

# Practical DSP for the Cortex-M4

This chapter looks at the DSP capabilities of the Cortex-M4. How best to use the DSP intrinsic functions for custom algorithms. ARM also publishes a free DSP library, and this chapter will look at implementing an FFT as well as Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) filters. Techniques for creating real-time DSP applications within an RTOS framework are also presented. The Cortex-M4 may also be fitted with a single precision FPU; it is important to understand how it has been implemented alongside the main Cortex CPU.

## Introduction

The Cortex-M4 is the most powerful member of the Cortex-M processor family. While the Cortex-M4 has the same features and performance level of 1.25 DMIPS/MHz as the Cortex-M3, it has a number of architectural enhancements that help dramatically boost its math ability. The key enhancements over the Cortex-M3 are the addition of “single instruction multiple data” or SIMD instructions, an improved MAC unit for integer math, and the optional addition of a single precision FPU. These enhancements give the Cortex-M4 the ability to run DSP algorithms at high enough levels of performance to compete with dedicated 16-bit DSP processors.

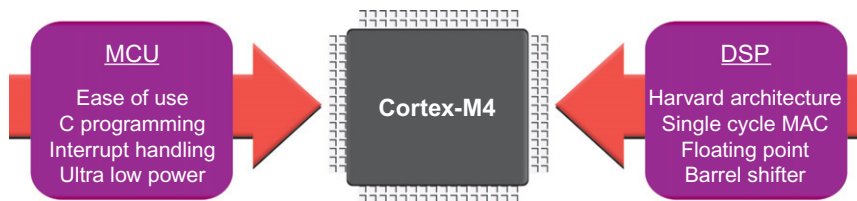


Figure 7.1

The Cortex-M4 extends the Cortex-M3 with the addition of DSP instructions and fast math capabilities. This creates a microcontroller capable of supporting real-time DSP algorithms and DSC.

## Cortex-M4 Hardware Floating Point Unit

One of the major features of the Cortex-M4 processor is the hardware FPU. The FPU supports single precision floating point arithmetic operations to the IEEE 7xx standard.

Initially, the FPU can be thought of as a coprocessor that is accessed by dedicated instructions to perform most floating point arithmetic operations in a few cycles.

Table 7.1: Floating Point Unit Performance

Operation	Cycle Count
Add/subtract	1
Divide	14
Multiply	1
MAC	3
Fused MAC	3
Square root	14

The FPU consists of a group of control and status registers and 31 single precision scalar registers. The scalar registers can also be viewed as 16 double word registers.

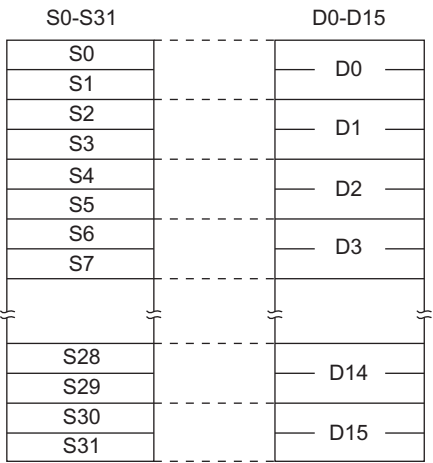


Figure 7.2

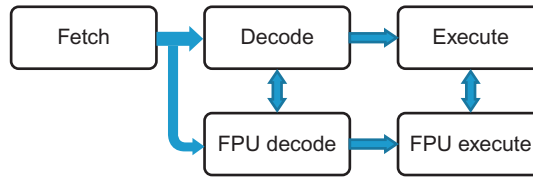
The FPU 32-bit scalar registers may also be viewed as 64-bit double word registers. This supports very efficient casting between C types.

While the FPU is designed for floating point operations, it is possible to load and store fixed point and integer values. It is also possible to convert between floating point and fixed point and integer values. This means C casting between floating point and integer values can be done in a cycle.

**FPU Integration**

While it is possible to consider the FPU as a coprocessor adjacent to the Cortex-M4 processor, this is not really true. The FPU is an integral part of the Cortex-M4

processor; the floating point instructions are executed within the FPU in a parallel pipeline to the Cortex-M4 processor instructions. While this increases the FPU performance, it is “invisible” to the application code and does not introduce any strange side effects.



**Figure 7.3**

The FPU is described as a coprocessor in the register documentation. In reality, it is very tightly coupled to the main instruction pipeline.

## FPU Registers

In addition to the scalar registers, the FPU has a block of control and status registers.

**Table 7.2: FPU Control Registers**

Register	Description
Coprocessor access control	Controls the privilege access level to the FPU
Floating point context control	Configures stacking and lazy stacking options
Floating point context address	Holds the address of the unpopulated FPU stack space
Floating point default status control	Holds the FPU condition codes and FPU configuration options
Floating point status control	Holds the default status control values

All of the FPU registers are memory mapped except the floating point status control register (FPSCR), which is a CPU register accessed by the MRS and MSR instructions. Access to this function is supported by a CMSIS core function:

```
uint32_t __get_FPSCR(void);
void __set_FPSCR (uint32_t fpscr).
```

The FPSCR register contains three groups of bits. The top 4 bits contain condition code flags N, Z, C, V that match the condition code flags in the xPSR. These flags are set and cleared in a similar manner by results of floating point operations. The next groups of bits contain configuration options for the FPU. These bits allow you to change the operation of the FPU from the IEEE 754 standard. Unless you have a strong reason to do this, it is recommended to leave them alone. The final group of bits are status flags for the FPU exceptions. If the FPU encounters an error during execution, an exception will be raised and

the matching status flag will be set. The exception line is permanently enabled in the FPU and just needs to be enabled in the NVIC to become active. When the exception is raised, you will need to interrogate these flags to work out the cause of the error. Before returning from the FPU exception, the status flags must be cleared. How this is done depends on the FPU exception stacking method.

### ***Enabling the FPU***

When the Cortex-M4 leaves the reset vector, the FPU is disabled. The FPU is enabled by setting the coprocessor 10 and 11 bits in the Co processor Access Control Register (CPACR). It is necessary to use the data barrier instruction to ensure that the write is made before the code continues. The instruction barrier command is also used to ensure that the pipeline is flushed before the code continues.

```
SCB->CPACR |= ((3UL << 10*2) | (3UL << 11*2)); // Set CP10 & CP11 Full Access
__DSB(); //Data barrier
__ISB(); //Instruction barrier
```

In order to write to the CPACR register, the processor must be in privileged mode. Once enabled, the FPU may be used in privileged and unprivileged modes.

### ***Exceptions and the FPU***

When the FPU is enabled, an extended stack frame will be pushed and an exception is raised. In addition to the standard stack frame, the Cortex-M4 also pushes the first 16 FPU scalar registers and the FPSCR. This extends the stack frame from 32 to 100 bytes. Clearly pushing this amount of data onto the stack, every interrupt would increase the interrupt latency significantly. To keep the 12 cycle interrupt latency, the Cortex-M4 uses a technique called lazy stacking. When an interrupt is raised, the normal stack frame is pushed onto the stack and the stack pointer is incremented to leave space for the FPU registers, but their values are not pushed onto the stack. This leaves a void space in the stack. The start address of this void space is automatically stored in the floating point context address register (FPCAR). If the interrupt routine uses floating point calculations, the FPU registers will be pushed into this space using the address stored in the FPCAR as a base address. The floating point context control register controls the stacking method used. Lazy stacking is enabled by default when the FPU is first enabled. The stacking method is controlled by the most significant 2 bits in the floating point context control register; these are the Automatic State Preservation Enable (ASPEN) and Lazy State Preservation Enable (LSPEN) bits.

Table 7.3: Lazy Stacking Options

LSPEN	ASPEN	Configuration
0	0	No automatic state preservation; only use when the interrupts do not use floating point
0	1	Lazy stacking disabled
1	0	Lazy stacking enabled
1	1	Invalid configuration

## Using the FPU

Once you have enabled the FPU, the compiler will start to use the FPU in place of software libraries. The exception is the square root instruction, which is part of the math.h library. If you have enabled the FPU, the ARM compiler provides an intrinsic instruction to use the FPU square root instruction.

```
float __sqrtf(float x);
```

**Note:** the intrinsic square root function differs from the ANSI `sqrt()` library function in that it takes and returns a float rather than a double.

## Exercise: Floating Point Unit

This exercise performs a few simple floating point calculations to test out the Cortex-M4 FPU.

**Open the project in c:\exercises\FPU.**

The code in the main loop is a mixture of math operations to exercise the FPU.

```
#include <math.h>
float a,b,c,d,e;
int f,g=100;
while(1){
    a = 10.1234;
    b = 100.2222;
    c = a*b;
    d = c-a;
    e = d+b;
    f = (int)a;
    f = f*g;
    a1 = (unsigned int) a;
    a = __sqrtf(e);
    //a = sqrt(e);
```

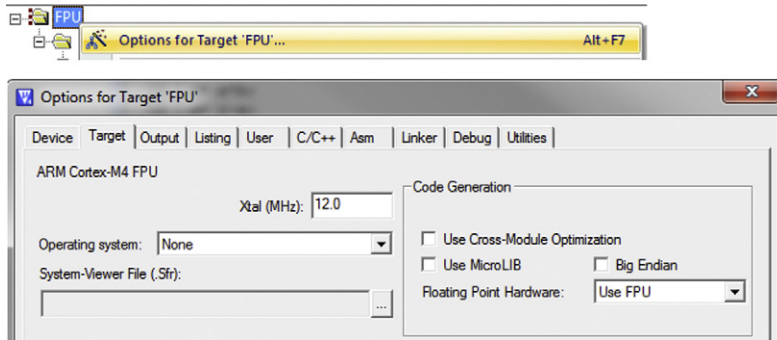
```

a = c/f;
e = a/0;
}
}

```

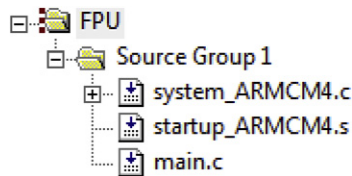
Before we can build this project, we need to make sure that the compiler will build code to use the FPU.

Open the Options for Target window and select the Target menu.



We can enable floating point support by selecting Use FPU in the Floating Point Hardware box. This will enable the necessary compiler options and load the correct simulator model.

**Close the Options for Target menu and return to the editor.**



In addition to our source code, the project includes the CMSIS startup and system files for the Cortex-M4.

**Now build the project and note the build size.**

```

Build target 'FPU'
compiling system_ARMCM4.c...
assembling startup_ARMCM4.s...
compiling main.c...
linking...
Program Size: Code=224 RO-data=224 RW-data=4 ZI-data=1028
"cortexM4.axf" - 0 Error(s), 0 Warning(s).

```

**Start the debugger.**

When the simulator runs the code to main, it will hit a breakpoint that has been preset in the system\_ARMCM4.c file.

```

61 void SystemInit (void)
62 {
63     #if (__FPU_USED == 1)
64         SCB->CPACR |= ((3UL << 10*2) | (3UL << 11*2) );
65     #endif
66
67     SystemCoreClock = __SYSTEM_CLOCK;
68
69 }

```

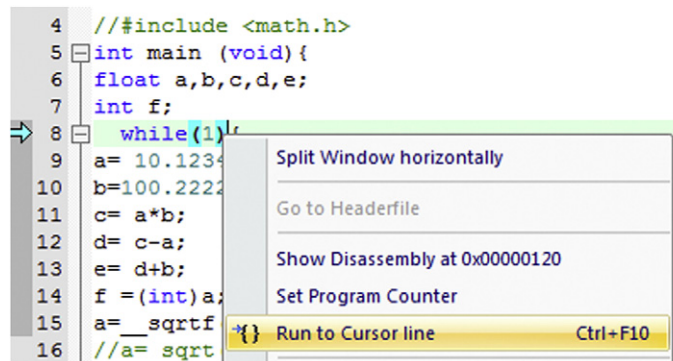
The standard microcontroller include file that will define the feature set of the Cortex processor including the availability of the FPU. If the FPU is present on the microcontroller, the SystemInit() function will make sure it is switched on before you reach your application's main function.

```

/* ===== */
/* ===== Processor and Core Peripheral Section ===== */
/* ===== */

/* —— Configuration of the Cortex M4 Processor and Core Peripherals —— */
#define __CM4_REV  0x0001  /*!< Core revision r0p1 */
#define __MPU_PRESENT  1  /*!< MPU present or not */
#define __NVIC_PRI0_BITS  3 /*!< Number of Bits used for Priority Levels */
#define __Vendor_SysTickConfig  0 /*!< Set to 1 if different SysTick Config is used */
#define __FPU_PRESENT  1  /*!< FPU present or not */

```

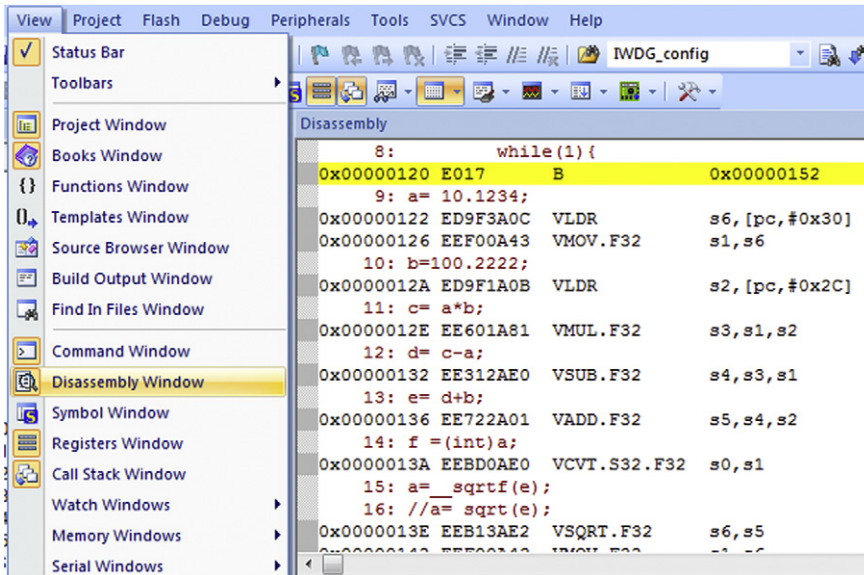
**Open the main.c module and run the code to the main while() loop.**


```

4  // #include <math.h>
5  int main (void){
6  float a,b,c,d,e;
7  int f;
8  while(1){
9  a= 10.1234;
10 b=100.2222;
11 c= a*b;
12 d= c-a;
13 e= d+b;
14 f= (int) a;
15 a= __sqrtf(f);
16 // a= sqrtf(f);

```

Now open the disassembly window.



This will show the C source code interleaved with the Cortex-M4 assembly instructions.

In the project window select the registers window.

FPU		FPU Reg		Float Format
S0	0x00000000	S0	0.000000	
S1	0x00000000	S1	0.000000	
S2	0x00000000	S2	0.000000	
S3	0x00000000	S3	0.000000	
S4	0x00000000	S4	0.000000	

The register window now shows the 31 scalar registers in their raw format and the IEEE 7xx format.

FPSCR		Internal	
N	0	Mode	Thread
Z	0	Privilege	Privileged
C	0	Stack	MSP
V	0	States	1905
AHP	0	Sec	0.00015875
DN	0		
FZ	0		
RM	RN		
IDC	0		
IXC	0		
UFC	0		
OFC	0		
DZC	0		
IOC	0		



The contents of the FPSCR are also shown. In this exercise, we will also be using the states (cycle count) value also shown in the register window.

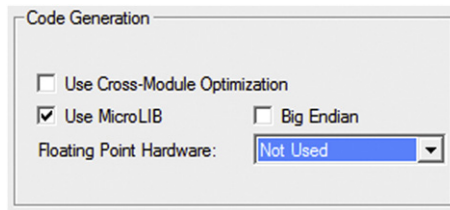
**Highlight the assembly window and step through each operation noting the cycle time for each calculation.**

**Now quit the debugger and change the code to use software floating point libraries.**

```
a1=(unsigned int) a;
//a=__sqrtf(e);
a= sqrt(e);
```

Here we need to include the math.h library, comment out the \_\_sqrtf() intrinsic and replace it with the ANSI C sqrt() function.

In the Options for Target\Target settings, we also need to remove the FPU support.



**Rebuild the code and compare the build size to the original version.**

```
Build target 'FPU'
compiling main.c...
linking...
Program Size: Code=1440 RO-data=224 RW-data=8 ZI-data=1024
"cortexM4.axf" - 0 Error(s), 0 Warning(s).
```

**Now restart the debugger run and run the code to main.**

In the disassembly window step through the code and compare the number of cycles used for each operation to the number of cycles used by the FPU.

By the end of this exercise, you can clearly see not only the vast performance improvement provided by the FPU but also its impact on project code size. The only downside is the additional cost to use a microcontroller fitted with the FPU and the additional power consumption when it is running.

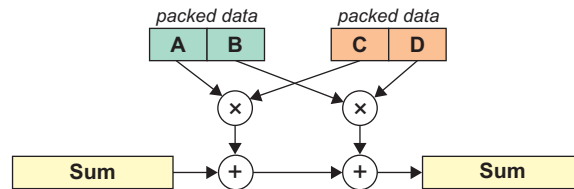
## ***Cortex-M4 DSP and SIMD Instructions***

The Thumb-2 instruction set has a number of instructions that are useful in DSP algorithms.

**Table 7.4: Thumb-2 DSP Instructions**

Instruction	Description
CLZ	Count leading zeros
REV, REV16, REVSH, and RBIT	Reverse instructions
BFI	Bit field insert
BFC	Bit field clear
UDIV and SDIV	Hardware divide
SXT and UXT	Sign and zero extend

The Cortex-M4 instruction set includes a new group of instructions that can perform multiple arithmetic calculations in a single cycle. The SIMD instructions allow multiple parallel arithmetic operations on 8 or 16 bit data quantites. The 8 and 16 bit values must be packet into two 32 bit words. So, for example, you can perform two 16-bit multiplies and a 32- or 64-bit accumulate or a quad 8-bit addition in one processor cycle. Since many DSP algorithms work on a pipeline of data, the SIMD instructions can be used to dramatically boost performance.



**Figure 7.4**

The SIMD instructions support multiple arithmetic operations in a single cycle. The operand data must be packed into 32-bit words.

The SIMD instructions have an additional field in the xPSR register. The “greater than or equal” (GE) field contains 4 bits, which correspond to the 4 bytes in the SIMD instruction result operand. If the result operand byte is GE to zero then the matching GE flag will be set.



**Figure 7.5**

The Cortex-M4 xPSR register has an additional GE field. Each of the four GE bits are updated when an SIMD instruction is executed.

The SIMD instructions can be considered as three distinct groups: add and subtract operations, multiply operations, and supporting instructions. The add and subtract operations can be performed on 8- or 16-bit signed and unsigned quantities. A signed and unsigned halving instruction is also provided; this instruction adds or subtracts the 8- or 16-bit quantities and then halves the result as shown in the following.

**Table 7.5: SIMD Add Halving and Subtract Halving Instructions**

Instruction	Description	Operation
UHSUB16	Unsigned halving 16-bit subtract	$\text{Res}[15:0] = (\text{Op1}[15:0] - \text{Op2}[15:0])/2$ $\text{Res}[31:16] = (\text{Op1}[31:16] - \text{Op2}[31:16])/2$
UHADD16	Unsigned halving 16-bit add	$\text{Res}[15:0] = (\text{Op1}[15:0] + \text{Op2}[15:0])/2$ $\text{Res}[31:16] = (\text{Op1}[31:16] + \text{Op2}[31:16])/2$

The SIMD instructions also include an add and subtract with exchange (ASX) and a subtract and add with exchange (SAX). These instructions perform an add and subtract on the two halfwords and store the results in the upper and lower halfwords of the destination register.

**Table 7.6: SIMD Add Exchange and Subtract Exchange Instructions**

Instruction	Description	Operation
USAX	Unsigned 16-bit subtract and add with exchange	$\text{Res}[15:0] = \text{Op1}[15:0] + \text{Op2}[31:16]$ $\text{Res}[31:16] = \text{Op1}[31:16] - \text{Op2}[15:0]$
UASX	Unsigned 16-bit add and subtract with exchange	$\text{Res}[15:0] = \text{Op1}[15:0] + \text{Op2}[31:16]$ $\text{Res}[31:16] = \text{Op1}[31:16] - \text{Op2}[15:0]$

A further group of instructions combine these two operations in a subtract and add (or add and subtract) with exchange halving instruction. This gives quite few possible permutations. A summary of the add and subtract SIMD instructions is shown below.

**Table 7.7: Permutations of the SIMD Add, Subtract, Halving, and Saturating Instructions**

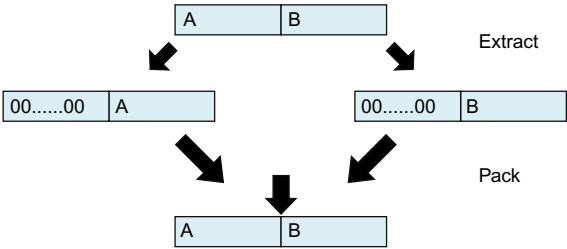
Prefix Instruction	S Signed	Q Signed Saturating	SH Signed Halving	U Unsigned	UQ Unsigned Saturating	UH Unsigned Halving
ADD8	SADD8	QADD8	SHADD8	UADD8	UQADD8	UHADD8
SUB8	SSUB8	QSUB8	SHSUB8	USUB8	UQSUB8	UHSUB8
ADD16	SADD16	QADD16	SHADD16	UADD16	UQADD16	UHADD16
SUB16	SSUB16	QSUB16	SHSUB16	USUB16	UQSUB16	UHSUB16
ASX	SASX	QASX	SHASX	UASX	UQASX	UHASX
SAX	SSAX	QSAX	SHSAX	USAX	UQSAX	UHSAX

The SIMD instructions also include a group of multiply instructions that operate on packed 16-bit signed values. Like the add and subtract instructions, the multiply instructions also support saturated values. As well as multiply and multiply accumulate the SIMD multiply instructions support multiply subtract and multiply add as shown in the following.

**Table 7.8: SIMD Multiply Instructions**

Instruction	Description	Operation
SMLAD	Q setting dual 16-bit signed multiply with single 32-bit accumulator	$X = X + (A \times B) + (C \times D)$
SMLALD	Dual 16-bit signed multiply with single 64-bit accumulator	$X = X + (A \times B) + (C \times D)$
SMLSD	Q setting dual 16-bit signed multiply subtract with 32-bit accumulator	$X = X + (A \times B) - (B \times C)$
SMLSXD	Q setting dual 16-bit signed multiply subtract with 64-bit accumulator	$X = X + (A \times B) - (B \times C)$
SMUAD	Q setting sum of dual 16-bit signed multiply	$X = (A \times B) + (C \times D)$
SMUSD	Dual 16-bit signed multiply returning difference	$X = (A \times B) - (C \times D)$

To make the SIMD instructions more efficient, a group of supporting pack and unpack instructions have also been added. The pack/unpack instructions can be used to extract 8- and 16-bit values from a register and move them to a destination register. The unused bits in the 32-bit word can be set to zero (unsigned) or one (signed). The pack instructions can also take two 16-bit quantities and load them into the upper and lower halfwords of a destination register.



**Figure 7.6**

The SIMD instruction group includes support instructions to pack 32-bit words with 8- and 16-bit quantities.

**Table 7.9: SIMD Supporting Instructions**

Mnemonic	Description
PKH	Pack halfword
SXTAB	Extend 8-bit signed value to 32 bits and add
SXTAB16	Dual extend 8-bit signed value to 16 bits and add
SXTAH	Extend 16-bit signed value to 32 bits and add
SXTB	Sign extend a byte
SXTB16	Dual extend 8-bit signed value to 16 bits and add
SXTH	Sign extend a halfword
UXTAB	Extend 8-bit signed value to 32 bits and add
UXTAB16	Dual extend 8 to 16 bits and add
UXTAH	Extend a 16-bit value and add
UXTB	Zero extend a byte
UXTB16	Dual zero extend 8 to 16 bits and add
UXTH	Zero extend a halfword

When an SIMD instruction is executed, it will set or clear the xPSR GE bits depending on the values in the resulting bytes or halfwords. An additional select (SEL) instruction is provided to access these bits. The SEL instruction is used to select bytes or halfwords from two input operands depending on the condition of the GE flags.

**Table 7.10: xPSR “GE” Bit Field Results**

GE Bit[3:0]	GE Bit = 1	GE Bit = 0
0	Res[7:0] = OP1[7:0]	Res[7:0] = OP2[7:0]
1	Res[15:8] = OP1[15:8]	Res[15:8] = OP2[15:8]
2	Res[23:16] = OP1[23:16]	Res[23:16] = OP2[23:16]
3	Res[31:24] = OP1[31:24]	Res[31:24] = OP2[31:24]

### Exercise: SIMD Instructions

In this exercise, we will have a first look at using the Cortex-M4 SIMD instructions. In this exercise, we will simply multiply and accumulate two 16-bit arrays first using an SIMD instruction and then using the standard add instruction.

**First open the CMSIS core documentation and the SIMD signed multiply accumulate intrinsic `__SMLAD`.**

The screenshot shows the CMSIS-CORE documentation for the `__SMLAD` intrinsic function. The left sidebar lists the navigation menu, with 'Intrinsic Functions for SIMD Instructions' selected. The main content area displays the function signature, description, parameters, returns, and operation.

```

res[7:0]  = val1[7:0]  - val2[7:0]  >> 1
res[15:8] = val1[15:8] - val2[15:8] >> 1
res[23:16] = val1[23:16] - val2[23:16] >> 1
res[31:24] = val1[31:24] - val2[31:24] >> 1
  
```

**uint32\_t \_\_SMLAD ( uint32\_t val1,  
uint32\_t val2,  
uint32\_t val3  
)**

This function enables you to perform two signed 16-bit multiplications, adding both results to a 32-bit accumulate operand. The Q bit is set if the addition overflows. Overflow cannot occur during the multiplications.

**Parameters:**  
**val1** first 16-bit operands for each multiplication.  
**val2** second 16-bit operands for each multiplication.  
**val3** accumulate value.

**Returns:**  
 the product of each multiplication added to the accumulate value, as a 32-bit integer.

**Operation:**

```

p1 = val1[15:0] * val2[15:0]
p2 = val1[31:16] * val2[31:16]
res[31:0] = p1 + p2 + val3[31:0]
  
```

**Open the project in `c:\exercises\CMSIS core SIMD`.**

The application code defines two sets of arrays as a union of 16- and 32-bit quantities.

```

union _test{
  int16_t Arry_halfword[100];
  int32_t Arry_word[50];
};
  
```

The code first initializes the arrays with the values 0–100.

```
for(n=0;n<100;n++){
    op1.Array_halfword[n] = op2.Array_halfword[n] = n; }
```

Then multiply accumulates first using the SIMD instruction, then the standard multiply accumulate.

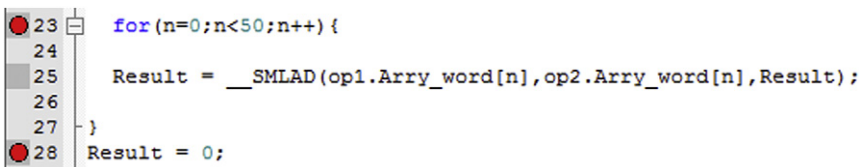
```
for(n=0;n<50;n++){
    Result=__SMLAD(op1.Array_word[n],op2.Array_word[n],Result);}
```

The result is then reset and the calculation is repeated without using the SIMD instruction.

```
Result=0;
for(n=0;n<100;n++){
    Result=Result+(op1.Array_halfword[n]*op2.Array_halfword[n]);}
```

**Build the code and start the debugger.**

**Set a breakpoint at lines 23 and 28.**



**Run to the first breakpoint and make a note of the cycle count.**

A screenshot of the 'Internal' window in a debugger. It displays the following data:

Mode	Thread
Privilege	Privileged
Stack	MSP
States	4066
Sec	0.00033883

**Run the code until it hits the second breakpoint. See how many cycles have been used to execute the SIMD instruction.**

A screenshot of the 'Internal' window in a debugger, showing the state after the second breakpoint. The data is as follows:

Mode	Thread
Privilege	Privileged
Stack	MSP
States	5172
Sec	0.00043100

Cycles used = 5172–4066 = 1106

Set a breakpoint at the final while loop.

```

30 for(n=0;n<100;n++){
31   Result = Result + (op1.Array_halfword[n] * op2.Array_halfword[n]);
32 }
33 while(1){

```

Run the code and see how many cycles are used to perform the calculation without using the SIMD instruction.

Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	7483
Sec	0.00062358

Cycles used =  $7483 - 5172 = 2311$

Compare the number of cycles used to perform the same calculation without using the SIMD instructions.

As expected, the SIMD instructions are much more efficient when performing calculations on large data sets.

The primary use for the SIMD instructions is to optimize the performance of DSP algorithms. In the next exercise, we will look at various techniques in addition to the SIMD instructions that can be used to boost the efficiency of a given algorithm.

### Exercise: Optimizing DSP Algorithms

In this exercise, we will look at optimizing an FIR filter. This is a classic algorithm that is widely used in DSP applications.

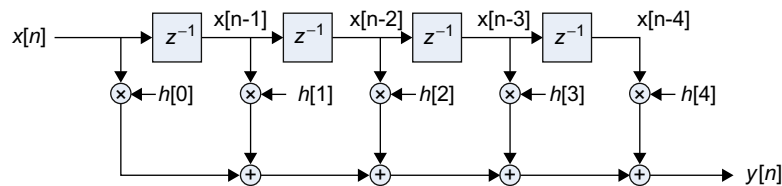


Figure 7.7

An FIR filter is an averaging filter with its characteristics defined by a series of coefficients applied to each sample in a series of “taps.”

The FIR filter is an averaging filter that consists of a number of “taps.” Each tap has a coefficient and as the filter runs each sample is multiplied against the coefficient in the first tap and then shifted to the next tap to be multiplied against its coefficient when the next sample arrives. The output of each tap is summed to give the filter output. Or to put it mathematically,

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]$$

**Open the project in c:\exercises\FIR\_optimization.**

**Build the project and start the debugger.**

The main function consists of four FIR functions that introduce different optimizations to the standard FIR algorithm.

```
int main(void){
    fir (data_in,data_out, coeff,&index, FILTERLEN,BLOCKSIZE);
    fir_block (data_in,data_out, coeff,&index, FILTERLEN,BLOCKSIZE);
    fir_unrolling (data_in,data_out, coeff,&index, FILTERLEN,BLOCKSIZE);
    fir_SIMD (data_in,data_out, coeff,&index, FILTERLEN,BLOCKSIZE);
    fir_SuperUnrolling (data_in,data_out, coeff,&index, FILTERLEN,BLOCKSIZE);
    while(1);
}
```

**Step into the first function and examine the code.**

The filter function is implemented in C as shown below. This is a standard implementation of an FIR filter written purely in C.

```
void fir(q31_t *in, q31_t *out, q31_t *coeffs, int *stateIndexPtr,
        int filtLen, int blockSize)
{
    int sample;
    int k;
    q31_t sum;
    int stateIndex = *stateIndexPtr;
    for(sample=0; sample<blockSize; sample++)
    {
        state[stateIndex++] = in[sample];
        sum = 0;
        for(k=0; k<filtLen; k++)
        {
```



```

    sum += coeffs[k] * state[stateIndex];
    stateIndex--;
    if (stateIndex < 0)
    {
        stateIndex = filtLen-1;
    }
}
out[sample] = sum;
}
*stateIndexPtr = stateIndex;
}

```

While this compiles and runs fine, it does not take full advantage of the Cortex-M4 DSP enhancements. To get the best out of the Cortex-M4, we need to optimize this algorithm, particularly the inner loop.

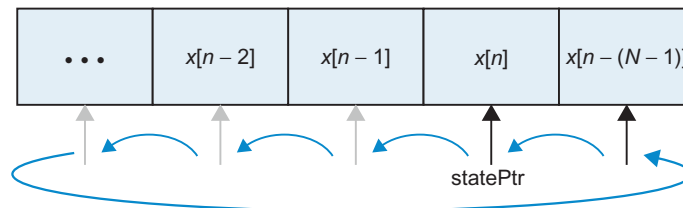
The inner loop performs the FIR multiply and accumulate for each tap.

```

for(k=0;k<filtLen;k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex--;
    if (stateIndex < 0)
    {
        stateIndex = filtLen-1;
    }
}

```

The inner loop processes the samples by implementing a circular buffer in the software. While this works fine, we have to perform a test for each loop to wrap the pointer when it reaches the end of the buffer.



**Figure 7.8**

Processing data in a circular buffer requires the Cortex-M4 to check for the end of the buffer on each iteration. This increases the execution time.

Run to the start of the inner loop and set a breakpoint.

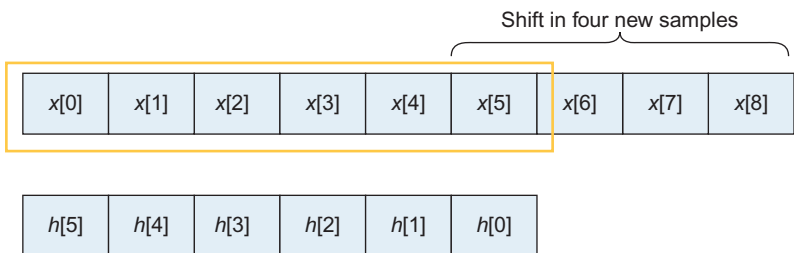
```

16         for(k=0; k<filtLen; k++)
17     {
18         sum += coeffs[k] * state[stateIndex];
19         stateIndex--;
20         if (stateIndex < 0)
21     {

```

Run the code so it does one iteration of the inner loop and note the number of cycles used.

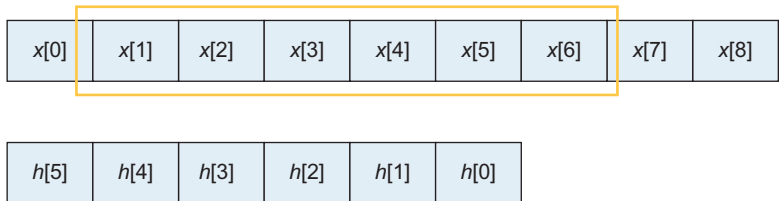
Circular addressing requires us to perform an end of buffer test on each iteration. A dedicated DSP device can support circular buffers in hardware without any such overhead, so this is one area that we need to improve. By passing our FIR filter function, a block of data rather than individual samples allows us to use block processing as an alternative to circular addressing. This improves the efficiency on the critical inner loop.



**Figure 7.9**

Block processing not only increases the size of the buffer but also increases the efficiency of the inner processing loop.

By increasing the size of the state buffer to number of filter taps + processing block size, we can eliminate the need for circular addressing. In the outer loop, the block of samples is loaded into the top of the state buffer.



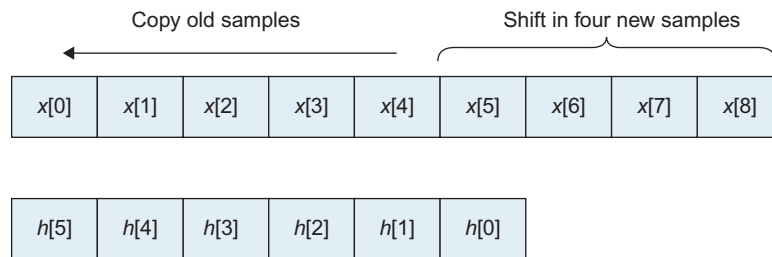
**Figure 7.10**

With block processing, the fixed size buffer is processed without the need to check for the end of the buffer.

The inner loop then performs the filter calculations for each sample of the block by sliding the filter window one element to the right for each pass through the loop. So the inner loop now becomes

```
for(k=0; k<filtLen; k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
}
```

Once the inner loop has finished processing, the current block of sample data held in the state buffer must be shifted to the right and a new block of data should be loaded.



**Figure 7.11**

Once the block of data has been processed, the outer loop shifts the samples one block to the left and adds a new block of data.

**Now step into the second FIR function.**

**Examine how the code has been modified to process blocks of data.**

There is some extra code in the outer loop, but this is only executed once per tap and becomes insignificant compared to the savings made within the inner loop particularly for large block sizes.

**Set a breakpoint on the same inner loop and record the number of cycles it takes to run.**

Next we can further improve the efficiency of the inner loop by using a compiler trick called “loop unrolling.” Rather than iterating round the loop for each tap, we can process several taps in each iteration by inlining multiple tap calculations per loop.

```
I = filtLen >> 2
for(k=0; k<filtLen; k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
```

```
sum += coeffs[k] * state[stateIndex];
stateIndex++;
sum += coeffs[k] * state[stateIndex];
stateIndex++;
sum += coeffs[k] * state[stateIndex];
stateIndex++;
}
```

**Now step into the third FIR function.**

**Set a breakpoint on the same inner loop and record the number of cycles it takes to run. Divide this by four and compare it to the previous implementations.**

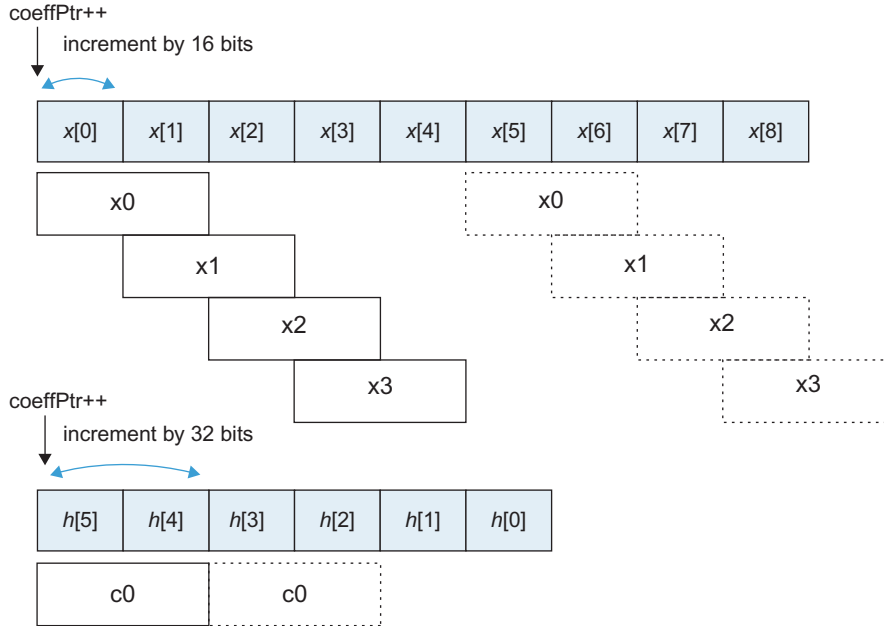
The next step is to make use of the SIMD instructions. By packing the coefficient and sample data into 32-bit words, the single signed multiply accumulates can be replaced by dual signed multiply accumulates, which allows us to extend the loop unrolling from four summations to eight for the same number of cycles.

```
for(k=0;k<filtLen;k++)
{
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
    sum += coeffs[k] * state[stateIndex];
    stateIndex++;
}
```

**Step into the fourth FIR function and again calculate the number of cycles used per tap for the inner loop.**

Remember we are now calculating eight summations, so divide the raw loop cycle count by eight.

To reduce the cycle count of the inner loop even further, we can extend the loop unrolling to calculate several results simultaneously.


**Figure 7.12**

Super loop unrolling extends loop unrolling to process multiple output samples simultaneously.

This is kind of a “super loop unrolling” where we perform each of the inner loop calculations for a block of data in one pass.

```
sample = blockSize/4;
do
{
    sum0 = sum1 = sum2 = sum3 = 0;
    statePtr = stateBasePtr;
    coeffPtr = (q31_t *)(&S->coeffs);
    x0 = *(q31_t *)(&statePtr++);
    x1 = *(q31_t *)(&statePtr++);
    i = numTaps >> 2;
    do
    {
        c0 = *(coeffPtr++);
        x2 = *(q31_t *)(&statePtr++);
        x3 = *(q31_t *)(&statePtr++);
        sum0 = __SMLALD(x0, c0, sum0);
        sum1 = __SMLALD(x1, c0, sum1);
        sum2 = __SMLALD(x2, c0, sum2);
```

```

    sum3=__SMLALD(x3, c0, sum3);
    c0=*(coeffPtr++);
    x0=(q31_t*)(statePtr++);
    x1=(q31_t*)(statePtr++);
    sum0=__SMLALD(x0, c0, sum0);
    sum1=__SMLALD(x1, c0, sum1);
    sum2=__SMLALD(x2, c0, sum2);
    sum3=__SMLALD(x3, c0, sum3);
} while(-i);
*pDst+=(q15_t)(sum0>>15);
*pDst+=(q15_t)(sum1>>15);
*pDst+=(q15_t)(sum2>>15);
*pDst+=(q15_t)(sum3>>15);
stateBasePtr= stateBasePtr+4;
} while(-sample);

```

**Now step into the final FIR function and again calculate the number of cycles used by the inner loop per tap.**

This time we are calculating eight summations for four taps simultaneously; this brings us close to one cycle per tap, which is comparable to a dedicated DSP device.

While you can code DSP algorithms in C and get reasonable performance, these kinds of optimizations are needed to get performance levels comparable to a dedicated DSP device. This kind of code development requires experience with the Cortex-M4 and the DSP algorithms. Fortunately, ARM provides a free DSP library already optimized for the Cortex-M4.

## *The CMSIS DSP Library*

While it is possible to code all of your own DSP functions, this can be time consuming and requires a lot of domain specific knowledge. To make it easier to add common DSP functions to your application, ARM have published a library of 61 common DSP functions as part of CMSIS. Each of these functions are optimized for the Cortex-M4 but can also be compiled to run on the Cortex-M3 and even on the Cortex-M0. The CMSIS DSP library is a free download and is licensed for use in any commercial or noncommercial project. The CMSIS DSP library is also included as part of the MDK-ARM installation and just needs to be added to your project. The installation includes a prebuilt library for each of the Cortex-M processors and all the source code.

Documentation for the library is included as part of the CMSIS help, which can be found in the Books tab of the  $\mu$ Vision project window.

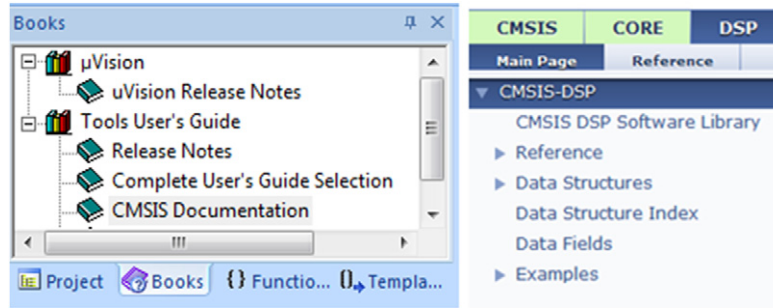


Figure 7.13

The CMSIS DSP documentation is available through the  $\mu$ Vision Books tab.

## CMSIS DSP Library Functions

The Cortex-M DSP library provides easy to use functions for the most commonly used DSP algorithms. The functions included in the library are shown below.

Table 7.11: CMSIS DSP Library Functions

<b>Basic math functions</b> Vector multiplication Vector subtraction Vector addition Vector scale Vector shift Vector offset Vector negate Vector absolute Vector dot product <b>Fast math functions</b> Cosine Sine Square root of number <b>Complex math functions</b> Complex conjugate Complex dot product Complex magnitude Complex magnitude squared Complex by complex multiplication Complex by real multiplication <b>Filters</b> Convolution Partial convolution Correlation FIR filter FIR decimation	<b>Matrix functions</b> Matrix initialization Matrix addition Matrix subtraction Matrix multiplication Matrix inverse Matrix transpose Matrix scale <b>Transforms</b> Complex FFT functions Real FFT functions DCT type IV functions <b>Controller functions</b> <b>Sine cosine</b> <b>PID</b> Vector park transform Vector inverse park transform Vector Clarke transform Vector inverse Clarke transform <b>Statistical functions</b> Power Root mean square Standard deviation Variance Maximum Minimum Mean
--	---

(Continued)

Table 7.11: (Continued)

FIR lattice filter	<b>Support functions</b>
Infinite impulse response lattice filter	Vector copy
FIR sparse filter	Vector fill
FIR filter interpolation	Convert 8-bit integer value
Biquad cascade IIR filter using direct form I structure	Convert 16-bit integer value
Biquad cascade IIR filter $32 \times 64$ using direct form I structure	Convert 32-bit integer value
Biquad cascade IIR filter using direct form II transposed structure	Convert 32-bit FPU
Least mean squares FIR filter	<b>Interpolation functions</b>
Least mean squares normalized FIR filter	Linear interpolation function
	Bilinear interpolation function

### Exercise: Using the Library

In this project, we will have a first look at using the CMSIS DSP library by setting up a project to experiment with the PID control algorithm.

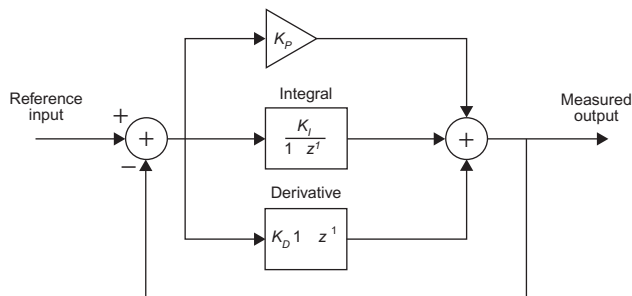
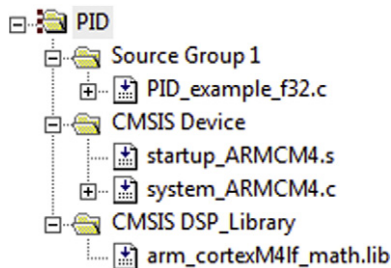


Figure 7.14

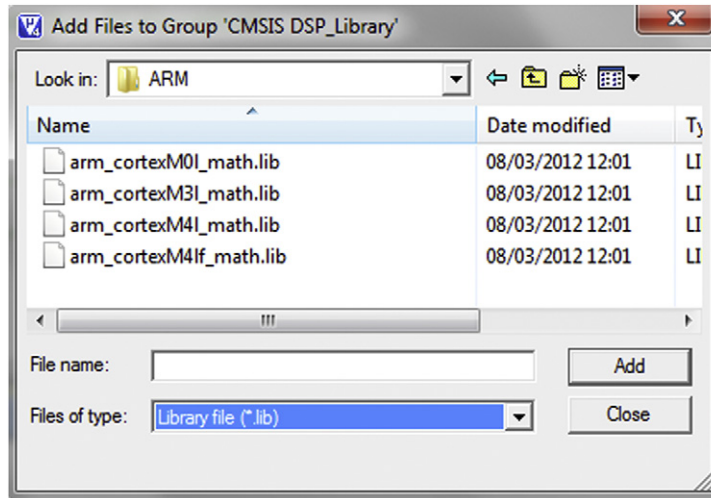
A PID control loop consists of proportional, integral, and derivative control blocks.

**Open the project in c:\exercises\PID.**



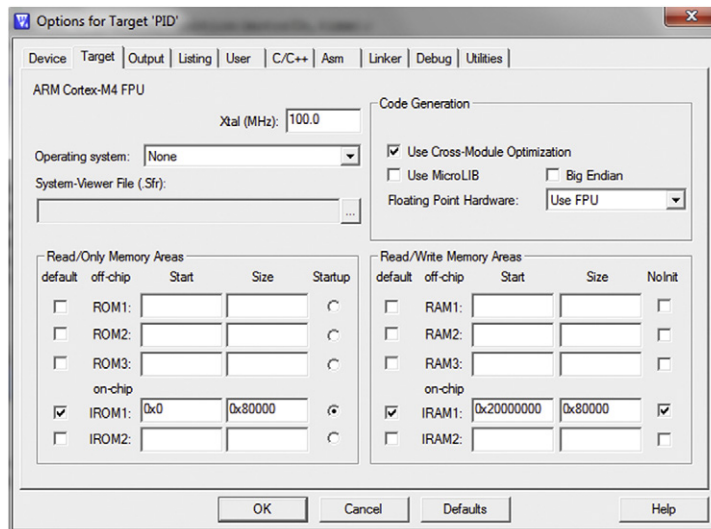
The project is targeted at a Cortex-M4 with FPU. It includes the CMSIS startup and system files for the Cortex-M4 and the CMSIS DSP functions as a precompiled library. The CMSIS DSP library is located in c:\CMSIS\lib with subdirectories for the ARM compiler and GCC versions of the library.



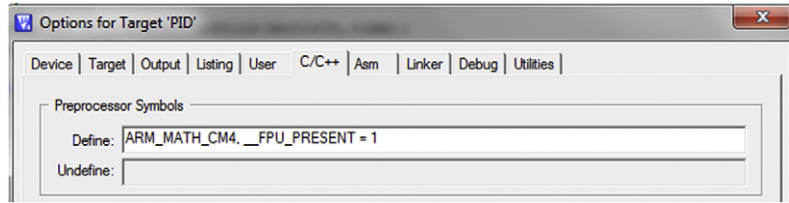


There are precompiled versions of the DSP library for each Cortex processor. There is also a version of the library for the Cortex-M4 with and without the FPU. For our project, the Cortex-M4 floating point library has been added.

**Open the Project\Options for Target dialog.**



In this example, we are using a simulation model for the Cortex-M4 processor only. A 512 K memory region has been defined for code and data. The FPU is enabled and the CPU clock has been set to 100 MHz.



In the compiler options tab, the `_FPU_PRESENT` define is set to one. Normally you will not need to configure this option as this will be done in the microcontroller include file. We also need to add a `#define` to configure the DSP header file for the processor we are using. The library defines are

```
ARM_MATH_CM4
ARM_MATH_CM3
ARM_MATH_CM0
```

The source code for the library is located in `c:\keil\arm\CMSIS\DSP_Lib\Source`.

**Open the file `PID_example_f32.c` which contains the application code.**

To access the library, we need to add its header file to our code.

```
#include "arm_math.h"
```

This header file is located in `c:\keil\arm\CMSIS\include`.

In this project, we are going to use the PID algorithm. All of the main functions in the DSP library have two function calls: an initializing function and a process function.

```
void arm_pid_init_f32 (arm_pid_instance_f32 *S, int32_t resetStateFlag)
__STATIC_INLINE void arm_pid_f32 (arm_pid_instance_f32 *s, float32_t in)
```

The initializing function is passed on a configuration structure that is unique to the algorithm. The configuration structure holds constants for the algorithm, derived values, and arrays for state memory. This allows multiple instances of each function to be created.

```
typedef struct
{
    float32_t A0;      /**< The derived gain, A0 = Kp + Ki + Kd. */
    float32_t A1;      /**< The derived gain, A1 = -Kp - 2 Kd. */
    float32_t A2;      /**< The derived gain, A2 = Kd. */
    float32_t state[3]; /**< The state array of length 3. */
    float32_t Kp;      /**< The proportional gain. */
    float32_t Ki;      /**< The integral gain. */
    float32_t Kd;      /**< The derivative gain. */
} arm_pid_instance_f32;
```

The PID configuration structure allows you to define values for the proportional, integral, and derivative gains. The structure also includes variables for the derived gains A0, A1, and A2 as well as a small array to hold the local state variables.

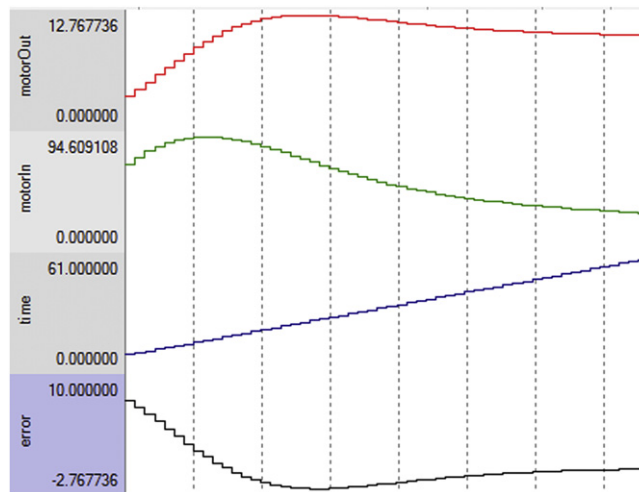
```
int32_t main(void){
    int i; S.Kp=1; S.Ki=1; S.Kd=1;
    setPoint=10;
    arm_pid_init_f32 (&S,0);
    while(1){
        error=setPoint-motorOut;
        motorIn=arm_pid_f32 (&S,error);
        motorOut=transferFunction(motorIn,time);
        time += 1;
        for(i=0;i<100000;i++);
    }}

```

The application code sets the PID gain values and initializes the PID function. The main loop calculates the error value before calling the PID process function. The PID output is fed into a simulated hardware transfer function. The output of the transfer function is fed back into the error calculation to close the feedback loop. The time variable provides a pseudo-time reference.

**Build the project and start the debugger.**

**Add the key variables to the logic analyzer and start running the code.**



The logic analyzer is invaluable for visualizing data in a real-time algorithm and can be used in the simulator or can capture data from the CoreSight data watch trace unit.

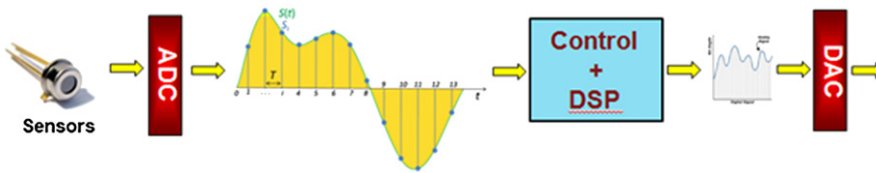
### Experiment with the gain values to tune the PID function.

The performance of the PID control algorithm is tuned by adjusting the gain values. As a rough guide, each of the gain values has the following effect.

- Kp Effects the rise time of the control signal.
- Ki Effects the steady state error of the control signal.
- Kd Effects the overshoot of the control signal.

### DSP Data Processing Techniques

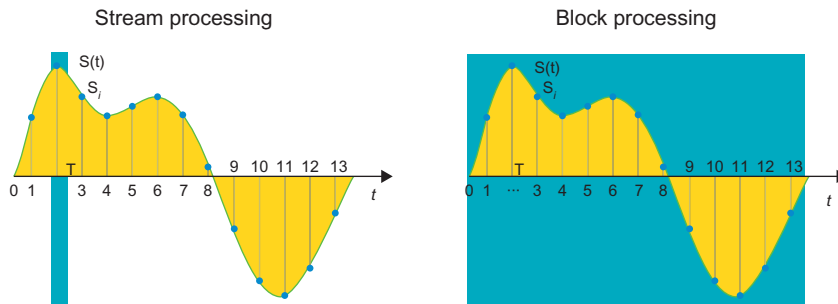
One of the major challenges of a DSP application is managing the flow of data from the sensors and ADC through the DSP algorithm and back out to the real world via the DAC.



**Figure 7.15**

A typical DSP system consists of an analog sample stage, microcontroller with DSP algorithm, and an output DAC. In this chapter, we concentrate on the microcontroller software in isolation from the hardware design.

In a typical system, each sampled value is a discrete value at a point in time. The sample rate must be at least twice the signal bandwidth or up to four times the bandwidth for a high-quality oversampled audio system. Clearly the volume of data is going to ramp up very quickly and it becomes a major challenge to process the data in real time. In terms of processing the sampled data, there are two basic approaches, stream processing and block processing.



**Figure 7.16**

Analog data can be processed as single samples with minimum latency or as a block of samples for maximum processing efficiency.

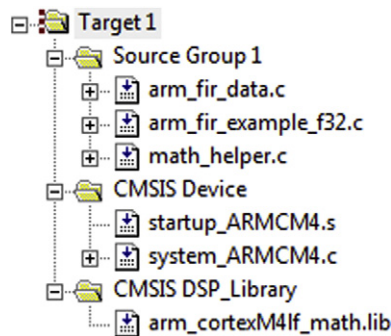
In stream processing, each sampled value is processed individually. This gives the lowest signal latency and also the minimum memory requirements. However, it has the disadvantage of making the DSP algorithm more complex. The DSP algorithm has to be run every time an ADC conversion is made, which can cause problems with other high-priority interrupt routines.

The alternative to stream processing is block processing. Here a number of ADC results are stored in a buffer, typically about 32 samples, and then this buffer is processed by the DSP algorithm as a block of data. This lowers the number of times that the DSP algorithm has to run. As we have seen in the optimization exercise, there are a number of techniques that can improve the efficiency of an algorithm when processing a block of data. Block processing also integrates well with the microcontroller DMA unit and an RTOS. On the downside, block processing introduces more signal latency and requires more memory than stream processing. For the majority of applications, block processing should be the preferred route.

### ***Exercise: FIR Filter with Block Processing***

In this exercise, we will implement an FIR filter, this time by using the CMSIS DSP functions. This example uses the same project template as the PID program. The characteristics of the filter are defined by the filter coefficients. It is possible to calculate the coefficient values manually or by using a design tool. Calculating the coefficients is outside the scope of this book, but the appendices list some excellent design tools and DSP books for further reading.

**Open the project in c:\examples CMSIS\_FIR.**



There is an additional data file that holds a sampled data set and an additional math\_helper.c file that contains some ancillary functions.

```
int32_t main(void)
{
    uint32_t i;
    arm_fir_instance_f32 S;
    arm_status status;
    float32_t *inputF32, *outputF32;
    /* Initialize input and output buffer pointers */
    inputF32 = &testInput_f32_1_kHz_15_kHz[0];
    outputF32 = &testOutput[0];
    /* Call FIR init function to initialize the instance structure. */
    arm_fir_init_f32(&S, NUM_TAPS, (float32_t *)&firCoeffs32[0], &firStateF32[0],
    blockSize);
    for(i = 0; i < numBlocks; i++)
    {
        arm_fir_f32(&S, inputF32 + (i * blockSize), outputF32 + (i * blockSize), blockSize);
    }
    snr = arm_snr_f32(&refOutput[0], &testOutput[0], TEST_LENGTH_SAMPLES);
    if (snr < SNR_THRESHOLD_F32)
    {
        status = ARM_MATH_TEST_FAILURE;
    } else {
        status = ARM_MATH_SUCCESS;
    }
}
```

The code first creates an instance of a 29-tap FIR filter. The processing block size is 32 bytes. An FIR state array is also created to hold the working state values for each tap. The size of this array is calculated as the block size + number of taps – 1. Once the filter has been initialized, we can pass it to the sample data in 32-byte blocks and store the resulting processed data in the output array. The final filtered result is then compared to a precalculated result.

**Build the project and start the debugger.**

**Step through the project to examine the code.**

**Look up the CMSIS DSP functions used in the help documentation.**

Set a breakpoint at the end of the project.

```

209     if( status != ARM_MATH_SUCCESS)
210     {
211         while(1);
212     }
213
214     while(1);
215 }

```

**Reset the project and run the code until it hits the breakpoint.**

Now look at the cycle count in the registers window.

Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	90260
Sec	0.00752167

**Open the project in c:\exercises\CMSIS\_FIR\CM3.**

This is the same project built for the Cortex-M3.

**Build the project and start the debugger.**

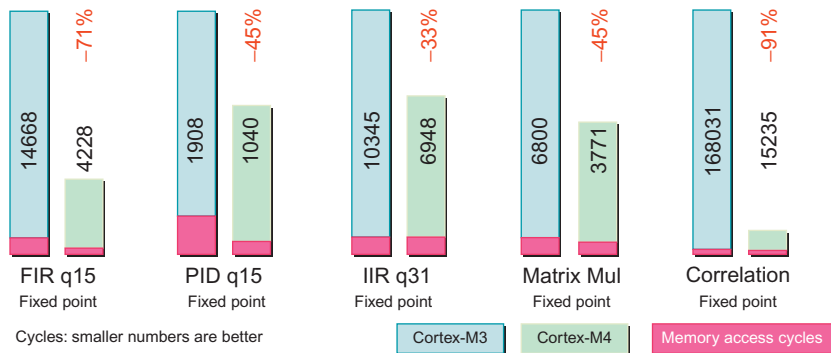
**Set a breakpoint in the same place as the previous Cortex-M4 example.**

**Reset the project and run the code until it hits the breakpoint.**

Now compare the cycle count used by the Cortex-M3 to the Cortex-M4 version.

Internal	
Mode	Thread
Privilege	Privileged
Stack	MSP
States	855964
Sec	0.07133033

When using floating point numbers, the Cortex-M4 is nearly an order faster than the Cortex-M3 using software floating point libraries. When using fixed point math, the Cortex-M4 still has a considerable advantage over the Cortex-M3.

**Figure 7.17**

The bar graphs show a comparison between the Cortex-M3 and Cortex-M4 for common DSP algorithms.

### Fixed Point DSP with Q Numbers

The functions in the DSP library support floating point and fixed point data. The fixed point data is held in a Q number format. Q numbers are fixed point fractional numbers held in integer variables. A Q number has a sign bit followed by a fixed number of bits to represent the integer value. The remaining bits represent the fractional part of the number.

Signed = S I I I I I I I I . F F F F F F

A Q number is defined as the number of bits in the integer portion of the variable and the number of bits in the fractional portion of the variable. Signed Q numbers are stored as two complement values. A Q number is typically referred to by the number of fractional bits it uses so Q10 has 10 fractional places. The CMSIS DSP library functions are designed to take input values between +1 and -1. The supported integer values are shown below.

**Table 7.12: Floating Point Unit Performance**

CMSIS DSP Typedef	Q Number
Q31_t	Q31
Q15_t	Q15
Q7_t	Q7

The library includes a group of conversion functions to change between floating point numbers and the integer Q numbers. Support functions are also provided to convert between different Q number resolutions.



Table 7.13: CMSIS DSP Type Conversion Functions

arm_float_to_q31	arm_q31_to_float	arm_q15_to_float	arm_q7_to_float
arm_float_to_q15	arm_q31_to_q15	arm_q15_to_q31	arm_q7_to_q31
arm_float_to_q7	arm_q31_to_q7	arm_q15_to_q7	arm_q7_to_q15

As a real-world example, you may be sampling data using a 12-bit ADC that gives an output from 1-0xFFF. In this case, we would need to scale the ADC result to be between +1 and -1 and then convert to a Q number.

```
Q31_t ADC_FixedPoint;
float temp;
```

Read the ADC register to a float variable. Scale between +1 and -1.

```
temp = ((float32_t)((ADC_DATA_REGISTER) & 0xFFF) / (0xFFF / 2)) - 1;
```

Convert the float value to a fixed point Q31 value.

```
arm_float_to_q31(&ADC_FixedPoint, &temp, 1);
```

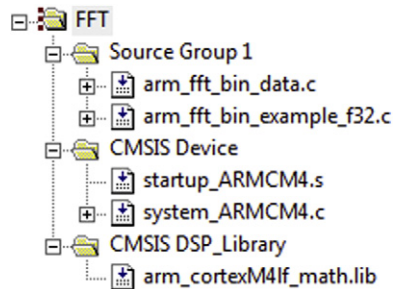
Similarly, after the DSP function has run, it is necessary to convert back to a floating point value before using the result. Here we are converting from a Q31 result to a 10-bit integer value prior to outputting the value to a DAC peripheral.

```
arm_q31_to_float(&temp, &DAC_Float, 1);
DAC_DATA_REGISTER = (((uint32_t)((DAC_Float))) & 0x03FF);
```

### ***Exercise: Fixed Point FFT***

In this project, we will use the FFT to analyze a signal.

**Open the project in c:\exercises\FFT.**



The FFT project uses the same template as the PID example with the addition of a data file that holds an array of sample data. This data is a 10 KHz signal plus random “white noise.”

```
int32_t main(void) {
    arm_status status;
    arm_cfft_radix4_instance_q31 S;
    q31_t maxValue;
    status = ARM_MATH_SUCCESS;
    /* Convert the floating point values to the Q31 fixed point format */
    arm_float_to_q31 (testInput_f32_10khz, testInput_q31_10khz, 2048);
    /* Initialize the CFFT/CIFFT module */
    status = arm_cfft_radix4_init_q31(&S, fftSize, ifftFlag, doBitReverse);
    /* Process the data through the CFFT/CIFFT module */
    arm_cfft_radix4_q31(&S, testInput_q31_10khz);
    /* Process the data through the Complex Magnitude Module for
    calculating the magnitude at each bin */
    arm_cmplx_mag_q31(testInput_q31_10khz, testOutput, fftSize);
    /* Calculates maxValue and returns corresponding BIN value */
    arm_max_q31(testOutput, fftSize, &maxValue, &testIndex);
    if(testIndex != refIndex)
    {
        status = ARM_MATH_TEST_FAILURE;
    }
}
```

The code initializes the complex FFT with a block size of 1024 output bins. When the FFT function is called, the sample data set is first converted to fixed point Q31 format and is then passed to the transform as one block. Next the complex output is converted to a scalar magnitude and then scanned to find the maximum value. Finally, we compare this value to an expected result.

**Work through the project code and look up the CMSIS DSP functions used.**

**Note:** In an FFT, the number of output bins will be half the size of the sample data size.

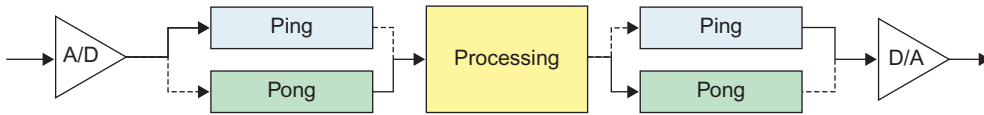
**Build the project and start the debugger.**

## *Designing for Real-Time Processing*

So far, we have looked at the DSP features of the Cortex-M4 and the supporting DSP library. In this next section, we will look at developing a real-time DSP program that can also support non-real-time features like a user interface or communication stack without disturbing the performance of the DSP algorithm.

## Buffering Techniques: The Double or Circular Buffer

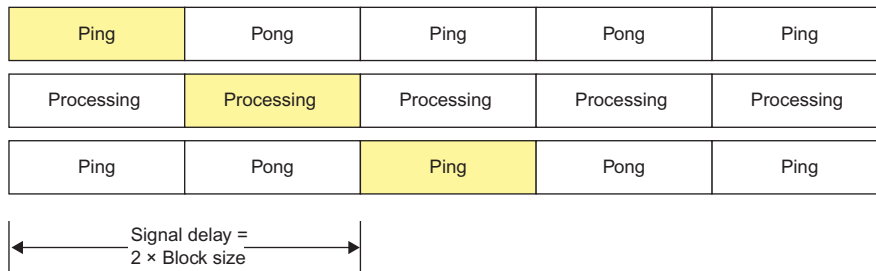
When developing your first real-time DSP application, the first decision you have to make is how to buffer the incoming blocks of data from the ADC and the outgoing blocks of data to the DAC. A typical solution is to use a form of double buffer as shown below.



**Figure 7.18**

A simple DSP system can use a double buffer to capture the stream of data.

While the ping buffer is being filled with data from the ADC, the DSP algorithm will be processing the data stored in the pong buffer. Once the ADC reaches the end of the ping buffer, it will start to store data into the pong buffer and the new data in the ping buffer may be processed. A similar ping-pong structure can be used to buffer the output data to the DAC.



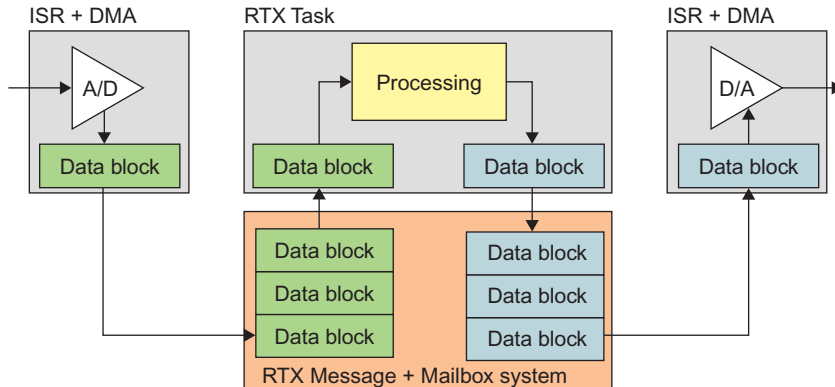
**Figure 7.19**

The double buffer causes minimum signal latency but requires the DSP algorithm to run frequently. This makes it hard to maintain real-time performance when other software routines need to be executed.

This method works well for simple applications; it requires the minimum amount of memory and introduces the minimal latency. The application code must do all of the necessary buffer management. However, as the complexity of the application rises, this approach begins to run into problems. The main issue is that you have a critical deadline to meet, in that the DSP algorithm must have finished processing the pong buffer before the ADC fills the ping buffer. If, for example, the CPU has to run some other critical code such as user interface or communications stack, then there may not be enough processing power available to run all the necessary codes and the DSP algorithm in time to meet the end of buffer deadline. If the DSP algorithm misses its deadline, then the ADC will start to overwrite the active buffer. You then have to write more sophisticated buffer management code to “catch up,” but really you are on to a losing game and data will at some point be lost.

## Buffering Techniques: FIFO Message Queue

An alternative to the basic double buffer approach is to use an RTOS to provide a more sophisticated buffering structure using the mailbox system to create a FIFO queue.



**Figure 7.20**

Block processing using RTOS mail queues increases the signal latency but provides reliable integration of a DSP thread into a complex application.

Using the RTX RTOS, we can create a task that is used to run the DSP algorithm. The inputs and outputs of this task are a pair of mailboxes. The ADC interrupt is used to fill a mailbox message with a block of ADC data and then post that message to the DSP task. These message blocks are then placed in a FIFO queue ready to be processed. Now if other critical code needs to run, the DSP task will be preempted and rather than risk losing data as in the double buffer case the blocks of data will be sitting in the message queue. Once the critical code has run the DSP, the task will resume running and can use all the CPU processing power to “catch up” with the backlog of data. A similar message queue is used to send blocks of data to the DAC. While you could code this yourself by extending the double buffer example, you would simply end up re inventing the wheel. The RTX RTOS already has message queues with buffer management implemented and as we shall see it results in simple and elegant application code with all the buffer management handled by the RTX RTOS.

In the first example program, the ADC is sampling at 40 KHz and is placed in a 32-byte message buffer.

The message queue is defined by allocating a block of memory as the message space. This region is formatted into a group of mailslots. Each mailslot is a structure in the shape of the data you want to send.

The shape of the mailslot can be any collection of data declared as a structure. In our case, we want an array of floating point values to be our block of data.

```
typedef struct {
    float32_t dataBuffer[BLOCK_SIZE]; //declare structure for message queue mailslot
} ADC_RESULT;
```

Next a block of memory is reserved for the mail queue. Here memory is reserved for a queue of 16 mailslots. Data can be written into each mailslot; the mailslot is then locked until the message is received and the buffer released.

```
osMailQDef(mpool_ADC, 16, ADC_RESULT);
```

Then the mailbox can be instantiated by formatting the message box and then initializing the message pointer queue.

```
osMailQId MsgBox;

MsgBox = osMailCreate(osMailQ(MsgBox), NULL);
```

Once this is done, the mailbox is ready to send data between tasks or ISR and tasks. The code below shows the ADC handler logging 32 conversions into a mailslot and then posting it to the DSP routine.

```
void ADC_IRQHandler(void)
{
    static ADC_RESULT *mptr; //declare a mailslot pointer
    static unsigned char index = 0;
    if(index == 0) {
        mptr = (ADC_RESULT*)osMailAlloc(MsgBox, osWaitForever); //allocate a new mailslot
    }
    mptr->dataBuffer[index] = (float)((LPC_ADC->ADGDR>>4)& 0xFFF); //place ADC
                                                                    data into the mailslot

    index++;
    if(index == BLOCK_SIZE){
        index = 0;
        osMailPut(MsgBox, mptr); //when mailslot is full send
    }
}
```

In the main DSP task, the application code waits for the blocks of data to arrive in the message queue from the ADC. As they arrive, each one is processed and the results are placed in a similar message queue to be fed into the DAC.

```
void DSP_thread (void)
{
    ADC_RESULT *ADCptr;    //declare mailslot pointers for ADC and DAC mailboxes
    ADC_RESULT *DACptr;
    while(1)
    {
        evt=osMailGet(MsgBox_ADC, osWaitForever);    //wait for a message from the ADC
        if (evt.status == osEventMail)
            ADCptr=(ADC_RESULT*)evt.value.p;
        DACptr=(ADC_RESULT*)osMailAlloc(MsgBox, osWaitForever);    //allocate a mailslot for
                                                                    the DAC message queue
        DSP_Alogrithm (ADCptr->dataBuffer, DACptr->dataBuffer);    //run the DSP algorithm
        osMailPut(MsgBox,DACptr);    //post the results to the DAC message queue
        osMailFree(MsgBox, ADCptr);    //release the ADC mailslot
    }
}
```

In the example code, a second task is used to read the DAC message and write the processed data to the DAC output register.

```
void DAC_task (void)
{
    unsigned int i;
    ADC_RESULT *DACrptr;
    while(1)
    {
        evt=osMailGet(MsgBox, osWaitForever); //wait for a mailslot of processed data
        if (evt.status == osEventMail)
            DACrptr=(ADC_RESULT*)evt.value.p;
        for(i=0;i<(BLOCK_SIZE);i++)
        {
            osSignalWait (0x01,osWaitForever); //synchronise with the ADC sample rate
            LPC_DAC->DACR =(unsigned int) DACrptr->dataBuffer[i]<<4;//write a processed
                                                                    data value to the DAC register
        }
        osMailFree(MsgBox, DACrptr); //free the mailslot
    }
}
```

The `osSignalWait()` RTX call causes the DAC task to halt until another task or ISR sets its event flags to the pattern `0x0001`. This is a simple triggering method between tasks that can be used to synchronize the output of the DAC with the sample rate of the ADC. Now if we add the line

```
osSignalSet (tsk_DAC,0x01); //set the event flag in the DAC task.
```

to the ADC ISR, this will trigger the DAC task each time the ADC makes a conversion. The DAC task will wake up each time its event flag is triggered and write an output value to the DAC. This simple method synchronizes the input and output streams of data.

### ***Balancing the Load***

The application has two message queues, one from the ADC ISR to the DSP task and one from the DSP task to the DAC task. If we want to build a complex system, say with a GUI interface and a TCP/IP stack, then we must accept that during critical periods the DSP task will be preempted by other tasks running on the system. Provided that the DSP thread can run before the DAC message queue runs dry, then we will not compromise the real-time data. To cope with this, we can use the message queues to provide the necessary buffering to keep the system running during these high demand periods. All we need to do to ensure that this happens is post a number of messages to the DAC message queue before we start running the DAC task. This will introduce additional latency, but there will always be data available for the DAC. It is also possible to use the timeout value in the `osMessageGet()` function. Here we can set a timeout value to act as a watchdog.

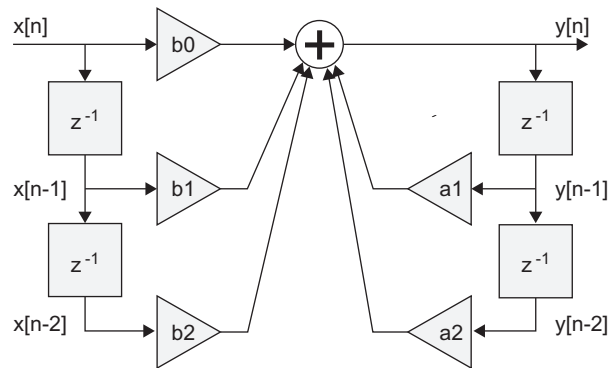
```
evt=osMailGet(MsgBox, WatchdogTimeout);
if(evt.status == osEventTimeout)
    osThreadSetPriority(t_DSP,osPriorityHigh);
```

Now we will know the maximum period at which the DAC thread must read a data from the message queue. We can set the timeout period to be  $\frac{3}{4}$  of this period. If the message has not arrived, then we can boost the priority of the DSP thread. This will cause the DSP thread to preempt the running thread and process and delayed packets of data. These will then be posted on to the DAC thread. Once the DAC has data to process, it can lower the priority of the DSP thread to resume processing as normal. This way we can guarantee the stream of real-time data in a system with bursts of activity in other threads.

The obvious downside of having a bigger depth of message buffering is the increased signal latency. However, for most applications, this should not be a problem. Particularly when you consider the ease of use of the mailbox combined with the sophisticated memory management of the RTX RTOS.

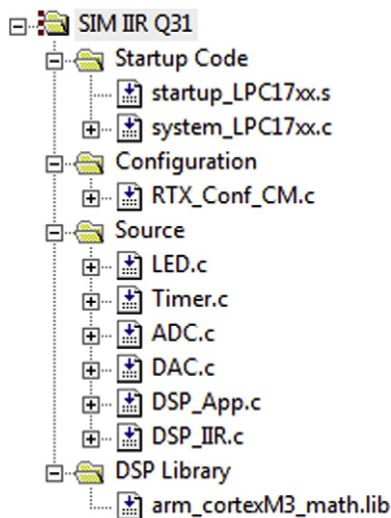
**Exercise: RTX IIR**

This exercise implements an IIR filter as an RTX task. A periodic timer interrupt takes data from a 12-bit ADC and builds a block of data that is then posted to the DSP task to be processed. The resulting data is posted back to the timer ISR. As the timer ISR receives processed data packets, it writes an output value per interrupt to the 10-bit DAC. The sampling frequency is 32 KHz and the sample block size is 256 samples.

**Figure 7.21**

An IIR filter is a feedback filter that is much faster than an FIR filter but can be unstable if incorrectly designed.

**Open the project in c:\exercises\CMSIS\RTX DSP.**



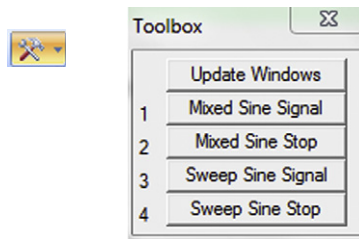


### First examine the code, particularly the DSP\_App.c module.

The project is actually designed for a Cortex-M3-based microcontroller which as a simulation model that includes an ADC and DAC. The modules ADC.c, Timer.c, and DAC.c contain functions to initialize the microcontroller peripherals. The module DSP\_App.c contains the initializing task that sets up the mail queues and creates the DSP task called “sigmod.” An additional “clock” task is also created and this task periodically flashes GPIO pins to simulate other activity on the system. DSP\_App.c also contains the timer ISR. The timer ISR can be split into two halves. The first section reads the ADC and posts sample data into the DSP task inbound mail queue. The second half receives messages from the outbound DSP task mail queue and writes the sampled data to the DAC. The CMSIS filter functions are in DSP\_IIR.c.

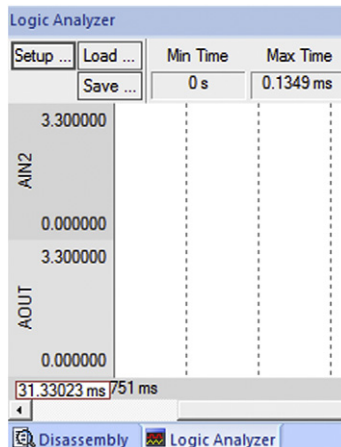
### Build the project and start the simulator.

When the project starts, it also loads a script file that creates a set of simulation functions. During the simulation, these functions can be accessed via buttons created in the toolbox dialog. If the toolbox dialog is not open when the simulation starts, it can be opened by pressing the “hammer and wrench” icon on the debugger toolbar.



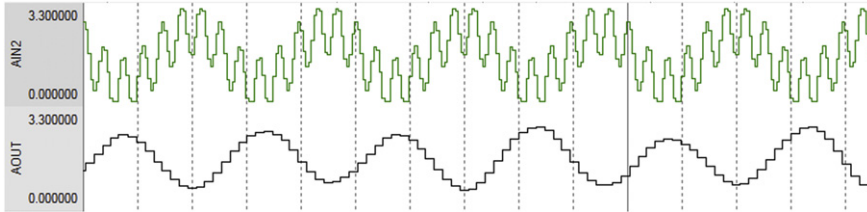
The simulation script generates simulated analog signals linked to the simulator time base and applies them to the simulated ADC input pin.

### Open the logic analyzer window.



The logic analyzer should have two signals defined, AIN2 and AOUT, each with a range of 0–3.3. These are not program variables but virtual simulation registers that represent the analog input pin and the DAC output pin.

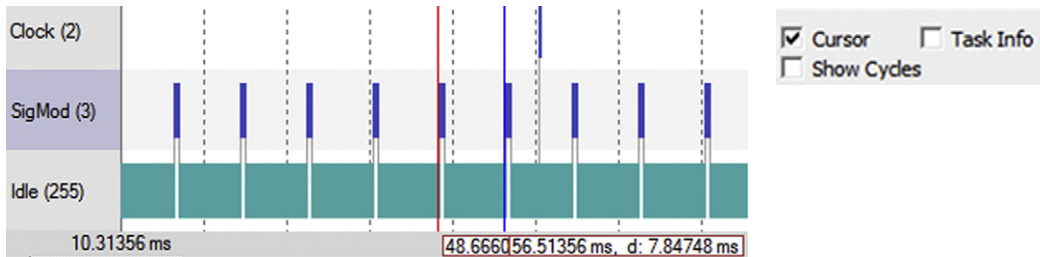
**Start running the simulator and press the “Mixed Signal Sine” button in the toolbox dialog.**



This will generate an input signal that consists of a mixed high- and low-frequency sine wave. The filter removes the high-frequency component and outputs the processed wave to the DAC.

**Now open the Debug\OS Support\Event Viewer window.**

In the event viewer window, we can see the activity of each task in the system. Tick the Cursor box on the event viewer toolbar. Now we can use the red and blue cursors to make task timing measurements. This allows us to see how the system is working at a task level and how much processing time is free (i.e., time spent in the idle task).



**Stop the simulation and wind back the logic analyzer window to the start of the mixed sine wave.**

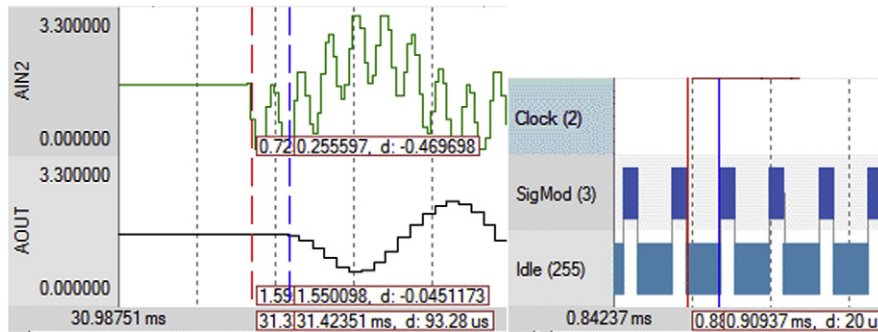


This shows a latency of around 8.6 ms between signal data being sampled and processed data being output from the DAC. While part of this latency is due to the DSP processing time, most of the delay is due to the sample block size used for the DSP algorithm.

**Exit the simulator and change the DSP block size from 256 to 1.**

The DSP blocksize is defined in IIR.h.

**Rebuild and rerun the simulation.**



Now the latency between the input and output signal is reduced to around 90  $\mu$ s. However, if we look at the event viewer, we can see that the SigMod task is running for every sample and is consuming most of the processor resources and is in fact blocking the clock task.

### ***Shouldering the Load, the Direct Memory Access Controller***

The Cortex-M3 and Cortex-M4 microcontrollers are designed with a number of parallel internal busses; this is called the “AHB bus matrix lite.” The bus matrix allows a Cortex-M-based microcontroller to be designed with multiple bus masters (i.e., a unit capable of initiating a bus access) that can operate in parallel. Typically a Cortex-M microcontroller will have the Cortex processor as a bus master and one or possibly several Direct Memory Access (DMA) units. The DMA units can be programmed by the Cortex-M processor to transfer data from one location in memory to another while the Cortex processor performs another task. Provided that the DMA unit and the Cortex-M processor are not accessing resources on the same arm of the bus matrix, they can run in parallel with no need for bus arbitration. In the case of peripherals, a typical DMA unit can, for example, transfer data from the ADC results register to incremental locations in memory. When working with a peripheral, the DMA unit can be slaved to the peripheral so that the peripheral becomes the flow controller for the DMA transfers. So each time there is an ADC result, the DMA unit transfers the new data to the next location in memory. Going back to our original example by using a DMA unit, we can replace the ADC ISR routine with a DMA transfer that places the ADC results directly into the current mailslot. When the DMA unit has

completed its 32 transfers, it will raise an interrupt. This will allow the application code to post the message and configure the DMA unit for its next transfer. The DMA can be used to manage the flows of raw data around the application leaving the Cortex-M processor free to run the DSP processing code.