



# Using ThreadX<sup>®</sup> With Freescale's MCF51MM, CodeWarrior IDE, and the EKG Medical Application Demo

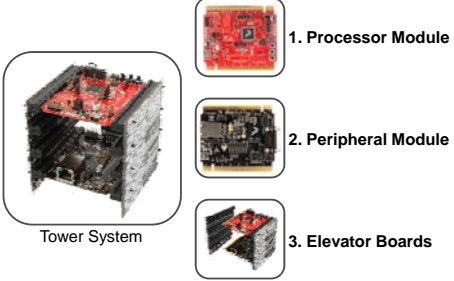
Presented at Freescale FTF  
June 23, 2010  
10:15AM – 11:15AM  
Tuscany F-H

## **Express Logic, Inc.**

11423 West Bernardo Court  
San Diego, CA 92127  
1-888-THREADX  
1-(858) 613-6640  
[www.rtos.com](http://www.rtos.com)



## Tower System Components




**1. Processor Module**

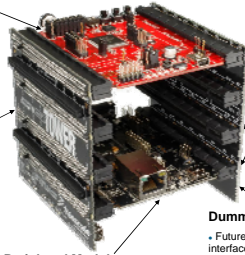
**2. Peripheral Module**

**3. Elevator Boards**

Tower System



## The Tower System



**Processor Module:**

- Tower controller board
- Works stand-alone or in Tower system
- Features new Open Source BDM (OSBDM) for easy programming and debugging via miniB USB cable

**Functional Elevator:**

- Common serial and expansion bus signals
- Two 2x80 connectors on backside for easy signal access and side-mounting board (e.g. LCD module)
- Power regulation circuitry
- Standardized signal assignments (e.g. UART, SPI, Timers, External Bus, I2C, CAN, GPIO, Ethernet, USB, etc.)

**Board Connectors:**


- Four card-edge connectors
- Uses PCI Express connectors (x16, 90mm/3.5" long, 164 pins)

**Dummy Elevator:**

- Future expansion for more serial interfaces and more complex MPU interfaces (e.g. RGB LCD, segment LCD, audio, enhanced Timer, etc.)
- "Dummy" shown with only GND connectivity. Used for structural integrity



**Peripheral Module:**

Standard peripheral boards compatible with all controller boards (e.g. Serial, Memory, etc.)



## Tower System Software


- CodeWarrior IDE
  - CW\_MCU\_6.3
- EKG Demo Application
  - EKG sensor data capture
  - Optional data simulation
  - A/D
  - Display
- ThreadX RTOS

© 2010 Express Logic, Inc. 9

## Express Logic's ThreadX RTOS


- Small, fast, easy-to-use RTOS for hard real-time applications. Widely used in Medical, Consumer, Industrial markets. FDA 501(k) and IEC-60601 approved.
- Footprint: 2KB – 15KB on MCF51MM
  - Automatically scales based on services used
- Speed: ~ 1-2µs context switch
  - Most services 50-200 cycles
- API: Intuitive, 66 functions in 8 categories



© 2010 Express Logic, Inc. 10

## ThreadX Services


- Threads
  - (aka Tasks) Semi-autonomous units of execution, unlimited in number, priority from 0-1024, unlimited threads at any priority
- Scheduling
  - Preemptive, priority-based, with round-robin or time-slicing for threads of equal priority
  - Preemption-Threshold™ to minimize context switches and avoid priority inversion
  - Sleep, Relinquish, Wait, Context Switch



© 2010 Express Logic, Inc. 11

## ThreadX Services

- Timers
  - Unlimited, one-shot, repeatable
- Message passing
  - Queues, send, receive, pend
- Semaphores/Mutexes
  - Priority inheritance optional
  - Inter-thread communication
- Memory management
  - Byte and block pool allocation



© 2010 Express Logic, Inc. 12

## ThreadX Technology

- Preemption-Threshold
  - Set lower limit for priority of preempting thread
  - See academic research: [http://www.cs.utah.edu/~regehr/reading/open\\_papers/preempt\\_thresh.pdf](http://www.cs.utah.edu/~regehr/reading/open_papers/preempt_thresh.pdf)
  - Useful for reducing number of context switches
  - Also useful for preventing priority inversion

© 2010 Express Logic, Inc.
13

## Preemption-Threshold Example

- Assume that "Low Thread" has a priority value of 20
- Assume that "Low Thread" has a preemption-threshold of 15
- N.B.: The highest priority value is 0

Priority

↑

0

15

20

"Low" can only be preempted by another thread with priority values 0 to 14

"Low" has a preemption-threshold of 15; thus "Low" cannot be preempted by another thread with a priority value less than or equal to 15

"Low" has a priority value of 20

© 2010 Express Logic, Inc.
14

## Test Case with/without Preemption-Threshold

- No Preemption-Threshold

The shaded area identifies priority inversion. Without Preemption-Threshold, High\_thread must wait for Medium\_thread, which has a lower priority.

- With Preemption-Threshold

With Preemption-Threshold, no priority inversion has been detected. High\_thread obtains the mutex without waiting for Medium\_thread.

© 2010 Express Logic, Inc.
15

## Preemption-Threshold Eliminates Priority Inversions

**Number of Priority Inversions For Test Case**

	Number of Non-Deterministic Priority Inversions
Without Preemption-Threshold	9
With Preemption-Threshold	0

In this test case, Preemption-Threshold eliminates both deterministic and non-deterministic priority inversions

© 2010 Express Logic, Inc.
16

## Preemption-Threshold Improves Efficiency

**Relative Time Between Mutex\_Get and Mutex\_Put Pairs for High\_thread**

	Average Time	Minimum Time	Maximum Time	Number of Get-Put Pairs
Without Preemption-Threshold	40.1	2	104	17
With Preemption-Threshold	2.0	2	2	52

In this test case, Preemption-Threshold significantly improves efficiency by reducing or eliminating the time High\_thread must wait for the mutex

© 2010 Express Logic, Inc.
17

## Other ThreadX Technology

- Picokernel Architecture
  - Non-layered implementation for size & speed
  - Deterministic processing, not affected by number of application objects
  - Automatic scaling
- Event chaining
  - Simplifies processing dependent on multiple events
  - Reduces number of threads required
- Performance metrics
  - Counts various system events and operations (context switches, etc.)
  - Execution Profile Kit
- File system, Network stack, USB stack host/device/OTG, Graphics
  - Full featured, fully integrated
- TraceX and StackX development tools
  - Innovative tools for real-time systems
- Optimized Interrupt Processing
  - Only scratch registers saved/restored if no preemption
  - No idle thread, hence no context save/restore when system is idle
  - Most of API available directly from ISRs
  - Optional timer thread or direct timer processing in ISR

© 2010 Express Logic, Inc.
18

## Performance (Thread-Metric)

Public Thread-Metric test suite, results reported by independent consultant (best in red)

	Cooperative Scheduling	Preemptive Scheduling	Interrupt Processing	Interrupt Preemption Processing	Message Processing	Synchronization Processing	Memory Processing
<b>ThreadX (3)</b>	<b>11,848,815</b>	<b>4,870,885</b>	<b>6,918,050</b>	<b>3,052,151</b>	<b>6,928,383</b>	<b>15,337,354</b>	<b>12,863,624</b>
µC/OS-II (1)	3,909,085	5,259,998	(1)	(3)	10,293,318	6,814,817	
TNKernel (1)	3,359,814	5,497,238	2,693,630	4,146,914	7,353,579	5,933,761	
AVA (1)	1,724,948	5,207,762	1,260,190	2,761,154	7,514,799	10,235,182	
FreeRTOS (1)	3,717,913	1,881,892	2,400,967	484,691	1,989,999	(2)	

1) Cooperative scheduling not supported  
 2) Not reported  
 3) µC/OS-II Message Processing test is not valid - copies 1 32-bit word instead of copying 4 32-bit words.

© 2010 Express Logic, Inc.
19

## ThreadX for CodeWarrior

- ThreadX delivered in 2 source code projects
  1. ThreadX Library (66 functions)
  2. Demo Application (with headers, processor-specific files, ThreadX object library, demo source)
- Build Library
  - Compiler settings (DEBUG, TRACE, ...)
- Build (Demo) Application
  - See standard demo and EKG demo
  - Reference Library
  - Compile, Link
- Download and Run/Debug

© 2010 Express Logic, Inc.
20

## Adapting an Existing Application

- Freescale EKG Demo for MCF51MM
  - Stand-alone code ("Bare Metal")
  - Uses "spin loops" for scheduling and delay
  - USB communication with Host
  - Interrupt Service Routines
- Run it under ThreadX
  - Make it a thread, or multiple threads
  - Replace loops with ThreadX services
  - Free up CPU cycles

© 2010 Express Logic, Inc.
21

## The Big Loop

- CPU time spent checking to see if any activity (thread) has work to do

© 2010 Express Logic, Inc.
22

## "RTOS-izing" Code

- Stand-alone code generally uses "event loops" to run functions

```

while(1) {
    if (condition_1) {
        function_1()
    }
    else if (condition_2) {
        function_2()
    }
    else if (condition_3) {
        function_3()
    }
    ...
    ...
    endif;
}
    
```

Time to evaluate each "condition\_n" expression plus decide and branch, adds up - delays response to "condition\_x". Plus, any new conditions or functions change timing of loop.

- With an RTOS
  - Run highest priority function (task/thread)
  - When that thread must wait for an event, suspend thread until "event" occurs
  - Enables other threads to get CPU cycles while waiting for event
- Event triggers interrupt. ISR processes event and calls scheduler
- Scheduler detects that waiting thread is now READY, and performs context switch to resume thread waiting for event
- Result is faster response to event plus better use of time between events

© 2010 Express Logic, Inc.
23

## Multithreading

- Enabling an activity to use the CPU while other activities don't need it - Event Loop

© 2010 Express Logic, Inc.
24

## "RTOS-izing" Code

- Stand-alone code often uses "Delay Loops"
 

```
for i=1, 1<10000, i++ {
    _____
    end;
}
```

Spin loop occupies CPU 100% for duration of delay period.
- Replace with call to tx\_thread\_sleep(n)
 

```
tx_thread_sleep (1);
```

Sleep call frees up CPU for other threads or for low-power operation.

  - Enables other (READY) threads to get CPU cycles
- In "n" timer ticks (1ms each, but can be any user-defined duration), suspended thread re-awakens
- Result is better use of delay time

**expresslogic** © 2010 Express Logic, Inc. 25

## Multithreading

- Enabling an activity to use the CPU while other activities don't need it – I/O Delay

The diagram shows a horizontal timeline. Thread A starts at the beginning. At a certain point, I/O starts. Thread A then enters a 'Thread A Waits' period. During this time, Thread B starts and runs. Once I/O finishes, Thread A resumes its execution.

**expresslogic** © 2010 Express Logic, Inc. 26

## Porting to an RTOS

- Start with "Bare Metal" code
- Add ThreadX library of services
- Convert Demo into a thread
- Add Initialization
- Add other threads as desired
- Build and run

**expresslogic** © 2010 Express Logic, Inc. 27

## Convert Demo Into Threads

- Add header files tx\_api.h, tx\_port.h, and tx\_tpm.h to the project.
- Add function files tx\_initialize\_low\_level.s, tx\_tpm.c (MCF51MM timer logic), and tx\_interrupt\_handlers.c (MCF51MM interrupt handlers) to the project.
- Add the ThreadX library tx.a to the project
- Add the following line to the Project.lcf file in order to identify the first unused RAM address for ThreadX:
 

```
__free_mem = __SP_AFTER_RESET;
```
- Modify main.c as follows:
  - Lines 40-68 define basic ThreadX data structures.
  - Lines 159 through 230 create the Test App thread and the Background Thread.
- Move all ISRs (from main.c, lines 800-928) into tx\_interrupt\_handlers.c
- Replace delay loops to use tx\_thread\_sleep instead of spinning. Spin code was changed in virtual\_com.c, in function usb\_init (lines 180-181), in function cdc\_putch (lines 652-673), and in Demo\_Measurement\_Engine.c, in function Display\_ME\_Results (lines 1029-1030).

**expresslogic** © 2010 Express Logic, Inc. 28

## Some Setup Items

```
/* Enable the ThreadX trace. */
tx_trace_enable(event_buffer, 4096, 10);
```

Event Buffer for trace events

```
/* Create the ThreadX thread for processing the main loop. */
tx_thread_create(&TestApp_Init_Thread, "Test App Thread", TestApp_Init_Thread_Entry,
0, pointer, 2048, 16, 16, 0, TX_AUTO_START);
```

Stack Size, Priority, Preemption-Threshold, Time Slice, Auto\_Start

```
/* Create the ThreadX background thread. */
tx_thread_create(&Background_Thread, "Background Thread", Background_Thread_Entry,
0, pointer, 512, 31, 31, 0, TX_AUTO_START);
```

**expresslogic** © 2010 Express Logic, Inc. 29

## Delay Loop

```
// Use TOD to delay 250ms
TODC_TODCLKS = 1;
TODC_TODPS = 0;
TODSC_QSECF = 1;
TODC_TODEN = 1;

while(TODSC_QSECF == 0){
    /* Use ThreadX to sleep and avoid wasting cycles. */
    tx_thread_sleep(1);
};
TODSC_QSECF = 1;
}
```

**expresslogic** © 2010 Express Logic, Inc. 30

## Let's See How it All Works

< live demo >

© 2010 Express Logic, Inc.

31

## Results Without ThreadX

© 2010 Express Logic, Inc.

32

## Results With ThreadX

© 2010 Express Logic, Inc.

33

## How ThreadX Operates

© 2010 Express Logic, Inc.

34

## RAM/ROM Measurements

- ROM
  - Demo (without ThreadX): 18KB
  - Demo (with ThreadX): 24KB
- RAM
  - Demo (without ThreadX): 1KB Stack
  - Demo (with ThreadX):
    - ThreadX RAM requirements: 416 bytes
    - Thread Control Blocks: 344 bytes
    - ThreadX Stacks: 2.5KB
    - TraceX Trace Buffer 4KB (user option)

© 2010 Express Logic, Inc.

35

## Benefits of An RTOS

- Reclaim CPU cycles – lower overhead
  - Polling keeps CPU at 100%
  - Sleep() reclaims almost all poll cycles
- Easily add new threads
  - Modular expansion through threads
- Trace real-time events
  - Use TraceX to view and profile system activity
- See Article, “Multitasking Mysteries”

© 2010 Express Logic, Inc.

36

## Medical Regulatory Requirements

**IEC-60601, IEC-62304, ISO-14971, FDA 510(k)**

- FDA/CDRH – Market Submission Compliance
  - Premarket Notification 510(k)
  - Premarket Approval
- IEC-62304
  - Class A/B/C
  - Normative requirement (software) of IEC-60601
- IEC-60601
  - Total Device Requirement
- ISO-14971
  - Normative requirement of IEC-60601
  - Normative requirement of IEC-62304

© 2010 Express Logic, Inc.
37

## An RTOS Helps Certification

- Complexity of developing and certifying bare metal code
  - Executive loop, low level memory management, etc
  - May be cheaper to use an RTOS and strip the application down
- Product roadmap
  - May not need an RTOS for v1 of a product
  - v2 may jump from a stand alone app to one requiring connectivity, etc.
  - Using an RTOS in v1 of the product eliminates massive rework for v2, v3, ...
- Predicate Usage Argument
  - Going from v1 with no RTOS to v2 with an RTOS would be considered more than "enhancing" a product
  - Reduces the argument for predicate usage of the device.
  - Second or third generation product's safety assumptions can be based on the incremental improvement model. i.e. if v1 was safe, and we only did a little to v2 and tested it, and it was safe then v3 must be safe also....)

© 2010 Express Logic, Inc.
38

## Meeting RTOS Requirements

- Addresses/eliminates/satisfies SOUP issues
- Independently validated
- Packaged for user submission with application
- Covers RTOS-related portion of product
- Express Logic Certification Pack™
  - For a specific regulatory standard
  - For a specific device classification (Class II or III)
  - For specific hardware and tools used in product
  - 100% Turnkey, Guaranteed acceptance

© 2010 Express Logic, Inc.
39

## What is a Certification Pack?

- Process and methodology documentation
  - Planning
  - Development
  - Requirements
  - Verification
  - Configuration management
  - Quality assurance
- Code
  - Source code, test code & scripts, object code
- Test
  - Code coverage and analysis
    - Unit/white-box, integration/black-box, acceptance testing
  - Plan for tool usage
- Results
  - Unit & integration test reports
  - Trace matrix under IBM Rational RequisitePro™

© 2010 Express Logic, Inc.
40

## Medical Certification Pack

IEC-60601, IEC-62304, FDA510(k), ISO-14971	
Software Safety Requirements Specification	Real Source Code unconditional use code
Software Configuration Management Plan	Unit Integration Test Scripts
Software Development Plan	Unit Integration Test Plans and Code
Software Quality Assurance Plan	Integration Test Procedures
Software Verification Plan	Integration Test Report
Hardware Source Code conditional use code	Integration Test Report Files
Software Requirements Document	Unit Software Unit Test Plan QALCC
Software Design Document	Unit Unit Test scripts
Integration Test Plans and Code	Unit Test Test Libraries prebuilt targets
Integration Test Scripts	Unit Test Report Files
Unit Test Plans and Code	Unit Test Procedures
Unit Test scripts	Unit Test Reports
Assembly Language Coding Standards	Safety Manual
C Language Coding Standards	Software Configuration Mgmt / Test Coverage Report
Software Design Standard	Software Life Cycle Environment Configuration (HyperSoftware CI)
Software Requirements Standard	Software Acceptance Summary
Code Review Procedures	Documentation Review Sheets
Documentation Procedures	Unit Review Sheets
Documentation Review Sheets	Product Review Checklist and Audit Reports
Code Review Sheets	Equivalent Pro objects and reports
Prebuilt Pro objects and reports	Unit Test & Unit Test Integration Results
Unit Software Safety Integration Plan	Unit review analysis configuration file
Unit Software Requirements Document	Complete Test Qualification Plan
Unit Software Design Document	Complete Test Qualification Plan

© 2010 Express Logic, Inc.
41

## Summary

- Select appropriate hardware and software for the job
  - "Less is More"
- Freescale MCF51MM
  - Range of ColdFire or Power Architecture processors
- Express Logic's ThreadX RTOS
  - Ideal for simple to moderate medical devices
  - Easily incorporate "bare metal" code
  - Certification Pack for FDA/IEC requirements

© 2010 Express Logic, Inc.
42

## Using ThreadX with the Stand-Alone MCF51MM Medical Demo

The use of an RTOS such as ThreadX can enable a more efficient utilization of CPU cycles, as opposed to polling loops or wait loops, and actually enable more work to be performed in the same period of time. Many programs, including the ECG Demo, utilize loops for delay, or to wait for an external condition to reach a desired state. ThreadX has basic services for “delay” and “suspend” that can perform the same functions as these loops, but in a manner that enables the CPU to do other work while the application is being delayed, or while it is waiting for some external event.

The stand-alone Flexix\_MM\_Demo\_Beta\_V2 demonstration effectively does all of its processing in a large polling loop inside of TestApp\_Init. The introduction of ThreadX allows this processing to be contained in a single thread (“Test App”) and allows the introduction of additional threads. In this example, we placed tx\_thread\_sleep calls where the demo software uses delay loops. This frees up considerable CPU time. In our example, we have added a “Background Thread” to use the free cycles.

We have identified 2 loops that are “delay loops,” and serve no other function. We have chosen a “sleep” period of 1ms, and we have added a “Background Thread” at a low priority that only runs if the “Test App” thread is suspended (in this case, sleeping). We have replaced these delay loops with calls to the ThreadX “sleep” service (“tx\_thread\_sleep()”), enabling ThreadX to allow our Background Thread to run until the higher priority Test App thread awakens 1 or 2ms later. We have traced the Test App thread and Background Thread activity using TraceX, to illustrate the CPU time we have recaptured.

A more comprehensive conversion of the demo to use ThreadX would involve replacing the loops that are waiting for an event with a “suspend” service call, and then to program the ISR for the event to trigger ThreadX to “resume” the demo. However, such an implementation would require a greater understanding of the structure and operation of the demo, and the periodic masking of interrupts. In the ECG Demo, there are interrupts at Level 7, which is non-maskable. Thus, the more comprehensive approach is not possible without major restructuring. Still, even the simplified approach used shows significant benefits.

### Adding ThreadX is Easy

Adding ThreadX was simple. The following steps were performed in order to adapt the standalone demo to use ThreadX:

1. Add header files tx\_api.h, tx\_port.h, and tx\_tpm.h to the project.
2. Add function files tx\_initialize\_low\_level.s, tx\_tpm.c (MCF51MM timer logic), and tx\_interrupt\_handlers.c (MCF51MM interrupt handlers) to the project.
3. Add the ThreadX library tx.a to the project
4. Add the following line to the Project.lcf file in order to identify the first unused RAM address for ThreadX:

```
___free_mem = ___SP_AFTER_RESET; # Mark ThreadX free memory here!
```

5. Modified main.c as follows:
  - a. Lines 40-68 define basic ThreadX data structures.
  - b. Lines 159 through 230 create the Test App thread and the Background Thread.
6. Moved all ISRs (main.c lines 800-928) into tx\_interrupt\_handlers.c
7. Replaced delay loops to use tx\_thread\_sleep instead of spinning. Spin code was changed in:
  - a. virtual\_com.c, in function usb\_init, lines 180-181;
  - b. Virtual\_c om.c, in function cdc\_putch (lines 652-673); and in
  - c. Demo\_Measurement\_Engine.c, in function Display\_ME\_Results (lines 1029-1030).

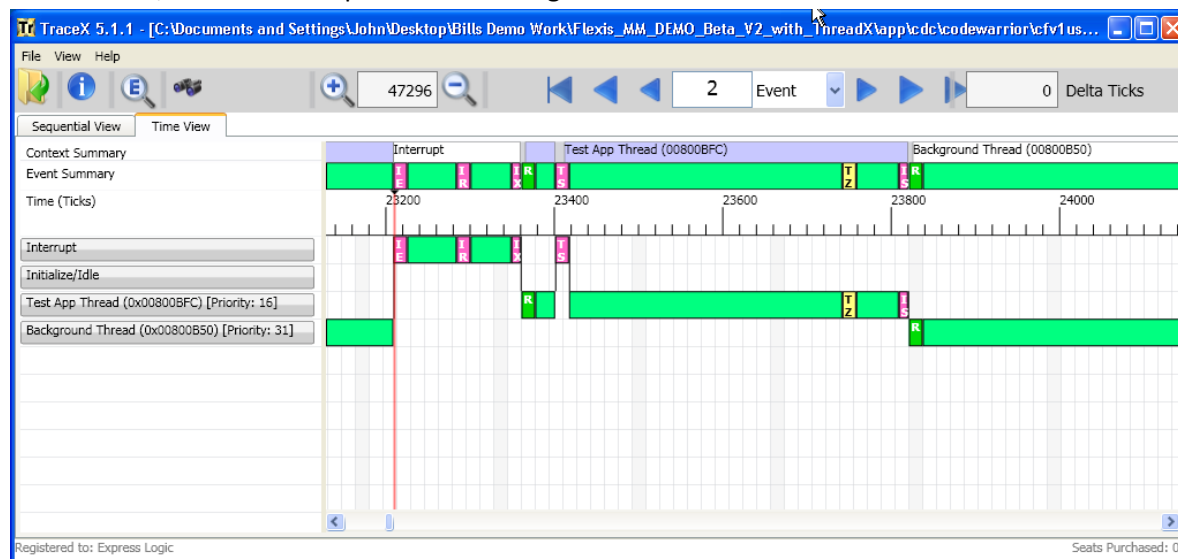
## Viewing the System Via TraceX

By default, event tracing is enabled in ThreadX and is captured in a trace buffer (defined in the ThreadX mods to main.c). The size of the trace buffer is up to the user and depends on how much memory is to be allocated for event trace information (the larger the memory the more events that can be stored). In this demo, we have allocated 4KB for the event trace buffer – enough to hold about 1,000 events. The address of the buffer is defined in `_tx_trace_control_header_ptr`. Add this variable to one of the "Data" windows in the debugger via a right-click and "Add Expression" operation. This will display the address of the event buffer. The demo system currently has the buffer at 0x801a68.

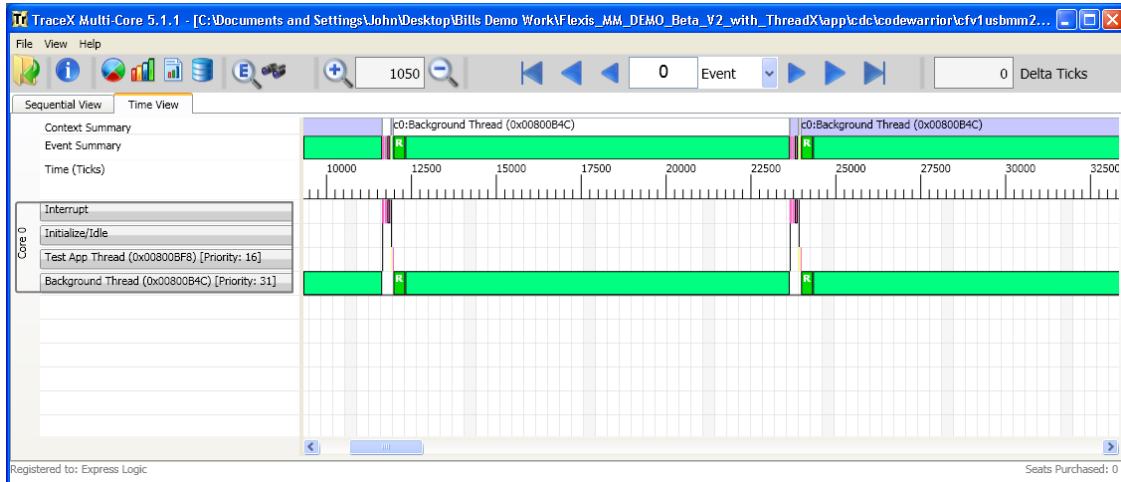
Exporting this trace buffer to the PC for display by TraceX can be done via the SAVE command entered in the command window:

```
> save 0x801a68..0x802a68 mm_demo_trace.trx
```

The trace file ("mm\_demo\_trace.trx") is saved in the "cfv1usbmm256" project directory. Then, launch TraceX and "Open" the trace file. This view shows the detailed sequence of events for each `tx_thread_sleep()` function, resulting in a timer interrupt, preemption of the background thread, resumption of the TestApp thread, and then the suspension of the TestApp thread for the next sleep function call, and the resumption of the Background thread.



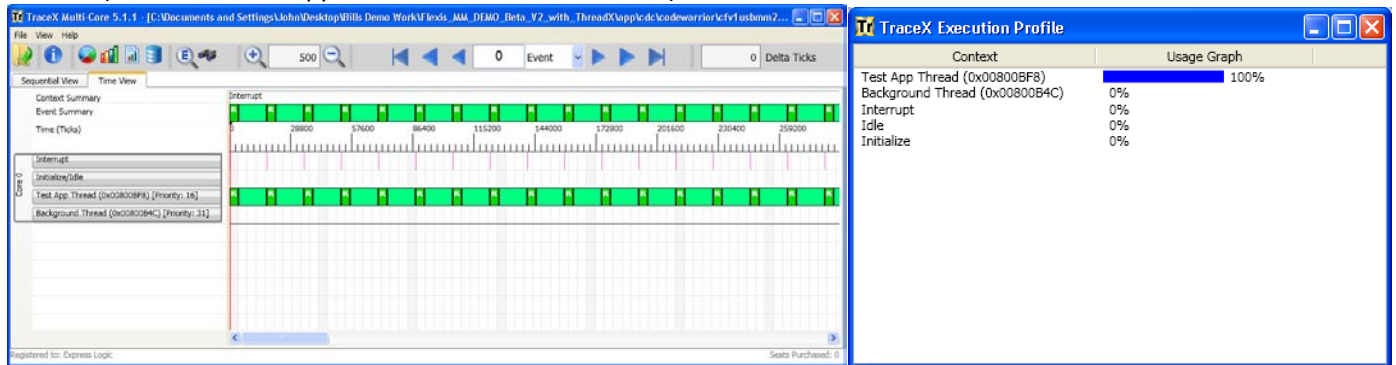
In this view, “zoomed-out,” we can see that the TestApp thread runs only for short bursts, then sleeps to allow the Background thread to run for the rest of the time.



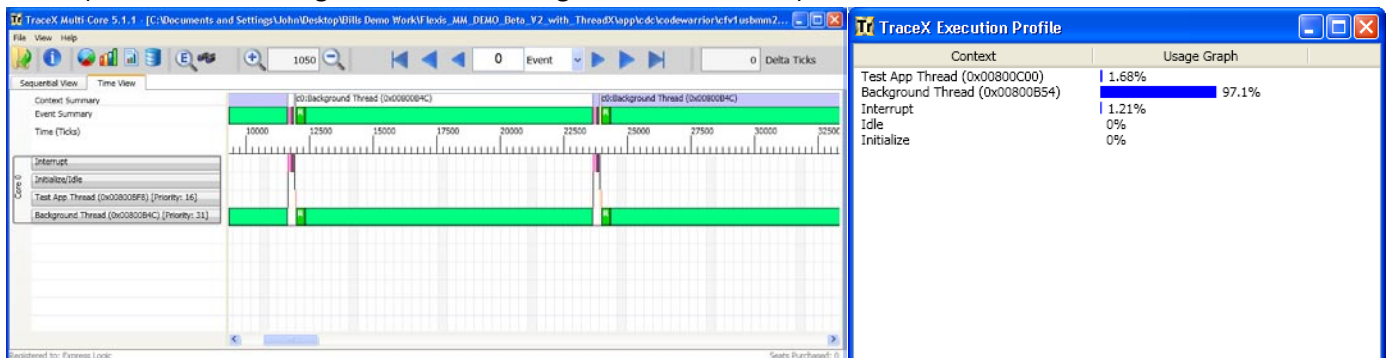
## Performance/Size Impacts

Using ThreadX greatly improves overall system performance by reducing the time spent in delay loop polling. In the execution of Demo 7, we have recaptured approximately 97% of all CPU time and made it available to the Background Thread. This time could also be used to run the processor in low-power mode, saving battery life in portable systems. The TraceX trace below shows the Before and After profile of the Test App thread and the Background Thread:

Before: (Note that Test App consumes 100% of the CPU)



After: (Note that the Background Thread now gets 97% of the CPU)



The demo code size without ThreadX is 18KB and ThreadX is roughly 6KB, for a total ROM of 24KB. The breakdown on RAM usage is:

	Before		After	
	ROM	RAM	ROM	RAM
Application	18KB	2KB	18KB	1KB
ThreadX requirements	N/A	N/A	6KB	416 bytes
2 thread control blocks				344 bytes
2 thread stacks				2.5KB
TraceX event buffer				4KB (size optional)
<b>TOTAL</b>	<b>18KB</b>	<b>2KB</b>	<b>24KB</b>	<b>8.3KB</b>

In this system, the stack size is almost a wash compared to the original code. The ISR stack size can be reduced since much of the stack usage is now in the thread's stack.

## Main (40-68)

```
*****//*
*
* @file main.c
*
* @author
*
* @version
*
* @date    May-28-2009
*
* @brief   This software is the USB driver stack for S08 family
*****

#include "derivative.h" /* include peripheral declarations */
#include "types.h"      /* User Defined Data Types */
#include "user_config.h"

#include "demo.h"      /* Demo application declarations */

/* Define ThreadX includes and global objects. */

#include "tx_api.h"
#include "tx_tpm.h"

/* Define thread control block for TestApp_Init. */

TX_THREAD TestApp_Init_Thread;
TX_THREAD Background_Thread;

/* Define thread entry prototypes. */

void TestApp_Init_Thread_Entry(ULONG id);
void Background_Thread_Entry(ULONG id);

/* Define the TraceX event buffer. */

UCHAR *event_buffer;

/* Define other thread counters. */

ULONG Background_Thread_Counter;

/* End of ThreadX information. */
```

## Main (159-230)

```

/*****
 * @name          main
 *
 * @brief         This routine is the starting point of the application
 *
 * @param         None
 *
 * @return        None
 *****/
*****
 * This function initializes the system, enables the interrupts and calls the
 * application
 *****

void main(void)
{
#if ((defined _MCF51MM256_H) || (defined _MCF51JE256_H))
    uint_32 *pdst,*psrc;
    uint_16 j;
#endif

    stop2recovery();

#if ((defined _MCF51MM256_H) || (defined _MCF51JE256_H))
    /* !! This section needs to be here to redirect interrupt vectors !! */

        // Set VBR to beginning of RAM
        asm (move.l  #VECTOR_RAM,d0);
        asm (movec  d0,vbr);

        // copy exception vectors from flash to RAM
        pdst=(uint_32*)VECTOR_RAM;
        psrc=(uint_32*)0;           // Flash base address

        for (j=0;j<256;j++,pdst++,psrc++)
        {
            *pdst=*psrc;
        }
        /* !! This section needs to be here to redirect interrupt vectors !! */
#endif

    Init_Sys();           /* initial the system */

#if MAX_TIMER_OBJECTS
    (void)TimerQInitialize(0);
#endif

    /* ThreadX initialization goes here! */

    /* Initialize ThreadX periodic timer interrupt. */
    TPM_Init();

    /* Enter ThreadX kernel. */
    tx_kernel_enter();

    /* This is no longer called from main, since it has been moved to a
    thread. */

```

```

    /* (void)TestApp_Init(); */ /* run the USB Test Application */
}

/* Define the ThreadX application define function. */

void tx_application_define(void *first_unused_memory)
{
    UCHAR *pointer;

    /* Pickup the first available RAM address. */
    pointer = (UCHAR *) first_unused_memory;

    /* Enable the ThreadX trace. */
    event_buffer = pointer;
    tx_trace_enable(event_buffer, 4096, 10);
    pointer = pointer + 4096;

    /* Create the ThreadX thread for processing the main loop. */
    tx_thread_create(&TestApp_Init_Thread, "Test App Thread",
        TestApp_Init_Thread_Entry, 0, pointer, 2048,
        16, 16, 4, TX_AUTO_START);
    pointer = pointer + 2048;

    /* Create the ThreadX background thread. */
    tx_thread_create(&Background_Thread, "Background Thread",
        Background_Thread_Entry, 0, pointer, 512,
        31, 31, 4, TX_AUTO_START);
    pointer = pointer + 512;

    /* All done with ThreadX initialization. */
}

/* Define the Test App Thread. ; */

void TestApp_Init_Thread_Entry(ULONG id)
{
    /* Simply call the application function from here! */
    TestApp_Init();
}

/* Define the Background Thread. ; */

void Background_Thread_Entry(ULONG id)
{
    /* Simply sit in a loop incrementing a counter for now... */

    uint_8 i = 3;
    while(1)
    {
        /* Increment the Background Thread Counter. */
        Background_Thread_Counter++;
    }
}

```

## usb\_init (180-181)

```
uint_8 cdc_putch(char ch)
{
    uint_8 status = 0;
    uint_16 delay;

    /* Send Data to USB Host*/

    g_curr_send_buf[0] = ch;

    g_send_size = 0;
    status = USB_Class_CDC_Interface_DIC_Send_Data(CONTROLLER_ID,
        g_curr_send_buf,1);
    if(status != USB_OK)
    {
        return 0xff; /* Send Data Error Handling Code goes here */
    }

    /* ThreadX mod... remove the delay and replace with a sleep. */

    // Original Code replaced By ThreadX Mod
    // for (delay = 0; delay < 10000; delay++)
    // {
    //
    // }
    //
    // End Original Code replaced By ThreadX Mod

    /* Call sleep, check for return status to make sure not in ISR. */
    if (tx_thread_sleep(2) != TX_SUCCESS) {

        /* Yes ISR calling, simply use the original delay. */
        for(delay = 0; delay < 10000; delay++)
        {

        }

    }

    /* End ThreadX mod. */

    return ch;
}
```

## usb\_putch (652-673)

```
void usb_init(void)
{
    uint_8 error;

    g_rcv_size = 0;
    g_send_size= 0;
    DisableInterrupts;
    #if (defined _MCF51MM256_H) || (defined _MCF51JE256_H)
        usb_int_dis();
    #endif
    /* Initialize the USB interface */
    error = USB_Class_CDC_Init(CONTROLLER_ID,USB_App_Callback,
                               NULL,USB_Notify_Callback);

    if(error != USB_OK)
    {
        /* Error initializing USB-CDC Class */
        return;
    }
    EnableInterrupts;
    #if (defined _MCF51MM256_H) || (defined _MCF51JE256_H)
        usb_int_en();
    #endif

    start_app = FALSE;
    start_transactions = FALSE;

    while((start_app!=TRUE) || (start_transactions!=TRUE))
    {
        /* Use ThreadX to sleep */
        tx_thread_sleep(1);
    }
}
```

## Display\_ME\_Results (1029-1030)

```
//-----  
//  
// void Display_ME_Results(void)  
//  
//   DESC:      After the Measurement Engine has been exercised,  
//              this function displays the results  
//  
//   INPUTS:  
//             none  
//  
//   OUTPUTS:  
//             none  
//  
//-----  
void Display_ME_Results(void) {  
  
    u8 i,p;  
    u32 convertedCPU;  
    u32 maxCPU;  
    u8 index;  
  
    Results_Index = ADC_Results;  
    Benchmark_Index = 0;  
    maxCPU = (u32)(Max_Cycles_Used * 0x10000 / (2*(PDBMOD + 1)));  
    maxCPU &= 0xFFFF;  
    p=0;  
    while(Benchmark_Index < (u16)NUM_PERIODS) {  
  
        // Convert CPU usage and display  
        convertedCPU = (u32)(Benchmark[Benchmark_Index++] * 0x10000 /  
(2*(PDBMOD + 1)));  
        convertedCPU &= 0xFFFF;  
  
        for(i=0; i<8; i++) {  
  
            if(p==0){  
  
                HexToASCII(DDATA[i]);  
                usb_out=",\0";  
                index=0;  
                while(usb_out[index] != '\0') {  
  
                    while((int)usb_out[index]!=cdc_putch(usb_out[index])){  
                        };  
                        index++;  
                    }  
                }  
  
                if(p!=0){  
  
                    HexToASCII(DDATA[i+8]);  
                    usb_out=",\0";  
                    index=0;  
                    while(usb_out[index] != '\0') {  
                        while((int)usb_out[index]!=cdc_putch(usb_out[index])){  
                            };  
                            index++;  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        };
        index++;
    }
}
// Display ADC Result
HexToASCII(*Results_Index);
usb_out=",\0";
index=0;
while(usb_out[index] != '\0') {
while((int)usb_out[index]!=cdc_putch(usb_out[index])){
    };
    index++;
}

HexToASCII((u16)convertedCPU);
usb_out=",\0";
index=0;
while(usb_out[index] != '\0') {

while((int)usb_out[index]!=cdc_putch(usb_out[index])){
    };
    index++;
}
HexToASCII((u16)maxCPU);
usb_out="\r\n\0";
index=0;
while(usb_out[index] != '\0') {

while((int)usb_out[index]!=cdc_putch(usb_out[index])){
    };
    index++;
}
//cdc_process();
Results_Index++;

}

p = !p;
if(!SELECT_PIN)
    return;

// Use TOD to delay 250ms
TODC_TODCLKS = 1;
TODC_TODPS = 0;
TODSC_QSECF = 1;
TODC_TODEN =1 ;

while(TODSC_QSECF == 0){

    /* Use ThreadX to sleep and avoid wasting cycles. */
    tx_thread_sleep(1);

};
TODSC_QSECF = 1;

}
}

```