APPLICATION NOTE

Atmel AT03157: SAM4E FPU and CMSIS DSP Library

Atmel 32-bit Microcontroller

Description

This application note helps users to get familiar with the Floating Point Unit (FPU) on SAM4E and the ARM® Cortex® Microcontroller Software Interface Standard (CMSIS) DSP library. It also introduces how to use the FPU and CMSIS DSP library in AS6, IAR™ and GCC toolchains. At last it will introduce a FFT example on SAM4E which use the CMSIS DSP library.

Features

- Floating-point introduction
- SAM4E Floating Point Unit (FPU) introduction
- CMSIS DSP_Lib introduction and usage
- SAM4E FFT example introduction

Table of Contents

1. Floating-point Introduction

1.1 Floating-Point Arithmetic

The term floating point refers to the fact that their radix point (decimal point, or, more commonly in computers, binary point) can "float"; that is, it can be placed anywhere relative to the significant digits of the number. Floating-point representation is similar in concept to scientific notation. The typical form is:

$$\textit{Significant digits} \times \textit{base}^{\textit{exponent}}$$

It consists of:

- A signed (meaning positive or negative) digit string of a given length in a given base (or radix) which is referred to as the significand, coefficient or, less often, the mantissa. The length of the significand determines the precision to which numbers can be represented
- A signed integer exponent also referred to as the characteristic or scale, which modifies the magnitude of the number

Floating-point representations are easier to use than fixed-point representations, because they can handle a wider dynamic range and do not require programmers to specify the number of digits after the radix point.

The C language offers the float and the double types for floating-point operations. On an FPUless processor, all these operations are done by software through the C compiler library and are not visible to the programmer, but the performances are very low. On a processor having an FPU, the operations can be done by hardware in a single cycle for most of the instructions. The C compiler does not use its own floating-point library but directly generates FPU native instructions.

1.2 IEEE Standard for Floating-Point Arithmetic (IEEE 754)

The IEEE® Standard for Floating-Point Arithmetic (IEEE 754) is a technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). Many hardware floating point units use the IEEE 754 standard. The current version is IEEE 754-2008 which is published in August 2008.

The standard defines:

- Arithmetic formats: sets of binary and decimal floating-point data, which consist of finite numbers (including signed zeros and subnormal numbers), infinities, and special "not a number" values (NaNs)
- Interchange formats: encodings (bit strings) that may be used to exchange floating-point data in an efficient and compact form
- Rounding rules: properties to be satisfied when rounding numbers during arithmetic and conversions
- operations: arithmetic and other operations on arithmetic formats
- Exception handling: indications of exceptional conditions (such as division by zero, overflow, etc.)

This standard provides a discipline for performing floating-point computation that yields results independent of whether the processing is done in hardware, software, or a combination of the two.

2. SAM4E FPU Introduction

2.1 SAM4E FPU

The SAM4E includes a Cortex-M4F FPU which implements the single precision variant of the ARMv7-M Floating-Point Extension (FPv4-SP).

The FPU provides floating-point computation functionality that is compliant with the ANSI/IEEE Std 754-2008, IEEE Standard for Binary Floating-Point Arithmetic, referred to as the IEEE 754 standard.

The FPU contains 32 single-precision extension registers, which can also be accessed as 16 doubleword registers for load, store, and move operations.

The FPU fully supports single-precision add, subtract, multiply, divide, multiply and accumulate, and square root operations. It also provides conversions between fixed-point and floating-point data formats, and floating-point constant instructions. Table 2-1 shows the floating-point instructions.These instructions are only available if the FPU is included, and enabled, in the system.

Table 2-1. Floating-point Instructions

Mnemonic	Description
VABS	Floating-point Absolute
VADD	Floating-point Add
VCMP	Compare two floating-point registers, or one floating-point register and zero
VCMPE	Compare two floating-point registers, or one floating-point register and zero with Invalid Operation check
VCVT	Convert between floating-point and integer
VCVT	Convert between floating-point and fixed point
VCVTR	Convert between floating-point and integer with rounding
VCVTB	Converts half-precision value to single-precision
VCVTT	Converts single-precision register to half-precision
VDIV	Floating-point Divide
VFMA	Floating-point Fused Multiply Accumulate
VFNMA	Floating-point Fused Negate Multiply Accumulate
VFMS	Floating-point Fused Multiply Subtract
VFNMS	Floating-point Fused Negate Multiply Subtract
VLDM	Load Multiple extension registers
VLDR	Loads an extension register from memory
VLMA	Floating-point Multiply Accumulate
VLMS	Floating-point Multiply Subtract
VMOV	Floating-point Move Immediate
VMOV	Floating-point Move Register
VMOV	Copy ARM core register to single precision
VMOV	Copy 2 ARM core registers to 2 single precision
VMOV	Copies between ARM core register to scalar
VMOV	Copies between Scalar to ARM core register
VMRS	Move to ARM core register from floating-point System Register
VMSR	Move to floating-point System Register from ARM Core register

VMUL	Multiply floating-point
VNEG	Floating-point negate
VNMLA	Floating-point multiply and add
VNMLS	Floating-point multiply and subtract
VNMUL	Floating-point multiply
VPOP	Pop extension registers
VPUSH	Push extension registers
VSQRT	Floating-point square root
VSTM	Store Multiple extension registers
VSTR	Stores an extension register to memory
VSUB	Floating-point Subtract

2.2 Using FPU

2.2.1 Overview

The SAM4E implement the FPU system registers as in Table 2-2:

Table 2-2. Floating Point Unit (FPU) Register Mapping

Offset	Register	Name	Access	Reset
0xE000ED88	Coprocessor Access Control Register	CPACR	Read-write	0x00000000
0xE000EF34	Floating-point Context Control Register	FPCCR	Read-write	0xC0000000
0xE000EF38	Floating-point Context Address Register	FPCAR	Read-write	–
–	Floating-point Status Control Register	FPSCR	Read-write	–
0xE000E01C	Floating-point Default Status Control Register	FPDSCR	Read-write	0x00000000

As we can see, there is an extra FPDSCR register for hold the default value for FPSCR.

2.2.2 Initialization

The FPU is disabled from reset. It must be enabled before any floating-point instructions can be used. We can enable and disable the FPU by setting the CPACR system register. The reference assembly code sequence for enabling the FPU can be as below:

FPU enable codes

```
; CPACR is located at address 0xE000ED88
LDR.W R0, =0xE000ED88
; Read CPACR
LDR R1, [R0]
; Set bits 20-23 to enable CP10 and CP11 coprocessors
ORR R1, R1, #(0xF << 20)
; Write back the modified value to the CPACR
STR R1, [R0];
; Wait for store to complete
DSB
; Reset pipeline now the FPU is enabled
ISB
```

In IAR toolchain, it implement above code in its own startup code.

In AS6 and GCC toochain, we implement above code in “fpu.h” file in “sam/utils/fpu” folder which includes APIs “fpu_enable”, “fpu_disable” and “fpu_is_enabled”. We call the “fpu_enable” API to open the FPU in “startup_sam4e.c” file. The “fpu.h” file can see in below:

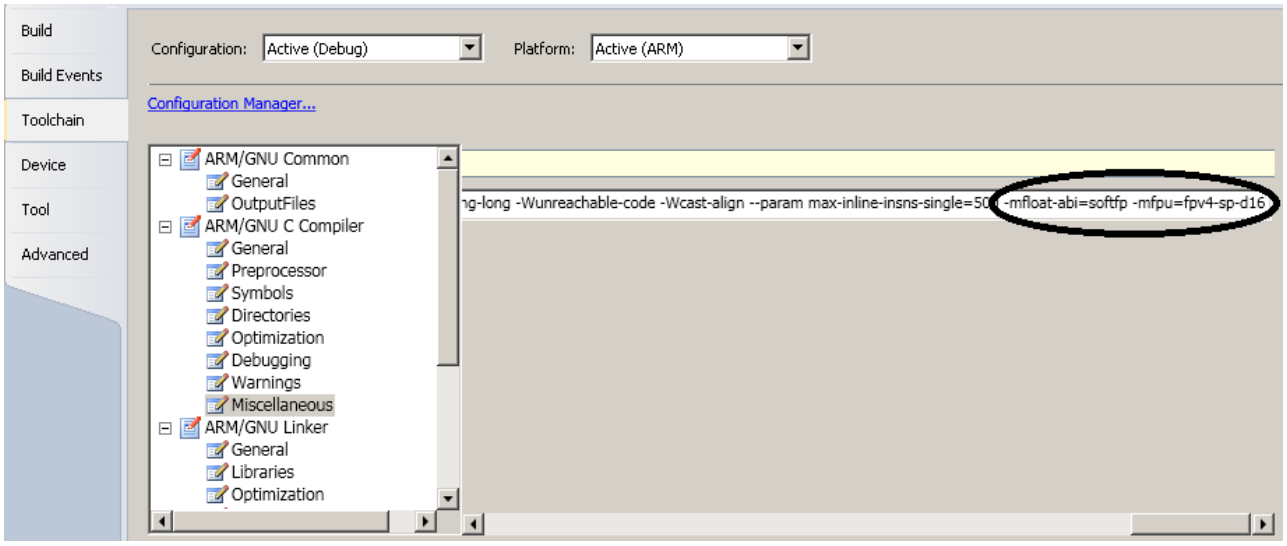
fpu.h

```
#include <compiler.h>

/** Address for ARM CPACR */
#define ADDR_CPACR 0xE000ED88

/** CPACR Register */
#define REG_CPACR  (*((volatile uint32_t *)ADDR_CPACR))

/**
 * Enable FPU
 */
__always_inline static void fpu_enable(void)
{
	irqflags_t flags;
	flags = cpu_irq_save();
	REG_CPACR |=  (0xFu << 20);
	__DSB();
	__ISB();
	cpu_irq_restore(flags);
}

/**
 * Disable FPU
 */
__always_inline static void fpu_disable(void)
{
	irqflags_t flags;
	flags = cpu_irq_save();
	REG_CPACR &= ~(0xFu << 20);
	__DSB();
	__ISB();
	cpu_irq_restore(flags);
}

/**
 * Check if FPU is enabled
 */
__always_inline static bool fpu_is_enabled(void)
{
	return (REG_CPACR & (0xFu << 20));
}
```

2.2.3 AS6 Compiling

When we use the Atmel® Studio 6, we must input the compiler and assembler flag '-mfpu=fpv4-sp-d16 -mfloat-abi=softfp' to generate hardware FPU instruction code with soft-float calling conventions. If there are libraries linked, the linker flag needs modification too. Note that the '-mfloat-abi' flag should be same for the whole project for it’s not link-compatible. If there are libraries with FPU related, we should use the same '-mfloat-abi' flag as the libraries or we should recompile the libraries with the flag we want. The flags for compiler can be added as shown in Figure 2-1. The assembler and linker flags can be added in the same way.

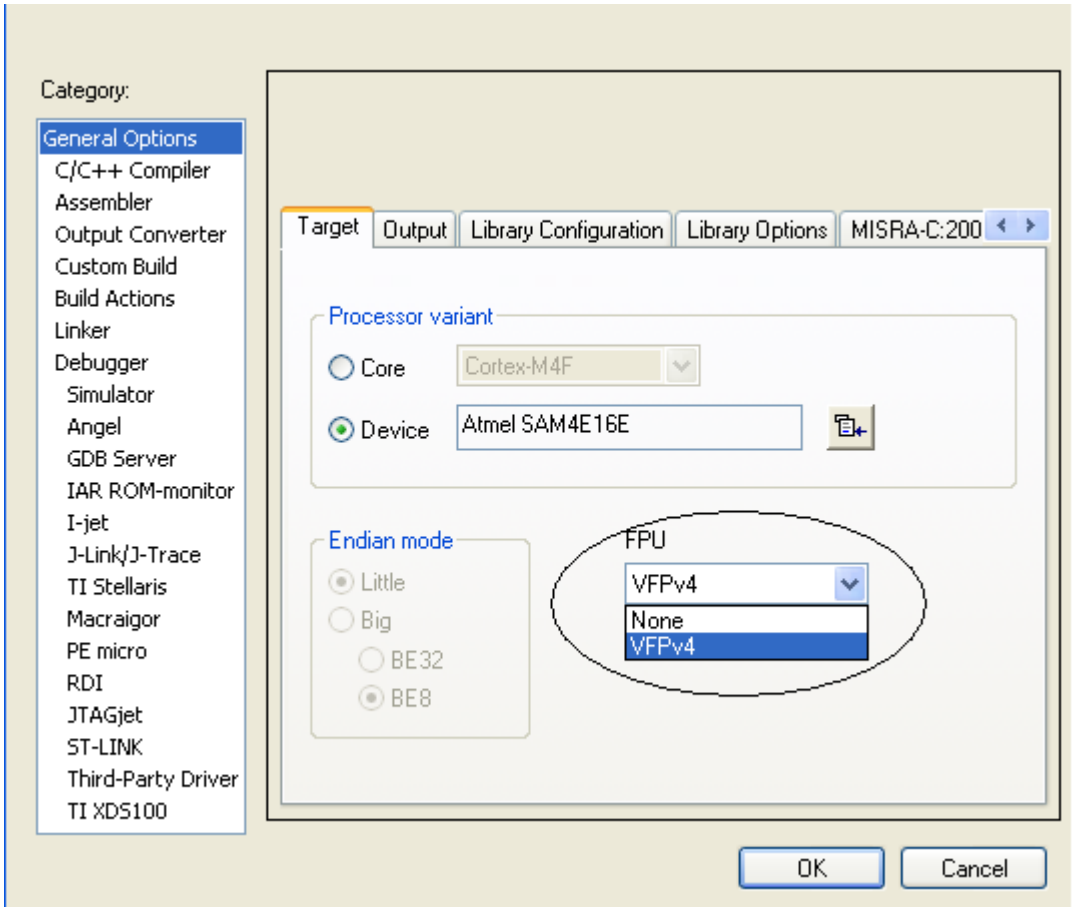
Figure 2-1. AS6 setting for FPU



2.2.4 IAR Compiling

When we use the IAR IDE, the SAM4E FPU support can be set as shown in Figure 2-2:

Figure 2-2. IAR setting for FPU



If we select 'VFPv4', the EWARM will enable FPU on startup and use hardware FPU instruction code; and if we select 'None', the EWARM will not enable FPU on startup and use normal instruction and software library.

2.2.5 GCC Compiling

When we use the GCC toolchain, we must modify makefile to add FPU support. The compiler flag '-mfpu=fpv4-sp-d16 -mfloat-abi=softfp' must be added to generate hardware FPU instruction code. If there are libraries with FPU related, we should use the same '-mfloat-abi' flag as the libraries or we should recompile the libraries with the flag we want. You can find in makefile such as:

```
# Extra flags to use when assembling.
ASFLAGS =  \
       -mfloat-abi=softfp \
       -mfpu=fpv4-sp-d16

# Extra flags to use when compiling.
CFLAGS =  \
       -mfloat-abi=softfp \
       -mfpu=fpv4-sp-d16

# Extra flags to use when linking.
LDFLAGS =  \
       -mfloat-abi=softfp \
       -mfpu=fpv4-sp-d16
```

3. CMSIS DSP Library Introduction

3.1 CMSIS Overview

The CMSIS is a vendor-independent hardware abstraction layer for the Cortex-M processor series. The CMSIS enables consistent and simple software interfaces to the processor and the peripherals, simplifying software re-use. It is not a huge software layer that introduces overhead and does not define standard peripherals. The silicon industry can therefore support the wide variations of Cortex-M processor-based devices with this common standard.

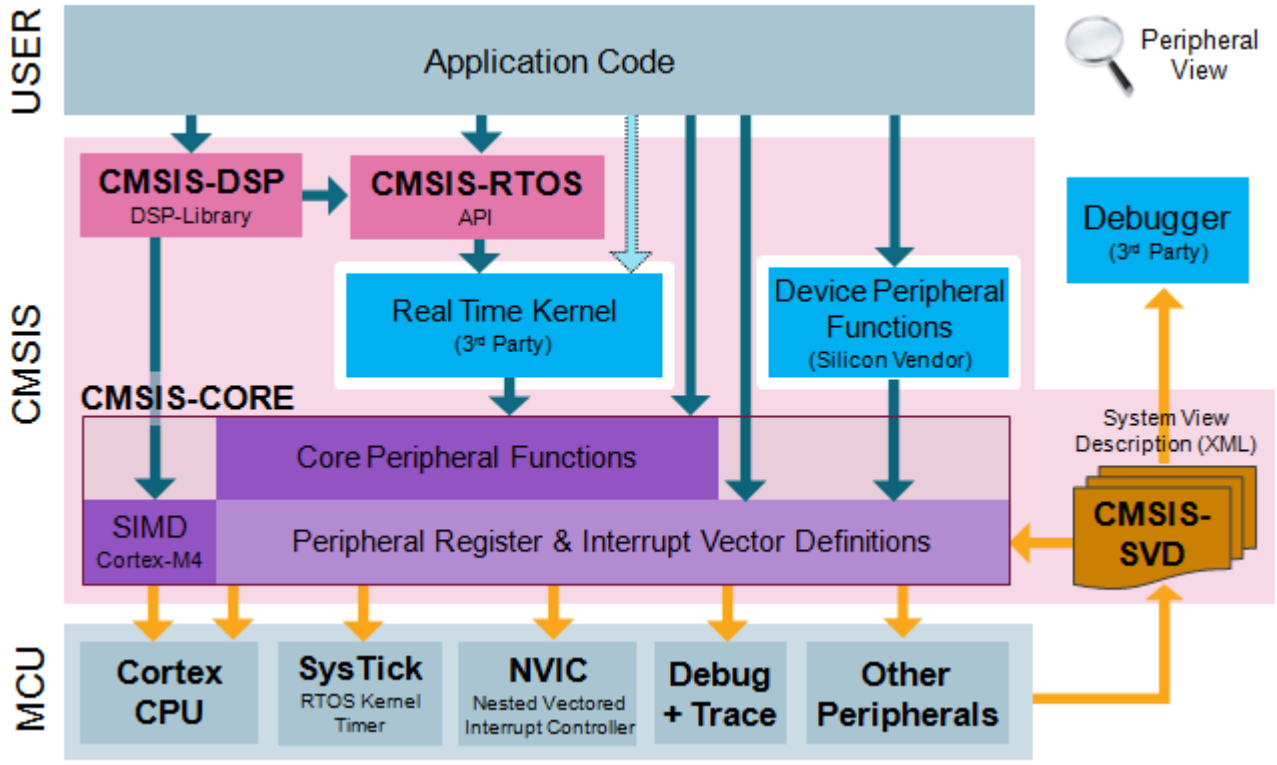
CMSIS is supported by all mainstream compilers (ARMCC, IAR, and GNU).

The CMSIS components are:

- CMSIS-CORE: API for the Cortex-M processor core and peripherals. It provides at standardized interface for Cortex-M0, Cortex-M3, Cortex-M4, SC000, and SC300. Included are also SIMD intrinsic functions for Cortex-M4 SIMD instructions.
- CMSIS-DSP: DSP Library Collection with over 60 Functions for various data types: fix-point (fractional q7, q15, q31) and single precision floating-point (32-bit). The library is available for Cortex-M0, Cortex-M3, and Cortex-M4. The Cortex-M4 implementation is optimized for the SIMD instruction set.
- CMSIS_RTOS API: Common API for Real-Time operating systems. It provides a standardized programming interface that is portable to many RTOS and enables therefore software templates, middleware, libraries, and other components that can work acrosss supported the RTOS systems.
- CMSIS-SVD: System View Description for Peripherals. Describes the peripherals of a device in an XML file and can be used to create peripheral awareness in debuggers or header files with peripheral register and interrupt definitions.

The whole CMSIS can be view as in Figure 3-1:

Figure 3-1. CMSIS overview



3.2 DSP Library Introduction

The CMSIS DSP software library is a suite of common signal processing functions for use on Cortex-M processor based devices.

The library is divided into a number of functions each covering a specific category:

- Basic math functions
- Fast math functions
- Complex math functions
- Filters
- Matrix functions
- Transforms
- Motor control functions
- Statistical functions
- Support functions
- Interpolation functions

The library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. The library ships with a number of examples which demonstrate how to use the library functions.

The source files for the library can been found in CMSIS folder “CMSIS\DSP_Lib\Source”.

The prebuilt versions of the libraries can also been found in the folder “CMSIS\Lib”. It now includes MDK-ARM, G++ and GCC toolchains.

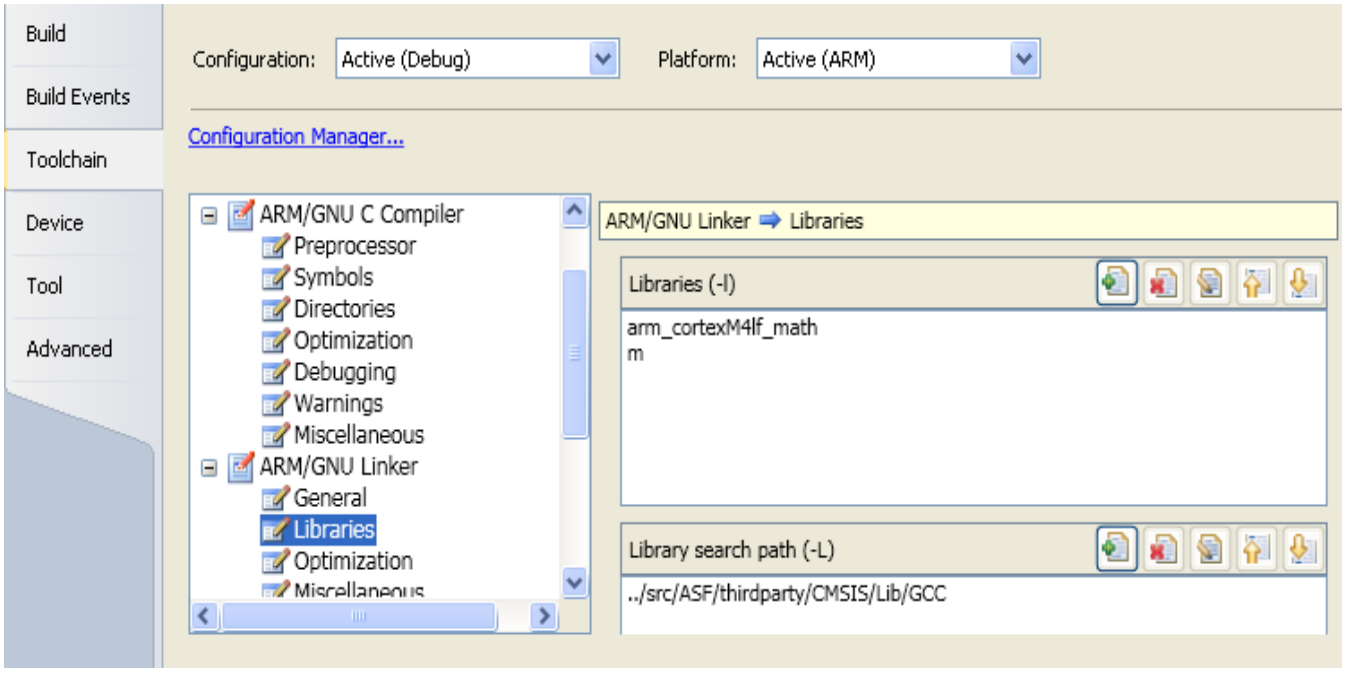
3.3 Using CMSIS DSP Library

The library functions are declared in the public file “arm_math.h” which is placed in the folder “CMSIS\Include”. To use the CMSIS DSP library we can simply include this file and link the appropriate library in the application and begin calling the library functions. The library supports single public header file “arm_math.h” for Cortex-M4/M3/M0 with little endian and big endian. Same header file will be used for floating point unit (FPU) variants. Define the appropriate pre processor MACRO ARM_MATH_CM4 or ARM_MATH_CM3 or ARM_MATH_CM0 which will be used in “arm_math.h” depending on the target processor in the application. Initialize macro “__FPU_PRESENT” with “1” in device header file such as “sam4e16e.h” when building on FPU supported Targets. Enable this macro for M4bf and M4lf libraries.

- When we use the Atmel Studio 6, we can add the library by:
  - Select the toolchain configuration in “.cproj” file
  - Select the “Libraries” in “ARM/GNU Linker”
  - Enter the path and name of the library files

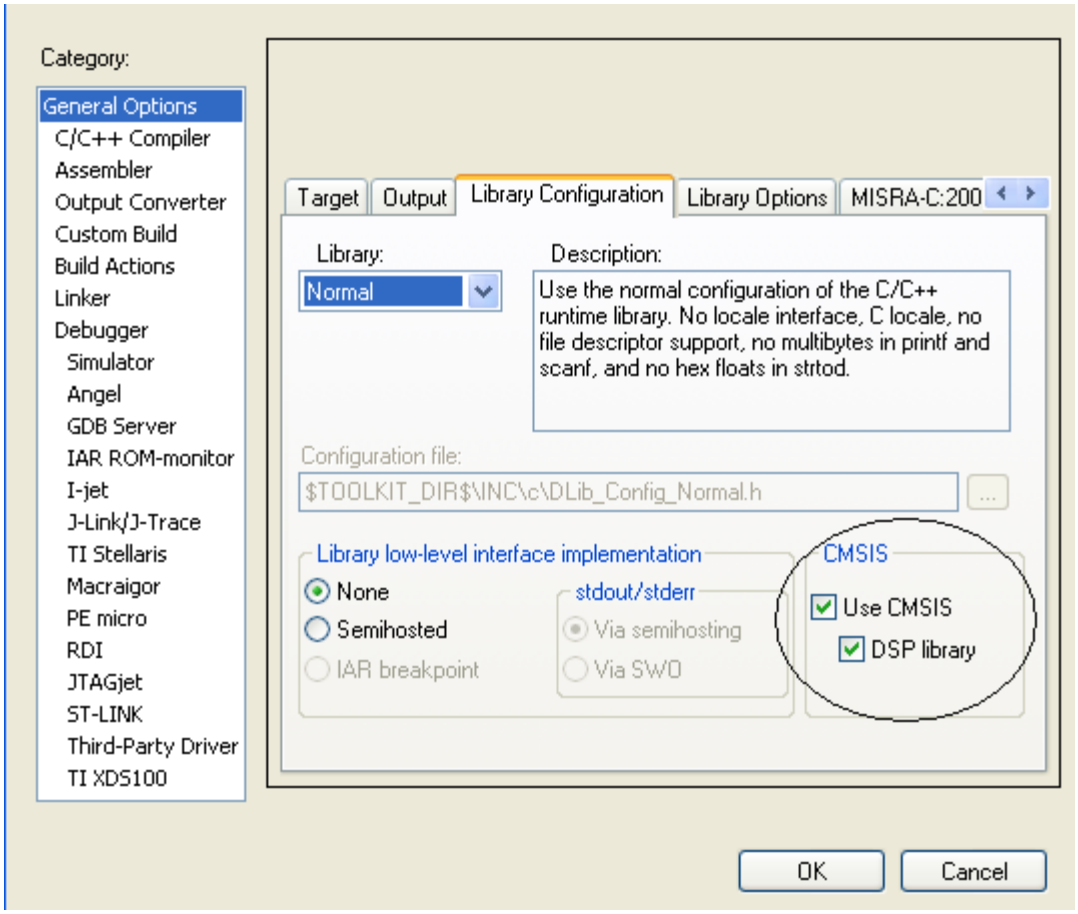
You can see in below after we add the CMSIS dsp library:

Figure 3-2. AS6 CMSIS DSP Library setting



- When we use the IAR IDE, we can add the CMSIS DSP library as in Figure 3-3

Figure 3-3. IAR CMSIS DSP Library setting



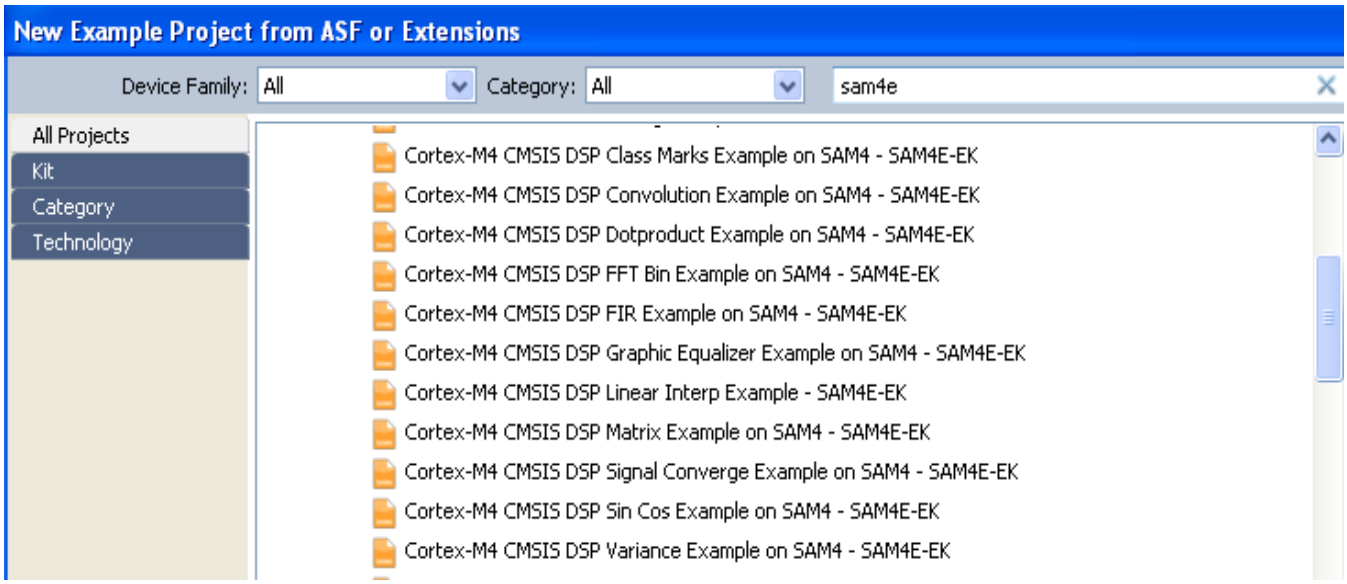
- When we use the GCC toolchain, we must modify makefile to include the CMSIS DSP library file "CMSIS\Lib\GCC\libarm_cortexM4lf_math.a" and math library "libm" like below:

```
# Additional search paths for libraries.
LIB_PATH =  \
       thirdparty/CMSIS/Lib/GCC

# List of libraries to use during linking.
LIBS =  \
       arm_cortexM4lf_math \
       m
```

4. DSP Application Example

The CMSIS offers some DSP application examples in folder “CMSIS\DSP_Lib\Examples” by Keil projects. We have added AS6, IAR and GCC version support for them on SAM4E in ASF which can be seen in Figure 4-1:

Figure 4-1. DSP Examples in AS6

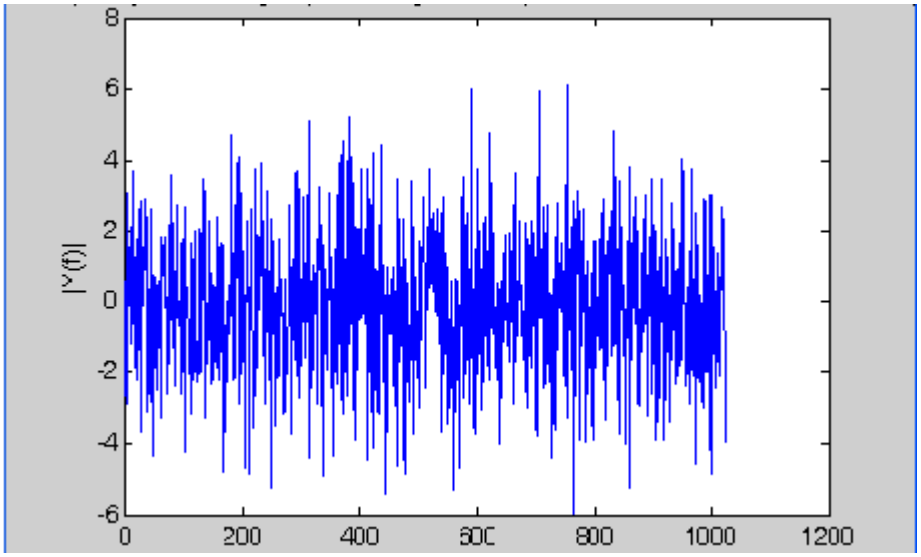


4.1 The FFT Example

The FFT is widely used in many applications. The “FFT Bin Example” is described here which shows how the CMSIS DSP library is used in a real application. The example uses the floating point library functions. They can also run in other SAM4 devices with proper library. But with the FPU in SAM4E, we can get the best performance.

The example uses an input 10kHz signal with uniformly distributed white noise. Figure 4-2 shows it in the time domain:

Figure 4-2. Input Signal: Time Domain



The example uses the radix-4 CFFT/CIFFT floating point APIs to get the result. At first it initializes the CFFT/CIFFT module:

```
status = arm_cfft_radix4_init_f32(&S, fftSize, ifftFlag, doBitReverse);
```

Then it uses the Complex FFT translate:

```
/* Process the data through the CFFT/CIFFT module */
      arm_cfft_radix4_f32(&S, testInput_f32_10khz);
```

Now the result is in the input buffer “testInput_f32_10khz”. Then it uses the Complex Magnitude function:

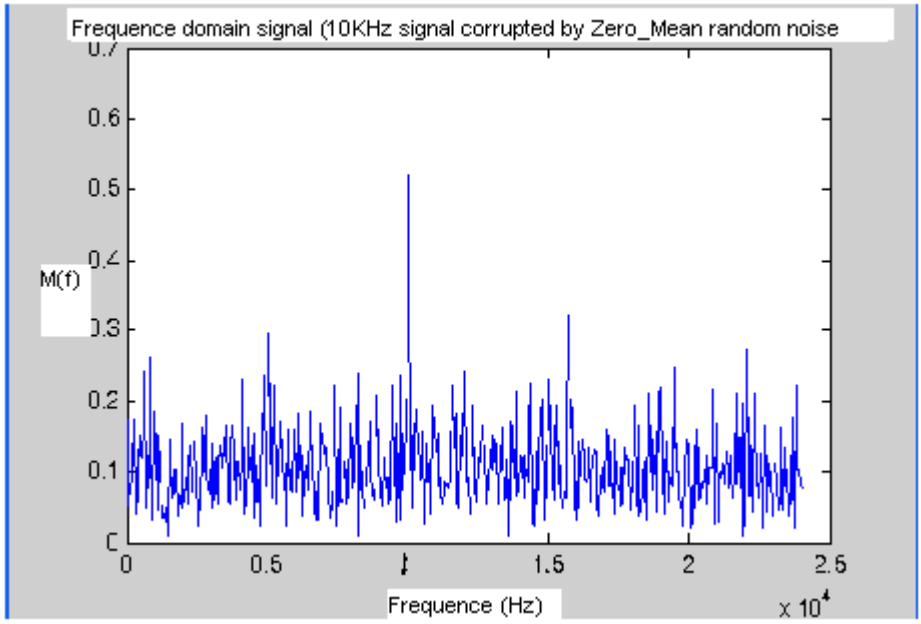
```
/* Process the data through the Complex Magnitude Module for
      calculating the magnitude at each bin */
      arm_cmplx_mag_f32(testInput_f32_10khz, testOutput, fftSize);
```

Now the result is in “testOutput” buffer. Then it calculates the maximum energy bin in the frequency domain of the input signal with the use of Maximum functions:

```
/* Calculates maxValue and returns corresponding BIN value */
      arm_max_f32(testOutput, fftSize, &maxValue, &testIndex);
```

We can see from the figure below that the bin with maximum energy corresponds to 10kHz signal.

Figure 4-3. Input Signal: Frequence Domain



4.2 Other Examples

For get more usage of the DSP library, we can check the other examples which are:

- Class Marks Example: Demonstrates the use of static initialization and the Maximum, Minimum, Mean, Standard Deviation, Variance and Matrix functions to calculate statistical values of marks obtained in a class;
- Convolution Example: Demonstrates the convolution theorem with the use of the Complex FFT, Complex-by-Complex Multiplication, and Support Functions.
- Dot Product Example: Demonstrates the use of the Multiply and Add functions to perform the dot product. The dot product of two vectors is obtained by multiplying corresponding elements and summing the products.

- FIR Lowpass Filter Example: Demonstrates how to configure an FIR filter and then pass data through it in a block-by-block fashion to removes high frequency signal components.
- Graphic Audio Equalizer Example: Demonstrates how a 5-band graphic equalizer can be constructed using the Biquad cascade functions. A graphic equalizer is used in audio applications to vary the tonal quality of the audio.
- Linear Interpolate Example: Demonstrates usage of linear interpolate modules and fast math modules. Method 1 uses fast math sine function to calculate sine values using cubic interpolation and method 2 uses linear interpolation function and results are compared to reference output. Result shows linear interpolation function can be used to get higher precision compared to fast math sin calculation.
- Matrix Example: Demonstrates the use of Matrix Transpose, Matrix Muliplication, and Matrix Inverse functions to apply least squares fitting to input data. Least squares fitting is the procedure for finding the best-fitting curve that minimizes the sum of the squares of the offsets (least square error) from a given set of data.
- Signal Convergence Example: Demonstrates the ability of an adaptive filter to "learn" the transfer function of a FIR lowpass filter using the Normalized LMS Filter, Finite Impulse Response (FIR) Filter, and Basic Math Functions.
- SineCosine Example: Demonstrates the Pythagorean trignometric identity with the use of Cosine, Sine, Vector Multiplication, and Vector Addition functions.
- Variance Example: Demonstrates the use of Basic Math and Support Functions to calculate the variance of an input sequence with N samples. Uniformly distributed white noise is taken as input.

5. Revision History

Doc. Rev.	Date	Comments
42144B	10/2013	The flag "-mfpu=vfpv4" used in GCC toolchain to support FPU was wrong for SAM4E device. The correct flag to be used is "-mfpu=fpv4-sp-d16".
42144A	06/2013	Initial document release

Atmel Corporation
1600 Technology Drive
San Jose, CA 95110
USA
Tel: (+1)(408) 441-0311
Fax: (+1)(408) 487-2600
www.atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Parkring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan G.K.
16F Shin-Osaki Kangyo Building
1-6-4 Osaki, Shinagawa-ku
Tokyo 141-0032
JAPAN
Tel: (+81)(3) 6417-0300
Fax: (+81)(3) 6417-0370