Using ThreadX® 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
Express Logic, Inc.
Medical Device Development
Using ThreadX® and Freescale’s MCF51MM Microcontroller

Abstract
Freescale’s MCF51MM microcontroller is ideal for Medical applications, and Express Logic’ ThreadX® RTOS provides an easy-to-use, inexpensive run-time software solution that enables rapid time to market and full FDA 510(k) regulatory compliance. In this class, attendees will learn how to use Freescale’s CodeWarrior IDE to develop a sample medical application, and run it on the MCF51MM. Full demo source code will be provided as well as a demo license for ThreadX.

Agenda
• Medical Device Development
  – Technical Characteristics of medical devices
  – Available Technologies
  – Regulatory Requirements
• Freescale’s MCF51MM
  – Microcontroller-based platform
  – Low cost, low power, small form factor
  – Options
• Express Logic’s ThreadX RTOS
  – Small footprint, easy to use, low cost, full source code
  – Basic functionality, technology, benefits
  – Building a ThreadX project with CodeWarrior
• Demo Application
  – EKG Demo
  – Adapted from stand-alone form
• Certification Pack™ from Express Logic
  – For use in securing FDA 510(k) PMA and IEC-60601 certification
  – Turnkey materials for RTOS software portion

Technical Characteristics
• Widely varied from CAT-Scanner to handheld
• Today, we will focus on small size, low cost, portable, low power
• Hardware: Memory << 1MB, small form factor
• Complexity: Low-moderate
• OS: Small footprint, Basic Functionality, FDA/IEC compliant

Freescale MCF51MM
Target Applications
• Blood Glucose Meters
• Blood Pressure Monitors
• Heart Rate Monitors
• One Lead EKG
• Telehealth Devices
• Pulse Oximeters
• Implantables
• Clinical Equipment
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 (x1, 90mm/3.5" long, 164 pins)

Board Connectors:
- Four card-edge connectors
- Uses PCI Express connectors (x1, 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

Tower System Software

- CodeWarrior IDE
  - CW_MCU_6.3
- EKG Demo Application
  - EKG sensor data capture
  - Optional data simulation
  - A/D
  - Display
- ThreadX RTOS

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

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

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
ThreadX Technology

- Preemption-Threshold
  - Set lower limit for priority of preempting thread
  - Useful for reducing number of context switches
  - Also useful for preventing priority inversion

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

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.

Preemption-Threshold Improves Efficiency

<table>
<thead>
<tr>
<th>Relative Time Between Mutex_Get and Mutex_Put Pairs for High_thread</th>
<th>Average Time</th>
<th>Minimum Time</th>
<th>Maximum Time</th>
<th>Number of Get-Put Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without Preemption-Threshold</td>
<td>40.1</td>
<td>2</td>
<td>104</td>
<td>17</td>
</tr>
<tr>
<td>With Preemption-Threshold</td>
<td>2.0</td>
<td>2</td>
<td>4</td>
<td>52</td>
</tr>
</tbody>
</table>

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

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 context registers saved/restored if preemption
  - No task thread, hence no context recreation when system is idle
  - Most API available directly from ISRs
  - Optional timer thread or direct timer processing in ISR
Performance (Thread-Metric)

<table>
<thead>
<tr>
<th></th>
<th>ThreadX (3)</th>
<th>μC/OS-II (1)</th>
<th>TNKernel (1)</th>
<th>AVA (1)</th>
<th>FreeRTOS (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cooperative Scheduling</td>
<td>11,848,815</td>
<td>4,870,885</td>
<td>6,928,383</td>
<td>1,724,948</td>
<td>3,717,913</td>
</tr>
<tr>
<td>Preemptive Scheduling</td>
<td>4,870,885</td>
<td>5,259,998</td>
<td>6,814,817</td>
<td>5,207,762</td>
<td>1,881,892</td>
</tr>
<tr>
<td>Interrupt Processing</td>
<td>3,052,151</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>Message Processing</td>
<td>6,918,050</td>
<td>(1)</td>
<td>(1)</td>
<td>(1)</td>
<td>(2)</td>
</tr>
<tr>
<td>Synchronization Processing</td>
<td>15,337,354</td>
<td>10,293,318</td>
<td>7,353,579</td>
<td>7,514,799</td>
<td>1,989,999</td>
</tr>
<tr>
<td>Memory Processing</td>
<td>12,863,624</td>
<td>6,814,817</td>
<td>5,933,761</td>
<td>10,235,182</td>
<td>(2)</td>
</tr>
</tbody>
</table>

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

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.

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

"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()
  } ...
}
```

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

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

The Big Loop

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

```
A?, B?, C?, D?, ...
```

"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()
  } ...
}
```

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

Multithreading

- Enabling an activity to use the CPU while other activities don’t need it – Event Loop
"RTOS-izing" Code

- Stand-alone code often uses "Delay Loops"
  ```c
  for(i=0, i<10000, i++) {
     // Spin loop occupies CPU 100% for duration of delay period.
  }
  ```
- Replace with call to `tx_thread_sleep(n)`
  ```c
  for(i=0, i<10000, i++) {
     // Sleep call frees up CPU for 25ms (1ms each, but can be any user-defined duration).,
     // suspended thread re-awakens
  }
  ```
- 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

Multithreading

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

Convert Demo Into Threads

1. Add header files `tx_api.h`, `tx_port.h`, and `tx_lpm.h` to the project.
2. Add function files `tx_inititalize_low_level.s`, `tx_lpm.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:
   ```c
   __free_mem           =  ___SP_AFTER_RESET;
   ```
5. Modify main.c as follows:
   ```c
   /* Enable the ThreadX trace. */
   tx_trace_enable(event_buffer, 4096, 10);
   
   /* Create the ThreadX thread for processing the main loop. */
   tx_thread_create(&TestA_Init_Thread, "Test A Init Thread", TestA_Init_Thread_Entry, 0, pointer, 2048, 16, 16, 0, TX_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);
   
   // Use TOD to delay 250ms
   TODC_TODCLKS = 1;
   TODC_TODPS = 0;
   TODC_QSECF = 1;
   TODC_TODEN = 1;
   while(TODC_QSECF == 0){
     // Use ThreadX to sleep and avoid wasting cycles.
     tx_thread_sleep(1);
   }
   ```
Let's See How it All Works

Results Without ThreadX

Results With ThreadX

How ThreadX Operates

RAM/ROM Measurements

<table>
<thead>
<tr>
<th>Component</th>
<th>Measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROM (without ThreadX)</td>
<td>18KB</td>
</tr>
<tr>
<td>ROM (with ThreadX)</td>
<td>24KB</td>
</tr>
<tr>
<td>RAM (without ThreadX)</td>
<td>1KB Stack</td>
</tr>
<tr>
<td>RAM (with ThreadX)</td>
<td></td>
</tr>
<tr>
<td>ThreadX RAM requirements</td>
<td>416 bytes</td>
</tr>
<tr>
<td>Thread Control Blocks</td>
<td>344 bytes</td>
</tr>
<tr>
<td>ThreadX Stacks</td>
<td>2.5KB</td>
</tr>
<tr>
<td>TraceX Trace Buffer</td>
<td>4KB (user option)</td>
</tr>
</tbody>
</table>

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”
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

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….

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

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™

Medical Certification Pack

<table>
<thead>
<tr>
<th>Device</th>
<th>Medical Certification Pack</th>
<th>Medical Certification Pack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
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_com.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:

<table>
<thead>
<tr>
<th></th>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ROM</td>
<td>RAM</td>
</tr>
<tr>
<td>Application</td>
<td>18KB</td>
<td>2KB</td>
</tr>
<tr>
<td>ThreadX requirements</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>2 thread control blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 thread stacks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TraceX event buffer</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>18KB</strong></td>
<td><strong>2KB</strong></td>
</tr>
</tbody>
</table>

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)

#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)

```c
/**
 * @name        main
 * @brief       This routine is the starting point of the application
 * @param       None
 * @return      None
**/

void main(void)
{
    #if ((defined _MCF51MM256_H) || (defined _MCF51JE256_H))
        uint_32 *p dst,*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  #VECT OR_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;
        }
    #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 */

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;
}
void usb_init(void)
{
    uint_8  error;

    g_recv_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) {

    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++;  
                    }
                }
            }


        }
    }
}

// DESC: After the Measurement Engine has been exercised,
// this function displays the results
// INPUTS: none
// OUTPUTS: none
//
};
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"
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;