Real-time Operating Systems

Original slides by Ben Nahill
Modified by Ashraf Suyyagh - Fall 2013

RTOS

- Trade performance for consistent operation
- Multi-tasking
- Provides tools for synchronization
- Utility for shared resources

CMSIS-RTOS

Common RTOS interface for Cortex-M systems

- Dozens of RTOS's out there (tinyOS, RTX, ChibiOS … etc.)
- Abstraction of common OS functionality
- Says nothing about implementation
- Currently only supported by ARM's own RTX OS... but maybe someday

Official CMSIS Documentation:

http://bennahill.com/docs/cmsis/CMSIS/Documentation/RTOS/html/

Examples

1. http://mbed.org/handbook/CMSIS-RTOS

Note however that those examples were based on older version, some changes in syntax for one or two definitions, but else is fine

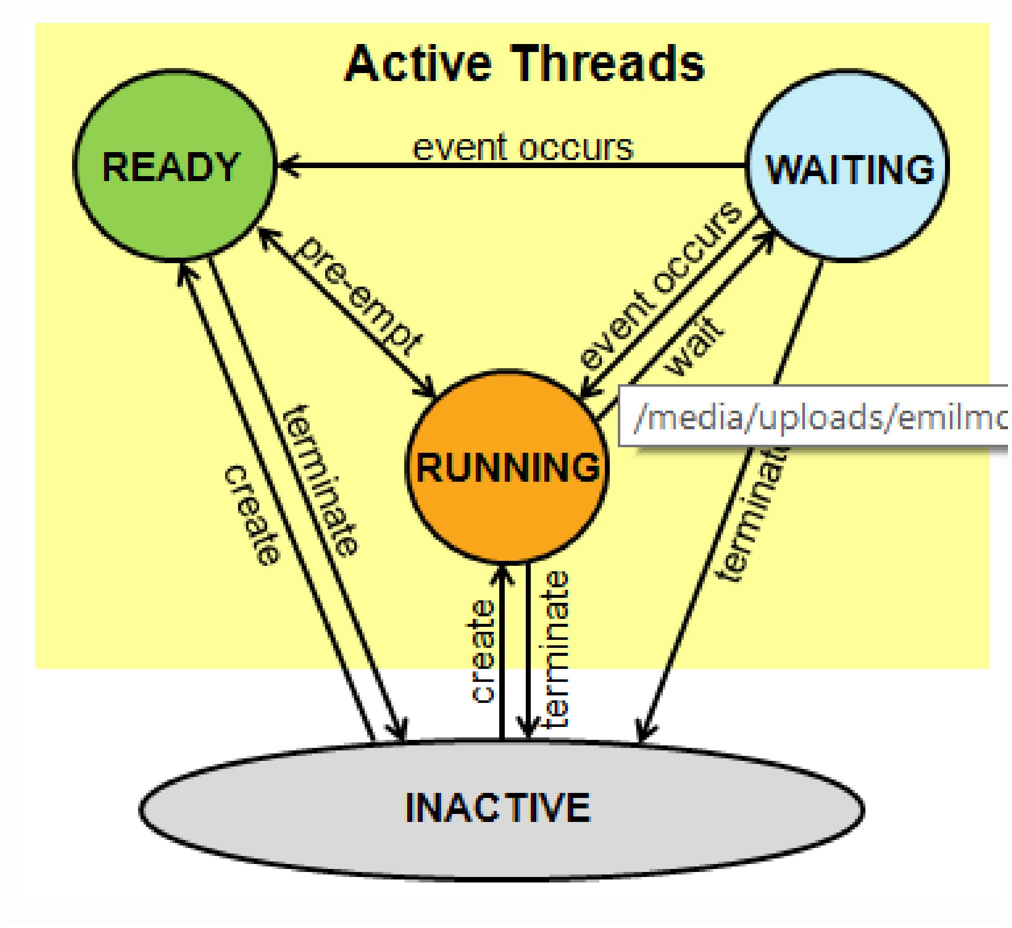
2. Tutorial Project Example

Threads

Concurrent tasks which may have interdependencies

- Single core, need scheduler
- Each task has its own stack and state buffer
- Assigned priorities to be used in scheduling
- Can sleep without costing the system much

Thread Model



Scheduler

Schedules which thread is running at any given moment

- May be preemptive or non-preemptive
- Priority based or fixed-priority
- Many different schemes
- RTX uses round robin (RR) preemptive scheduler with priorities

Pre-emptive Scheduler (Round-robin)

Scheduler based on tick

- Reschedules on events
  - Can be tick, resource release, message, etc
- Highest-priority task is running
- Equal-priority tasks will alternate
- Very common scheduler scheme that works well for most systems

RR Scheduler (cont'd)

- Ticks can be at arbitrary frequency
- Timebase generated by hardware
- Low frequency reduces context switch overhead
- High frequency allows for more even distribution of computational resources
- You can control many of these scheduling variables for CMSIS-RTOS in the file RTX_CONF_CM.c

Thread Priority

Decides who gets to run when possible

- Must be carefully assigned
  - Not necessarily constant
- Improper assignment could lead to starvation of lower-priority task
- Schedulers offer techniques to limit the above case (priority inheritence)

Thread Functions

- **osThreadCreate** : Start execution of a thread function.
- **osThreadTerminate** : Stop execution of a thread function.
- **osThreadYield** : Pass execution to next ready thread function.
- **osThreadGetId** : Get the thread identifier to reference this thread.
- **osThreadSetPriority** : Change the execution priority of a thread function.

Timers

Scheduler also provides facilities to create reasonably accurate delays

- Create both one-shot and periodic timers
- Timers usually a multiple of tick frequency for RR schedulers

Mutual Exclusion without RTOS

Need to protect certain operations from interruption which can cause conflict

- Disable interrupts during operation

Serialize access to a shared resource

- A shared mutex can guarantee exclusive access to a resource while held
- With threads waiting on a mutex, priority determines which will get it first
- Priority inheritance used

Semaphores

Thread-safe record of arbitrary quantifiable resource

- Used in producer-consumer model
- Serialized access
- Counting semaphores
  - General case, count usually has a limit
- Binary semphores
  - Counting semaphores with limit 1

Binary Semaphore vs Mutex

Mutex has notion of an owner and manages priorities accordingly

- Semaphore is associated more with the resource that it repressents
- Semaphore has less overhead for implementation

Signals

Boolean flags sent to specific threads

- Each signal has a per-thread ID
- Can wait for specific flag or set of flags

osSignalSet : Set signal flags of a thread.

osSignalClear : Reset signal flags of a thread.

osSignalGet : Read signal flags of a thread.

osSignalWait : Suspend execution until specific signal flags are set.

Queues

Need safe way to pass messages to/from threads/interrupts

We have message queues and mail queues

Message Queue

Send uint32_t to a target thread

- osMessageCreate : Define and initialize a message queue.
- osMessagePut : Put a message into a message queue.
  - Sleep until there is room for this message
- osMessageGet : Get a message or suspend thread execution until message arrives.
  - Sleep until a message arrives or perform simple poll

Value is copied since it is very small

Mail Queue

Communicate with pointers to data

- Items may be dynamically allocated
  - Allocated by sender, freed by recipient
  - Includes dedicated memory pool management

osMailCreate : Define and initialize a mail queue with fix-size memory blocks.

osMailAlloc : Allocate a memory block.

osMailCAlloc : Allocate a memory block and zero-set this block.

osMailPut : Put a memory block into a mail queue.

osMailGet : Get a mail or suspend thread execution until mail arrives.

osMailFree : Return a memory block to the mail queue.

Interrupt-thread Communication

- Interrupts can't wait for *anything*
  - Can receive from queues, check mutexes, set or check semaphores but *cannot* wait on any of them
- Need to use queues or signals to communicate

Examples: Event Handler

Need to handle external events detected using interrupts without processing any data in ISRs

- Perform different actions for different events
- Don't spend so long on one event that you miss another one

Event Handler: without RTOS

All processing must occur in a single loop

```
static volatile int new_acc_data = 0;
static volatile int new_button_press = 0;

while(1){
   if(new_acc_data){
      new_acc_data = 0;
      // Compute some stuff with it
   }
   if(new_button_press){
      new_button_press = 0;
      // Handle button press
   }
}

void button_isr(){
   new_button_press = 1;
}

void acc_transfer_complete_isr(){
   new_acc_data = 1;
}
```

Event Handler: with RTOS

Use threads to divide behavior

```
void acc_handler_thread(){
   void *mail;
   while(1){
      // Wait forever for new accelerometer data
      mail = osMailGet(acc_mailbox_id, osWaitForever);
      acc_handle(mail);
      osMailFree(acc_mailbox_id, mail);
   }
}

void button_handler_thread(){
   while(1){
      // Wait forever for button signal
      osSignalWait(BUTTON_PRESSED_SIGNAL, osWaitForever);
   }
}

void button_isr(){
   osSignalSet(button_handler_id, BUTTON_PRESSED_SIGNAL);
}

void acc_transfer_complete_isr(){
   void *mail = osMailAlloc(acc_mailbox_id, 0);
   // Read the values...
   if(mail){
      // Put values into mail...
      osMailPut(acc_mailbox_id, mail);
   }
}
```

Keil Support for debugging

- Keil offers visualization of thread execution and in depth information about what is going on per thread
- First, you need to tell Keil that there is RTOS being used

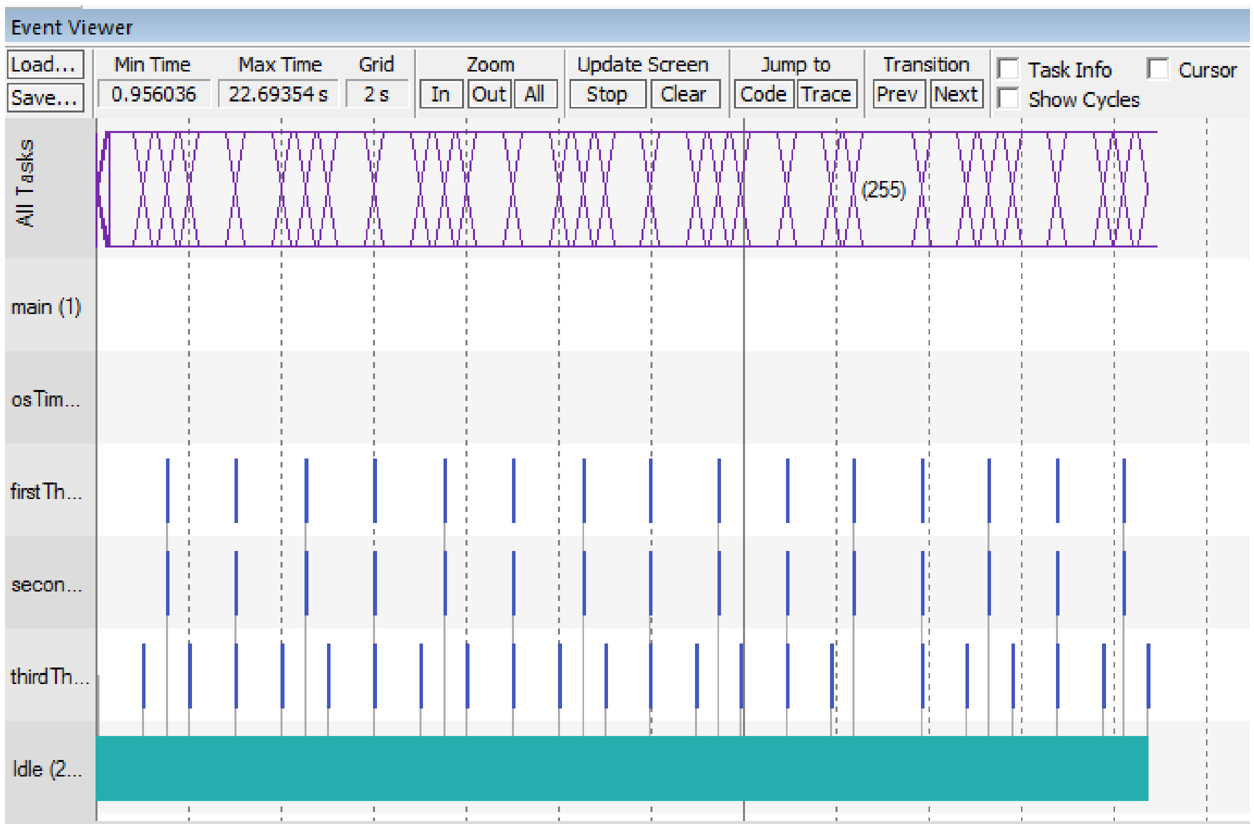
Right click on your project → Options → Target

Then select RTX in the Operating Systems menu

If you don’t do that, the next screens won’t work

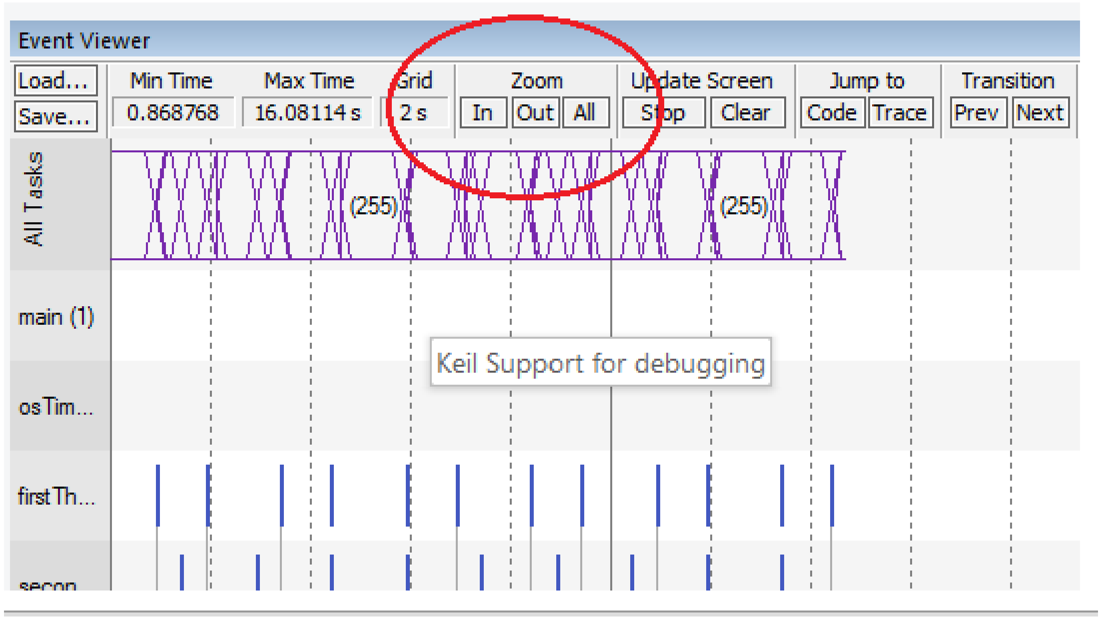
Keil Support for debugging

- While in Debug mode
- From the menu bar Debug ! OS Support ! Event Viewer, you should see a screen like this (though it starts empty before running the code)



Keil Support for debugging

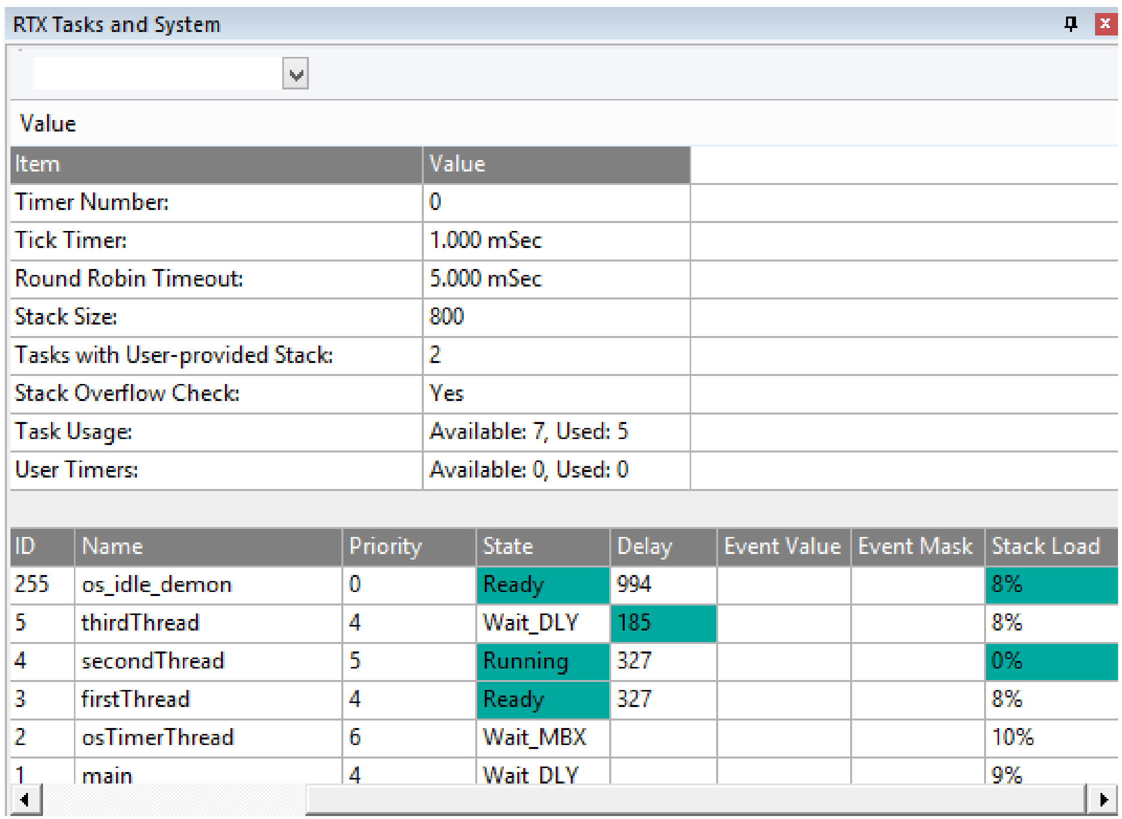
- When you run the code and don't see something similar to the previous screen, you need to zoom all (or zoom out) to see the big picture



Keil Support for debugging

To see more details per thread (for example, to see if your thread overflows the allocated stack and you need to increase it) go to

Debug → OS Support → RTX Tasks and system



RTX Tasks and System

Value

Item	Value
Timer Number:	0
Tick Timer:	1.000 mSec
Round Robin Timeout:	5.000 mSec
Stack Size:	800
Tasks with User-provided Stack:	2
Stack Overflow Check:	Yes
Task Usage:	Available: 7, Used: 5
User Timers:	Available: 0, Used: 0

ID	Name	Priority	State	Delay	Event Value	Event Mask	Stack Load
255	os_idle_demon	0	Ready	994			8%
5	thirdThread	4	Wait_DLY	185			8%
4	secondThread	5	Running	327			0%
3	firstThread	4	Ready	327			8%
2	osTimerThread	6	Wait_MBX				10%
1	main	4	Wait_DLY				9%