CMSIS DSP Library and Example Code

Application Note for NuMicro® Cortex®-M4 Family

Document Information

Abstract	This document introduces the basic operation instructions of NuMicro Cortex-M4 DSP, including the usage of the DSP library and its common acceleration performance that helps to program more higher-performance projects with the NuMicro Cortex-M4 family.
Apply to	NuMicro Cortex-M4 Family

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

Table of Contents

List of Figures

List of Tables

1 Overview

There are two types of DSP. The first type is digital signal processing, a technology that expresses and processes digital signals through numbers. It is not a special processor, but a method of measuring or filtering continuous analog signals in the real world. The other type is the digital signal processor, which is a microprocessor used specifically for digital signal processing. It has special signal processing hardware and is responsible for fast signal processing and conversion.

The DSP of Arm Cortex belongs to a kind of digital signal processor. It combines several advanced multiplex operation instructions to combine certain digital signal processing functions to speed up operations.

The main multiplex operation instructions of the DSP extension of Cortex-M4 are Multiply-and-Accumulate (MAC) and Single Instruction Multiple Data (SIMD). Digital signal processing uses many value operations and improves the MCU's ability to process and analyze digital signals through multiplex operations. At the same time, Arm provides the CMSIS (Cortex Microcontroller Software Interface Standard) library from DSP. It contains optimized, powerful DSP algorithms and includes many common mathematical functions sets for users to use directly without having to combine multiplexing instructions themselves.

1.1 Computing Advantages of Floating-Point and DSP by Cortex-M4

Table 1-1 compares the Cortex-M series microcontrollers, and shows that the Cortex-M4 has obvious advantages over the Cortex-M3 due to its extensive instruction sets: Hardware Multiplier, Hardware Divider, Digital Signal Processing, and Floating-Point Operation.

Arm Cortex-M	Hardware Multiplier	Hardware Divider	DSP Processor	Operation of Floating-Point
Cortex-M0+	Single Clock Rate	Option	-	-
Cortex-M3	Single Clock Rate	√	-	-
Cortex-M4	Single Clock Rate	√	√	Option

Table 1-1 Comparison of Cortex-M Series Microcontroller

The DSP processors can be further divided into two categories:

- Digital Signal Processor
- Digital Signal Controller

The Digital Signal Processor is a normal digital signal processor whose architecture is fully optimized for digital signal processing and is actually capable of performing multiplication and accumulation operations with a single clock. The operands required for multiplication and accumulation are loaded and executed in the next clock cycle without having to wait for the time to load the operands from memory, and the ultimate computing power can be used. However, the design complexity is relatively high, and in general, the power savings are not as good as a microcontroller.

Most DSP processors only support fixed-point operations. Because of the same area requirements and power consumption, fixed-point operations can provide higher computational performance, and signal processing applications do not require such a large dynamic range of floating-point numbers. But floating-point DSP still has its advantages, especially in some applications where development time must be reduced. The floating-point capability makes it unnecessary for programmers to spend too much effort and time on fixed-point operations.

The second category is the integration of microcontrollers with digital signal processing functions. It mainly consists of microcontrollers and is characterized by the addition of the multiplication and accumulation function, which is often used by DSP, and the single-instruction multiple data stream function, which also has a wealth of peripheral support and I/O pins. The Arm Cortex-M4 falls into this category, and since the Arm architecture is widely supported, the Arm Cortex-M4 has a wealth of tools.

1.2 DSP Functions

1.2.1 Multiply-and-Accumulate

Multiply-and-Accumulate (MAC) is a common operation used in signal processing, such as matrix multiplication (1.1), convolution, finite impulse response (FIR) filter (1.2), and fast Fourier transform (FFT, especially in Butterfly diagram (1.3) and (1.4)), which are all dependent on MAC.

$$\begin{bmatrix} a11 & a12 & a13 \\ a21 & a22 & a23 \\ a31 & a32 & a33 \end{bmatrix} \times \begin{bmatrix} b11 & b12 & b13 \\ b21 & b22 & b23 \\ b31 & b32 & b33 \end{bmatrix} = \begin{bmatrix} a11 \times b11 + a12 \times b21 + a13 \times b31 & \dots & \dots \\ a21 \times b11 + a22 \times b21 + a23 \times b31 & \dots & \dots \\ a31 \times b11 + a32 \times b21 + a33 \times b31 & \dots & \dots \end{bmatrix} \quad (1.1)$$

$$\mathrm{y[n]} = \sum_{k=0}^{N-1} h[k]x[n-k] \quad (1.2)$$

$$\mathrm{Y}[k_1] = \mathrm{X}[k_1] + \mathrm{X}[k_2] \quad (1.3)$$

$$\mathrm{Y}[k_2] = (\mathrm{X}[k_1] - \mathrm{X}[k_2])\mathrm{e}^{-\mathrm{j}\omega} \quad (1.4)$$

The Cortex-M4 is equipped with a 32-bit MAC that can perform the most difficult operation of multiplying two 32-bit and adding 64-bit operations or twice multiplying two 16-bit in one cycle.

The instruction sets for multiply or multiply-accumulate in the Arm Cortex-M4 are listed in Table 1-2. The following instructions are all fixed-point operations and can all be executed in one cycle.

Operation	Instruction	Command List
16×16=32	Signed Multiply (SMUL)	SMULBB, SMULBT, SMULTB, SMULTT
16×16+32=32	Signed Multiply and Accumulate (SMLA)	SMLABB, SMLABT, SMLATB, SMLATT
16×16+64=64	16-bit Signed Multiply with 64-bit Accumulate (SMLAL)	SMLALBB, SMLALBT, SMLALTB, SLMALTT
16×32=32	16-bit by 32-bit signed multiply returning 32 MSB, i.e. word (SMULW)	SMULWB, SMULWT
(16×32)+32=32	Q setting 16-bit by 32-bit signed multiply with 32-bit accumulate (SMLAW)	SMLAWB, SMLAWT
(16×16)+(16×16)=32	Q setting sum of dual 16-bit signed multiply (SMU-AD)	SMUAD, SMUADX
(16×16)-(16×16)=32	Dual 16-bit signed multiply returning difference (SMU-SD)	SMUSD, SMUSDX
(16×16)+(16×16)+32=32	Q setting dual 16-bit signed multiply with single 32-bit accumulator (SMLAD)	SMLAD, SMLADX
(16×16)-(16×16)+32=32	Q setting dual 16-bit signed multiply subtract with 32-bit accumulate (SMLSD)	SMLSD, SMLSDX
(16×16)+(16×16)+64=64	Dual 16-bit signed multiply with single 64-bit accumulator (SMLALD)	SMLALD, SMLALDX
(16×16)-(16×16)+64= 64	Q setting dual 16-bit signed multiply subtract with 64-bit accumulate (SMLSLD)	SMLSLD, SMLSLDX
32×32=32	Multiply	MUL
32+(32×32)=32	Multiply accumulate	MLA
32-(32×32)=32	Multiply subtract	MLS

32×32=64	Long signed/unsigned multiply	SMULL, UMULL
(32×32)+64=64	Long signed/unsigned accumulate	SMLAL, UMLAL
(32×32)+32+32=64	32-bit unsigned multiply with double 32-bit accumulation yielding 64-bit result	UMAAL
32+(32×32) = 32 MSB	32-bit multiply with 32-most-significant-bit accumulate	SMMLA, SMMLAR
32-(32×32) =32 MSB	32-bit multiply with 32-most-significant-bit subtract	SMMLS, SMMLSR
(32×32)=32 MSB	32-bit multiply returning 32-most-significant-bits	SMMUL, SMMULR

Table 1-2 Instruction Sets of Multiply or Multiply-Accumulate in Arm Cortex-M4

1.2.2 Single Instruction Multiple Data (SIMD)

Single Instruction Multiple Data can improve the computational efficiency of the Cortex-M4 when performing DSP operations, which is not supported on the Cortex-M3. The mathematical expression for the Cortex-M4's SIMD instruction is (1.5), which can execute four 8-bit multiples or two 16-bit multiples in parallel in one cycle. The commands are listed in the gray bottom of Table 1-2, such as SMUAD, SMUADX, SMLSD, SMLSDX...etc.

$$\mathrm{Sum} = \mathrm{Sum} + (a \times c) + (b \times d) \tag{1.5}$$

1.2.3 Floating Point Unit (FPU)

The Arm Cortex-M4 also has a floating-point unit that conforms to the IEEE -754 specification, and single-precision floating-point numbers play an important role in signal processing algorithms. Table 1-3 lists the operations and number of clocks of single-precision floating-point numbers. It can be seen that multiplying floating-point numbers is time-consuming and the Arm Cortex-M4 provides the fused MAC instruction to increase accuracy.

Single-Precision Floating-Point Operation	Instruction	Number of Clocks
Addition and Subtraction	VADD.F32, VSUB.F32	1
Multiplication	VMUL.F32	3
Multiply and accumulate	VMLA.F32, VMLS.F32, VNMLA.F32, VNMLS.F32	3

Fused MAC	VFMA.F32, VFMS.F32, VFNMA.F32, VFNMS.F32	3
Square root	VSQRT.F32	14
Division	VDIV.F32	14

Table 1-3 Operations and Clocks of Single-Precision Floating-Point Number

1.3 CMSIS DSP Library Overview

The CMSIS DSP provides more than 60 types of DSP algorithm functions for Arm Cortex-M4 and is mainly classified in Table 1-4 as follows:

Function Class	Function Content
Basic Math Functions	Contains vector addition, subtraction, multiplication, division, inversion, absolute value, displacement...etc.
Fast Math Functions	Include Cosine and Sine values to reduce time by a look-up table.
Complex Math Functions	Contains complex products, complex conjugates, absolute values of complex numbers, etc.
Filtering Functions	Including convolution, correlation coefficient, least mean square filter, finite response filtering, wireless response filtering...etc.
Matrix Functions	Contains addition, subtraction, and multiplication of matrix, operations, inverse matrices, transposed matrix...etc.
Transform Functions	Includes Fast Fourier Transform of real and complex numbers, and Discrete Cosine Transform
Controller Functions	Contains PID controller, Clarke transform, and Park transform.
Statistics Functions	Including taking maximum, minimum, exponential operation, standard deviation, variance, quadratic mean...etc.
Support Functions	Contains Copying Vector Values, Filling Vector Values, and Converting Numeric Formats.
Interpolation Functions	Contains linear and bilinear differences.

Table 1-4 Classification of DSP Library and Its Content

2 Using DSP and CMSIS Library

The NuMicro Cortex-M4 DSP instruction functions can be used in three ways:

- Use CMSIS DSP library.
- Compose operational equations in a fixed format to make the system identify and apply the DSP instructions.
- Programming of assembly language.

Since the library functions are fully functional and have an optimized internal algorithm, they can effectively reduce execution time and facilitate direct development and application for users. Therefore, this chapter describes in detail how to use the CMSIS DSP library.

2.1 DSP Source Code

All NuMicro Cortex-M4 BSPs contain the source code of the DSP library located under the "..\Library\CMSIS\DSP_Lib\Source" path as shown in Figure 2-1. In addition, the DSP sample code provided by Arm is located under the "..\Library\CMSIS\DSP_Lib\Examples" path.

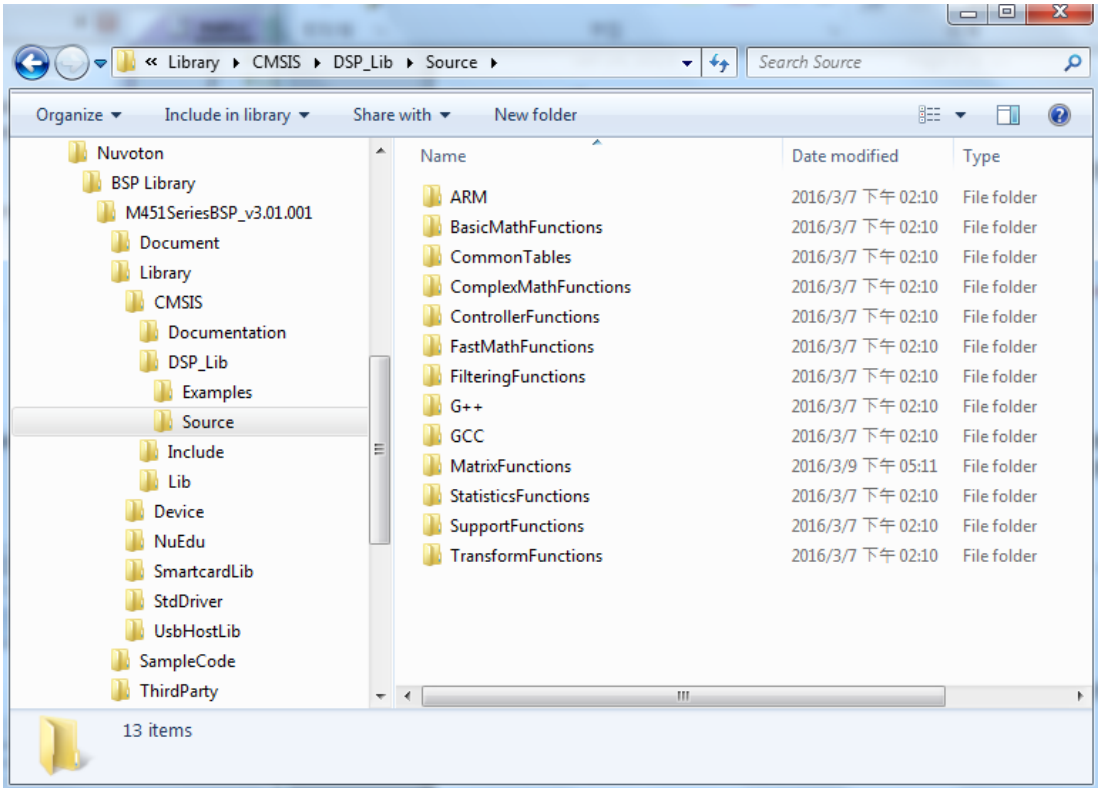


Figure 2-1 Libraries Location of Cortex-M4 DSP

2.2 Using DSP though Keil

- **Enable the DSP Function**

1. Open the "Options of the target" window and switch to the "**C/C++**" tab, import the text "ARM_MATH_CM4=1" and "__FPU_PRESENT=1(if you want to activate the FPU function)" into the definition line of the preprocessor symbols.

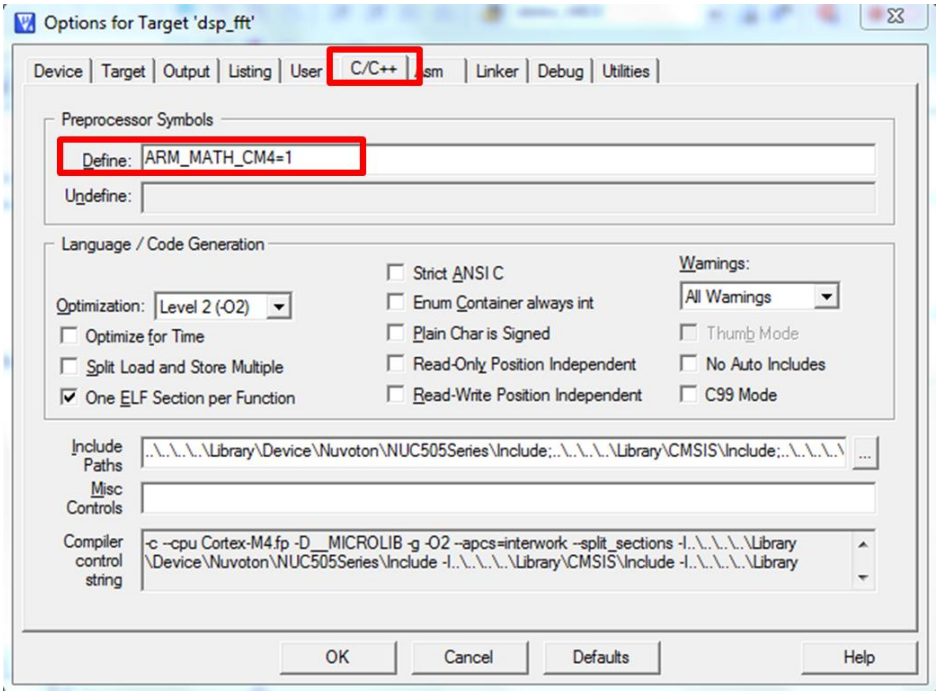


Figure 2-2 "Options for Target" Window

2. Open the “Projects” window, add the library "arm_cortexM4lf_math.lib" under the "..\Library\CMSIS\Lib\ ARM" path.

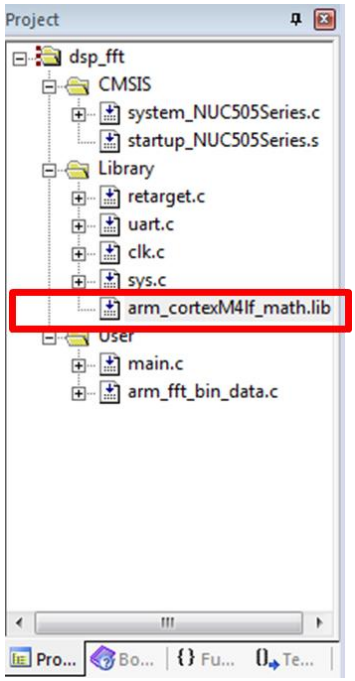


Figure 2-3 Projects Window

3. Open the file main.c and include the DSP library "arm_math.h". Then you can call the functions of this DSP library.

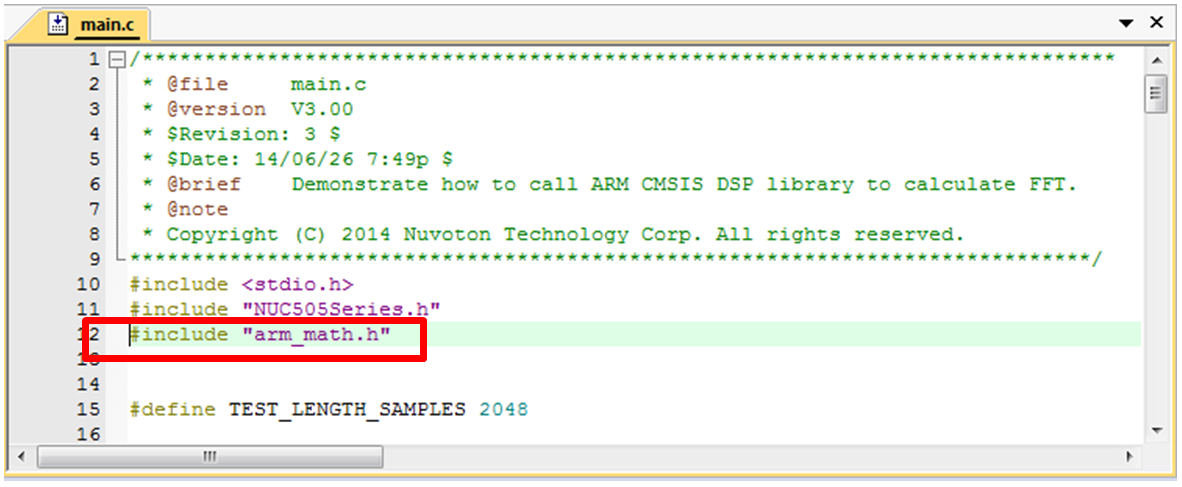


Figure 2-4 “main.c” Function

2.3 Using DSP though IAR

- **Enable the DSP Function**

1. Open the "Options for node" window and change the category to General Options, select the **Library Configuration** tab, and enable the "**Use CMSIS**" and "**DSP Library**" options.

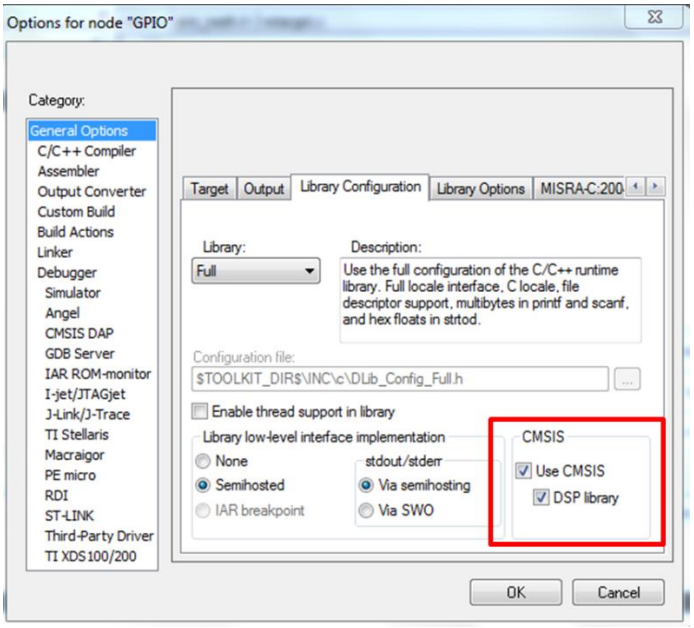


Figure 2-5 "Options for node" Window

2. Open the "Files" window, add the library "arm_cortexM4lf_math.lib" under the "..\Library\CMSIS\Lib\ARM" path.

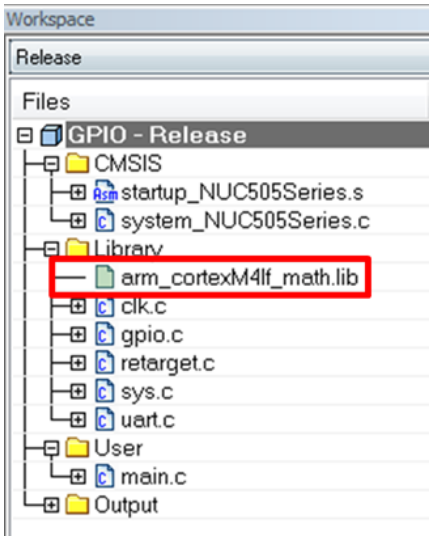
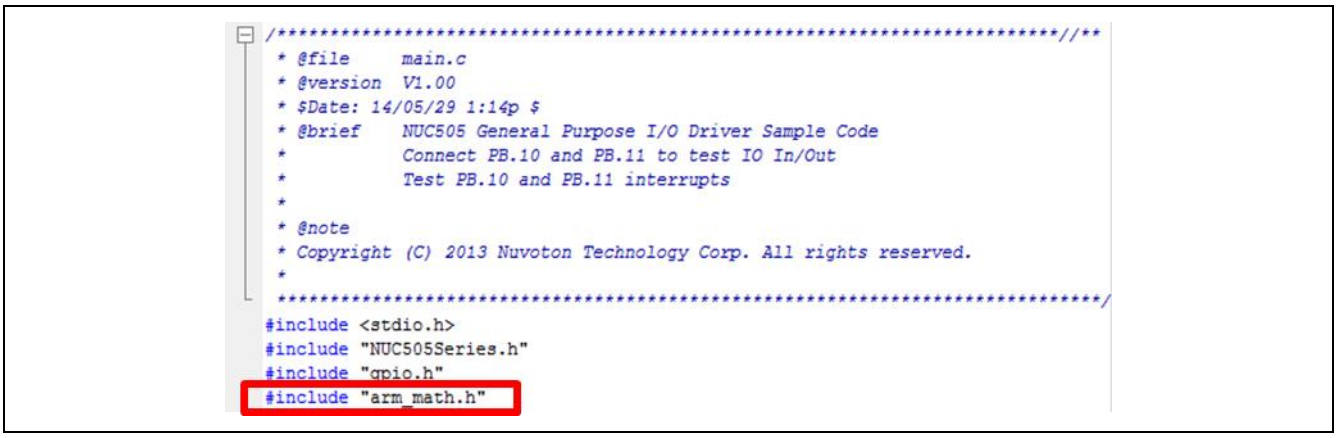


Figure 2-6 "Files" Window

3. Open the main.c file and include the DSP library "arm_math.h". Then, you can call functions of this DSP library.



```
/*****************************************************************************//**
 * @file     main.c
 * @version  V1.00
 * $Date: 14/05/29 1:14p $
 * @brief    NUC505 General Purpose I/O Driver Sample Code
 *           Connect PB.10 and PB.11 to test IO In/Out
 *           Test PB.10 and PB.11 interrupts
 *
 * @note
 * Copyright (C) 2013 Nuvoton Technology Corp. All rights reserved.
 *
 *****************************************************************************/
#include <stdio.h>
#include "NUC505Series.h"
#include "gpio.h"
#include "arm_math.h"
```

Figure 2-7 “main.c” Function

Note: Cortex-M4 DSP libraries are already included in “arm_cortexM4lf_math.lib”.

3 DSP Example Code

All example code presented in this chapter for the NuMicro Cortex-M4 DSP can be downloaded from Nuvoton official website. In the DSP function library provided by the Cortex-M4, most of the operation functions have four numeric formats: q31, q15, q7, and f32. Users can convert the formats according to their requirements using the functions , as shown in Table 3-1 ~ Table 3-4. In this document, sample code in f32 format are presented.

arm_float_to_q15 (float32_t *pSrc, q15_t *pDst, uint32_t blockSize) **arm_float_to_q31 (float32_t *pSrc, q31_t *pDst, uint32_t blockSize)** **arm_float_to_q7 (float32_t *pSrc, q7_t *pDst, uint32_t blockSize)**		
parameters	*pSrc	[in] numeric format of float
	*pDst	[out] numeric format of q15, q31, q7
	blockSize	[in] sample number
return		null

Table 3-1 Program Settings of Converting Float to q15, q31, and q7

arm_q15_to_float (q15_t *pSrc, float32_t *pDst, uint32_t blockSize) **arm_q15_to_q31 (q15_t *pSrc, q31_t *pDst, uint32_t blockSize)** **arm_q15_to_q7 (q15_t *pSrc, q7_t *pDst, uint32_t blockSize)**		
parameters	*pSrc	[in] numeric format of q15
	*pDst	[out] numeric format of float, q31,q7
	blockSize	[in] sample number
return		null

Table 3-2 Program Settings of Converting q15 to float, q31, and q7

arm_q31_to_float (q31_t *pSrc, float32_t *pDst, uint32_t blockSize) **arm_q31_to_q15 (q31_t *pSrc, q15_t *pDst, uint32_t blockSize)** **arm_q31_to_q7 (q31_t *pSrc, q7_t *pDst, uint32_t blockSize)**		
parameters	*pSrc	[in] numeric format of q31
	*pDst	[out] numeric format of float, q15,q7
	blockSize	[in] sample number
return		null

Table 3-3 Program Settings of Converting q31 to float, q15, and q7

arm_q7_to_float (q7_t *pSrc, float32_t *pDst, uint32_t blockSize) **arm_q7_to_q15 (q7_t *pSrc, q15_t *pDst, uint32_t blockSize)** **arm_q7_to_q31 (q7_t *pSrc, q31_t *pDst, uint32_t blockSize)**		
parameters	*pSrc	[in] numeric format of q7
	*pDst	[out] numeric format of float, q15,q31
	blockSize	[in] sample number
return		null

Table 3-4 Program Settings of Converting q7 to float, q15, and q31

3.1 Basic Math Operations

There are many basic vector math operations in the Cortex-M4 DSP library, including:

- Absolute value of vector
- Addition of vectors
- Subtraction of vectors
- Multiplication of vectors
- Vector dot product
- Inverse vector
- Vector scaling
- Vector offset

Users can easily use these functions to implement their own mathematical equations. The settings of each function program are listed in Table 3-5 to Table 3-12.

arm_abs_f32(float32_t *pSrc, float32_t *pDst, uint32_t blockSize)		
pDst[n] = abs(pSrc[n]), 0 <= n < blockSize		
Parameters	*pSrc	[in] vector data
	*pDst	[out] absolute value of vectors
	blockSize	[in] samples of vector
Return		null

Table 3-5 Program Settings of Vectors' Absolute Value

arm_add_f32(float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t blockSize)		
pDst[n] = pSrcA[n] + pSrcB[n], 0 <= n < blockSize		
Parameters	*pSrcA	[in] the first vector data
	*pSrcB	[in] the second vector data
	*pDst	[out] results of calculation
	blockSize	[in] samples of vector
Return		null

Table 3-6 Program Settings of Vectors Addition

arm_dot_prod_f32 (float32_t *pSrcA, float32_t *pSrcB, uint32_t blockSize, float32_t *result)		
sum = pSrcA[0]*pSrcB[0] + pSrcA[1]*pSrcB[1] + ... + pSrcA[blockSize-1]*pSrcB[blockSize-1]		
Parameters	*pSrcA	[in] the first vector data
	*pSrcB	[in] the second vector data
	blockSize	[in] samples of vector
	*result	[out] results of calculation
Return		null

Table 3-7 Program Settings of Vector Dot Product

arm_mult_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t blockSize)		
pDst[n] = pSrcA[n] * pSrcB[n], 0 <= n < blockSize		
Parameters	*pSrcA	[in] the first vector data
	*pSrcB	[in] the second vector data
	*pDst	[out] results of calculation
	blockSize	[in] samples of vector
Return		null

Table 3-8 Program Settings of Vectors Multiplication

arm_negate_f32 (float32_t *pSrc, float32_t *pDst, uint32_t blockSize)		
pDst[n] = -pSrc[n], 0 <= n < blockSize		
Parameters	*pSrc	[in] calculate vector
	*pDst	[out] results of calculation
	blockSize	[in] samples of vector
Return		null

Table 3-9 Program Settings of Inverse Vector

arm_offset_f32 (float32_t *pSrc, float32_t offset, float32_t *pDst, uint32_t blockSize)		
pDst[n] = pSrc[n] + offset, 0 <= n < blockSize		
Parameters	*pSrc	[in] calculate vector
	offset	[in] offset number
	*pDst	[out] results of calculation
	blockSize	[in] samples of vector
Return		null

Table 3-10 Program Settings of Vector Offset

arm_scale_f32 (float32_t *pSrc, float32_t scale, float32_t *pDst, uint32_t blockSize)		
pDst[n] = pSrc[n] * scale, 0 <= n < blockSize		
Parameters	*pSrc	[in] calculate vector
	scale	[in] scalling value
	*pDst	[out] results of calculation
	blockSize	[in] samples of vector
Return		null

Table 3-11 Program Settings of Vector Scaling

arm_sub_f32 (float32_t *pSrcA, float32_t *pSrcB, float32_t *pDst, uint32_t blockSize)		
pDst[n] = pSrcA[n] - pSrcB[n], 0 <= n < blockSize		
Parameters	*pSrcA	[in] the first vector data
	*pSrcB	[in] the second vector data
	*pDst	[out] results of calculation
	blockSize	[in] samples of vector
Return		null

Table 3-12 Program Settings of Vectors Subtraction

The following example code compares the execution time of each function using the DSP library and the C library, where the sample number calculation is 32. The result compares the difference in execution time with and without using the DSP library, as shown in Table 3-13. It shows that using the DSP function can greatly reduce the calculation time.

```
/*  Calculate inner product (sample number=32) */
arm_dot_prod_f32(srcA_buf_f32, srcB_buf_f32, blockSize, &dotoutput_f32);

/* Calculate absolute value (sample number=32) */
arm_abs_f32(srcA_buf_f32, absoutput, blockSize);

/* Computational addition (sample number=32) */
arm_add_f32(srcA_buf_f32, srcB_buf_f32, addoutput, blockSize);

/* Computational Subtraction (sample number=32) */
arm_sub_f32 (srcA_buf_f32, srcB_buf_f32, suboutput, blockSize);

/* Calculate the multiplication (sample number=32) */
arm_mult_f32 (srcA_buf_f32, srcB_buf_f32, multoutput, blockSize);

/* Calculate scaling (sample number=32) */
arm_scale_f32 (srcA_buf_f32, 10, scaleoutput, blockSize);

/* Calculate the offset (sample number=32) */
arm_offset_f32 (srcA_buf_f32, 10, offsetoutput, blockSize);
```

```
/* Calculate the inverse (sample number=32) */
arm_negate_f32 (srcA_buf_f32, negoutput, blockSize);
```

Function(with 32 sample)	DSP(Time unit)	CPU(Time unit)	CPU/DSP
Dot Product	52	105	2.02
Absolute Value	34	78	2.29
Addition	46	100	2.17
Subtraction	47	100	2.13
Multiplication	47	100	2.13
Scale	36	78	2.167
Offset	37	79	2.14
Negate	37	79	2.14

Table 3-13 Compare Execution Time of Basic Operations with and without Using DSP Library

3.2 Trigonometric Functions Operations

The Cortex-M4 DSP library provides the function of sine and cosine, reduced calculation times by look-up table, and the detailed formula as shown in Table 3-14, Table 3-15 and Table 3-16.

arm_cos_f32(float32_t x)		
Parameters	x	[in] calculate radian
Return		cos(x)

Table 3-14 Program Settings of Calculated Cosine by Radian

arm_sin_f32(float32_t x)		
Parameters	x	[in] calculate radian
Return		sin(x)

Table 3-15 Program Settings of Calculated Sine by Radian

arm_sin_cos_f32(float32_t theta, float32_t *pSinVal, float32_t *pCosVal)		
Parameters	theta	[in] calculate angle
	*pSinVal	[out] output of sine
	*pCosVal	[out] output of cosine
Return		null

Table 3-16 Program Settings of Calculated Sine and Cosine by Angle

The following example code compares the computation times between the DSP library and the C library, where the number of computation examples is 32. The results are listed in Table 3-17 and show that using the DSP function can significantly reduce the computation time.

```
/* calculate of sin, cos (sample number=32) */
for(i=0; i< blockSize; i++)
{
    /* input (radian) */
    cosOutput[i] = arm_cos_f32(testInput_f32[i]);
    sinOutput[i] = arm_sin_f32(testInput_f32[i]);
```

```
    /* input (angle) */
    arm_sin_cos_f32(testInput_f32[i], &sinOutput1[i], &cosOutput1[i]);
}
```

Function(with 32 sample)	DSP(Time unit)	CPU(Time unit)	CPU/DSP
Sine	551	23251	42.2
Cosine	551	23720	43.05
Sine and Cosine	413	46561	112.74

Table 3-17 Compare Execution Time of Trigonometric Functions Operations with and without Using DSP Library

3.3 Linear Interpolation

Assuming that the coordinates (x0, y0) and (x1, y1) are currently known, the value of a certain position x in the line interval on the straight line is to be obtained. According to Figure 3-1, get:

$$\frac{y - y0}{x - x0} = \frac{y1 - y0}{x1 - x0}$$

Since the x value is known, the value can be obtained from the formula.

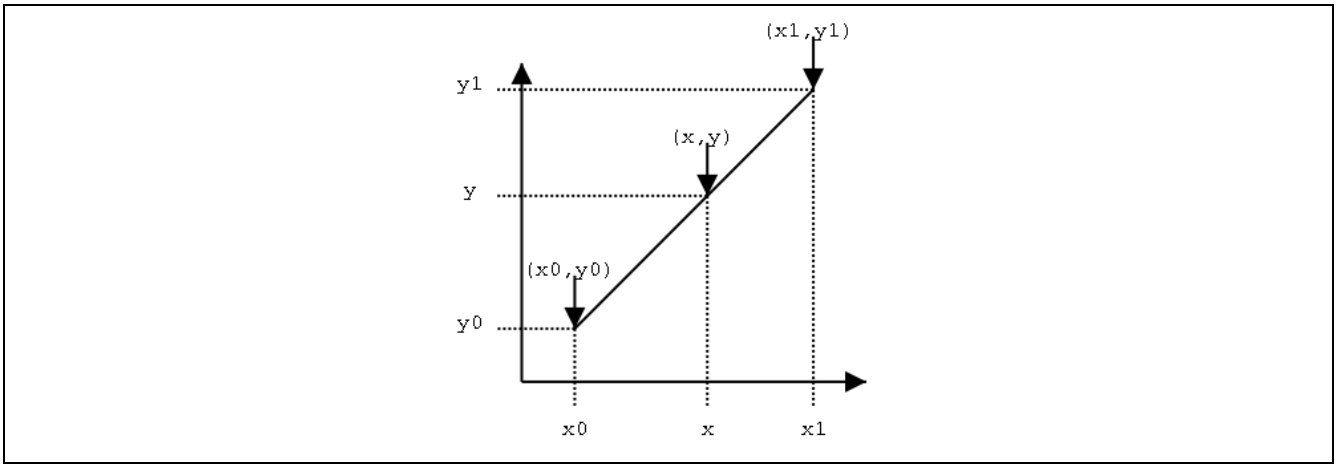


Figure 3-1 Schematic of Linear Interpolation

Use the linear interpolation function of the Cortex-M4 library DSP. First, specify the number of samples of x0, the initial value of x0, and the interval value of x, and then specify the value of y. The program settings are shown in Table 3-18 and Table 3-19. After setting the structure for floating-point interpolation, enter the value of x to run the program and then obtain the value of y by linear interpolation.

arm_linear_interp_instance_f32 S = {uint32_t nValues, float32_t x0, float32_t xSpacing, float32_t *pYData}		
Parameters	nValues	sample number of x
	x0	starting number of x
	xSpacing	interval value of x
	*pYData	y value
Return		null

Table 3-18 Structure Setting of Linear Interpolation

arm_linear_interp_f32(arm_linear_interp_instance_f32 *S, float32_t x)		
Parameters	*S	[in, out] structure of floating-point interpolation
	x	[in] input value of x
Return		linear interpolation result of y

Table 3-19 Program Settings of Linear Interpolation

The following example code shows the difference in processing time with or without using the DSP function, as shown in Table 3-20.

```
/* Set the structure variable S: define samples number = 2.
Its starting value is 0 and the value interval is 10 (x0=0, x1=10), and the y value is
defined as arm_linear_interep_table*/
arm_linear_interp_instance_f32 S = {2, 0, 10, (float32_t *)&arm_linear_interep_table[0]};


/* Set the sample number = 10, for linear interpolation and outputs the result to
testLinIntOutput[] */
for(i=0; i< TEST_LENGTH_SAMPLES; i++)
{
        testLinIntOutput[i] = arm_linear_interp_f32(&S, testInputSin_f32[i]);
}
```

Function(with 10 sample)	DSP(Time unit)	CPU(Time unit)	CPU/DSP
Linear Interpolation	138	155	1.12

Table 3-20 Compare Execution Time of Linear Interpolation Operations with and without Using DSP Library

3.4 Statistical Operations

There are statistical operations in the Cortex-M4 DSP library, including:

- Take the maximum
- Take the minimum
- Take the average
- Quadratic mean
- Standard deviation
- Variance

Two of the most commonly used functions are Square Mean and Standard Deviation.

Quadratic mean, also known as Root Mean Square (RMS), its mathematical expression, and the program settings are listed in Table 3-21:

$$M = \sqrt{\frac{\sum_{i=1}^{n} x_i^2}{n}}$$

Standard Deviation (SD) is most commonly used in probability statistics to measure the degree of dispersion of a group of values. Its mathematical expression is as follows, where u is the average value of x, and the program settings are listed in Table 3-22:

$$SD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - u)^2}$$

arm_rms_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)		
Parameters	*pSrc	[in] matrix of calculating
	blockSize	[in] sample number of matrix
	*pResult	[out] calculated of result
Return		null

Table 3-21 Program Settings of Quadratic Mean

arm_std_f32 (float32_t *pSrc, uint32_t blockSize, float32_t *pResult)		
Parameters	*pSrc	[in] calculate value
	blockSize	[in] sample number
	*pResult	[out] calculated of result
Return		null

Table 3-22 Program Settings of Standard Deviation

The compared operation time with and without DSP is shown in Table 3-23. The sample code is as follows:

```
/* calculated of Quadratic mean (sample number=32) */
arm_rms_f32 (testInput_f32, blockSize, &testoutput_f32);
```

```
/* calculated of Standard Deviation (sample number=32) */
arm_std_f32(testMarks_f32, blockSize, &std);
```

Function	DSP(Time unit)	CPU (Time unit)	CPU/DSP
RMS (with 32 sample)	45	413	9.18
Standard Deviation (with 32 sample)	116	1024	8.83

Table 3-23 Compare Execution Time of RMS and SD with and without Using DSP Library

3.5 Matrix Operations

The Cortex-M4 DSP library has many matrix operations, including:

- Matrix addition
- Matrix subtraction
- Matrix multiplication
- Inverse matrix
- Matrix scaling
- Transpose matrix

First, declare the matrix variables with arm_matrix_instance_f32() and initialize the used matrix. The number of rows, columns, and the value of the matrix can be defined with arm_mat_init_f32(). After that, the user can perform operations with each matrix. The program settings are listed in Table 3-24 ~ Table 3-30.

arm_mat_init_f32(arm_matrix_instance_f32 *S, uint16_t nRows, uint16_t nColumns, float32_t *pData)		
Parameters	*S	[in, out] a pointer to declared float matrix
	nRows	[in] rows of matrix
	nColumns	[in] columns of matrix
	*pData	[in] data of the matrix
Return		null

Table 3-24 Initial Settings of Matrix

arm_mat_add_f32(const arm_matrix_instance_f32 *pSrcA, const arm_matrix_instance_f32 *pSrcB, arm_matrix_instance_f32 *pDst)		
Parameters	*pSrcA	[in] augend matrix
	*pSrcB	[in] addend matrix
	*pDst	[out] matrix of result
Return		If successful, ARM_MATH_SUCCESS else ARM_MATH_SIZE_MISMATCH (matrix incompatibility)

Table 3-25 Program Settings of Matrix Addition

arm_mat_sub_f32(const arm_matrix_instance_f32 *pSrcA, const arm_matrix_instance_f32 *pSrcB, arm_matrix_instance_f32 *pDst)		
Parameters	*pSrcA	[in] minuend matrix
	*pSrcB	[in] subtrahend matrix
	*pDst	[out] matrix of result
Return		If successful, ARM_MATH_SUCCESS else ARM_MATH_SIZE_MISMATCH (matrix incompatibility)

Table 3-26 Program Settings of Subtraction Matrix

arm_mat_mult_f32(const arm_matrix_instance_f32 *pSrcA, const arm_matrix_instance_f32 *pSrcB, arm_matrix_instance_f32 *pDst)		
Parameters	*pSrcA	[in] multiplicand matrix
	*pSrcB	[in] multipliers matrix
	*pDst	[out] matrix of result
Return		If successful, ARM_MATH_SUCCESS, else ARM_MATH_SIZE_MISMATCH (matrix incompatibility)

Table 3-27 Program Settings of Matrix Multiplication

arm_mat_inverse_f32(const arm_matrix_instance_f32 *pSrc, arm_matrix_instance_f32 *pDst)		
Parameters	*pSrc	[in] matrix of calculating
	*pDst	[out] inverse matrix for the output
Return		If successful, ARM_MATH_SUCCESS else ARM_MATH_SIZE_MISMATCH (matrix incompatibility), or ARM_MATH_SINGULAR (matrix irreversible)

Table 3-28 Program Settings of Inverse Matrix

arm_mat_scale_f32(const arm_matrix_instance_f32 *pSrc, float32_t scale, arm_matrix_instance_f32 *pDst)		
Parameters	*pSrc	[in] matrix of calculating
	scale	[in] parameter of scaling
	*pDst	[out] matrix of result
Return		If successful, ARM_MATH_SUCCESS

Table 3-29 Program Settings of Matrix Scaling

arm_mat_trans_f32(const arm_matrix_instance_f32 *pSrc, arm_matrix_instance_f32 *pDst)		
Parameters	*pSrc	[in] matrix of calculating
	*pDst	[out] transpose matrix for the output
Return		If successful, ARM_MATH_SUCCESS

Table 3-30 Program Settings of Transpose Matrix

The compared operation time with and without DSP is shown in Table 3-31. The sample code is as follows:

```
arm_matrix_instance_f32 A;          /* Declare the structure parameter of matrix A */
arm_matrix_instance_f32 AT;         /* matrix AT(A transpose) */
arm_matrix_instance_f32 ATMA;       /* matrix ATMA(AT multiply with A)*/
arm_matrix_instance_f32 ATMAI;      /* matrix ATMAI(Inverse of ATMA) */

/* Init and set the matrix A: size=5*5, data=A_f32 */
arm_mat_init_f32(&A, 5, 5, A_f32);
/* Init and set the matrix AT: size=5*5, data=A_f32AT_f32 */
arm_mat_init_f32(&AT, 5, 5, AT_f32);
/* Matrix A is transposed and stored in matrix AT */
arm_mat_trans_f32(&A, &AT);
/* Init and set the matrix ATMA: size=5*5, data=ATMA_f32 */
arm_mat_init_f32(&ATMA, 5, 5, ATMA_f32);
/* The calculated result of AT×A and stored in the matrix ATMA */
```

```
arm_mat_mult_f32(&AT, &A, &ATMA);
/* Init and set the matrix ATMAI: size=5*5, data=ATMAI_f32 */
arm_mat_init_f32(&ATMAI, 5, 5, ATMAI_f32);
/* Matrix A performs inverse matrix operation and stores it in matrix ATMAI */
arm_mat_inverse_f32(&A, &ATMAI);
```

Function(with 5*5 sample)	DSP(Time unit)	CPU(Time unit)	CPU/DSP
Transpose	47	74	1.57
Multiply	288	483	1.67
Inverse	725	7174	9.9

Table 3-31 Compare Execution Time of Matrix with and without Using DSP Library

3.6 Convolution

Convolution is a mathematical operator that generates a third function h(x) from two functions f(x) and g(x), denoted as h(x)=(f*g)(x). It is the integral of the product of one function with another that has been translated and flipped, thus:

$$(f * g)(t) \stackrel{\text{def}}{=} \int f(\tau)g(t-\tau)d\tau$$

The following example illustrates its calculation method and process:

1. There are two functions *f(x)={1,2,3,4,5}* and *g(x)={-1,-2,-3,-4,-5}*, and flips *g(x)* to *g(-x)*

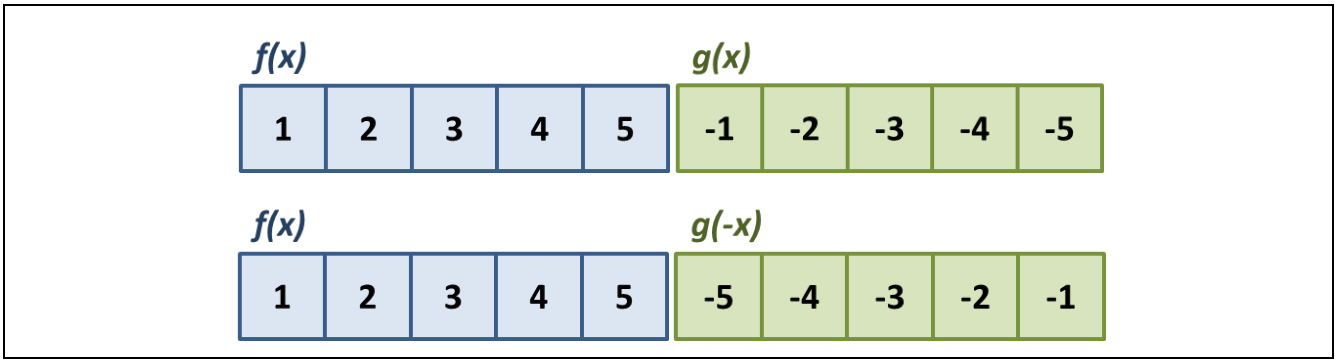


Figure 3-2 Functions and its flips

2. Multiply the two translation functions *f(x)* and *g(-x)* to produce *h(x)*, which is calculated as follows:

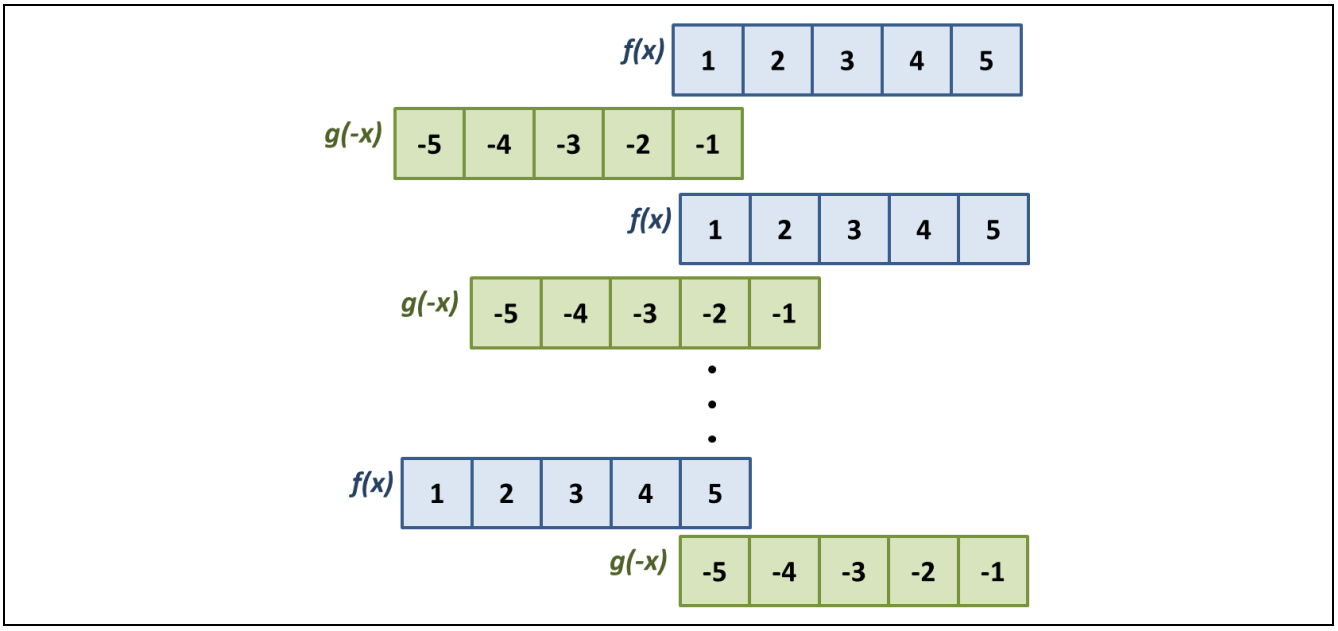


Figure 3-3 Processing of Calculate

h(1)=f(1)×g(1)= -1

h(2)=f(2)×g(1)+ f(1)×g(2)= -4

h(3)=f(3)×g(1)+ f(2)×g(2)+ f(1)×g(3)= -10

And so on, so *h(x)={-1,-4,-10,-20,-35,-44,-46,-40,-25}*

Users can call the function "***arm_conv_f32()"*** of the DSP library to implement the convolution operation defined in the Table 3-32:

arm_conv_f32(float32_t *pSrcA, uint32_t srcALen, float32_t *pSrcB, uint32_t srcBLen, float32_t *pDst)		
Parameters	*pSrcA	[in] the first function f(x)
	srcALen	[in] sample number of f(x)
	*pSrcB	[in] the second function g(x)
	srcBLen	[in] sample number of g(x)
	*pDst	the output result of h(x), and the sample number as srcALen+ srcBLen-1
Return		null

Table 3-32 Program Settings of Convolution

The compared operation time with and without DSP is shown in Table 3-33. The sample code is as follows:

```
    /* Convolution calculation */
    arm_conv_f32(testInput_f32_1kHz_15kHz, TEST_LENGTH_SAMPLES, firCoeffs32, NUM_TAPS,
conoutput);
```

Function	DSP(Time unit)	CPU(Time unit)	CPU/DSP
Convolution	7442	27703	3.72

Table 3-33 Compare Execution Time of Convolution with and without Using DSP Library

3.7 Finite Impulse Response

The finite impulse response (FIR) filter is a type of digital filter, also known as a FIR digital filter, whose operation is based on the convolution theorem. The finite impulse response filter is a linear system with input signals x(0), x(1)...x(n) and the output y(n) passed through the system can be expressed as follows:

$$y(n) = h_0 x(n) + h_1 x(n-1) + \cdots + h_N x(n-N)$$

where h_0 、h_1... h_N are impulse responses of the filter, referred to as filter coefficients. N is the order of the filter; the formula can be expressed as:

$$y(n) = \sum_{k=0}^{N} h_k \, x(n-k)$$

In the DSP library, using ***arm_fir_init_f32()*** function to initialize the settings, input the impulse response coefficient h and the number of samples of the filter, and then input the signal x into arm_fir_f32() to execute it. Users will then get the result of the signal x processed by the FIR filter. Table 3-34 and Table 3-35 describe the program settings for the FIR operations:

arm_fir_init_f32(arm_fir_instance_f32 *S, uint16_t numTaps, float32_t *pCoeffs, float32_t *pState, uint32_t blockSize)		
Parameters	*S	[in,out] a pointer to structure of FIR filter
	numTaps	[in] samples of filter coefficients h
	*pCoeffs	[in] matrix of filter coefficients h
	*pState	[in] state matrix
	blockSize	[in] calculating number of samples
Return		null

Table 3-34 Initial Settings of Finite Impulse Response

arm_fir_f32(const arm_fir_instance_f32 *S, float32_t *pSrc, float32_t *pDst, uint32_t blockSize)		
Parameters	*S	[in] a pointer to structure of FIR filter
	*pSrc	[in] calculating value
	*pDst	[out] Output result by FIR calculated
	blockSize	[in] calculating number of samples
Return		null

Table 3-35 Program Setting of Finite Impulse Response

The compared operation time with and without DSP is shown in Table 3-36. The sample code is as follows:

```
/* Declare the structure parameter of FIR */
arm_fir_instance_f32 S;
/*Init FIR */
arm_fir_init_f32(&S, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0], blockSize);
/* Set the loop times for execute */
for(k=0; k < numBlocks; k++)
{     /* number of samples to perform at once FIR operation */
      arm_fir_f32(&S, inputF32 + (k * blockSize), outputF32 + (k * blockSize),blockSize);
}
```

Function	DSP(Time unit)	CPU(Time unit)	CPU/DSP
FIR	7496	27703	3.7

Table 3-36 Compare Execution Time of Finite Impulse Response with and without Using DSP Library

In the example code, the input signal has a 1 kHz sine wave with 15 kHz noise, as shown in Figure 3-4, and the input of the FIR filter is calculated as a low-pass filter that filters out signals above 6 kHz. Thus, after the signal passes through the FIR filter, it outputs a 1 kHz signal and 15 kHz noise eliminated, as shown in Figure 3-5.

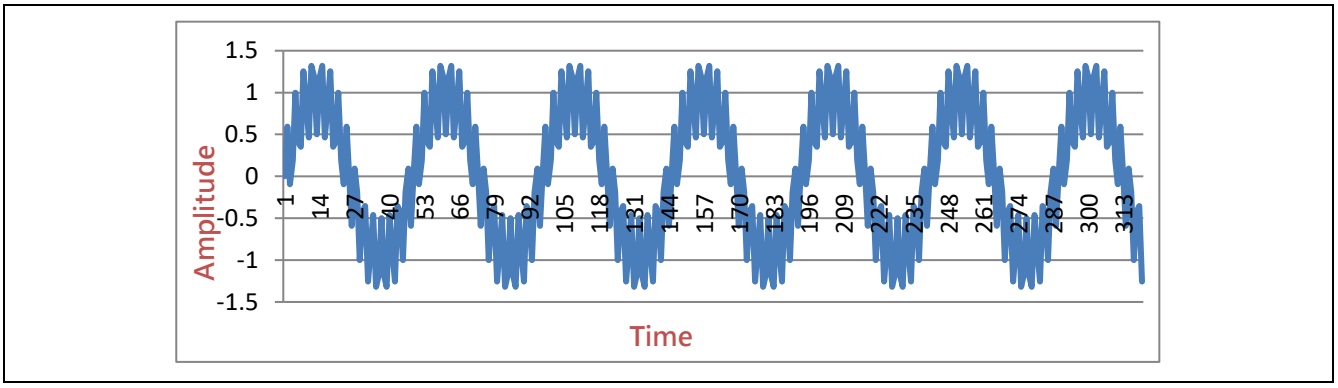


Figure 3-4 Input Signal (a 1 kHz Sine Wave with 15 kHz Noise)

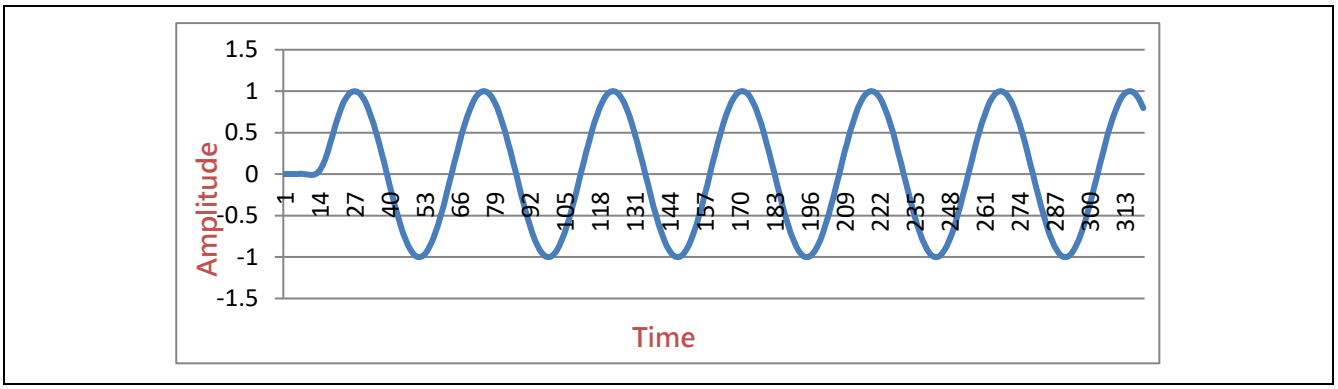


Figure 3-5 Output Signal Processed by FIR Filter (1 kHz Sine Wave)

3.8 Proportional - Integral - Derivative Controller (PID Controller)

The proportional-integral-differential controller consists of a proportional unit P, an integral unit I, and a differential unit D, which correspond to the current error, the past accumulated error, and the future error, and controlling feedback response by parameters Kp, Ki, and Kd. This controller is a common component of the feedback loop in industrial control applications. This controller compares the measured data to a target value and converts the difference to a new input by the PID controller to allow the system data to reach or maintain the target value. The PID controller can adjust the input value according to the historical data and the frequency of occurrence of the difference, making the system more accurate and stable.

The sample code from Nuvoton shows how to use the PID controller. First, declare matrix variables with the arm_pid_instance_f32 () function and initialize the PID controller settings. Next, enter the reference target value as input to arm_pid_f32() and for operation (to get the output value after the PID controller). Then determine the error value by subtracting the first output value and the target reference value, and inputting it into the PID operation. The output value can approach the target reference value by going down. The execution flow is shown in Figure 3-6, and the program is listed in Table 3-37 and Table 3-38.

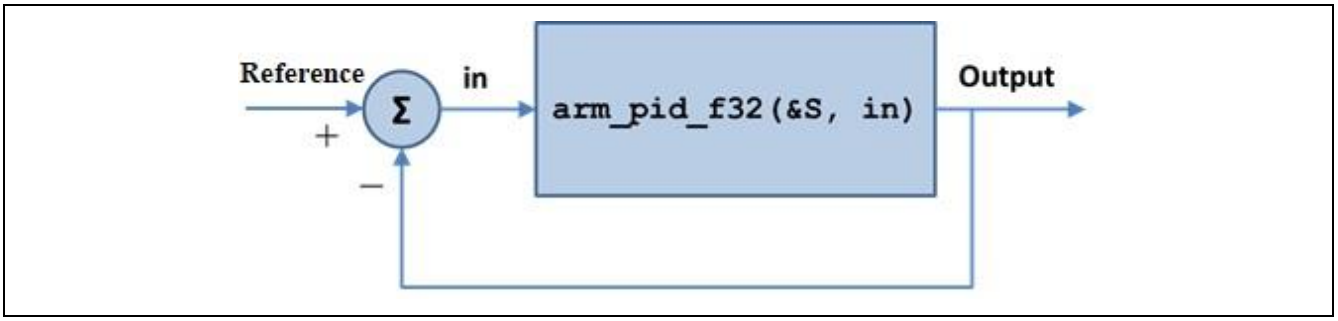


Figure 3-6 Execution Flow Chart of PID Control

arm_pid_init_f32(arm_pid_instance_f32 *S, int32_t resetStateFlag)		
Parameters	*S	[in, out] Declared structure parameter of PID
	resetStateFlag	[in] "0" is a state of not reset state, "1" is a state of reset
Return		null

Table 3-37 Initial Settings of PID

arm_pid_f32(arm_pid_instance_f32 *S, float32_t in)		
Parameters	*S	[in] Declared structure parameter of PID
	in	[out] The input value for operation (reference value minus return value)
Return		The output after PID

Table 3-38 Program Setting of PID

The compared operation time with and without DSP is shown in Table 3-39. The sample code is as follows:

```
/* declare structure variable */
arm_pid_instance_f32 PIDS;
/* declare values of Kp, Ki, Kd */
PIDS.Kp=0.4;
PIDS.Ki=0.4;
PIDS.Kd=0;
/* Declare target value */
target=500;
/* declare start value */
ival=0;
/* declare error value */
ee=target-ival;
/*Init of PID*/
arm_pid_init_f32(&PIDS,0);
for(i=1;i<100;i++)
{
   /* Enter the previous error value(ee) for the operation and get the output*/
   output[i]=arm_pid_f32(&PIDS,ee);
   /Calculate the current error value */
   ee=target-output[i-1];
 }
```

Function	DSP(Time unit)	CPU(Time unit)	CPU/DSP
PID	939	1702	1.81

Table 3-39 Compare Execution Time of PID with and without Using DSP Library

In this example, the motor speed starts at 0 and reaches a value of 500 after PID control. The values for Kp, Ki, and Kd are set to 0.4, 0.4, and 0, respectively. After the motor has been

operated 100 times, the result is shown in Figure 3-7.

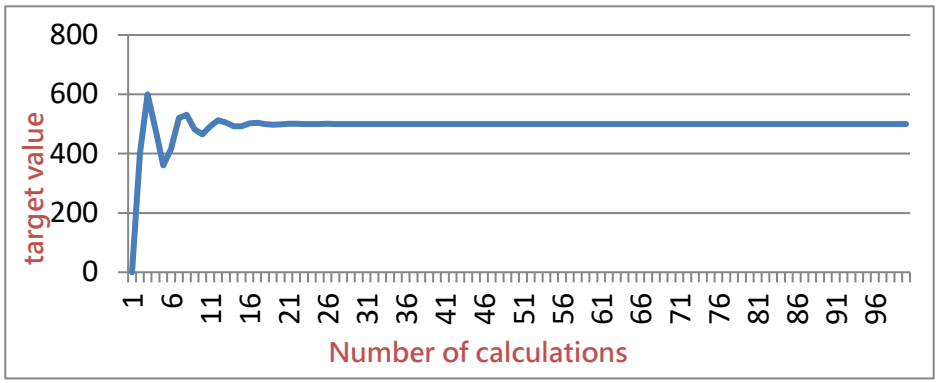


Figure 3-7 Results of Example Program by PID Operation

3.9 Fast Fourier Transform (FFT)

The mathematician Fourier derived a transformation formula to convert the data originally belonging to the time domain into the frequency domain. This transformation is a great help in the observation and improvement of signals. After the signal is transformed by FFT, we can obviously see the main frequency position of the signal and the relative relationship with other frequency intensities.

The fast Fourier transform can be used to decompose the signal from the time domain to the frequency domain. As shown in Figure 3-8, it is a kind of square wave f, which consists of many sine waves. The components of the sine wave are shown in Figure 3-9, and after the FFT, the frequencies and intensities of the sine waves (as shown in Figure 3-10). Thus, after a signal has been subjected to a fast Fourier transform, its composition can be clearly understood and signal processing can be performed according to the result, e.g., using filters.

Figure 3-8 Square Wave “ f ”

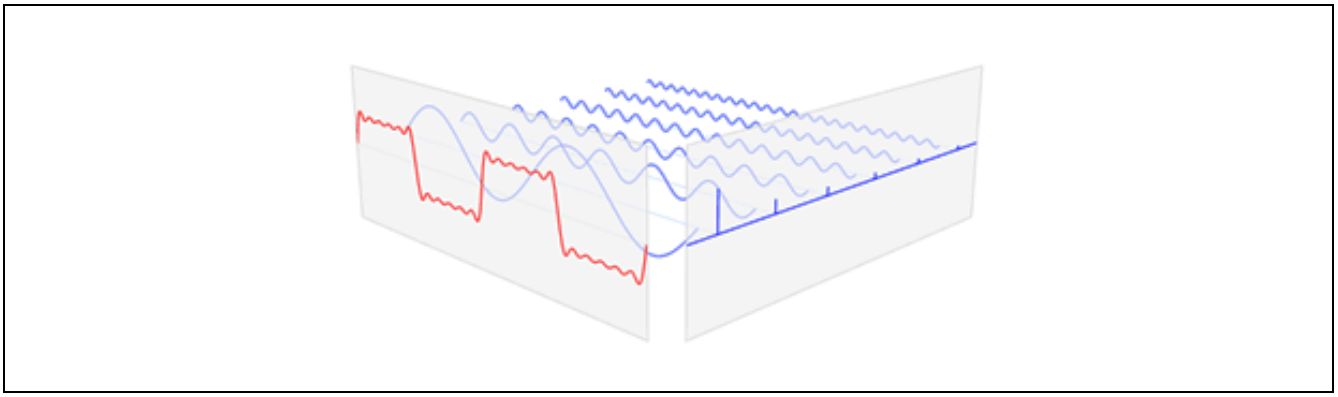


Figure 3-9 A Square Wave that Consist of Many Sine Waves

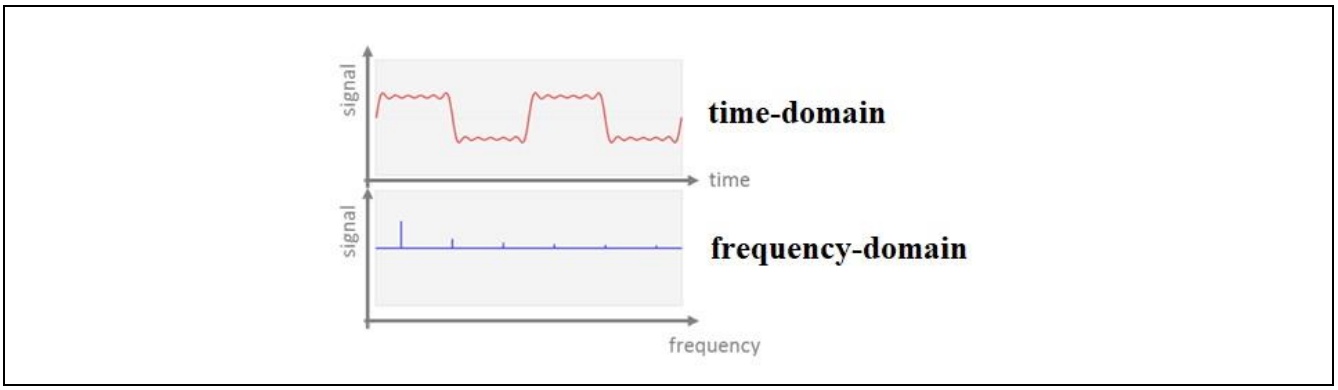


Figure 3-10 Combine Frequency and Intensity of Each Sine Wave into a Square Wave

The Cortex-M4 DSP library provides the function of FFT. Users can input the data to get the maximum frequency intensity and position. The following shows sample code for users to reference. The equations used are defined in Table 3-40 to Table 3-43.

arm_cfft_radix4_init_f32(arm_cfft_radix4_instance_f32 *S, uint16_t fftLen, uint8_t ifftFlag, uint8_t bitReverseFlag)		
Parameters	*S	[in, out] structure of complex FFT.
	fftLen	[in] sample number of FFT calculation (must be a power of 2)
	ifftFlag	[in] Perform a forward or inverse FFT operation (0 is positive, 1 is negative)
	bitReverseFlag	[in] Whether inverted the bits (1 is positive, 0 is negative).
Return		If the initialization successful, ARM_MATH_SUCCESS. If the number of samples is incorrect, ARM_MATH_ARGUMENT_ERROR.

Table 3-40 Initial Settings of Complex FFT

arm_cfft_radix4_f32(const arm_cfft_radix4_instance_f32 *S, float32_t * pSrc)		
Parameters	*S	[in] structure of complex FFT.
	*pSrc	[in, out] calculating array, and return the calculation result to the original array, the array format is (real number, complex number, real number...)
Return		null

Table 3-41 Program Settings of Complex FFT

arm_cmplx_mag_f32(float32_t *pSrc, float32_t *pDst, uint32_t numSamples)		
Parameters	*pSrc	[in] calculating complex array
	*pDst	[out] an array of real numbers after calculated
	numSamples	[in] Enter the number of samples of the complex array
Return		null

Table 3-42 Program Settings of Getting Complex Absolute Value

arm_max_f32(float32_t *pSrc, uint32_t blockSize, float32_t *pResult, uint32_t *pIndex)		
Parameters	*pSrc	[in] calculating array
	blockSize	[in] sample number of calculation
	*pResult	[out] maximum of result
	*pIndex	[out] location of maximum
Return		null

Table 3-43 Program Settings of Getting Maximum

The compared operation time with and without DSP is shown in Table 3-44. The sample code is as follows:

```
  /* Declare the structure parameter of FFT */
arm_cfft_radix4_instance_f32 S;
float32_t maxValue;
 /* Init FFT/IFFT */
arm_cfft_radix4_init_f32(&S, fftSize, ifftFlag, doBitReverse);
 /* Perform an FFT operation and return the result to the original input array */
arm_cfft_radix4_f32(&S, testInput_f32_10khz);
  /* Take the absolute value of each calculation to get intensity of each frequency */
arm_cmplx_mag_f32(testInput_f32_10khz, testOutput, fftSize);
 /* Take the maximum value as the main frequency position and value */
arm_max_f32(testOutput, fftSize, &maxValue, &testIndex);
```

Function(1024 sample)	DSP(Time unit)	CPU(Time unit)	CPU/DSP
FFT	18027	233893	12.98

Table 3-44 Compare Execution Time of FFT with and without Using DSP Library

After the fast Fourier transform, the complex value of each frequency is obtained, then the absolute value of the complex value, and finally the intensity distribution of each frequency. The details are shown in Figure 3-11 ~ Figure 3-12. Therefore, you can see the frequency intensity distribution of the cluttered signal and find that the input signal is a 10 kHz signal with noise.

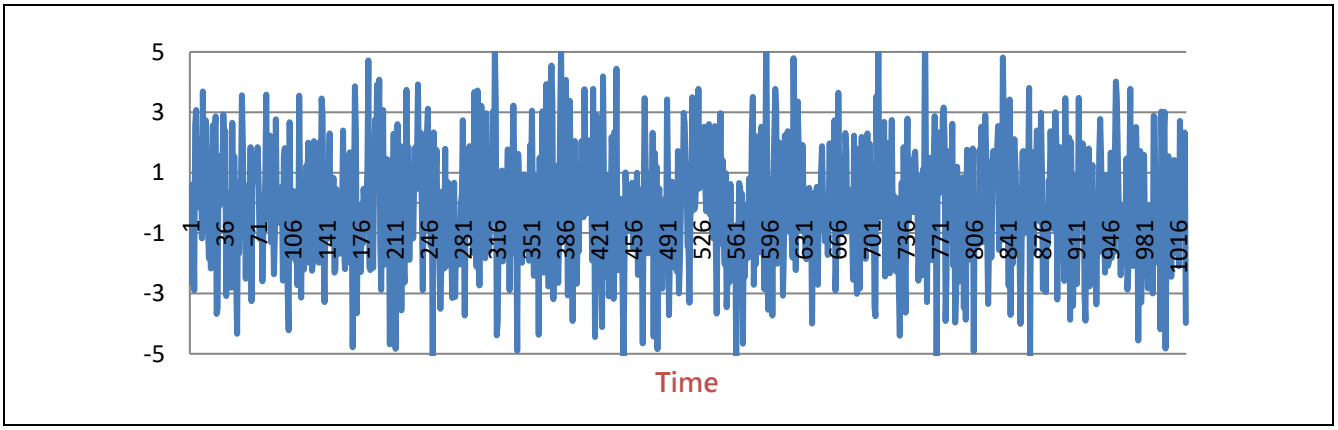


Figure 3-11 Input Signal of the Sample Program (Time-Domain)

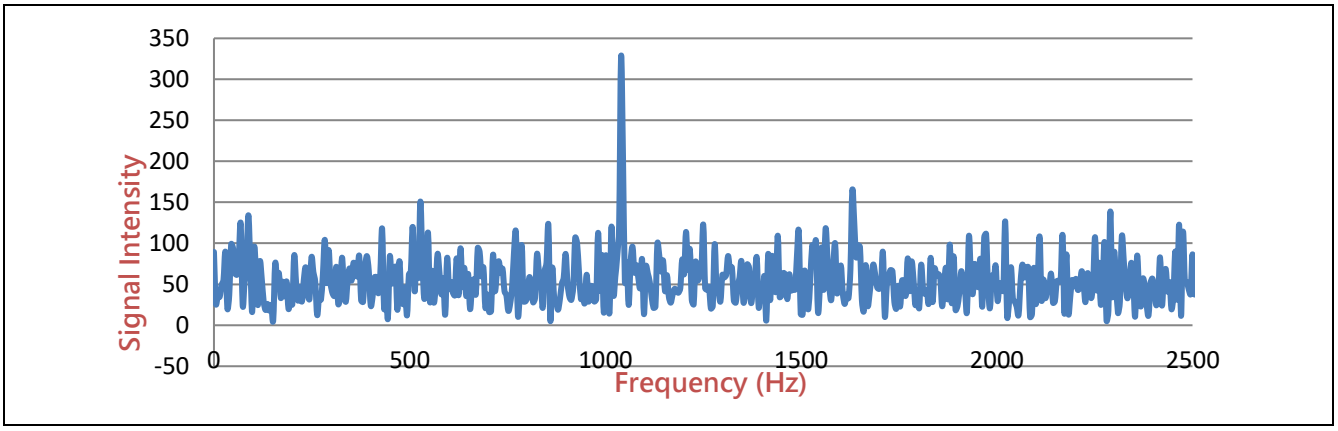


Figure 3-12 Signal after FFT Conversion (Frequency-Domain)

4 Conclusion

The Cortex-M4 DSP library introduced by this document is based on a microcontroller and adds the common functions of digital signal processing such as multiply-accumulate and single-instruction-multiple-data scheduling.

The Cortex-M4 DSP has rich peripherals and I/O pins and supports floating-point arithmetic. Besides, the CMSIS DSP library also offers more than 60 types of mathematical formulas that can effectively reduce the runtime of algorithms and promote the development work.

This document compares the operation time of 20 kinds of commonly mathematical equations as shown in Table 4-1. The equations set the sample number as 32 when it was not specified. There are three calculation methods. The first one uses Cortex-M4 with the DSP library enabled and calls the CMSIS library for calculation. The second one uses Cortex-M4 without using the DSP library. The last one uses the Cortex-M3 architecture without any DSP capabilities and does not support floating-point arithmetic. Ratios of operation time with M4/M4-DSP and M3/M4-DSP are shown in the last two lines of Table 4-1, where the maximum of reduced time can be a hundredfold, proving that using the Cortex-M4 with the DSP library can reach a faster execution speed for users.

Function (with 32 sample)	M4 with DSP (Time unit)	M4 (Time unit)	M3 without FPU	M4/M4 with DSP	M3/M4 with DSP
Dot Product	52	105	614	2.02	11.8
Absolute Value	34	78	67	2.29	1.97
Addition	46	100	382	2.17	8.3
Subtraction	47	100	406	2.13	8.64
Multiplication	47	100	327	2.13	6.96
Scale	36	78	330	2.167	9.167
Offset	37	79	327	2.14	8.84
Negate	37	79	67	2.14	1.81
Sine	551	23251	22165	42.2	40.23
Cosine	551	23720	22622	43.05	41.06
Sine and Cosine	413	46561	44742	112.74	108.33

Linear Interpolation (with 10 sample)	138	155	1479	1.12	10.72
RMS (with 32 sample)	45	413	951	9.18	21.13
Standard Deviation (with 80 sample)	116	1024	6676	8.83	57.55
Matrix Transpose(5*5)	47	74	65	1.57	1.38
Matrix Multiply(5*5)	288	483	1335	1.67	4.64
Matrix Inverse(5*5)	725	7174	7264	9.9	10.02
Convolution (320*29 sample)	7442	27703	171283	3.72	23.02
FIR(320*29 sample)	7496	27703	171283	3.7	22.85
PID (Calculate 100 times)	556	1014	7139	1.82	12.84
FFT (with 1024 sample)	18027	233893	237829	12.98	13.2

Table 4-1 Compare Execution Time of Cortex-M4 and Cortex-M3 with and without Using DSP Library

Revision History

Date	Revision	Description
2022.09.28	1.00	1. Initially issued.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.