RTOS Examples

EE5182 Microcontrollers and Embedded Systems

Includes content from:
Introduction to Real Time OSes by Mark Brehob
Introduction to FreeRTOS V6.0.5 by Amr Ali Abdel-Naby
The mC/OS-II Real-Time Operating System by NC State University

µC/OS

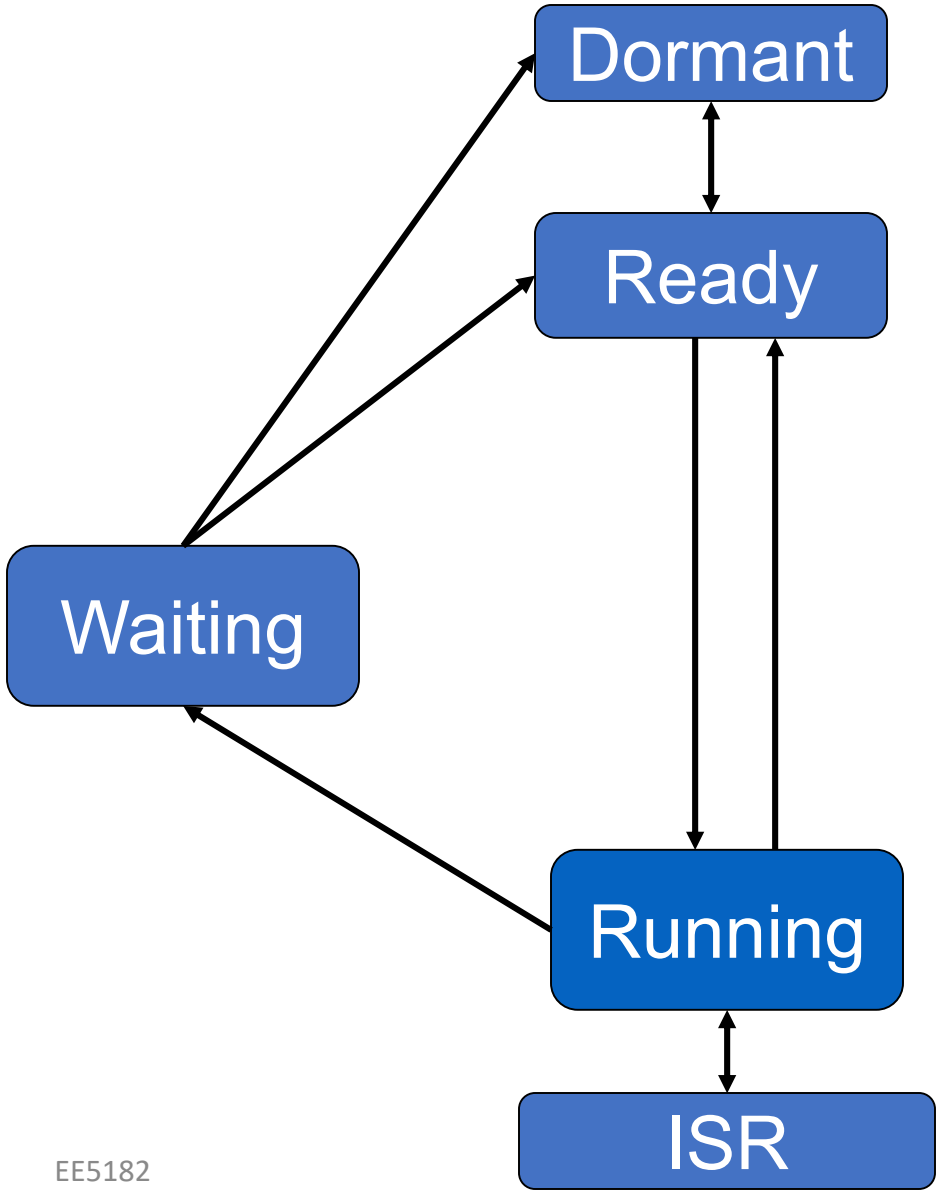
Pronounced “micro C OS”, a full-featured embedded operating system.

µC/OS-II and µC/OS-III are pre-emptive, highly portable, and scalable real-time kernels.

https://www.micrium.com/rtos/

Task States

- Five possible states for a tasks:
  - Dormant – not yet visible to OS (use OSTaskCreate(), etc.)
  - Ready
  - Running
  - Waiting
  - ISR – preempted by an Interrupt Service Routine (ISR)



Task Scheduling

- Scheduler runs highest-priority task using OSSched()
  - OSRdyTbl has a set bit for each ready task
  - Checks to see if context switch is needed
  - Macro OS_TASK_SW performs context switch
    - Implemented as software interrupt which points to OSCtxSw
    - Save registers of task being switched out
    - Restore registers of task being switched in

Task Scheduling

- Scheduler locking
  - Can lock scheduler to prevent other tasks from running (ISRs can still run)
    - OSSchedLock()
    - OSSchedUnlock()
  - Nesting of OSSchedLock possible
  - Don’t lock the scheduler and then perform a system call which could put your task into the WAITING state!
- Idle task
  - Runs when nothing else is ready
  - Automatically has prioirty OS_LOWEST_PRIO
  - Only increments a counter for use in estimating processor idle time

Task States

- Task status OSTCBStat

```
/* TASK STATUS (Bit definition for OSTCBStat)           */
#define  OS_STAT_RDY       0x00  /* Ready to run          */
#define  OS_STAT_SEM       0x01  /* Pending on semaphore */
#define  OS_STAT_MBOX      0x02  /* Pending on mailbox   */
#define  OS_STAT_Q         0x04  /* Pending on queue     */
#define  OS_STAT_SUSPEND   0x08  /* Task is suspended    */
```

Enabling Interrupts

- Macros OS_ENTER_CRITICAL, OS_EXIT_CRITICAL
- Note: three methods are provided in os_cpu.h
  - #1 doesn't restore interrupt state, just renables interrupts
  - #2 saves and restores state, but stack pointer must be same at enter/exit points – use this one!
  - #3 uses a variable to hold state
    - Is not reentrant
    - Should be a global variable, not declared in function StartSystemTick()

System Clock Tick

- OS needs periodic timer for time delays and timeouts
- Recommended frequency 10-200 Hz
  - trade off overhead vs. response time (and accuracy of delays)
- Must enable these interrupts after calling OSStart()
- OSTick() ISR
  - Calls OSTimeTick()
    - Calls hook to a function of your choosing
    - Decrements non-zero delay fields (OSTCBDly) for all task control blocks
    - If a delay field reaches zero, make task ready to run (unless it was suspended)
  - Increments counter variable OSTime (32-bit counter)
  - Then returns from interrupt
- Interface
  - OSTimeGet(): Ticks (OSTime value) since OSStart was called
  - OSTimeSet(): Set value of this counter

Overview of Writing an Application

- Scale the OS resources to match the application
  - See os_cfg.h
- Define a stack for each task
- Write tasks
- Write ISRs
- Write main() to Initialize and start up the OS (main.c)
  - Initialize MCU, display, OS
  - Start timer to generate system tick
  - Create semaphores, etc.
  - Create tasks
  - Call OSStart()

Configuration and Scaling

- For efficiency and code size, default version of OS supports limited functionality and resources
- When developing an application, must verify these are sufficient (or may have to track down strange bugs)
  - Can’t just blindly develop program without considering what’s available
- Edit os_cfg.h to configure the OS to meet your application’s needs
  - # events, # tasks, whether mailboxes are supported, etc.

Task Creation

- OSTaskCreate() in os_task.c
  - Create a task
  - Arguments: pointer to task code (function), pointer to argument, pointer to top of stack (use TOS macro), desired priority (unique)
- OSTaskCreateExt() in os_task.c
  - Create a task
  - Arguments: same as for OSTaskCreate(), plus
    - id: user-specified unique task identifier number
    - pbos: pointer to bottom of stack. Used for stack checking (if enabled).
    - stk_size: number of elements in stack. Used for stack checking (if enabled).
    - pext: pointer to user-supplied task-specific data area (e.g. string with task name)
    - opt: options to control how task is created.

More Task Management

- OSTaskSuspend()
  - Task will not run again until after it is resumed
  - Sets OS_STAT_SUSPEND flag, removes task from ready list if there
  - Argument: Task priority (used to identify task)
- OSTaskResume()
  - Task will run again once any time delay expires and task is in ready queue
  - Clears OS_STAT_SUSPEND flag
  - Argument: Task priority (used to identify task)
- OSTaskDel()
  - Sets task to DORMANT state, so no longer scheduled by OS
  - Removed from OS data structures: ready list, wait lists for semaphores/mailboxes/queues, etc.
- OSTaskChangePrio()
  - Identify task by (current) priority
  - Changes task's priority
- OSTaskQuery()
  - Identify task by priority
  - Copies that task's TCB into a user-supplied structure
  - Useful for debugging

FreeRTOS

Market Leading, De-facto Standard and Cross Platform RTOS kernel for embedded devices.

Ported to 35 microcontroller platforms, distributed under the MIT License.

https://www.freertos.org/

Source Code

- High quality
- Neat
- Consistent
- Organized
- Commented

```
signed portBASE_TYPE xTaskRemoveFromEventList( const xList * const pxEventList )
{
tskTCB *pxUnblockedTCB;
portBASE_TYPE xReturn;

    /* THIS FUNCTION MUST BE CALLED WITH INTERRUPTS DISABLED OR THE
    SCHEDULER SUSPENDED.  It can also be called from within an ISR. */

    /* The event list is sorted in priority order, so we can remove the
    first in the list, remove the TCB from the delayed list, and add
    it to the ready list.

    If an event is for a queue that is locked then this function will never
    get called - the lock count on the queue will get modified instead.  This
    means we can always expect exclusive access to the event list here.

    This function assumes that a check has already been made to ensure that
    pxEventList is not empty. */
    pxUnblockedTCB = ( tskTCB * ) listGET_OWNER_OF_HEAD_ENTRY( pxEventList );
    configASSERT( pxUnblockedTCB );
    vListRemove( &( pxUnblockedTCB->xEventListItem ) );

    if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE )
    {
        vListRemove( &( pxUnblockedTCB->xGenericListItem ) );
        prvAddTaskToReadyQueue( pxUnblockedTCB );
    }
    else
    {
        /* We cannot access the delayed or ready lists, so will hold this
        task pending until the scheduler is resumed. */
        vListInsertEnd( ( xList * ) &( xPendingReadyList ), &( pxUnblockedTCB->xEventListItem ) );
    }
```

Portable

- Highly portable C
- 35 architectures supported
- Assembly is kept minimum
- Ports are freely available in source code
- Other contributions do exist

Scalable

- Only use the services you only need.
  - FreeRTOSConfig.h
- Pretty darn small for what you get.
  - ~6000 lines of code (including a lot of comments, maybe half that without?)
- Minimum footprint = 4 KB

Preemptive and Cooperative Scheduling

- Preemptive scheduling:
  - Fully preemptive
  - Always runs the highest priority task that is ready to run
  - Comparable with other preemptive kernels
  - Used in conjunction with tasks

- Cooperative scheduling:
  - Context switch occurs if:
    - A task/co-routine blocks
    - Or a task/co-routine yields the CPU
  - Used in conjunction with tasks/co-routines

Multitasking

- No software restriction on:
  - # of tasks that can be created
  - # of priorities that can be used

- Priority assignment
  - More than one task can be assigned the same priority.
  - RR with time slice = 1 RTOS tick

Advanced Features

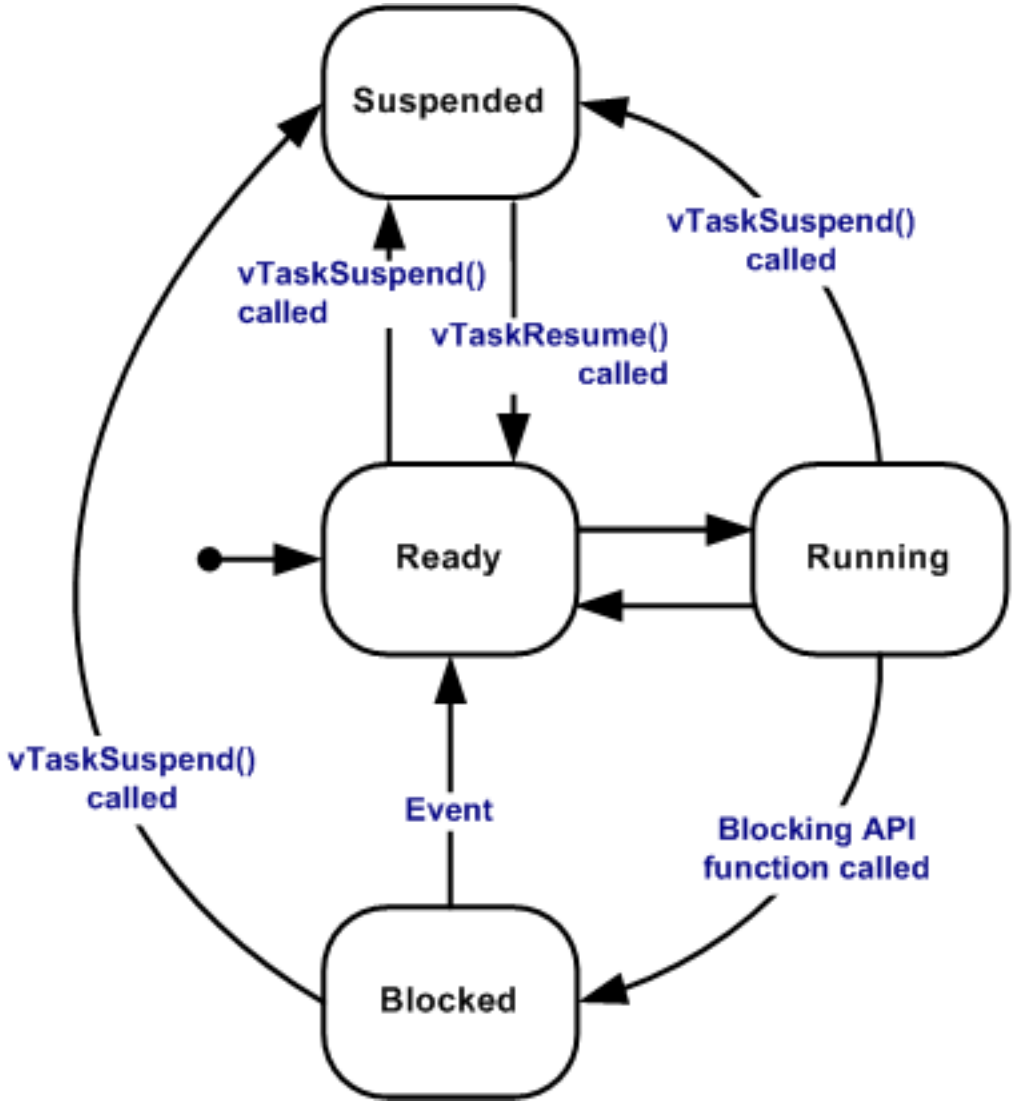
- Execution tracing
- Run time statistics collection
- Memory management
- Memory protection support
- Stack overflow protection

Device support in related products

- Connect Suite from High Integrity Systems
  - TCP/IP stack
  - USB stack
    - Host and device
  - File systems
    - DOS compatible FAT

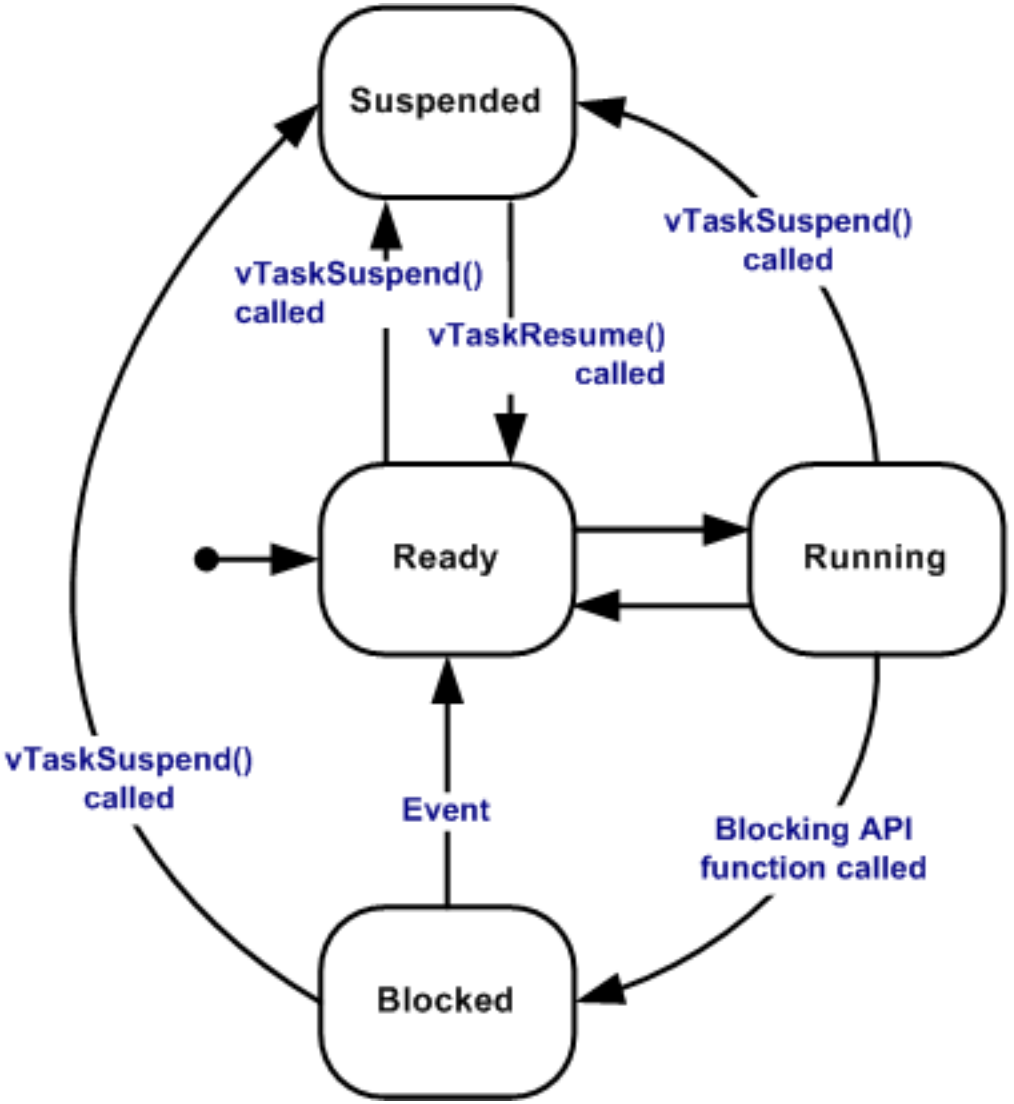
Task status in FreeRTOS

- Running
  - Task is actually executing
- Ready
  - Task is ready to execute but a task of equal or higher priority is Running.



Task status in FreeRTOS

- Blocked
  - Task is waiting for some event.
    - Time: if a task calls vTaskDelay() it will block until the delay period has expired.
    - Resource: Tasks can also block waiting for queue and semaphore events.
- Suspended
  - Much like blocked, but not waiting for anything.
  - Tasks will only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.



RTOS for Arduino

Real Time Operating Systems (RTOS) for ATmega

Arduino and the Bootloader

- Arduino, the ATmega variant, lacks an OS
- Rather, a simple program:
  - Waits for a new incoming sketch over serial.
  - If a new sketch is uploaded, the bootloader loads it into flash memory.
  - If no new sketch is received, jumps to memory beyond the bootloader (i.e. the current sketch) and executes it.

The Big Loop

- Once execution moves past the bootloader, the master loop is entered.
- There is a single “thread” of execution.
- Functions that need to be executed in the background can be implemented via ISRs.
- May be entirely appropriate for certain applications!
  - Ask yourself,
  - Does the app in question need multiple threads of execution?
  - Do I really need a RTOS?

RTOS – Advantages

- Built-in support for threads
  - Though the processor only handles on thing at a time, rapid switching gives the illusion of simultaneous execution
- Can result in more efficient use of processor time (assuming efficient scheduling)
- Enables integration of numerous modules. Developers have confidence that threads will be managed efficiently and safely.

RTOS – Advantages

- Threading syntactically more accessible to developers, who may be intimidated / put off by interrupts.
- Formal prioritization of threading (if present) can ensure high priority threads don’t wait on those that are less important

RTOS – Disadvantages

- Memory Footprint
  - The RTOS and application code for managing threads will, undoubtedly consume extra memory
- Processor Overhead
  - Incurred during thread management, etc.
- Priority Inversion
  - If the RTOS doesn’t have built-in prioritization and a mechanism for enforcing it, a higher-priority thread can find itself waiting for one of lower priority

RTOS – Disadvantages

- Deadlock
  - In any multi-threaded system, the danger of deadlocks arising from resource contention is an issue
- Complexity
  - Coding and debugging can be more difficult when threads are involved
- Learning curve
  - In addition the programming language, OS-specific calls and syntax must be learned

Embedded RTOS Examples

- FreeRTOS
  - http://www.freertos.org/a00098.html
  - Boasts support for numerous devices
    - Support for ATMega seems limited
  - Threading
    - Unlimited # of tasks
    - Unlimited # of priorities and flexible assignment
- Femto OS
  - http://www.femtoos.org/
  - Small footprint
  - Wide Atmel support

Embedded RTOS Examples

- Nut/OS
  - http://www.ethernut.de/en/software/index.html
  - Support for numerous ATMega versions
  - Two relevant implementations
    - EtherNut for wired networking
    - BTNut for wireless communication via Bluetooth
- DuinOS
  - https://github.com/DuinOS/DuinOS
  - RTOS for Arduino
  - “Native” to Arduino and meant for use in Arduino programming environment
  - Good Code Example using “task loops”