Express Logic, Inc.

Thread-Metric Benchmark Suite

expresslogic

Who Needs An RTOS?

- Applications with multiple threads
  - Modular design organizes functionality
  - Schedules processing to handle workload
  - Priority-based preemption
  - Message-passing
- Applications with external events
  - Interrupt response, resumption management
  - Enables rapid response to external events
- Applications with network communications
  - Handles thread-thread communications
  - Handles Internet communications

Most Important RTOS Capabilities

- Evans Data Corporation’s Surveys reveal the *Top **5** RTOS Features* most valued by developers:

  1. Real-time responsiveness (33.2%)
  2. Royalty-free pricing (14.7%)
  3. Source code availability (10.6%)
  4. Tools integration (IDE) (10.1%)
  5. Microprocessor coverage (7.8%)

  Source:
  http://www.evansdata.com/n2/surveys/embedded/2003_1/embedded_xmp1.shtml

Measuring RTOS Performance

- RTOS Performance is the speed with which an RTOS completes services for an application
- RTOS Performance is
  - platform-sensitive
  - processor-sensitive
  - clock-speed sensitive
  - compiler-sensitive
  - design-sensitive
- Performance also is “context sensitive”
  - What exactly is being measured?
  - How is it being measured?

Thread-Metric: An Open RTOS Benchmark

- “Thread-Metric,” is a free-source benchmark suite for measuring RTOS performance
- The Thread-Metric suite is freely available from Express Logic
- Performance of Express Logic’s ThreadX® RTOS is provided for reference
- Thread-Metric is easily adapted to other RTOSes

What Is Being Measured?

- The time taken by the RTOS to perform specific services
  - This is the RTOS “Overhead”
- To be applicable to multiple RTOSes, for comparison, a set of common services has been selected
  - Representative across all services
  - Commonly found among most RTOSes
- Selected RTOS Services
  - Cooperative Context Switching
  - Preemptive Context Switching
  - Interrupt Processing Without Preemption
  - Interrupt Processing With Preemption
  - Message Passing
  - Semaphore Processing
  - Memory Allocation/Deallocation

How Are Services Measured?

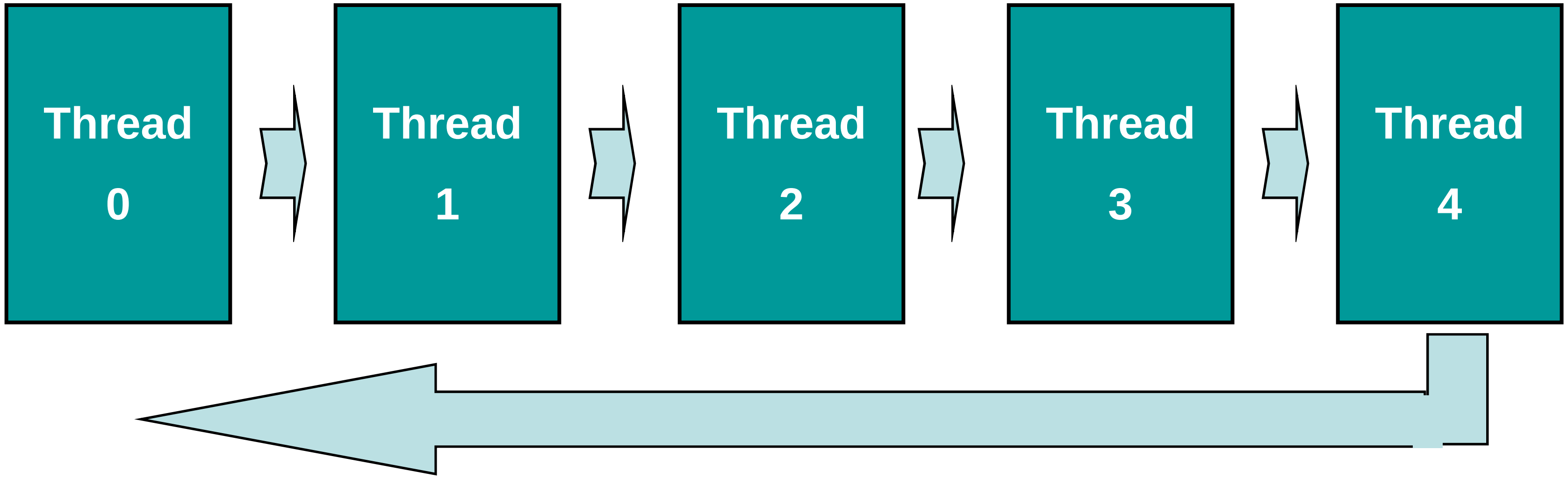
- Lots of iterations, 30-second reporting
  - Execute (service, inverse) in pairs
    - Send message; get message
    - Allocate memory; de-allocate memory
    - Keep counts in local variables
    - “Printf” results to host every 30 seconds
- Calibration run to establish baseline
- No special hardware required
- Easily ported to new environments
- Coded in “Vanilla” C
  - Tested with various compilers
  - RTOS functions identified for adaptation

Implicitly Measured

- Service entry design
  - Trap
    - Uses unimplemented instruction trap to force interrupt
    - ISR examines parameters and transfers to appropriate RTOS service
    - Similar to debugger software-breakpoint technique
    - Locks out interrupts for a time
  - Call
    - Uses processor branch instruction, no interrupts
    - Low overhead
    - Requires linking

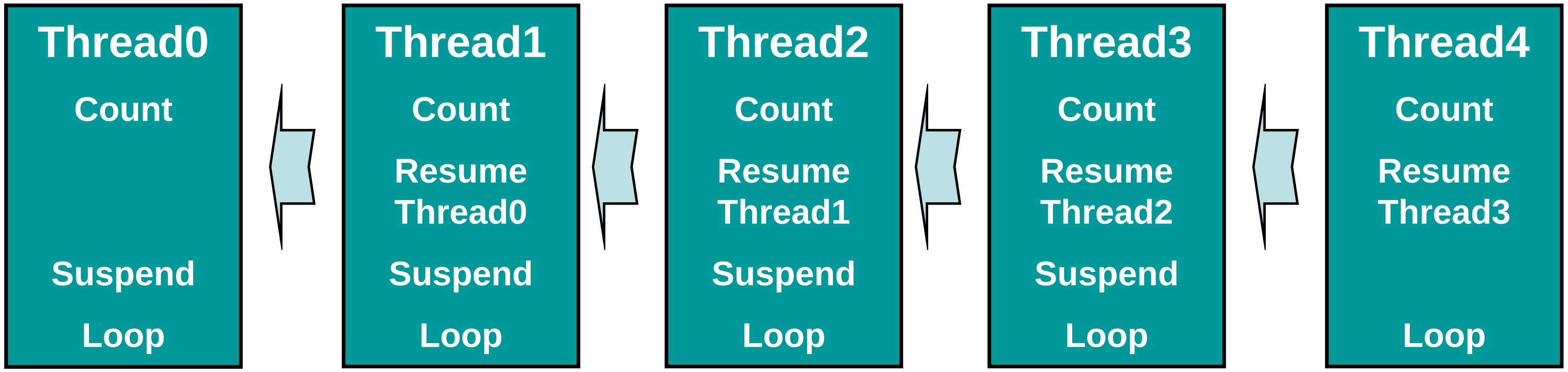
Cooperative Context Switching

- Threads run to completion and then exit
- This test consists of 5 threads created at the same priority
  - Each thread runs to completion and then voluntarily releases control.
  - Threads run in a round-robin fashion.
  - Each thread increments its run counter and then relinquishes to the next thread



Preemptive Context Switching

- Threads run until preempted by a higher priority thread.
- This test consists of 5 threads that each have a unique priority.
  - All threads except the lowest priority thread are in a suspended state.
  - The lowest priority thread resumes the next highest priority thread.
  - That thread resumes the next highest priority thread and so on until the highest priority thread executes.
  - Each thread increments its run count and then suspends.
  - Once processing returns to the lowest priority thread, it increments its run counter and once again resumes the next highest priority thread - starting the whole process over again.



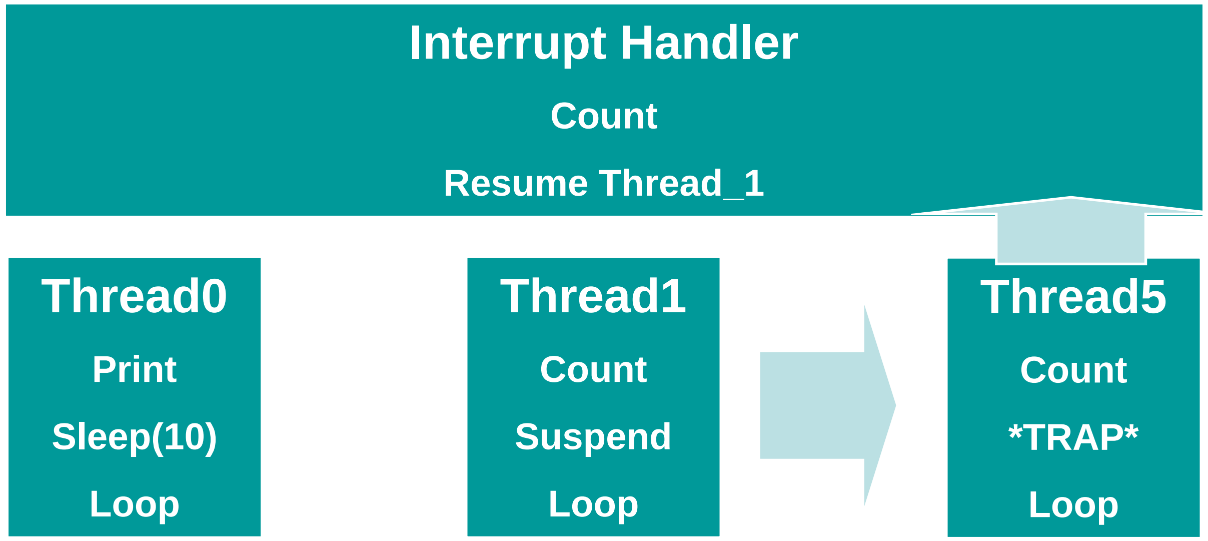
Interrupt Handling

- Must consider two components
    - Interrupt Latency ("Time to ISR")
        - How long are interrupts disabled?
    - Task Activation Overhead ("Time to Task")
        - How quickly can a thread/task respond?

- "Hurry Up and Wait" Can Be Misleading
    - Low interrupt latency, but delayed task activation
        - Can OS services be called from ISR?
    - "Split-Level" Interrupt Handling
        - Low level ISR responds to hardware
        - High Level ISR/Task calls OS services
        - Task performs processing

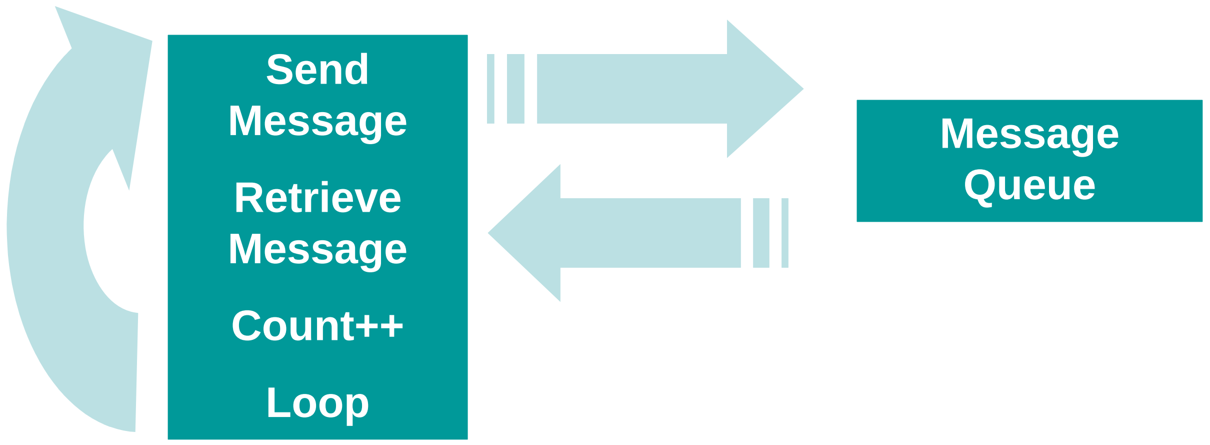
Interrupt Handling

- Threads use software interrupts to trigger preemption
- Reporting thread “prints” results every 30 seconds
  - Thread1 suspends
  - Thread5 forces interrupt
  - Interrupt handler resumes Thread1
  - Print thread is top priority, runs every 30 seconds



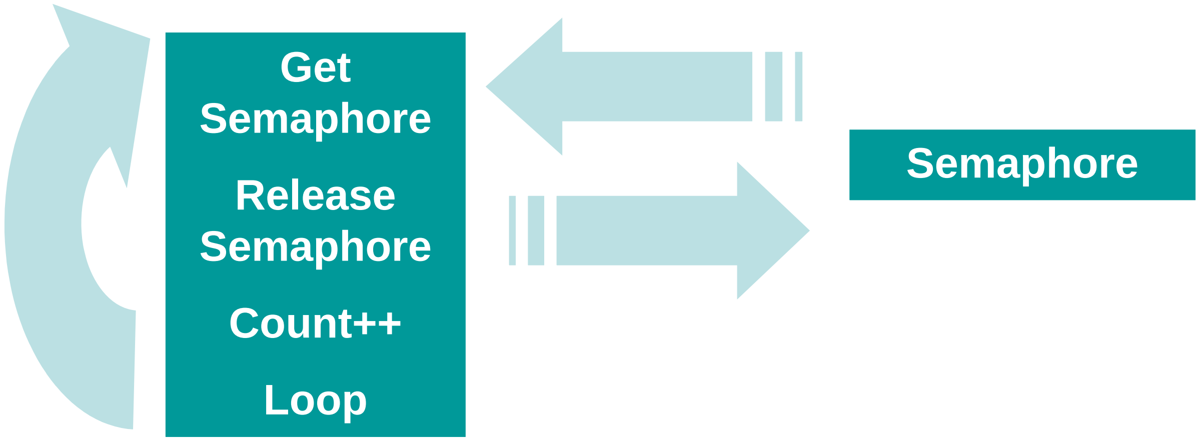
Message Processing

- A Thread sends a message to a queue from which it is retrieved by the same thread.
- This test consists of a thread sending a 16 byte message to a queue and retrieving the same 16 byte message from the queue.
  - After the send/receive sequence is complete, the thread increments its run counter



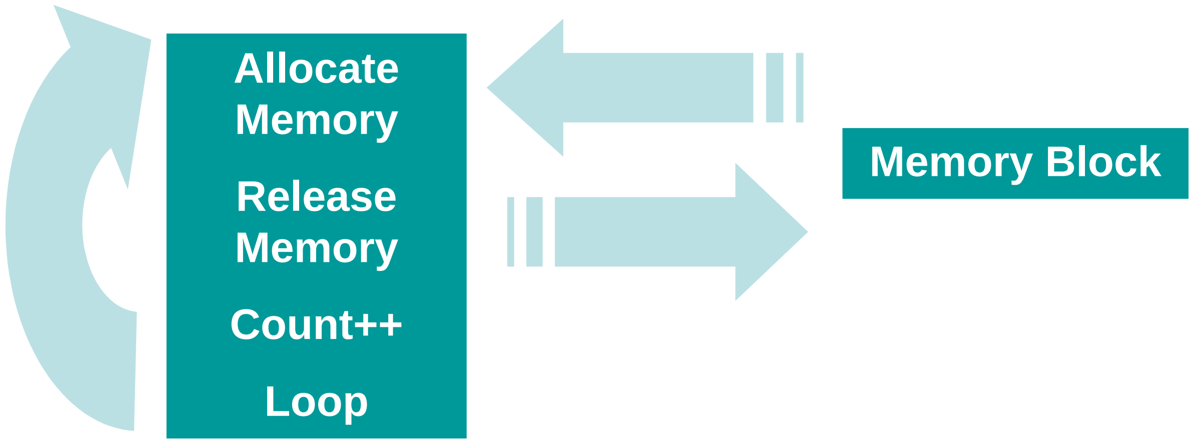
Semaphore Processing

- A semaphore is a system resource used to guarantee a task exclusive access to a critical resource. Semaphores synchronize asynchronous activities
- This test consists of a thread getting a semaphore and then immediately releasing it.
  - After the get/release cycle completes, the thread increments its run counter



Memory Allocation

- Applications commonly must allocate memory dynamically to avoid the need to plan for the maximum possible memory needs of all tasks at the same time
- This test consists of a thread allocating a 128-byte block of memory and releasing the same block.
  - After the block is released, the thread increments its run counter



RTOS Adaptation Layer

- tm_porting_layer.c
  - contains shells of the generic RTOS services used by each of the actual tests
  - The shells provide the mapping between the tests and the underlying RTOS
  - Must be adapted for specific RTOS
  - ThreadX adaptation source provided as example

```
/* This function sends a 16-byte message to the specified queue.  If successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */

int  tm_queue_send (int queue_id, unsigned long *message_ptr)
{
UINT    status;

    /* Send the 16-byte message to the specified queue.  */

    status =  tx_queue_send(&tm_queue_array[queue_id], message_ptr, TX_NO_WAIT);

    /* Determine if the queue send was successful.  */

    if (status == TX_SUCCESS)
        return(TM_SUCCESS);
    else
        return(TM_ERROR);
}
```

Test Environment

- ARM Versatile Board, ARM926 Processor, 200MHz
- 10ms Timer
- No Caching
- ARM RealView Compiler NO-OPT
- Express Logic’s ThreadX® RTOS

ThreadX Performance

- 30 Second Period

– Test	Iterations	TM Ratio
Calibration Test	11,482	-
Cooperative Context Switch	1,585,032	138
Preemptive Context Switch	579,190	50
Message Processing	856,342	75
Semaphore Processing	1,566,675	136
Memory Allocation	1,501,724	131
Interrupt Handling	908,127	79
Interrupt Preemption	358,460	31
Synchronization Processing	1,766,942	154

How To Get The Code

- Download from Express Logic’s web site: www.expresslogic.com

Conclusion

- Real-time performance is important to enable application to meet deadlines
- Simple, consistent method of measurement is equally important
- Here is an example of such a program
- Try it!
- Let us know your ideas for improvement
  - jcarbone@expresslogic.com
  - **1-888-THREADX**