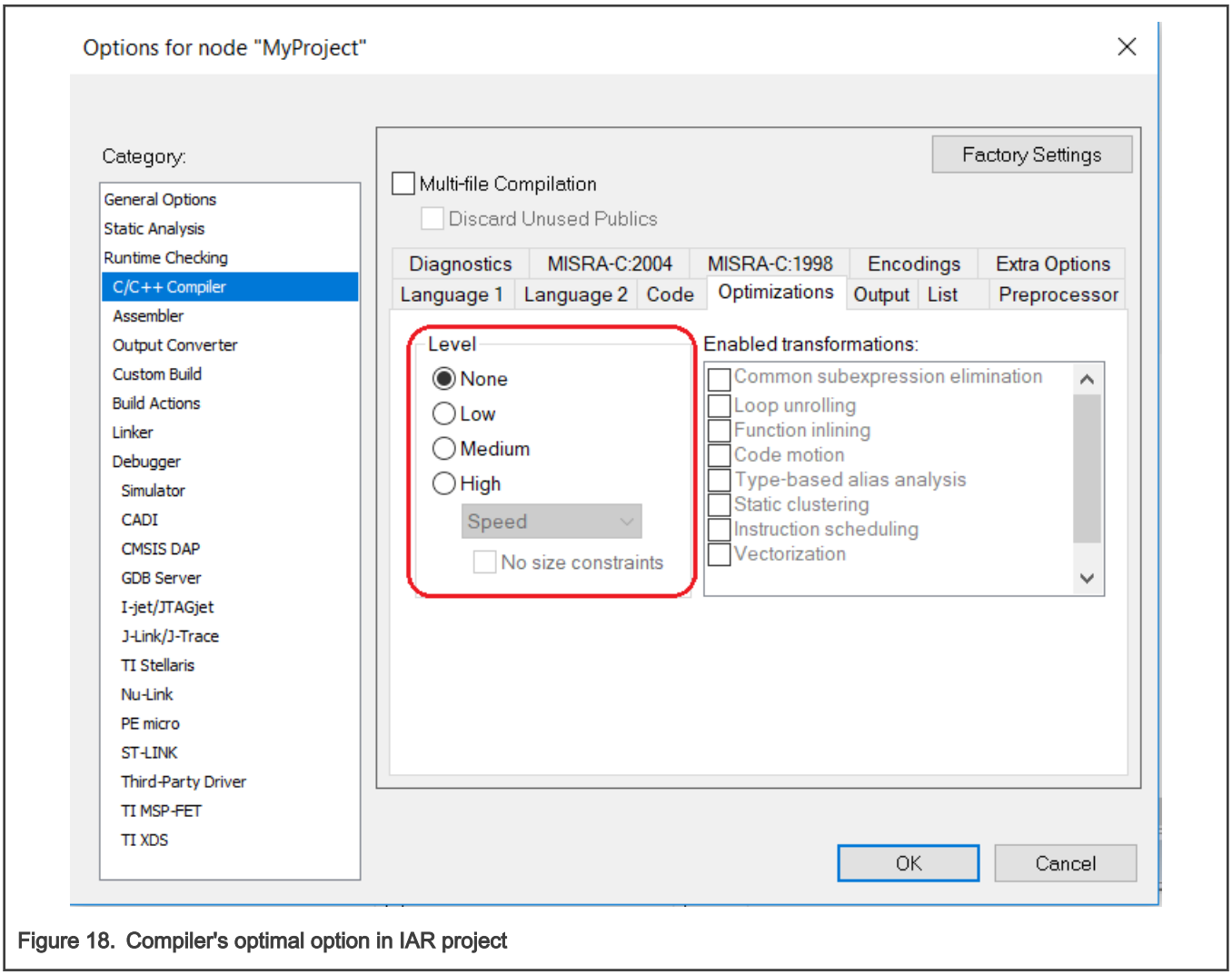

Figure 18. Compiler's optimal option in IAR project

Table 5 summarizes the measuring time.

Table 5. Measuring time on optimal conditions

Demo cases	none		low		mediume		high (speed)		none (FPU disabled)	
	cycles	us	cycles	us	cycles	us	cycles	us	cycles	us
App_CmsisDsp_CFFT_F32_Example	545274	5679	392081	4084	310262	3231	291130	3032	3382749	35236

Table continues on the next page...

Table 5. Measuring time on optimal conditions (continued)

Demo cases	none		low		mediume		high (speed)		none (FPU disabled)	
	cycles	us	cycles	us	cycles	us	cycles	us	cycles	us
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
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:

- PowerQuad's computing are much faster than the CMSIS-DSP functions. About x100 times faster in measuring values.
- PowerQuad's timing performance is stable for the FFT computing with different format numbers, with different compiling optimization conditions. But the CMSIS-DSP software's performance varies a lot depending on the compiling optimization condition. For CMSIS-DSP software's implementation, the higher level's optimization is not always making the code running faster (in the case of App_CmsisDsp_CFFT_Q15_Example, the low level's optimization runs 1876 us, while the medium level's optimization runs 1978 us.
- The fixed-point computing is not always faster than the floating-point's. When the hardware FPU is disabled, the floating-point's computing needs more CPU cycles with the general fixed-point instructions. On this condition, the fixed-point

algorithm would run more smoothly. However, the FPU is enabled for compiler, the floating-point computing instruments can save more time and calculate the floating-point number directly in one instrument, while the fixed-point one needs to more instruments to convert the big number's calculation into several steps and cost more time. So, that is reason why App_CmsisDsp_CFFT_F32_Example demo case runs faster than App_CmsisDsp_CFFT_Q31_Example when the FPU is enabled for compiler, but much slower when the FPU is disabled.

- The format conversion between floating-point numbers and fixed-point numbers costs a lot of time, almost the same level, both for the CMSIS-DSP software and the 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. However, the complex floating-point FFT is more recommended since it run much faster but with a little additional memory. Or modify the original data format to fixed-point number in application, then it can achieve the best performance.

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		mediume		high (speed)		none (FPU disabled)	
	cycles	us	cycles	us	cycles	us	cycles	us	cycles	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

Table continues on the next page...

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

Demo cases	none		low		mediume		high (speed)		none (FPU disabled)	
	cycles	us	cycles	us	cycles	us	cycles	us	cycles	us
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

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