AN12383

Computing FFT with PowerQuad and CMSIS-DSP on LPC5500

Rev. 1 — 7 September 2023 Application note

Document Information

Information	Content
Keywords	LPC5500, PowerQuad, FFT, DSP
Abstract	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.

1 Introduction

Fast Fourier 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 signals is transformed from timing field to frequency field, some useful and interesting attributes appear. They 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 applications 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 the software, even if it is well optimized. It means that the computing time depends on the optimization conditions of the compiler and the performance of the CPU. Also, the computing time of the complex process, like FFT purely by the software, is usually not short, which 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 executed by the specific hardware other than the Arm core, it runs faster 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. Therefore, the development based on it is friendly.

About the usage of the fixed-point FFT and the floating-point FFT, they have their specific implementations and applications in different fields. The fixed-point FFT is used to process the audio, video, and other data captured from hardware sensor modules like ADC, while the original direct sample value for these conditions is a fixed point. For the floating-point FFT, it is commonly used to process the longitude and latitude with high accuracy and high resolution in a 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 Transform (DFT) and Discrete Cosine Transform (DCT). It is implemented with a Radix-8 butterfly structure FFT engine using the fixed-point arithmetic at a resolution of 24 bits.

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

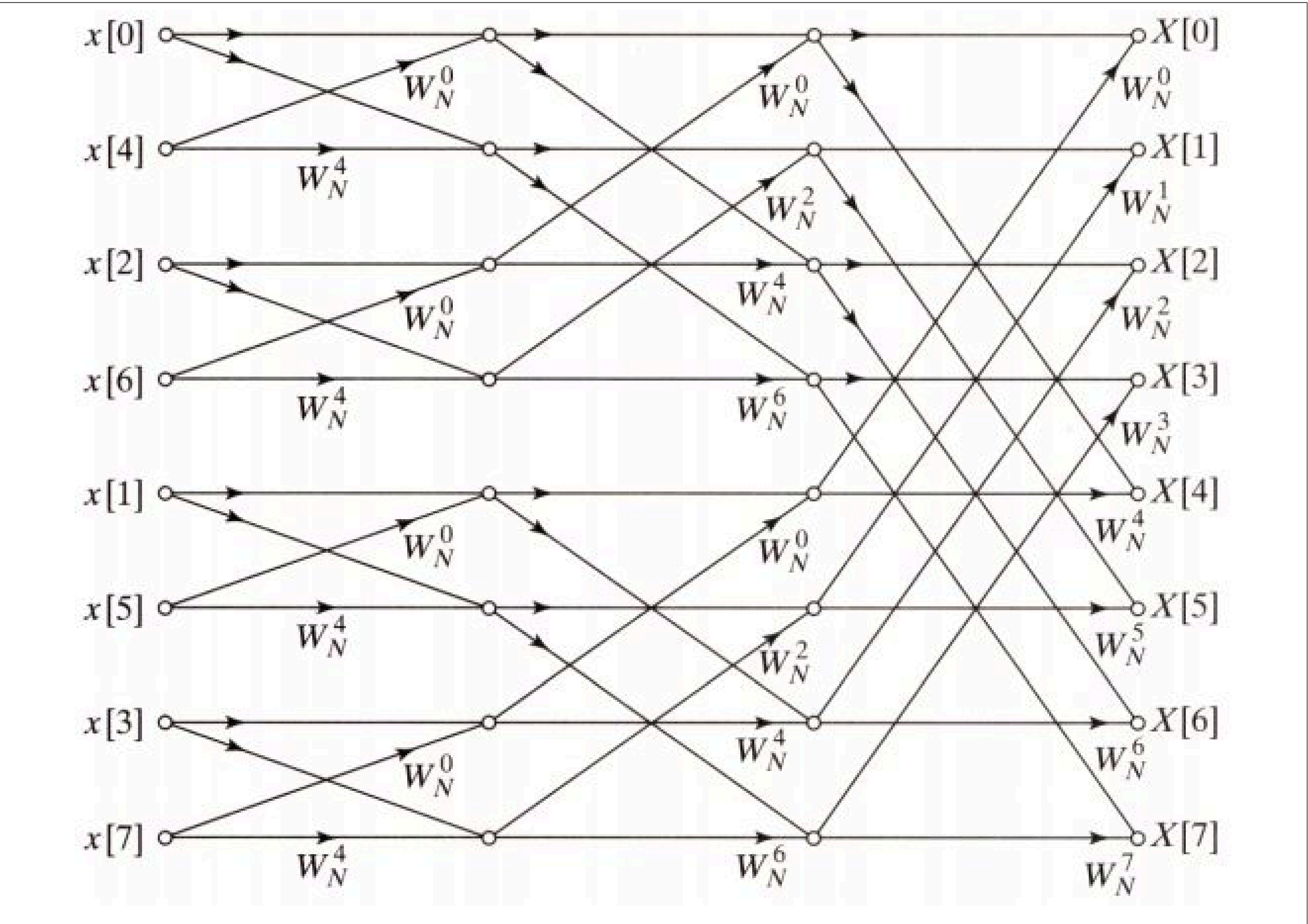


Figure 1. Radix-8 butterfly structure of PowerQuad FFT engine

2.1 Computing equations

DFT transforms a sequence of N complex numbers:

$x_0, x_1, x_2, ..., x_{N-1}$

Into another sequence of N complex numbers:

$X_0, X_1, X_2, ..., X_{N-1}$

which is defined by:

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

The inverse transform is given by:

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

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:

$$X_{N-k} = X_{-k} = X_k^* \qquad (3)$$

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 the FFT computing engine of the PowerQuad can support the DCT by hardware as well, it is not so popular as the FFT. It can be computed by the matrix way in a simpler way, which is also supported by the PowerQuad matrix computing engine. Using the matrix computing engine to compute the DCT is easier and more flexible than FFT computing engine. So, this application note does not describe the DCT in details, 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

The PowerQuad FFT engine can only use fixed-point numbers 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 prescaling must not exceed this to avoid saturation.*

If the FFT of the floating-point numbers is required in the application, the 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 matrix engine of the PowerQuad provides a function of matrix scale, which would accelerate the conversion by mixed format computing.

2.2.2 Input and output sequences 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}), ... (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}, ..., 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, ... x_{N-1}}

The output sequence is always stored in memory organized as an array of complex numbers where the imaginary parts are 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 are not overflowed 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 set up the hardware prescaler for the INPUT A region.

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

When the application is willing to replace the FFT API of CMSIS-DSP used in the existing project, to keep the input and output data to be aligned, manually add the prescaler. However, if the application is newly designed, this step can be omitted, 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 the memory exclusively without any arbitration delay, so that to accelerate the whole process of computing as fast as possible. As the PowerQuad accesses the four banks of memory with 32-bit bus simultaneously in an interleave way, it can achieve an equivalent 128-bit bus band wide. Using private RAM is encouraged. It means that PowerQuad can access the data quicker and improve the performance by accessing one operand from RAM and one from the system at the same time.

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 the format of a memory, the handler 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 is at general memory space. Private memory is used as temporary storage for the 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 reads the input and writes the output with 4 words (128-bit) once a time, the allocated memory address for the PowerQuad memory handler is 4-word (or 16 bytes) 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 mean 512 complex pairs, and it must align 1024 words.

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

3 Measuring time in a demo project

Considering the functions are usually running fast, the interrupt-based timing method is not suitable in the demo case. However, a tip here is that in some test projects special 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, the SysTick timer is chosen as the hardware timer, so that the code here could be 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 clock source of the SysTick timer, 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 conditions. The measuring time is summarized to show the computing performance.

4 Computing cases in a 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 1.0 f or 2.0 f.
- For the complex fixed-point numbers, they are the complex numbers (1, 0) or (2, 0).
- For the complex floating-point numbers, they are the complex number (1.0 f, 0.0 f) or (2.0 f, 0.0 f). All in all, the values of inputs are the same for different computing cases.

4.2 [OUTPUT]

The output array of values is all zero except for:

- The 0th number is 765.
- The 256th number is -256.

This output makes sense. As shown 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 1/2, and the phase would be negative. No other frequency factors.

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

However, using the general mathematic calculator (like Matlab) simplifies the step of 1/N when outputting the result. That means, the direct output multiples **N** from the final result. In the case for this paper, the actual result is: result [0] = 768, result [256] = -256.

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

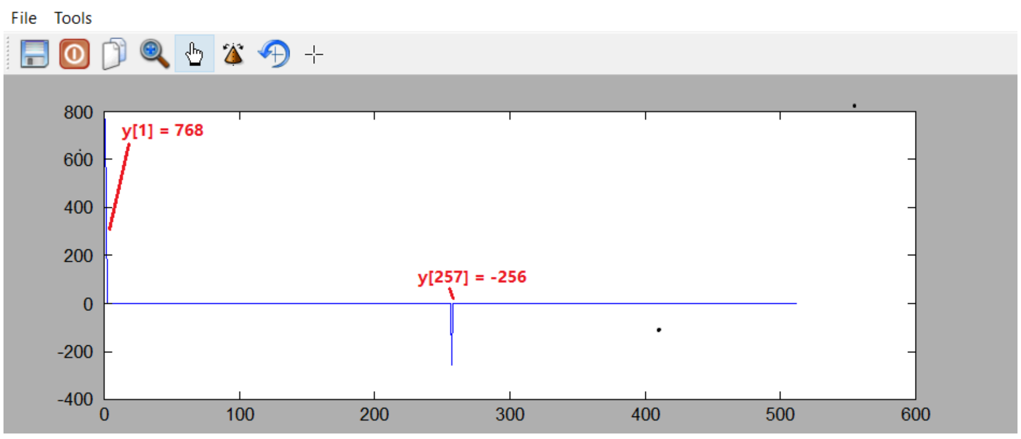


Figure 2. FFT calculation by FreeMat

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 FFT is an efficient algorithm for computing the 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 is also 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 results are contained in the same array and the frequency domain values 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, 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 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.

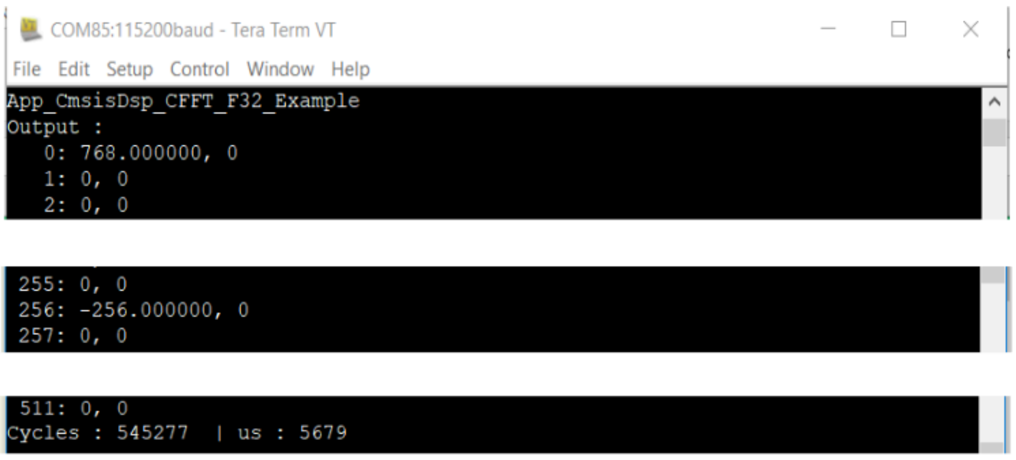
Pre-initialized data structures containing twiddle factors and bit reversal tables are provided and defined in the source file, *arm_const_structs.h*. Include this header in your function and then pass one of the constant structures as an argument to `arm_cfft_f32`. For example:

```
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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_CmsisDsp_CFFT_F32_Example
Output :
   0: 768.000000, 0
   1: 0, 0
   2: 0, 0

 255: 0, 0
 256: -256.000000, 0
 257: 0, 0

 511: 0, 0
Cycles : 545277   | us : 5679
```

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 computing function modifies the memory of `inputF32[]` and the output numbers cover the input numbers. The output number is using two items as the real part and the 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 cases use the result without 1/fftLen scale as the common target.
- The running time goes with no compiling optimization. Table 5 summarizes all the computing time in different optimal condition.

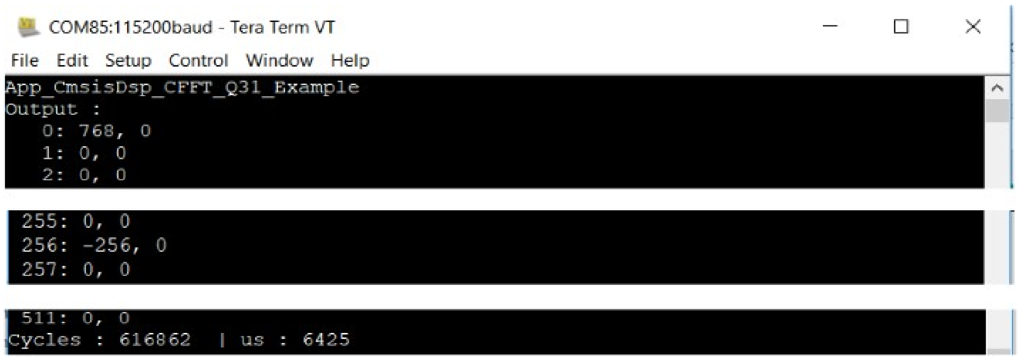
5.1.2 Computing FFT with complex Q31 numbers

The version FFT of Q31 is implemented differently from the floating-point one. Also, the range of fixed-point number is confusing, because the Q31 number is in the range of (-1, 1). 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, since the output of FFT would be mostly used as normal values to feed the following procedure, unless the whole application is 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_CmsisDsp_CFFT_Q31_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0
 255: 0, 0
 256: -256, 0
 257: 0, 0
 511: 0, 0
Cycles : 616862  | us : 6425
```

Figure 4. Terminal log for `App_CmsisDsp_CFFT_Q31_Example`

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

- The FFT of the fixed-point version 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 must 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 the CFFT/CIFFT process. Therefore, 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

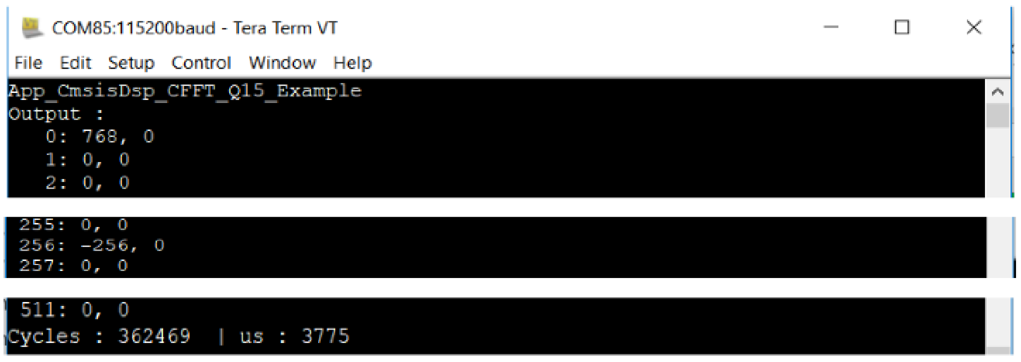
5.1.3 Computing FFT with complex Q15 numbers

The FFT of Q15 version in CMSIS-DSP is expected to cost less memory and time, but with less significant figures. It is also suitable to process the data, whose 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 the case of the Q31 version 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_CmsisDsp_CFFT_Q15_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0

 255: 0, 0
 256: -256, 0
 257: 0, 0

 511: 0, 0
Cycles : 362469   | us : 3775
```

Figure 5. Terminal log for `App_CmsisDsp_CFFT_Q15_Example`

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

- The FFT of the Q15 version does the scale of 1/fftLen inside the function like the Q31 version. 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

CFFT size	Input format	Output format	Number of bits to upscale
16	1.15	5.11	0

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

CFFT size	Input format	Output format	Number of bits to upscale
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], img[2], ... real[(N/2) - 1], imag[(N/2) - 1}

It happens that the first complex number (`real[0], imag[0]`) is pure real, while the `real[0]` represents the DC offset and `imag[0]` are 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 that 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.

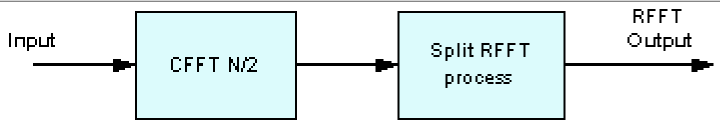


Figure 6. Real Fast Fourier Transform

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

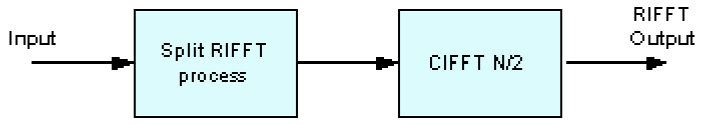


Figure 7. Real inverse Fast Fourier Transform

As a summary for using the N point real FFT:

- The length of the input array is N, with N real numbers.

- The length of the 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 packed with the two real numbers, `real[0]` and `real[N/2]`.

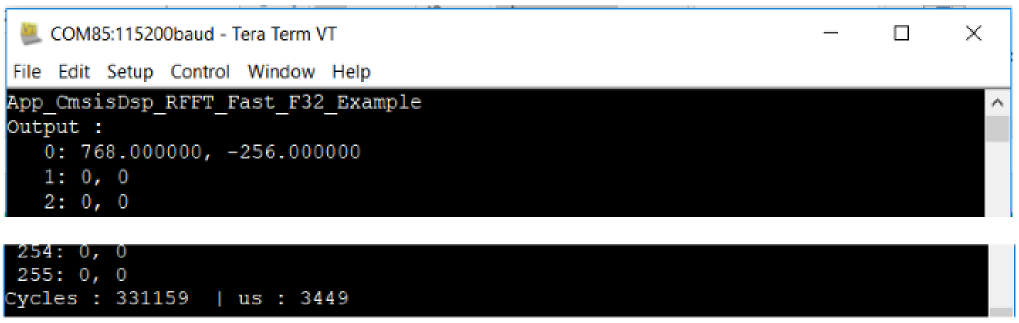
5.2.1 Computing FFT with real F32 numbers

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 8 shows the result.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_CmsisDsp_RFFT_Fast_F32_Example
Output :
   0: 768.000000, -256.000000
   1: 0, 0
   2: 0, 0

 254: 0, 0
 255: 0, 0
Cycles : 331159  | us : 3449
```

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 the 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 the 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]` is 0. (`real[1]`, `imag[1]`) is the fundamental frequency, (`real[2]`, `imag[2]`) is the first harmonic, and so on.

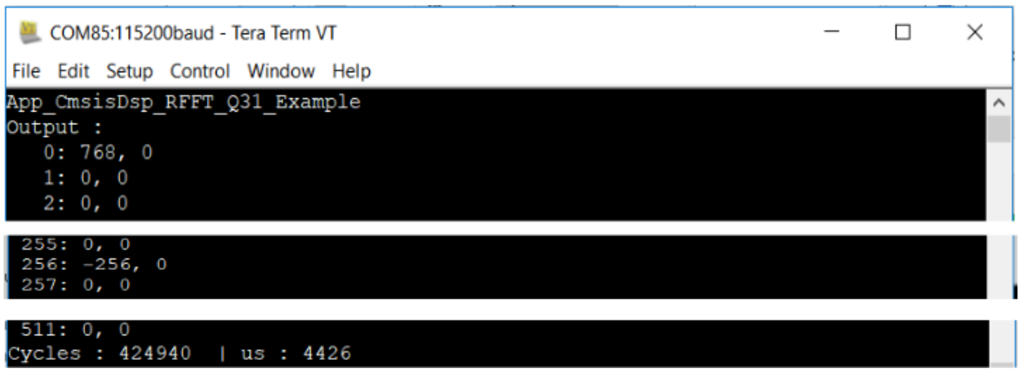
5.2.2 Computing FFT with real Q31 numbers

The real FFT of Q31 is different from the floating-point version using a fast way. It uses the old format like in the 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 the 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_CmsisDsp_RFFT_Q31_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0
 255: 0, 0
 256: -256, 0
 257: 0, 0
 511: 0, 0
Cycles : 424940   | us : 4426
```

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 the available input array is 512 for the 512 real numbers. The length of the 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 did.

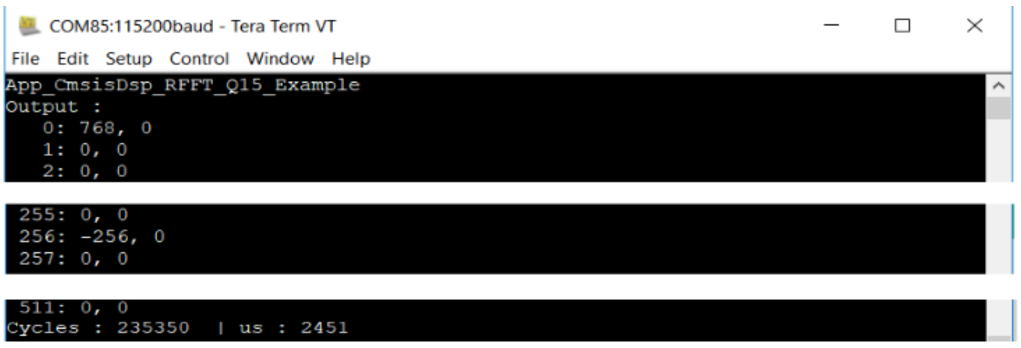
5.2.3 Computing FFT with real Q15 numbers

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

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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_CmsisDsp_RFFT_Q15_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0

 255: 0, 0
 256: -256, 0
 257: 0, 0

 511: 0, 0
Cycles : 235350   | us : 2451
```

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 architecture of the Arm core (the narrow memory bus) and the performance of the compiler (the optimizing condition of different level). But on the other side, the hardware implements and optimizes the computing engines (including the FFT engine) of PowerQuad. 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 coprocessor, the PowerQuad can also run with the Arm core parallel if necessary, to meet the requirements in the real-time system.

The MCUXpresso SDK software library of NXP already supports 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 set the format of various fixed points.

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 discusses the usage of APIs.

6.1 Fixed-point complex FFT transforms

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

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. Both input number and output number have their owner hardware prescaler setting. In this case, for the 512-point FFT, the prescaler number is 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 handlers, in `pq_cfg.inputAFormat`, `pq_cfg.tmpFormat`, and `pq_cfg.outputFormat`, are for the fixed-point, such as, `kPQ_32Bit` or `kPQ_16Bit`. In this case, they are `kPQ_32Bit`. The setting for the output memory handler is ignored for the FFT engine. Also, the input and output array must be the 32-bit words.

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

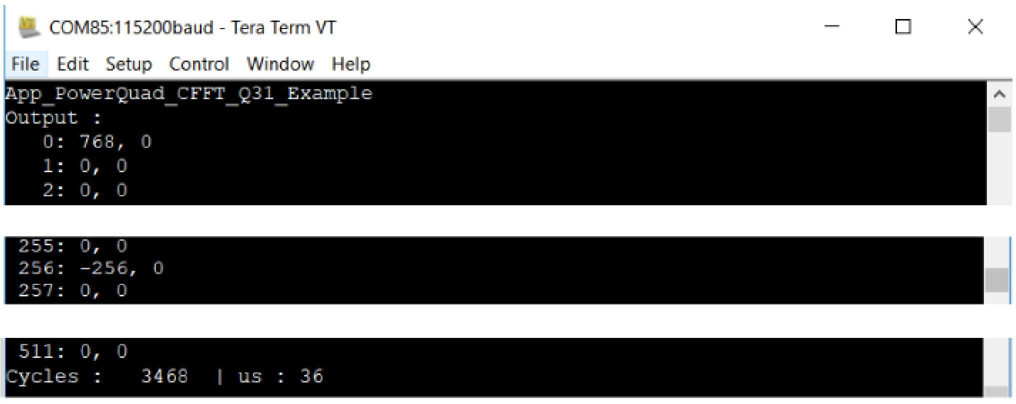
To keep the intermediate data during computing, the Temp memory handler uses the private RAM starting from `0xE000_0000`. For the 512-point FFT, to keep the 512 complex numbers with 1 K 32-bit word, reserve 4 kB memory in the private RAM.

The critical function in this case is the `PQ_TransformRFFT()` 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_PowerQuad_CFFT_Q31_Example
Output :
    0: 768, 0
    1: 0, 0
    2: 0, 0

  255: 0, 0
  256: -256, 0
  257: 0, 0

  511: 0, 0
Cycles :    3468   | us : 36
```

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 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 the output of CMSIS-DSP fixed-point FFT.

6.1.2 Computing FFT with complex Q15 numbers

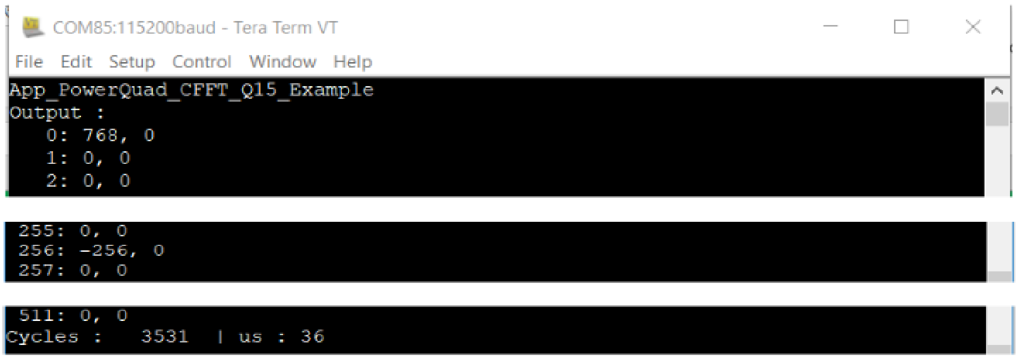
With the PowerQuad FFT engine, the complex Q15 task is almost the same with the complex Q31 task while the difference is:

- 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_PowerQuad_CFFT_Q15_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0

 255: 0, 0
 256: -256, 0
 257: 0, 0

 511: 0, 0
Cycles :   3531  | us : 36
```

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 do not reduce the workload of PowerQuad hardware.

6.2 Fixed-point real FFT transforms

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

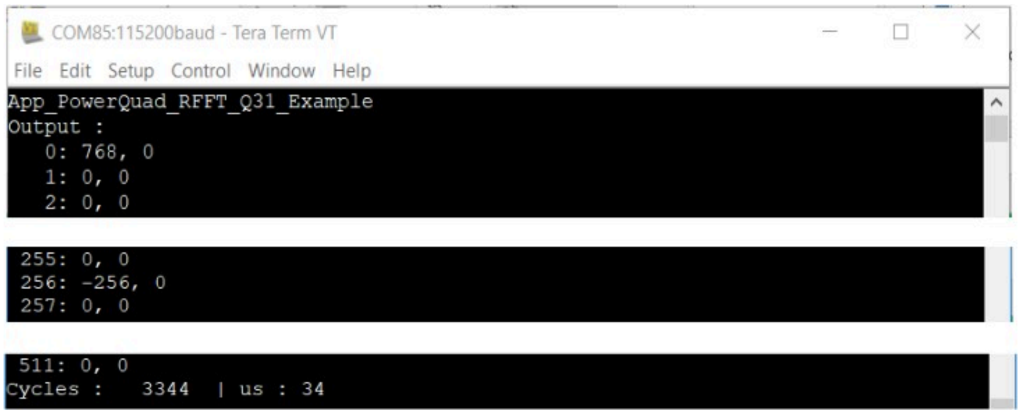
6.2.1 Computing FFT with real Q31 numbers

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 ] = APP_FFT_LEN_512 * (1 + i%2); /* 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_PowerQuad_RFFT_Q31_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0

 255: 0, 0
 256: -256, 0
 257: 0, 0

 511: 0, 0
Cycles :   3344  | us : 34
```

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 is simpler for users so that no special format is used against the complex FFT computing.

6.2.2 Computing FFT with real Q15 numbers

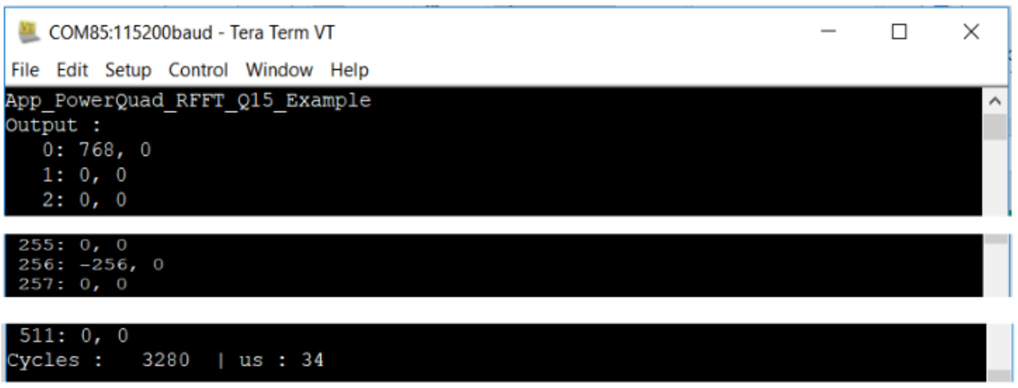
With the PowerQuad FFT engine, the read Q15 task is almost the same with the real Q31 task while the difference is:

- 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_PowerQuad_RFFT_Q15_Example
Output :
   0: 768, 0
   1: 0, 0
   2: 0, 0
```

```
 255: 0, 0
 256: -256, 0
 257: 0, 0
```

```
 511: 0, 0
Cycles :   3280  | us : 34
```

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 users 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 matrix scale function of the PowerQuad can help to deal with the format conversion by hardware. 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.

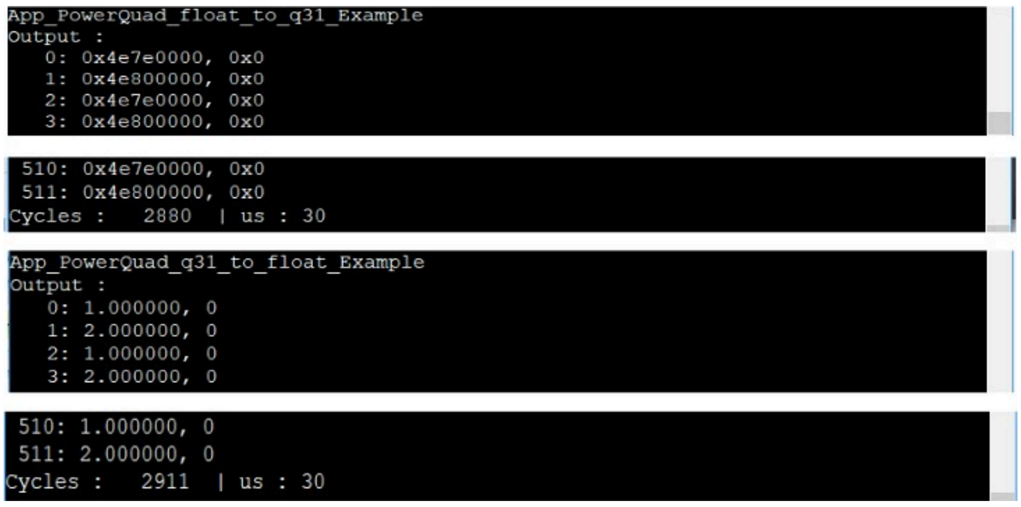
6.3.1 Format conversion using PowerQuad matrix scale function

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 1.0 f, which means the value is not 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.



```
App_PowerQuad_float_to_q31_Example
Output :
   0: 0x4e7e0000, 0x0
   1: 0x4e800000, 0x0
   2: 0x4e7e0000, 0x0
   3: 0x4e800000, 0x0

 510: 0x4e7e0000, 0x0
 511: 0x4e800000, 0x0
Cycles :   2880  | us : 30

App_PowerQuad_q31_to_float_Example
Output :
   0: 1.000000, 0
   1: 2.000000, 0
   2: 1.000000, 0
   3: 2.000000, 0

 510: 1.000000, 0
 511: 2.000000, 0
Cycles :   2911  | us : 30
```

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 conversion functions of PowerQuad. However, there are some limitations when using the conversion function:

- For CMSIS-DSP APIs, the fixed-point numbers cannot 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 a longer array must be processed, the matrix scale function is 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 times. After running the hardware FFT to get the output fixed-point numbers, the 256-point matrix scale functions of another four times are called to get the floating-point output number.

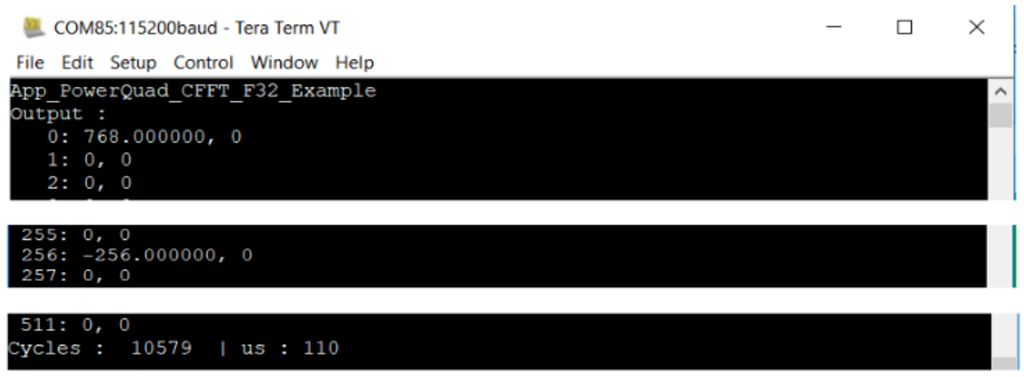
The code for the task is:

```
/* 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.


COM85:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
App_PowerQuad_CFFT_F32_Example
Output :
   0: 768.000000, 0
   1: 0, 0
   2: 0, 0

 255: 0, 0
 256: -256.000000, 0
 257: 0, 0

 511: 0, 0
Cycles :  10579  | us : 110
```

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 the floating complex FFT result of CMSIS-DSP.
- 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

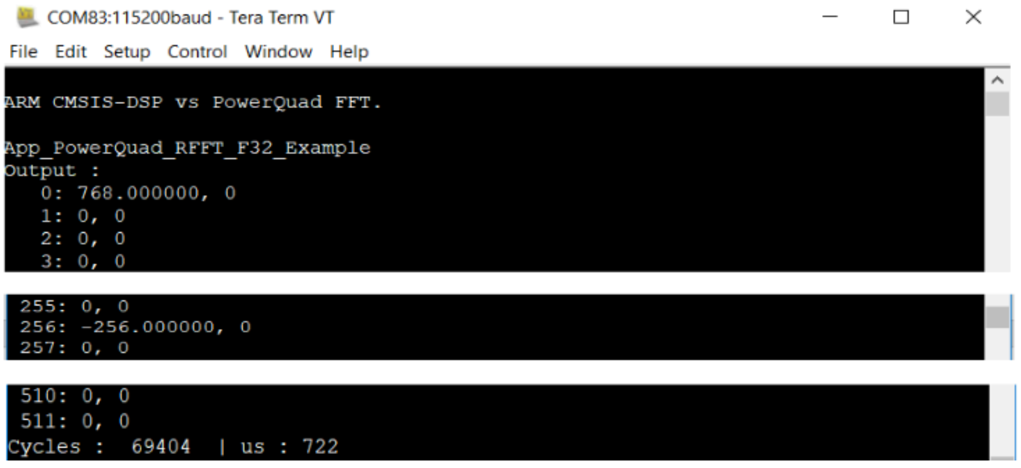
In this case, the input array of packed real floating numbers is converted to the Q31 numbers, then computed by the FFT engine of PowerQuad with the `PQ_TransformRFFT()` function to get the output of Q31 numbers, finally converted to the floating-point format with the matrix scale function of PowerQuad by the 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.


COM83:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
ARM CMSIS-DSP vs PowerQuad FFT.

App_PowerQuad_RFFT_F32_Example
Output :
   0: 768.000000, 0
   1: 0, 0
   2: 0, 0
   3: 0, 0

 255: 0, 0
 256: -256.000000, 0
 257: 0, 0

 510: 0, 0
 511: 0, 0
Cycles :  69404  | us : 722
```

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

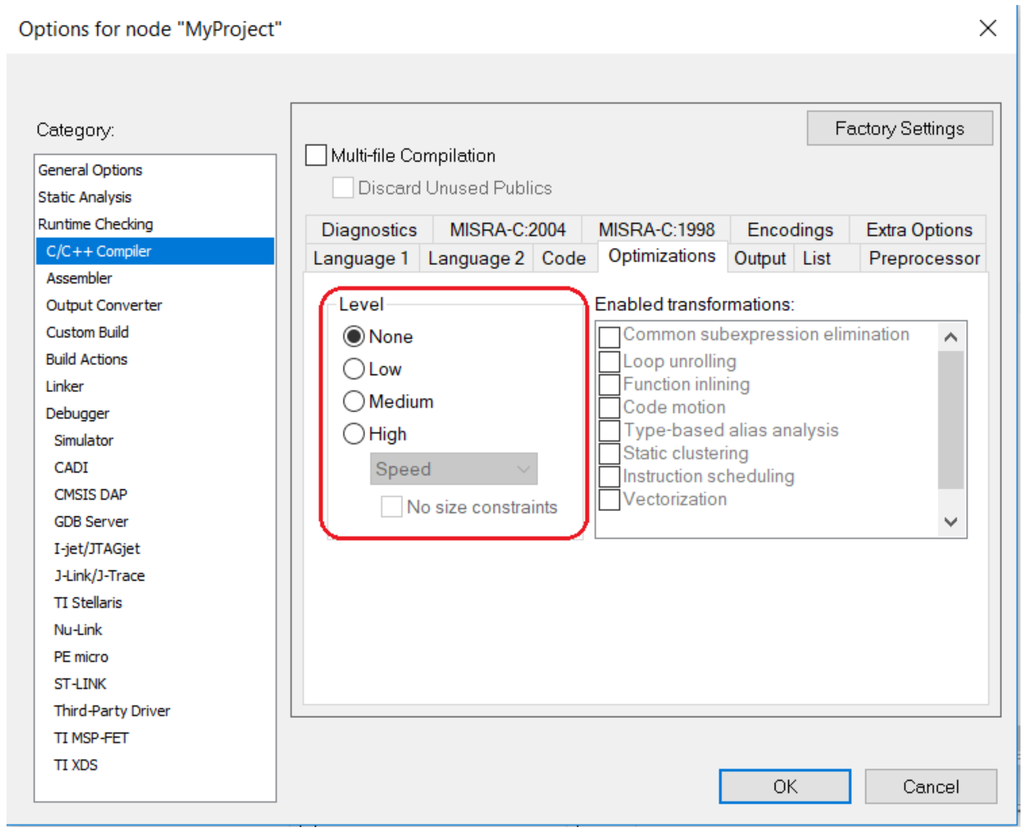
Per the code and terminal log shown for this case, we can see:

- Per the conversion number usage of Arm CMSIS-DSP, zoom down the input numbers to 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 though, it still runs faster than the implementation of the pure software. The time comparison is discussed in Section 7.

7 Summary and conclusion

Until 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. Nevertheless, the demo cases showed that the PowerQuad runs faster than the CMSIS-DSP.

Table 5 summarizes the timing characters for the demo cases, to show the accumulation capability of PowerQuad. The different compiling optimization conditions are set in the **Project Option** dialog box in the IAR IDE, as shown in Figure 18.


Options for node "MyProject"

Category: C/C++ Compiler

Optimizations

[ ] Multi-file Compilation

[ ] Discard Unused Publics

Level: [✓ None] Low Medium High

Speed

[ ] No size constraints

Enabled transformations:

[ ] Common subexpression elimination
[ ] Loop unrolling
[ ] Function inlining
[ ] Code motion
[ ] Type-based alias analysis
[ ] Static clustering
[ ] Instruction scheduling
[ ] Vectorization

Figure 18. Optimal option of the compiler in IAR project

Table 5 summarizes the measuring time.

Table 5. Measuring time in 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_Q31_ 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

All information provided in this document is subject to legal disclaimers.

© 2023 NXP B.V. All rights reserved.

Table 5. Measuring time in 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

As shown in Table 5:

- The PowerQuad computes faster than the CMSIS-DSP. About x100 times faster in measuring values.
- The timing performance of PowerQuad is stable for FFT computing with different format numbers, with different compiling optimization conditions. But the performance of CMSIS-DSP software varies a lot depending on the compiling optimization condition. For the implementation of the CMSIS-DSP software, the higher-level optimization is not always making the code run faster (in the case of `App_CmsisDsp_CFFT_Q15_Example`. The low-level optimization runs 1876 µs, while the medium level runs 1978 µs.
- The fixed point does not always compute faster than the floating-point. When the hardware FPU is disabled, the computing of the floating point needs more CPU cycles with the general fixed-point instructions. On this condition, the fixed-point algorithm would run more smoothly. However, if the FPU is enabled for the 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 need more instruments to convert the calculation of big numbers into several steps and cost more time. This is the reason that the `App_CmsisDsp_CFFT_F32_Example` demo case runs faster than `App_CmsisDsp_CFFT_Q31_Example` when the FPU is enabled for the compiler, but 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, for both the CMSIS-DSP software and the PowerQuad hardware.
- For the `App_PowerQuad_RFFT_F32_Example` demo case, even with the software workaround about the format conversion issue, and replaced with part of the 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, because it runs faster but with more memory. Or, modify the original data format to a fixed-point number in the application, and then it can achieve the best performance.

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

Table 6. Measuring time in various conditions with a 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_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
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

8 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2023 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

9 Revision history

Table 7 summarizes the revisions to this document.

Table 7. Revision history

Revision number	Release date	Description
1	07 September 2023	Updated
0	November 2019	Initial public release

10 Legal information

10.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

10.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

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.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Suitability for use in non-automotive qualified products — Unless this data sheet expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP B.V. - NXP B.V. is not an operating company and it does not distribute or sell products.

10.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μVision, Versatile — are trademarks and/or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

MATLAB — is a registered trademark of The MathWorks, Inc.

Contents

Please be aware that important notices concerning this document and the product(s) described herein, have been included in section 'Legal information'.

© 2023 NXP B.V. **All rights reserved.**

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

Date of release: 7 September 2023
Document identifier: AN12383