Freescale Semiconductor
Application Note

Document Number: AN4489
Rev. 0, 3/2012

Using CMSIS-DSP Algorithms with MQX

by: **Kelben Zhang**
IM&M FAE Team, GSM
China

Contents

1 Introduction 1
2 CMSIS DSP Library 1
3 Freescale MQX RTOS 5
4 CMSIS DSP Library Examples 6
5 Conclusion 18

1 Introduction

Kinetis products use the ARM® Cortex-M4 processor which supports a single cycle 16.32-bit MAC (DSP function). Using a DSP function such as signal processing, motor control, and data analyze can reduce core resources and enhance core performance.

This application note discusses the CMSIS DSP library combined with Freescale MQX RTOS. Included are also the CMSIS and CMSIS DSP library introductions. The application note is based on the CMSIS DSP library version 2.10 using the IAR ARM Workbench Rev.6.21.

2 CMSIS DSP Library

2.1 ARM® Cortex-M4 Core

The ARM® Cortex™-M4 processor has a large variety of highly efficient signal processing features applicable to digital signal control markets. The combination of a high-efficiency signal processing function with the low-power, low cost, and ease-of-use benefits of the Cortex-M4 processors is to satisfy the emerging category of flexible solutions. This specifically targets the motor control, automotive, power management, embedded audio, and industrial automation markets.

© 2012 Freescale Semiconductor, Inc.

freescale semiconductor

2.2 CMSIS

The ARM® Cortex™ Microcontroller Software Interface Standard (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 peripherals. It simplifies software re-use by reducing the learning curve for new microcontroller developers and reducing the time to market for new devices.

Figure 1. CMSIS mark

The CMSIS is divided into three basic function layers:

- Core Peripheral Access Layer (CPAL)
- Middleware Access Layer (MWAL)
- Device Peripheral Access Layer (DPAL)

The basic structure and the functional flow is shown in Figure 2.

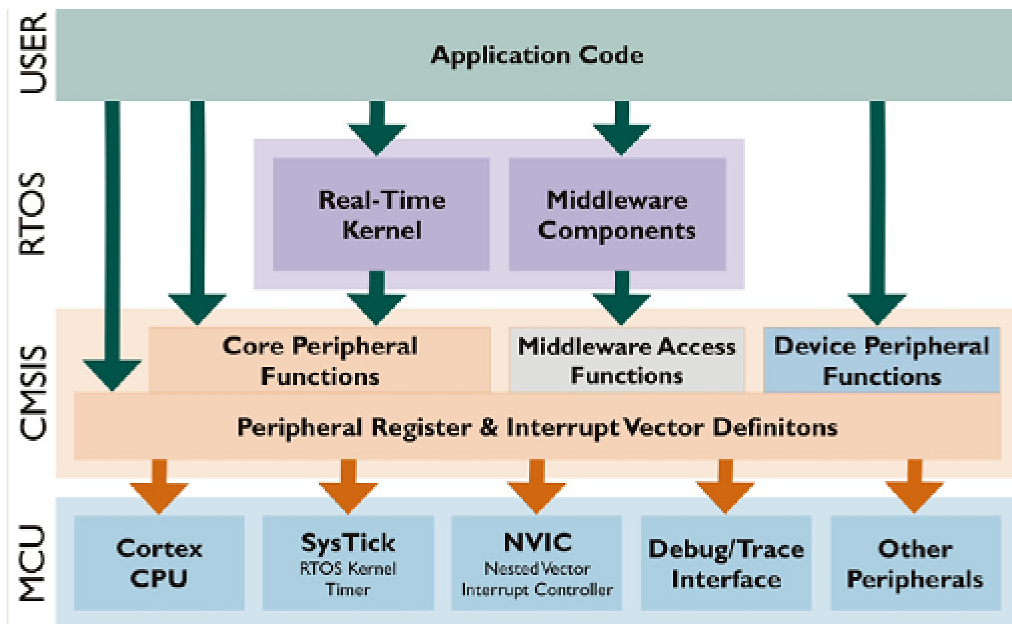

Figure 2. CMSIS structure function flow

Core Peripheral Access Layer (CPAL)

The lowest level defines addresses and accesses methods for common components and functions in every Cortex-M system. Access to core registers, NVIC, and the debug subsystem is provided by this layer. Tool specific access to special purpose registers (for example CONTROL, xPSR), is provided in the form of inline functions or compiler intrinsics. This layer is provided by ARM.

Middleware Access Layer (MWAL)

This layer is also defined by ARM, but is adapted by silicon vendors for their respective devices. The Middleware Access Layer defines a common API for accessing peripherals. The Middleware Access Layer is still under development, no further information is available at this point.

Device Peripheral Access Layer (DPAL)

Hardware register addresses, device specific access functions, and other definitions are defined in this layer. The device peripheral access layer is similar to the Core Peripheral Access Layer and is provided by the silicon vendor. Access methods provided by CPAL may be referenced. The vector table is adapted to include the device specific exception handler address.

DPAL is intended to be extended by the silicon vendor. Cortex-M based FPGA products effectively place developers in the position of a silicon vendor.

In MCU based systems, developers should treat the entire PCB system as a monolithic block. There is no reason to differentiate between a memory mapped register inside the MCU and a memory mapped register external to the MCU connected via an external memory interface. The benefit of applying a standard like CMSIS are the guidelines on how to access these devices and set a clear goal on how to implement and integrate critical parts of the software.

2.3 CMSIS DSP Library

The CMSIS-DSP 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 modules each covering a specific category:

- Basic math functions
  - Vector Absolute Value
  - Vector Addition
  - Vector Dot Product
  - Vector Multiplication
  - Vector Negate
  - Vector Offset
  - Vector Scale
  - Vector Shift
  - Vector Subtraction
- Fast math functions
  - Cosine
  - Sine
  - Square Root
- Complex math functions
  - Complex Conjugate
  - Complex Dot Product
  - Complex Magnitude
  - Complex Magnitude Squared
  - Complex-by-Complex Multiplication
  - Complex-by-Real Multiplication
- Filters
  - Biquad Cascade IIR Filters Using Direct Form I Structure
  - Biquad Cascade IIR Filters Using a Direct Form II Transposed Structure
  - High Precision Q31 Biquad Cascade Filter
  - Convolution
  - Partial Convolution
  - Correlation
  - Finite Impulse Response (FIR) Decimator
  - Finite Impulse Response (FIR) Filters
  - Finite Impulse Response (FIR) Lattice Filters

    - Finite Impulse Response (FIR) Sparse Filters
    - Infinite Impulse Response (IIR) Lattice Filters
    - Least Mean Square (LMS) Filters
    - Normalized LMS Filters
    - Finite Impulse Response (FIR) Interpolator

- Matrix functions
    - Matrix Addition
    - Matrix Initialization
    - Matrix Inverse
    - Matrix Multiplication
    - Matrix Scale
    - Matrix Subtraction
    - Matrix Transpose

- Transforms
    - Complex FFT Functions
    - DCT Type IV Functions
    - Real FFT Functions

- Motor control functions
    - Sine Cosine
    - PID Motor Control
    - Vector Clarke Transform
    - Vector Inverse Clarke Transform
    - Vector Park Transform
    - Vector Inverse Park transform

- Statistical functions
- Support functions
- Interpolation functions
    - Linear Interpolation
    - Bilinear Interpolation

The library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integers, and 32-bit floating-point values.

The Cortex-M4 processor implementation uses the ARM DSP SIMD (Single Instruction Multiple Data) instruction set and floating-point hardware to fully enable the Cortex-M4 processor capabilities for the signal processing algorithms. The optimized CMSIS-DSP library is written entirely in C and is delivered with source code which enables software programmers to adapt algorithms for specific application requirements. The library has been developed and tested with MDK-ARM version 4.21. The library is also tested in GCC and IAR toolchains.

The library installer contains prebuilt versions of the libraries in the Lib folder.

- arm_cortexM4lf_math.lib (Little endian and Floating Point Unit on Cortex-M4)
- arm_cortexM4bf_math.lib (Big endian and Floating Point Unit on Cortex-M4)
- arm_cortexM4l_math.lib (Little endian on Cortex-M4)
- arm_cortexM4b_math.lib (Big endian on Cortex-M4)
- arm_cortexM3l_math.lib (Little endian on Cortex-M3)
- arm_cortexM3b_math.lib (Big endian on Cortex-M3)
- arm_cortexM0l_math.lib (Little endian on Cortex-M0)
- arm_cortexM0b_math.lib (Big endian on Cortex-M3)

The library functions are declared in the public file <arm_math.h> which is in the Include folder. 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 the Cortex-M4/M3/M0 with little endian and big endian. The same header file is used for floating point unit (FPU) variants. Define the appropriate pre-processor MACRO ARM_MATH_CM4 or ARM_MATH_CM3 or ARM_MATH_CM0, this will depend on the target processor in the application.

Kinetis CMSIS 2.10 installer file can be downloaded from: http://compass.freescale.net/go/cmsis

3 Freescale MQX RTOS

The Freescale MQX Real-Time Operating System (RTOS) provides real-time performance within a small, configurable footprint. This RTOS allows you to configure and balance code size with performance requirements. The easy-to-use API and out-of-box experience ensures that first-time RTOS users can start developing their application on the day software is installed. For experienced OS developers, it is easy to migrate legacy application code to a Freescale MQX-based platform. The RTOS is tightly integrated with the latest ColdFire® processors from Freescale and provided with commonly used device drivers. Powerful design and development tools are integrated with CodeWarrior™ tools to provide additional profiling and debugging capability.

The Freescale MQX RTOS has modern component-based microkernel architecture that allows for customization by feature, size, and speed, allowing engineers to select the components they want to include while meeting the tight memory constraints of embedded systems.

3.1 Key benefits

- Small code density – The Freescale MQX RTOS is designed for speed and size efficiency in embedded systems. The RTOS delivers true real-time performance with context switch and low-level interrupt routines hand-optimized in assembly. It can be configured to take as little as 12 KB of ROM and 2.5 K RAM on the CFV2 including the kernel, two task applications, 1 LW semaphore, interrupt stack, queues, and memory manager.
- Component-based architecture – Provides a fully-functional RTOS core with optional services. Freescale MQX RTOS includes 25 components, eight are core components, and 17 are optional. Components are linked only if needed. This prevents unused functions from bloating the memory footprint.
- Full and lightweight components – Key components are included in both full and lightweight versions for further size control, RAM and ROM utilization, and performance options. These components include — lightweight semaphores, events, timers, logs, and a memory component.
- Real-time, priority-based preemptive, multi-threading – In RTOS, threads execute in order of their priority. If a high-priority thread becomes ready to run, it can, and within a small and bounded time interval take over the CPU from any lower-priority thread that may be executing. Moreover, the high-priority thread can run uninterrupted until it has finished what it needs to do. This approach known as priority-based preemptive scheduling, allows high-priority threads to meet their deadlines consistently, no matter how many other threads are competing for CPU time.
- Optimized for Freescale architecture – Optimized assembly code to accelerate key real-time portions of the RTOS, such as context switching.
- Scheduling – Freescale MQX RTOS provides faster development time by relieving engineers from creating or maintaining an efficient scheduling system and interrupt handling. It is also significantly useful if multiple communication protocols like USB or TCP/I are required.
- Code Reuse – Freescale MQX RTOS provides a framework with a simple API to build and organize the features across Freescal's broad portfolio of embedded processors.
- Intuitive API – Writing code for Freescale MQX RTOS is straight forward with a complete API and available reference documentation.
- Fast boot sequence – A fast boot sequence ensures the application is running quickly after the hardware has been reset
- Simple Message Passing – Messages can be passed from either a system pool or a private pool, can be sent with either an urgent status, or a user define priority, and can be broadcast or task specific. For maximum flexibility, a receiving task can operate on either the same CPU as the sending task or on a different CPU within the same system.

3.2 Download Freescale MQX RTOS

To download the MQX installer:

1. Visit www.freescale.com and register with Freescale to get the required access.
2. Go to http://www.freescale.com/mqx, login, and download the MQX installer.

3.3 Install Freescale MQX RTOS

To install Freescale MQX RTOS:

1. Execute the installation wizard. The default settings install Freescale MQX in the path: C:\ProgramFiles \Freescale \Freescale MQX 3.x
2. You must read the FSL_MQX_release_notes.pdf file that has important information about the source code and example code delivered in the release.

3.4 Setup Freescale MQX RTOS

Freescale MQX has a directory structure. The detailed information about this structure can be found in the FSL_MQX_release_notes.pdf file. The developer can change the board support package (BSP) and platform support package (PSP) settings using the user_config.h file. Adding and removing flags in this file enables you to add or remove components, change settings for the different drivers, and add or remove logging settings. After a modification is done in the user_config.h, it is necessary to recompile the libraries.

4 CMSIS DSP Library Examples

This application note is based on the Freescale MQX RTOS Rev.3.7, IAR Workbench Rev 6.21, and the TWR-K40X256 board. It uses four tasks named main_task (auto start task with high priority), triangle_task, matrix_task, and the fft_task (the last three tasks have the same priotiry). The default task stack size is 1000 bytes. The three tasks (triangle_task, matrix_task and fft_task) show how to use the CMSIS DSP library basic math functions, matrix functions, and transforms functions.

4.1 Add CMSIS DSP library to the MQX project

At this point you can use any MQX project in IAR or an example project, for Freescale MQX 3.7 MQX examples search on www.freescale.com.

This case uses a “ hello ” project that can be found at this path …\Freescale\Freescale MQX 3.7 \mqx \examples\hello. The following steps show how to add the CMSIS DSP library to the MQX project.

1. Open hello_twrk40x256.eww workspace.
2. Add CMSIS DSP library file to hello project.
3. Add CMSIS DSP library related header file path to hello project by going to the menu Project->Option. On the C/C++ Compiler and select the tab Preprocessor. Then add paths to, Additional include directories. Below the paths and for reference, add a CMSIS DSP library Rev.2.10 installation path. C:\Program Files\Freescale\CMSIS 2.1 for Freescale Kinetis MCUs\KINETIS_CMSIS_2.10\CMSIS\Include C:\Program Files\Freescale\CMSIS 2.1 for Freescale Kinetis MCUs\KINETIS_CMSIS_2.10\Device\FSL \MK40DZ10\Include

4. Enable the CMSIS DSP library on IAR by going to menu Project->Options. On General Options select the tab Library Configuration. Then check the Use CMSIS checkbox, and afterwards the DSP library checkbox. For more details see Figure 3.

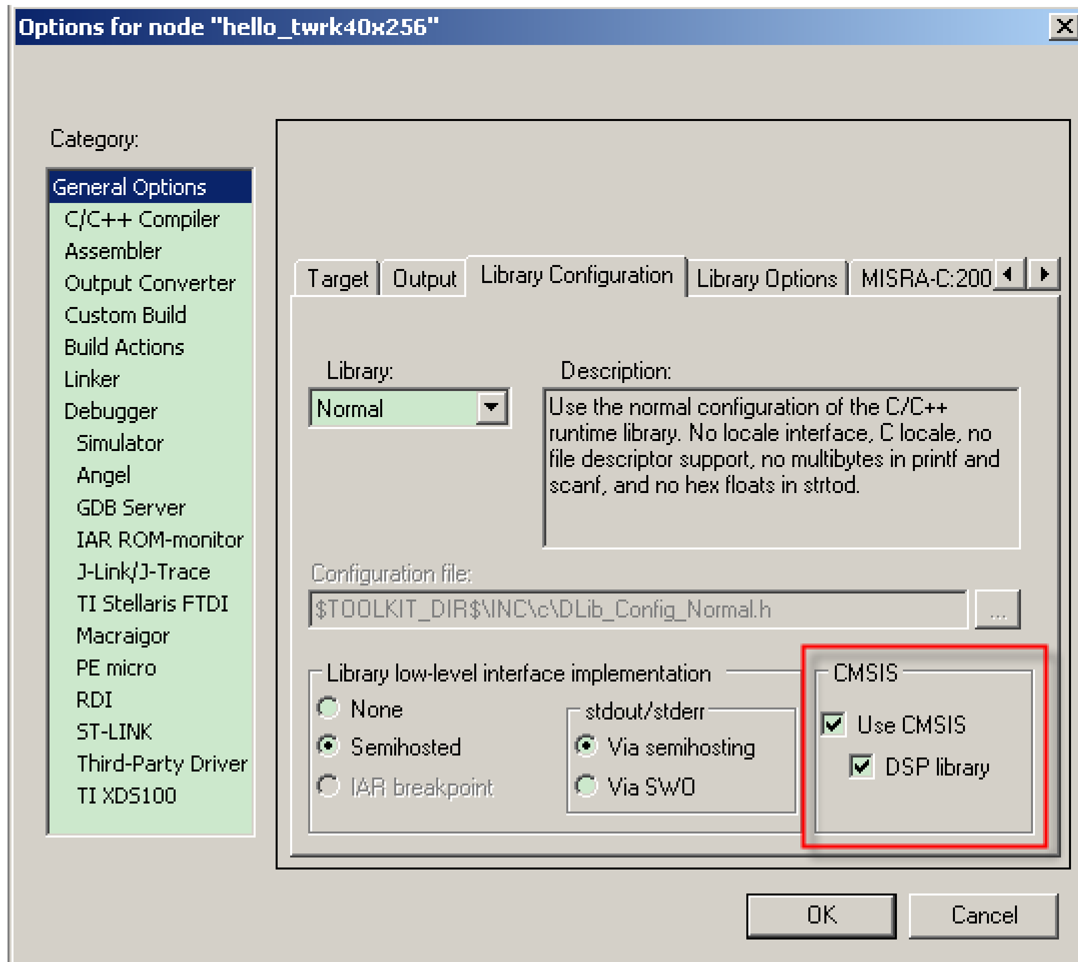

Figure 3. Enable CMSIS DSP library on IAR

5. Add #include <arm_math.h> in the hello.c file. You can now use CMSIS DSP library functions.

4.2 CMSIS DSP library basic math functions example (triangle_task)

This example demonstrates sin and cos calculation for the input signal. The trigonometric function equation, $\sin^2(x) + \cos^2(x) = 1$, whatever the input x value is, the calculation must equal 1. It uses 32-bit floating point numbers, adds multiple calculation functions, and sin(x), cos(x) functions.

The example uses 32-bit floating point type variables, such as float32_t cosOutput;

The code below shows input x in radian to obtain the related cos(x) value:

```
cosOutput = arm_cos_f32(testInput_f32[i]);
```

API introduction:

```
float32_t arm_cos_f32  ( float32_t  x )
```

Fast approximation to the trigonometric cosine function for floating-point data.

Parameters:
    [in] x input value in radians.

Returns:
    cos(x)

The code below shows the obtained $\sin^2(x)$ value:

```
        arm_mult_f32(&sinOutput, &sinOutput, &sinSquareOutput, 1);
```

API introduction:

```
void arm_mult_f32  ( float32_t *  pSrcA,
  float32_t *  pSrcB,
  float32_t *  pDst,
  uint32_t  blockSize
 )
```

Floating-point vector multiplication.

Parameters:
    [in] *pSrcA points to the first input vector
    [in] *pSrcB points to the second input vector
    [out] *pDst points to the output vector
    [in] blockSize number of samples in each vector

Returns:
    None

You can get a detailed example code at the related software <triangle.c> file.

4.3 CMSIS DSP library matrix functions example (matrix_task)

This example will demonstrate matrix calculation. Using matrix equal: $(AxB)^{\tau} = B^{\tau}xA^{\tau}$ to check matrix calculation. There are two matrix named A and B, A is 3X2 matrix, B is 2X3 matrix, AB will be 3X3 matrix, (AB)T is 3X3 matrix.

The example uses 32-bit floating point structure variables, such as the arm_matrix_instance_f32 A. The arm_matrix_instance_f32 is the CMSIS DSP library instance structure for the floating-point matrix.

uint16_t numRows

uint16_t numCols

float32_t * pData

The code below shows how to initialize Matrix A:

```
const float32_t A_f32[6] = { 1.6, 2.7, 0.1, 1.6, -3.6, -4.3 };  //Matrix A data
          srcRows = 3;
          srcColumns = 2;
          arm_mat_init_f32(&A, srcRows, srcColumns, (float32_t *)A_f32);
```

The initialization matrix A as 3X2 matrix below:

$$\begin{bmatrix} 1.6 & 2.7 \\ 0.1 & 1.6 \\ -3.6 & -4.3 \end{bmatrix}$$

API introduction:

```
void arm_mat_init_f32  ( arm_matrix_instance_f32 *  S,
  uint16_t  nRows,
  uint16_t  nColumns,
  float32_t *  pData
 )
```

Floating-point matrix initialization.

Parameters:
[in,out] *S points to an instance of the floating-point matrix structure
[in] nRows number of rows in the matrix
[in] nColumns number of columns in the matrix
[in] *pData points to the matrix data array

Returns:
None

The code below shows how to calculate matrix multiplication. Matrix AB is matrix A multiply matrix B result.

```
status = arm_mat_mult_f32(&A, &B, &AB);
```

API introduction:

```
arm_status arm_mat_mult_f32  ( const arm_matrix_instance_f32 *  pSrcA,
  const arm_matrix_instance_f32 *  pSrcB,
  arm_matrix_instance_f32 *  pDst
 )
```

Floating-point matrix multiplication.

Parameters:
[in] *pSrcA points to the first input matrix structure
[in] *pSrcB points to the second input matrix structure
[out] *pDst points to output matrix structure

Returns:
The function returns either ARM_MATH_SIZE_MISMATCH or ARM_MATH_SUCCESS based on the outcome of size checking.

The code below shows how to transpose matrix.

```
status = arm_mat_trans_f32(&A, &AT);
arm_status arm_mat_trans_f32  ( const arm_matrix_instance_f32 *  pSrc,
  arm_matrix_instance_f32 *  pDst
 )
```

Floating-point matrix transpose.

Parameters:
[in] *pSrc points to the input matrix
[out] *pDst points to the output matrix

Returns:
The function returns either ARM_MATH_SIZE_MISMATCH or ARM_MATH_SUCCESS based on the outcome of size checking.

Matrix calculation result:

Matrix A is initialization as 3X2 matrix below:

$$\begin{bmatrix} 1.6 & 2.7 \\ 0.1 & 1.6 \\ -3.6 & -4.3 \end{bmatrix}$$

Matrix B is initialization as 2X3 matrix below:

$$\begin{bmatrix} -2.0 & 3.0 & 1.6 \\ -4.3 & 0.73 & -3.6 \end{bmatrix}$$

The actual calculation result of Matrix A multiple matrix B transpose below:

$$\begin{bmatrix} -14.81 & -7.08 & 25.69 \\ 6.771 & 1.468 & -13.939 \\ -7.16 & -5.6 & 9.72 \end{bmatrix}$$

Using CMSIS DSP libray matrix function, Matrix A multiple matrix B transpose calculation result below:

$$\begin{bmatrix} -14.8100004 & -7.0800004 & 25.69000024 \\ 6.77100038 & 1.46800017 & -13.93989991 \\ -7.15999984 & -5.5999999 & 9.72000122 \end{bmatrix}$$

Using CMSIS DSP libray matrix function, Matrix B transpose multiple matrix A transpose calculation result below:

$$\begin{bmatrix} -14.8100004 & -7.0800004 & 25.69000024 \\ 6.77100038 & 1.46800017 & -13.93989991 \\ -7.15999984 & -5.5999999 & 9.72000122 \end{bmatrix}$$

The test result shows using the CMSIS DSP library matrix $(AxB)^T$ is the same with $B^T x A^T$.

Using the CMSIS DSP library calculation result with the actual calculation result, the difference is below 0.000001.

Obtain detailed example code at this application note related software <matrix.c> file.

4.4 CMSIS DSP Library transforms functions example (fft_task)

The Fourier transform is a mathematical operation that decomposes a signal into its constituent frequencies. The original signal depends on time, and therefore is called the time domain representation of the signal, whereas the Fourier transform depends on frequency and is called the frequency domain representation of the signal. The term Fourier transform refers both to the frequency domain representation of the signal and the process that transforms the signal to its frequency domain representation.

This example uses the CMSIS DSP Library complex FFT functions. Complex Fast Fourier Transform(CFFT) and Complex Inverse Fast Fourier Transform(CIFFT) is an efficient algorithm to compute Discrete Fourier Transform(DFT) and Inverse Discrete Fourier Transform(IDFT). This example uses 1024 points and a 32-bit floating point sin wave as an input signal, using Complex Fast Fourier Transform(CFFT) functions to transform the input signal from the time domain to frequency domain. It then uses the Complex Inverse Fast Fourier Transform(CIFFT) function to transform the previous frequency signal back to the time domain. After, compare the signal after FFT-IFFT with the original signal.

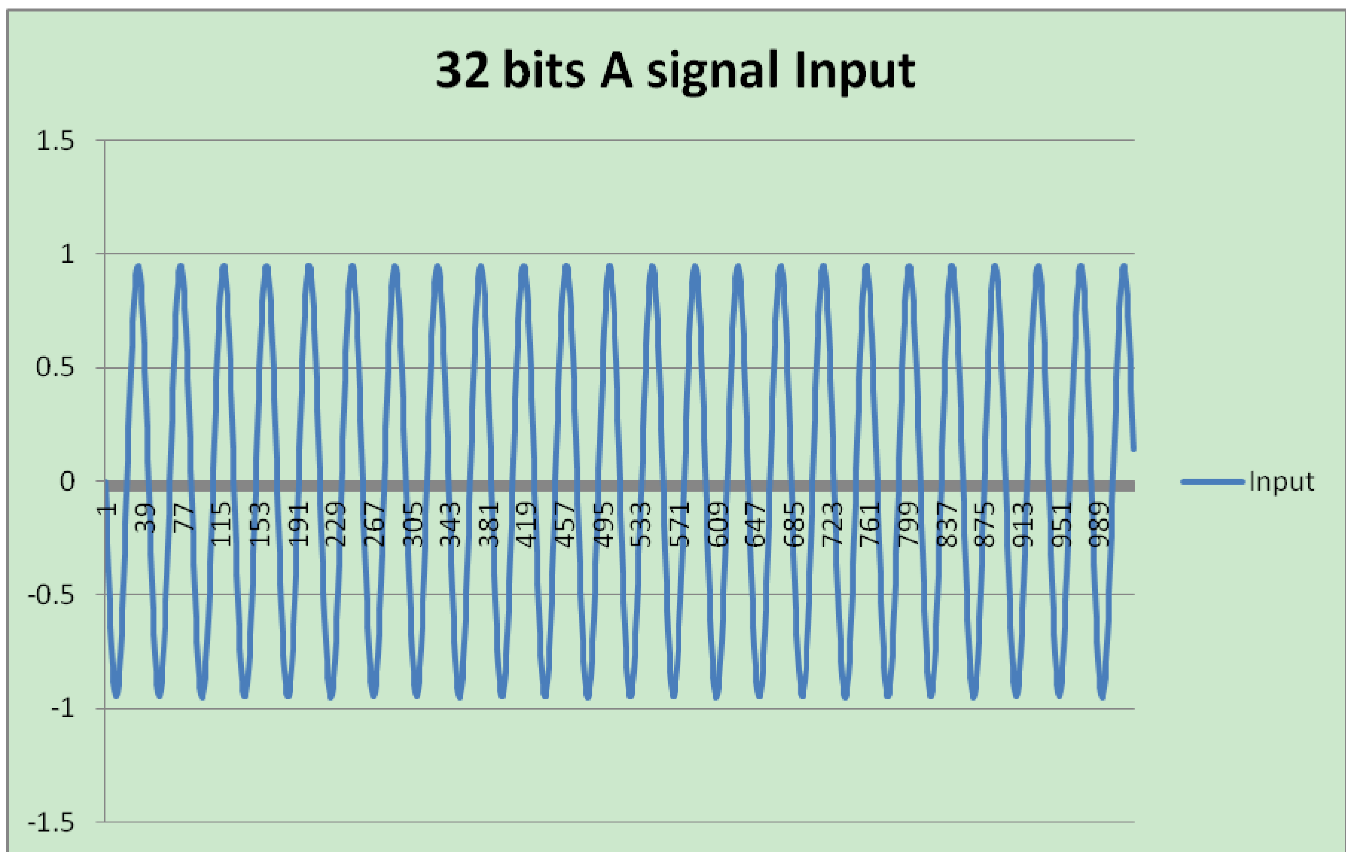

Figure 4. 32-bit 1024 point input signal A

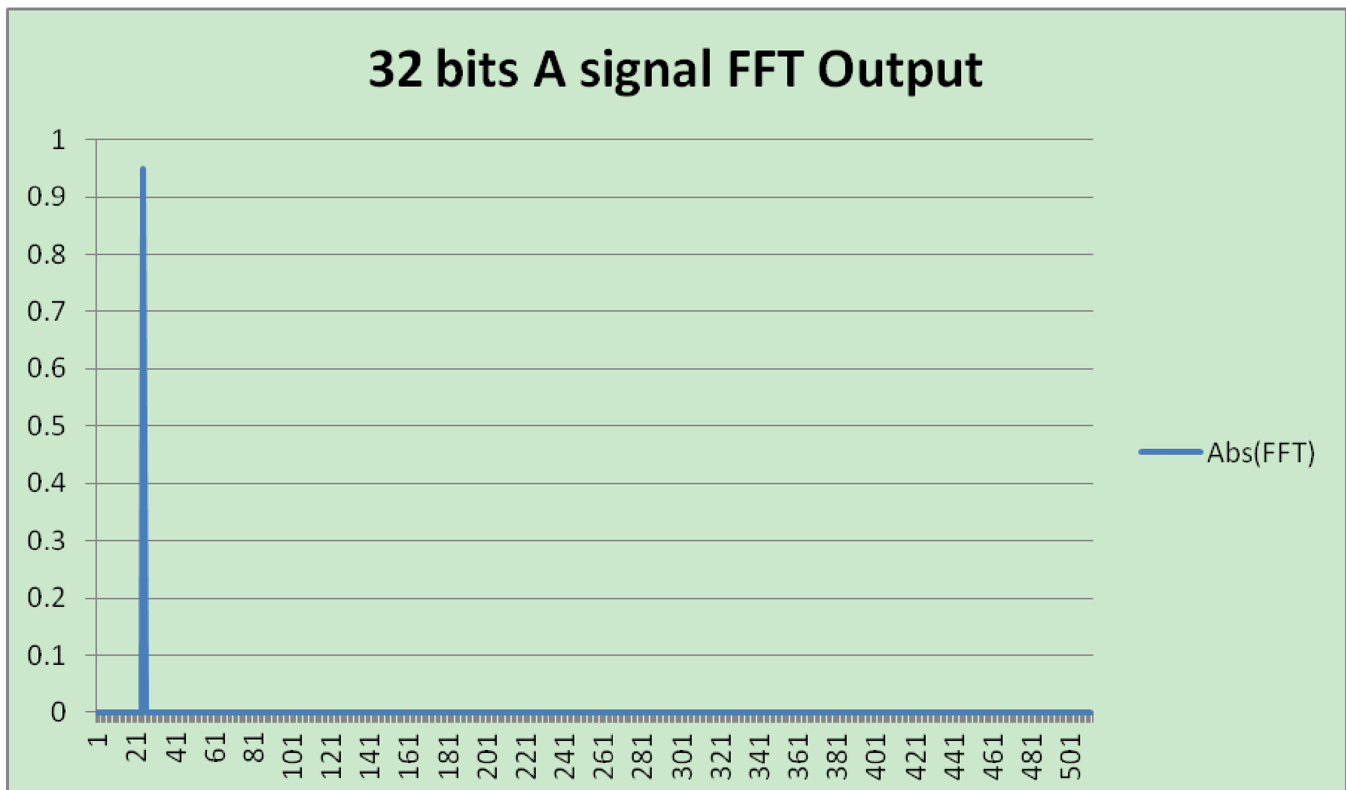

Figure 5. 32-bit A signal FFT output

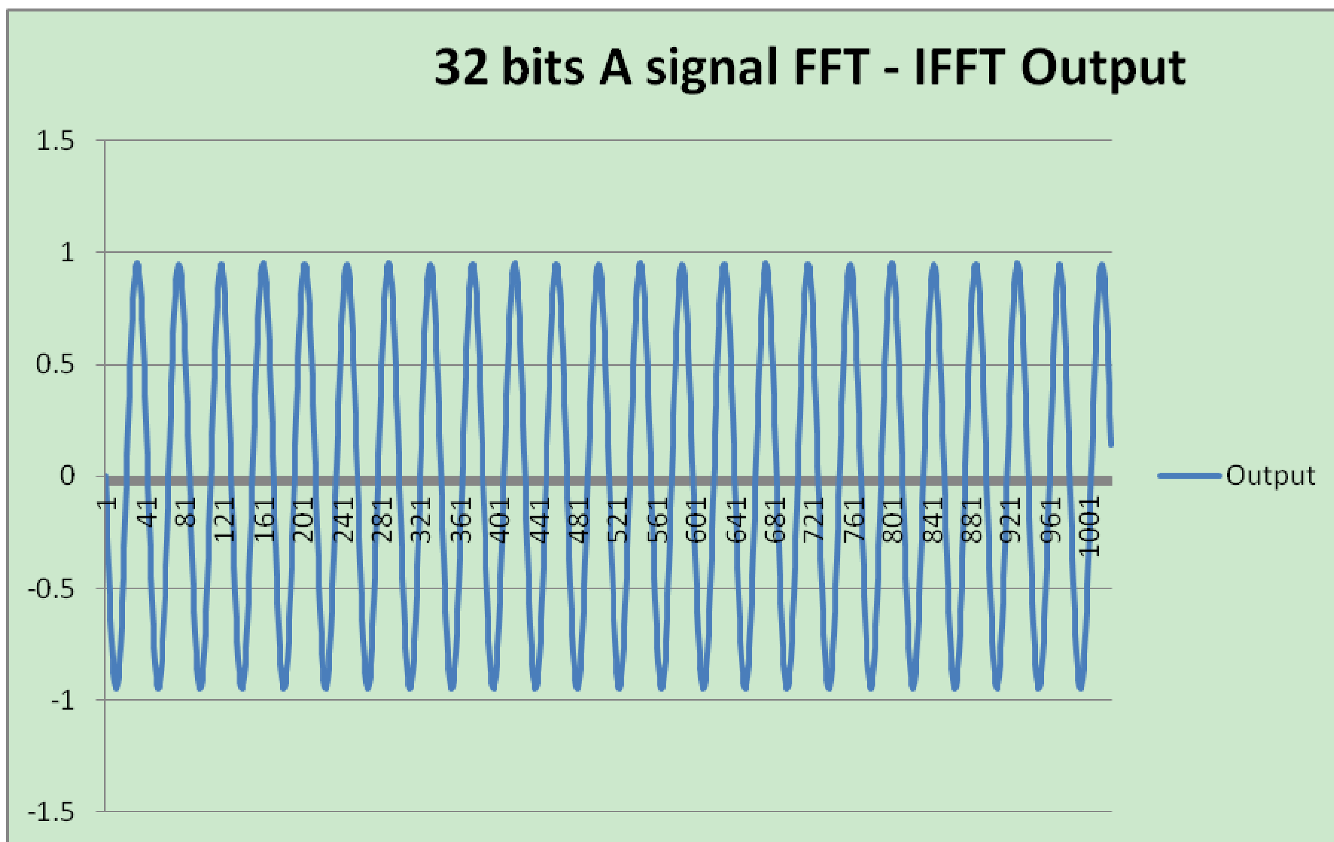

Figure 6. 32-bit A signal FFT-IFFT output

The CMSIS DSP library complex FFT functions operate on blocks of input and output data, each call to the function processes 2*fftLen samples through the transform.

The code below shows how to copy the input value to the fft input buffer.

```
#define MAX_BLOCKSIZE   2048
arm_copy_f32(testInputA_f32,  A, MAX_BLOCKSIZE/2);
API introduction:
void arm_copy_f32  ( float32_t *  pSrc,
  float32_t *  pDst,
  uint32_t  blockSize
 )
```

Copies the elements of a floating-point vector.

Parameters:
  [in] *pSrc points to input vector
  [out] *pDst points to output vector
  [in] blockSize length of the input vector

Returns:
  None

The code below shows how to intialize the CFFT function to compute a 1024 point FFT and transform input A signals from time domain to frequency domain.

```
#define MAX_BLOCKSIZE   2048
status = arm_cfft_radix4_init_f32(cfft_instance_ptr, MAX_BLOCKSIZE/2, 0, 1);
arm_cfft_radix4_f32(cfft_instance_ptr, A);
```

API introduction:

```
arm_status arm_cfft_radix4_init_f32  ( arm_cfft_radix4_instance_f32 *  S,
  uint16_t  fftLen,
  uint8_t  ifftFlag,
  uint8_t  bitReverseFlag
 )
```

Initialization function for the floating-point CFFT/CIFFT.

Parameters:

[in,out] *S points to an instance of the floating-point CFFT/CIFFT structure.
[in] fftLen length of the FFT.
[in] ifftFlag flag that selects forward (ifftFlag=0) or inverse (ifftFlag=1) transform.
[in] bitReverseFlag flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output.

Returns:

The function returns ARM_MATH_SUCCESS if initialization is successful or ARM_MATH_ARGUMENT_ERROR if fftLen is not a supported value.

Description:

The parameter ifftFlag controls whether a forward or inverse transform is computed. Set(=1) ifftFlag for calculation of CIFFT or otherwise CFFT is calculated
The parameter bitReverseFlag controls whether output is in normal order or bit reversed order. Set(=1) bitReverseFlag for output to be in normal order otherwise the output is in bit reversed order.
The parameter fftLen specifies the length of CFFT/CIFFT process. Supported FFT lengths are 16, 64, 256, 1024.
This function also initializes the twiddle factor table pointer and bit reversal table pointer.
void arm_cfft_radix4_f32 (const arm_cfft_radix4_instance_f32 * S, float32_t * pSrc)
Processing function for the floating-point CFFT/CIFFT.

Parameters:

[in] *S points to an instance of the floating-point CFFT/CIFFT structure.
[in,out] *pSrc points to the complex data buffer of size 2*fftLen. Processing occurs in-place.

Returns:

None

The code below shows how to initialize the CIFFT function to compute 1024 point A signal FFT output and transform it back to the time domain. The called functions are the same with the previous CFFT, and different at the third parameter: [in] ifftFlag flag that selects forward (ifftFlag=0) or inverse (ifftFlag=1) transform.

```
#define MAX_BLOCKSIZE   2048
status = arm_cfft_radix4_init_f32(cfft_instance_ptr, MAX_BLOCKSIZE/2, 1, 1);
arm_cfft_radix4_f32(cfft_instance_ptr, A_FFT);
```

Obtain detailed example code at this application note related software <fft.c> file.

4.5 MQX tasks scheduling

This application note creates four tasks, main_task (auto start task with high priority), triangle_task, matrix_task, fft_task (last three tasks with same priority) and uses FIFO task scheduling. FIFO is the default scheduling policy. In FIFO scheduling, the task that runs (becomes active) next is the highest-priority task that has been waiting the longest time.

The active task runs until any of the following occurs:

- The active task voluntarily relinquishes the processor, because it calls a blocking MQX function.
- An interrupt occurs that has higher priority than the active task
- A task that has higher priority than the active task becomes ready

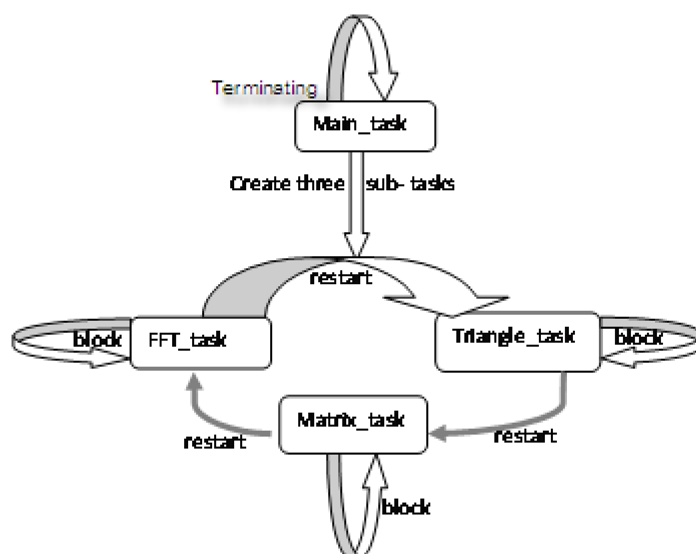

Figure 7. task scheduling

The Main_task creates three sub-tasks (triangle_task, matrix_task, fft_task) with the same priority, after that main_task terminates itself the three ready sub-tasks start to run. The three sub-tasks run in a circle, each task runs and blocks itself until another task restarts it. For detailed scheduling tasks see Figure 7.

The Freescale MQX provides at task-aware debugging tool. The task-aware debugging is an advanced kernel analysis tool that allows you to gain greater visibility into the embedded system. Developers can obtain detailed data about system performance enabling optimization work that can reduce potential performance bottlenecks in their embedded application

.

Key features:

- Task summary display
- Task stacks display
- Task semaphore display
- Task ready List display
- Tasks queues display
- Memory pools display
- Memory blocks display
- Memory partitions display
- Semaphore display
- Mutexes display
- Events display
- Logs display
- IO devices display
- Interrupts display

Figure 8 shows the task summary display while the CMSIS DSP library tasks run.

A	I	Name	ID	TD	Priority	State	Task Error Code
		_mqx_idle_task	0x10001	0x20000f5c	10	Ready	OK (0x0000)
		main	0x10002	0x2000111c	8	Blocked	OK (0x0000)
		triangle	0x10003	0x200015cc	9	Blocked	OK (0x0000)
		matrix	0x10004	0x20001a7c	9	Ready	OK (0x0000)
		fft	0x10005	0x20001f2c	9	Active	OK (0x0000)
		NO TASK					

Task List

Figure 8. TAD tool task list

Tasks are the key building blocks of an RTOS. The Freescale MQX RTOS task states include four logical states:

- Blocked
- Active
- Ready
- Terminated

The first state is Blocked. This means that the task is not ready and is waiting on something. It may be waiting on a signal to occur, a timeout to expire, or a condition to become true. Another state a task can be in is the active state. This means the task is ready and is currently in control of the processor because it is the highest priority ready task.

The Ready state happens when a task can run, but it is not the active task because it is not the highest priority task.

A task can be in the Terminated state. This means a task has finished all its work and no longer exists, or it was explicitly destroyed.

The Active task can go into the terminated state if the task finishes its work or becomes explicitly terminated. If it calls a blocking call, then it goes into the blocked state. This means it is now waiting for some condition to become true.

Finally, if a higher priority task becomes ready, its time expires, or an interrupt occurs, then the active tasks go into a ready state. At that point a context switch occurs, and a task that was in the ready queue gets promoted to being the active task. If a task becomes unblocked because that condition became true, then it goes into the ready state and the schedule determines if it is the highest priority task that could be running, and does a context switch if that is the case.

When a task is first created, it goes into the ready state. Figure 9 shows the visual representation of the different task states.

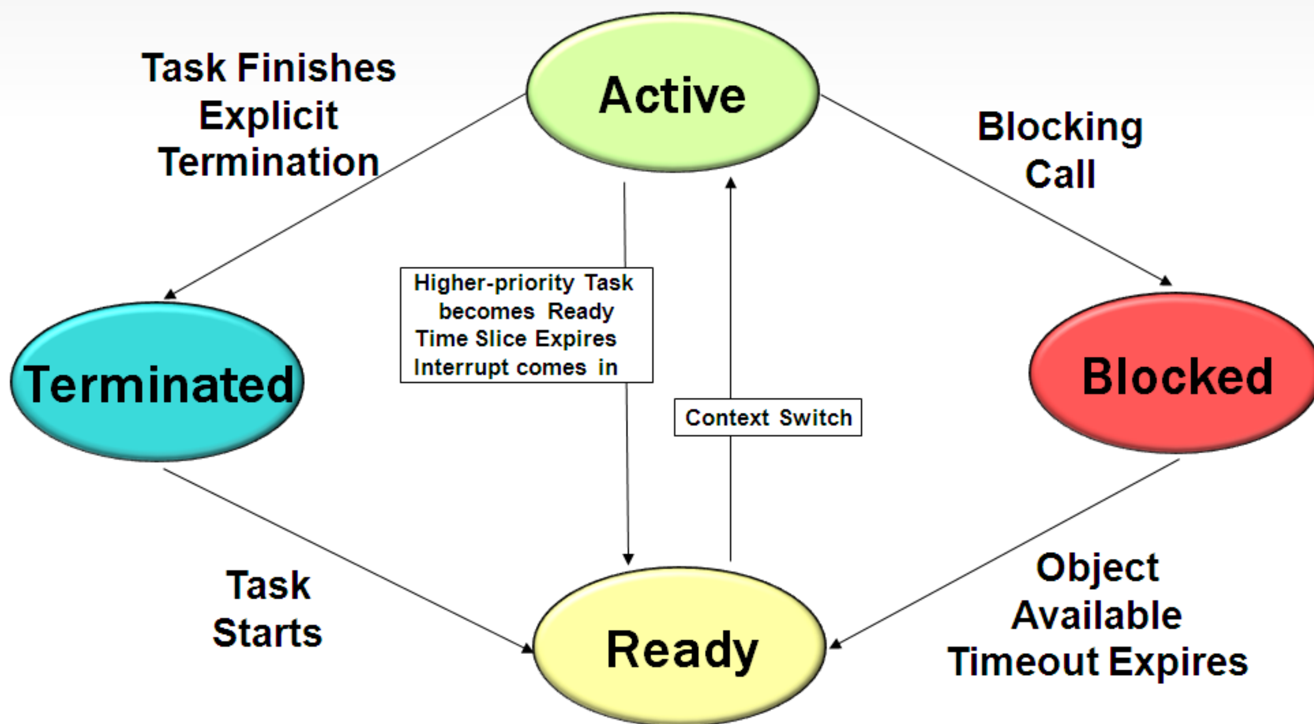

Figure 9. Task states

The Main_task is an auto start task. When the MQX starts, the main_task is in an active state. Then the main_task will create three sub-tasks with low priority. It calls the _task_destory() function to terminate the main_task and the three sub-tasks start to run.

Using the Task-Aware Debugging tool, you can find the main_task state changes from Active to Terminating after the _task_destroy(main_id); code is executed.

Figure 10 shows the main_task state is Active before _task_destroy(), and the other three sub-tasks in Ready state.

Task Name	Task ID	TD	Priority	State	Task Error Code
_mqx_idle_task	0x10001	0x20000f5c	10	Ready	OK (0x0000)
main	0x10002	0x2000111c	8	Active	OK (0x0000)
triangle	0x10003	0x200015cc	9	Ready	OK (0x0000)
matrix	0x10004	0x20001a7c	9	Ready	OK (0x0000)
fft	0x10005	0x20001c6c	9	Ready	OK (0x0000)

Task Summary

Figure 10. main_task states 1

After executing _task_destroy(main_id); code, the main_task state becomes Terminated. The main_task is then removed from the Task-Aware Debugging tool task summary list. Figure 11 shows there is no main_task.

The function _task_destroy does the following for the task being destroyed:

- Releases memory resources that the task allocated with functions from the _mem and _partition families

- Closes all queues that the task owns and frees all the queued elements
- Releases any other component resources that the task owns

Prototype

_mqx_uint _task_destroy(_task_id task_id)

Parameters

task_id [IN] — One of the following: task IDs of the task to be destroyed

MQX_NULL_TASK_ID (destroy the calling task)

Returns

MQX_OK

MQX_INVALID_TASK_ID

Task Name	Task ID	TD	Priority	State	Task Error Code
_mqx_idle_task	0x10001	0x20000f5c	10	Ready	OK (0x0000)
triangle	0x10003	0x200015cc	9	Blocked	OK (0x0000)
matrix	0x10004	0x20001a7c	9	Ready	OK (0x0000)
fft	0x10005	0x20001c6c	9	Active	OK (0x0000)

Task Summary

Figure 11. main_task states 2

4.6 MQX tasks stack size management

During the MQX task creation phase it needs to allocate each task stack size. The customer can gauge each task stack size usage status and modification task stack size to enhance memory usage efficiency with the Task-Aware Debugging tool assistant.

Four tasks are created with default 1000 bytes stack size. Figure 12 shows the task stack usage.

Task	Stack Base	Stack Limit	Stack Used	% Used	Overflow?
_mqx_idle_task	0x20001100	0x20001020	0x2000109c	44 %	No
main	0x200015b0	0x200011c0	0x200014e8	19 %	No
triangle	0x20001a60	0x20001670	0x20001978	23 %	No
matrix	0x20001f10	0x20001b20	0x20001eac	9 %	No
fft	0x200023c0	0x20001fd0	0x200022c0	25 %	No
Interrupt	0x200008e0	0x200004e0	0x200008a0	6 %	No

Stack Usage Summary

Figure 12. TAD tool stack usage summary

The Matrix_task stack used is only 9%, the current 1000 bytes stack size is too big for the matrix_task. Modify matrix_task size to 300 byte to enhance memory use.

Modify matrix_task stack size at <CMSIS_dsp_lib_main.c> file:

```
   /* Task Index,           Function,          Stack,  Priority,  Name,       Attributes,
Param, Time Slice */
{ MATRIX_TASK,    matrix_task,      1000,   9,         "matrix",    0,
0,     0 },
```

Modify stack size to 100

```
{ MATRIX_TASK,    matrix_task,      300,   9,         "matrix",    0,
0,     0 },
```

Afterwards, re-compile the project and use the TAD tool to find the matrix_task stack used. Figure 13 shows the matrix_task stack used the percentage enhanced.

Task	Stack Base	Stack Limit	Stack Used	% Used	Overflow?
_mqx_idle_task	0x20001100	0x20001020	0x2000109c	44 %	No
main	0x200015b0	0x200011c0	0x200014e8	19 %	No
triangle	0x20001a60	0x20001670	0x20001978	23 %	No
matrix	0x20001c50	0x20001b20	0x20001bec	32 %	No
fft	0x20002100	0x20001d10	0x20002000	25 %	No
Interrupt	0x200008e0	0x200004e0	0x200008a0	6 %	No

Stack Usage Summary

Figure 13. TAD tool stack usage summary

5 Conclusion

Many customers use MQX for different applications, as well as motor control and other DSP applications. The ARM CMSIS DSP algorithms is a good tool for DSP development that supports MQX,.

The ARM CMSIS DSP library works easily with MQX RTOS. The CMSIS DSP software library API function description and parameters information in the CMSIS DSP library help file can be found in the Freescale CMSIS software installation ..\KINETIS_CMSIS_2.10\CMSIS\Documentation folder. This folder provides examples on how to use the CMSIS DSP software library. The Speex recorder demo is another Freescale application lab with the ARM CMSIS DSP library FIR filter function. It demonstrates how to use the QEDesign tool and the ARM CMSIS DSP library implementing a band pass filter from 600–2000 Hz in the Speex decoder output.

Freescale MQX RTOS includes Real-Time TCP/IP Communication Suite (RTCS), Freescale MQX File System (MFS), and Freescale MQX USB Host/Device Stack. The RTCS provides IP networking for Freescale MQX Software Solutions. Freescale MQX RTCS provides an assortment of TCP/IP networking application protocols and uses the Freescale MQX RTOS drivers for Ethernet and serial connectivity. The Freescale MQX File System (MFS) is an embedded FAT file system compatible with Microsoft Windows and MS-DOS file systems. It can format, read, write, and exchange files with any operating systems running a FAT-12, FAT-16, or FAT-32 file system. Freescale MQX USB Host/Device Stack supports USB class drivers and implementation designs.

For more detailed information about Freescale MQX RTOS software visit www.freescale.com\mqx link.

Freescale MQX RTOS and ARM CMSIS DSP library speeds up customer design to market. Use the Freescale provided Task-Aware Debugging tool to monitor Freescale MQX tasks, memory states, and optimize memory resource usage efficiency.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductors products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claims alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-complaint and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale's Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.

© 2012 Freescale Semiconductor, Inc.

Document Number: AN4489
Rev. 0, 3/2012

freescale semiconductor