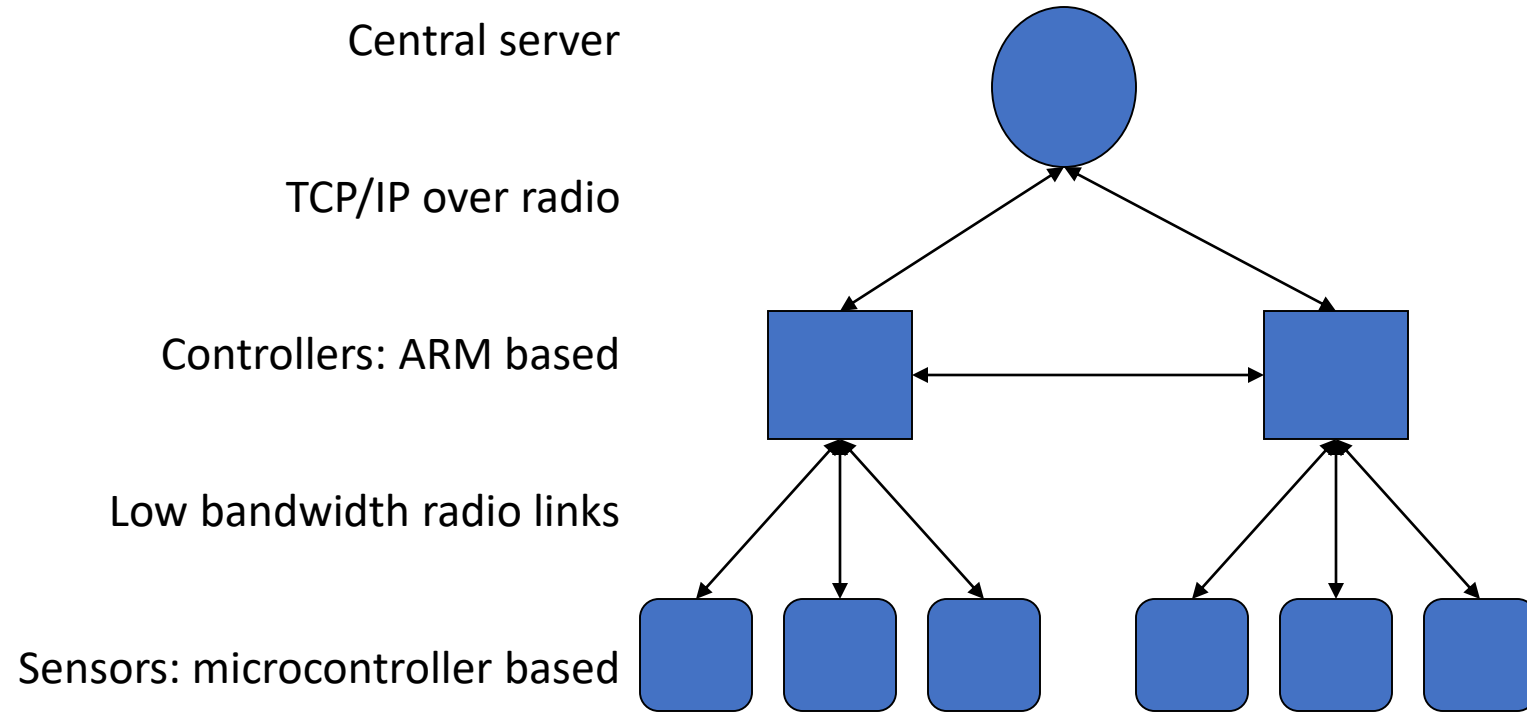


Real Time Operating Systems (RTOS)

ET2223 Microprocessors, Microcontrollers, and Embedded Systems

*Based on content by
Subhashis Banerjee, Indian Institute of Technology, Delhi &
Akos Ledeczki, Vanderbilt University*

Fire alarm system: An example



Fire alarm system: An example

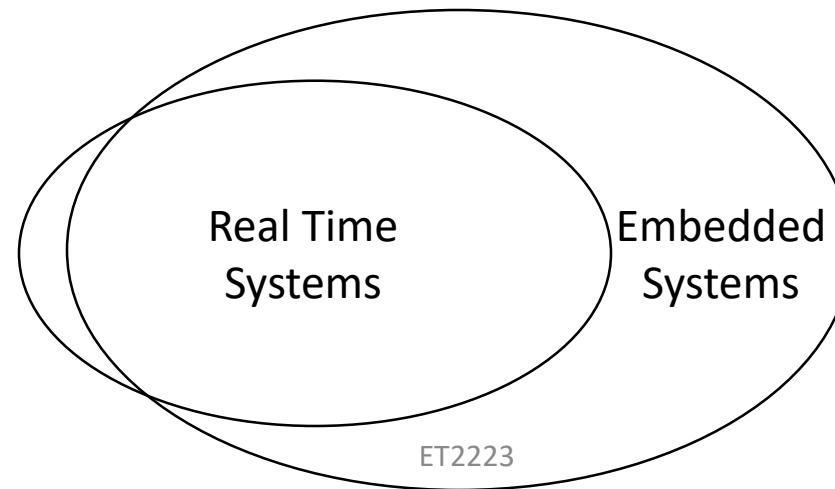
- Problem
 - Hundreds of sensors, each fitted with Low Range Wireless
 - Sensor information to be logged in a server & appropriate action initiated
- Possible Solution
 - Collaborative Action
 - Routing
 - Dynamic – Sensors/controllers may go down
 - Auto Configurable – No/easy human intervention.
 - Less Collision/Link Clogging
 - Less number of intermediate nodes – Fast Response Time
 - Secure

Real Time Systems

- A system is said to be Real Time if it is required to complete it's work and deliver it's services on time.
- Example – Flight Control System
 - All tasks in that system must execute on time.
- Non Example – PC system

Embedded vs. Real Time Systems

- Embedded system:
 - A computer system that performs a limited set of specific functions.
 - It often interacts with its environment.
- Real Time Systems:
 - Correctness of the system depends not only on the logical results, but also on the time in which the results are produced.



Examples?

Examples

- Real Time Embedded:
 - Nuclear reactor control; Flight control; Basically any safety critical system
 - GPS; Mobile phone
- Real Time, but not Embedded:
 - Stock trading system
 - Skype
- Embedded, but not Real Time:
 - Home temperature control
 - Sprinkler system
 - Washing machine, refrigerator, etc.
 - Blood pressure meter

Characteristics of Real Time Systems

- Event-driven, reactive
- High cost of failure
- Concurrency/multiprogramming
- Stand-alone/continuous operation
- Reliability/fault-tolerance requirements
- Predictable behavior

What's Important in Real Time Systems

- Metrics for real-time systems differ from that for time-sharing systems.

	Time-Sharing Systems	Real-Time Systems
Capacity	High throughput	Schedulability
Responsiveness	Fast average response	Ensured worst-case response
Overload	Fairness	Stability

- schedulability is the ability of tasks to meet all hard deadlines
- latency is the worst-case system response time to events
- stability in overload means the system meets critical deadlines even if all deadlines cannot be met

Hard and Soft Real Time Systems (Qualitative Definition)

- Hard Real Time System
 - Failure to meet deadlines is fatal
 - example : Flight Control System
- Soft Real Time System
 - Late completion of jobs is undesirable but not fatal.
 - System performance degrades as more & more jobs miss deadlines
 - Online Databases

Hard and Soft Real Time Systems (Operational Definition)

- Hard Real Time System
 - Validation by provably correct procedures or extensive simulation that the system always meets the timings constraints
- Soft Real Time System
 - Demonstration of jobs meeting some statistical constraints suffices.
- Example – Multimedia System
 - 25 frames per second on an average

Role of an OS in Real Time Systems

- Standalone Applications
 - Often no OS involved
 - Micro controller based Embedded Systems
- Some Real Time Applications are huge & complex
 - Multiple threads
 - Complicated Synchronization Requirements
 - Filesystem / Network / Windowing support
 - OS primitives reduce the software design time
- Real Time Operating System (RTOS)

Features of RTOS

- Scheduling
- Resource Allocation
- Interrupt Handling
- Other issues like kernel size

Scheduling in RTOS

- More information about the tasks are known
 - No of tasks
 - Resource Requirements
 - Release Time
 - Execution time
 - Deadlines
- Being a more deterministic system better scheduling algorithms can be devised.

Scheduling Algorithms in RTOS

- Clock Driven Scheduling
 - All parameters about jobs (release time/ execution time/deadline) known in advance.
- Weighted Round Robin Scheduling
 - Jobs scheduled in FIFO manner
 - Example use : High speed switching network
- Priority Scheduling
 - Processor allocated to processes according to priorities
 - (Greedy / List / Event Driven)

Priority Scheduling

- Earliest Deadline First (EDF)
 - Process with earliest deadline given highest priority
- Least Slack Time First (LSF)
 - $\text{slack} = \text{relative deadline} - \text{execution left}$
- Rate Monotonic Scheduling (RMS)
 - For periodic tasks
 - Tasks priority inversely proportional to it's period

Resource Allocation in RTOS

- Resource Allocation
 - The issues with scheduling applicable here.
 - Resources can be allocated in
 - Weighted Round Robin
 - Priority Based
- Some resources are non preemptible
 - Example : semaphores
- Priority Inversion if priority scheduling is used

Other RTOS issues

- Interrupt Latency should be very small
 - Kernel has to respond to real time events
 - Interrupts should be disabled for minimum possible time
- For embedded applications Kernel Size should be small
 - Should fit in ROM
- Sophisticated features can be removed
 - No Virtual Memory
 - No Protection

RTOS Examples

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

RTLinux

Linux for Real Time Applications

- Scheduling
 - Priority Driven Approach
 - Optimize average case response time
 - Interactive Processes Given Highest Priority
 - Aim to reduce response times of processes
 - Real Time Processes
 - Processes with high priority
 - No notion of deadlines
- Resource Allocation
 - No support for handling priority inversion

Interrupt Handling in Linux

- Interrupts are disabled in ISR/critical sections of the kernel
- No worst case bound on interrupt latency available
 - eg: Disk Drivers may disable interrupt for few hundred milliseconds
- Not suitable for Real Time Applications
 - Interrupts may be missed

Other Problems with Linux

- Processes are non preemptible in Kernel Mode
 - System calls like fork take a lot of time
 - High priority thread might wait for a low priority thread to complete it's system call
- Processes are heavy weight
 - Context switch takes several hundred microseconds

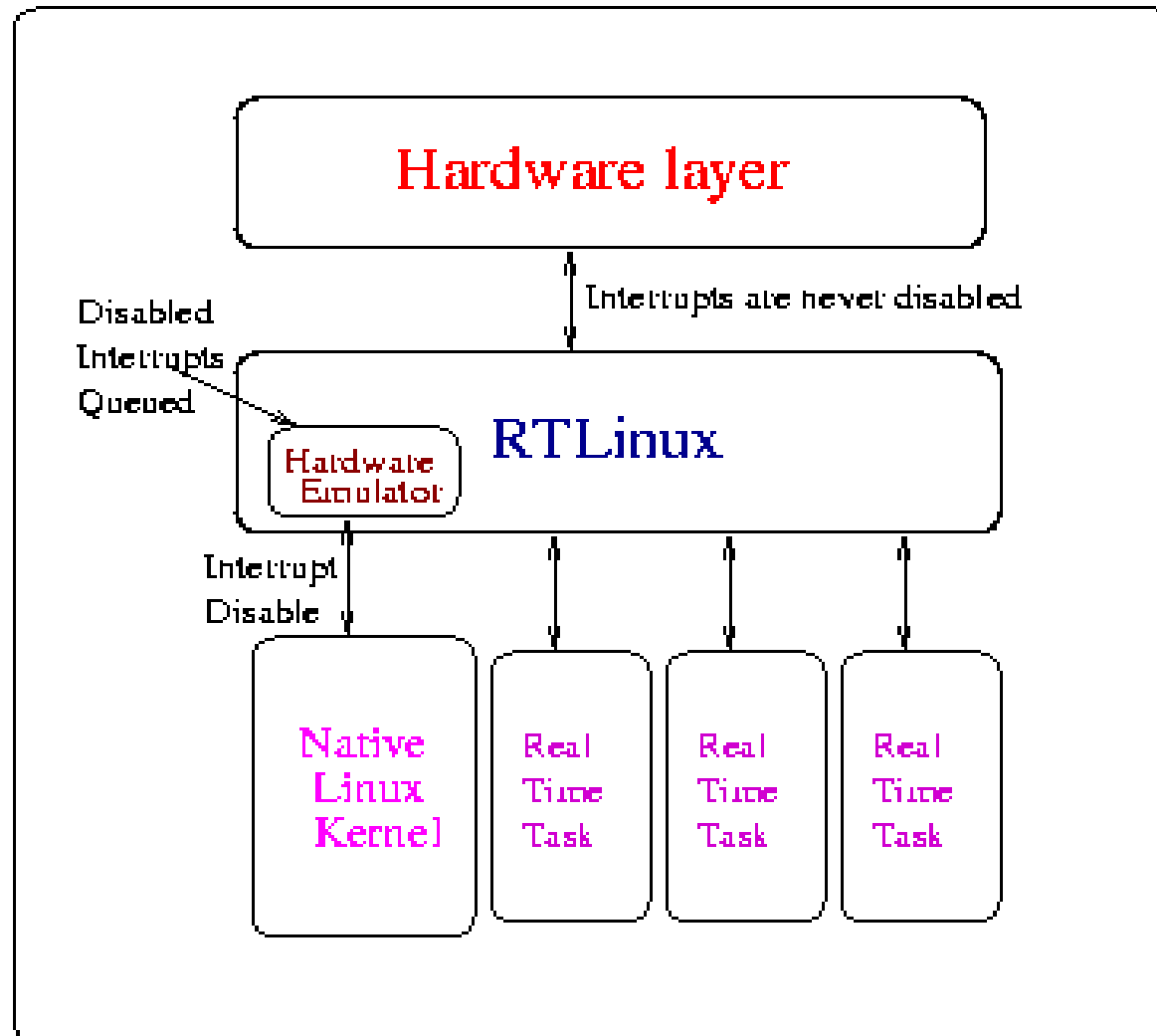
RTLinux

- Real Time Kernel at the lowest level
- Linux Kernel is a low priority thread
 - Executed only when no real time tasks
- Interrupts trapped by the Real Time Kernel and passed onto Linux Kernel
 - Software emulation to hardware interrupts
 - Interrupts are queued by RTLinux
 - Software emulation to `disable_interrupt()`

RTLinux (contd)

- Real Time Tasks
 - Statically allocate memory
 - No address space protection
- Non Real Time Tasks are developed in Linux
- Communication
 - Queues
 - Shared memory

RTLinux Framework



RTX

RTX

- Royalty-free, deterministic, open source RTOS
- High-Speed real-time operation with low interrupt latency
- Flexible Scheduling: round-robin, pre-emptive, and collaborative
- Small footprint for resource constrained systems
- Compatible with ARM cores (from ARM7, ARM9 to Cortex-M processors) and software tools (Keil MDK-ARM)
- Support for multithreading and thread-safe operation
- Kernel aware debug support in Keil MDK-ARM
- Dialog-based setup using μ Vision Configuration Wizard

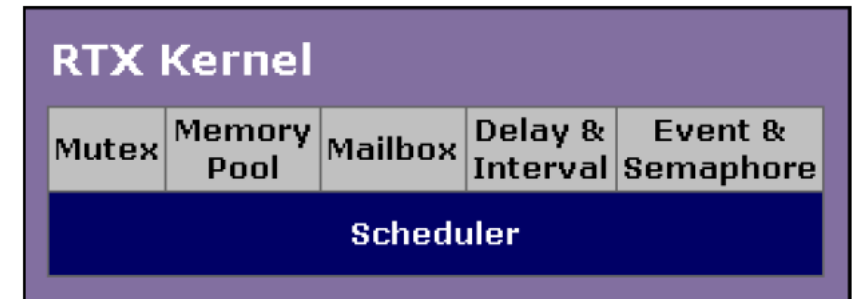
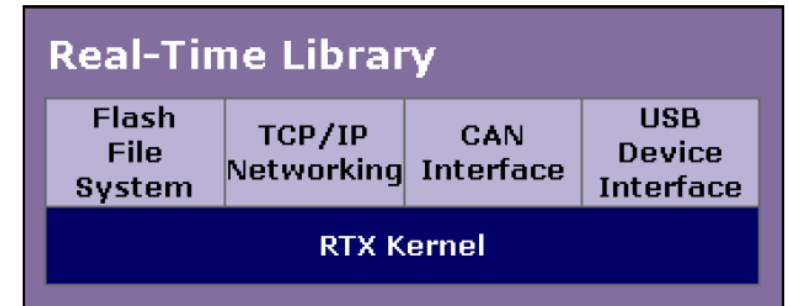
RTX Structure

- Keil Real-Time Library (RTL)

- RTX Kernel
- Flash file system
- Networking
- CAN interface
- USB device interface

- RTX Kernel

- Scheduler is the core of the RTX kernel
- Supports for mutex, memory pool, mailbox, timing functions, events and semaphores



μC/OS

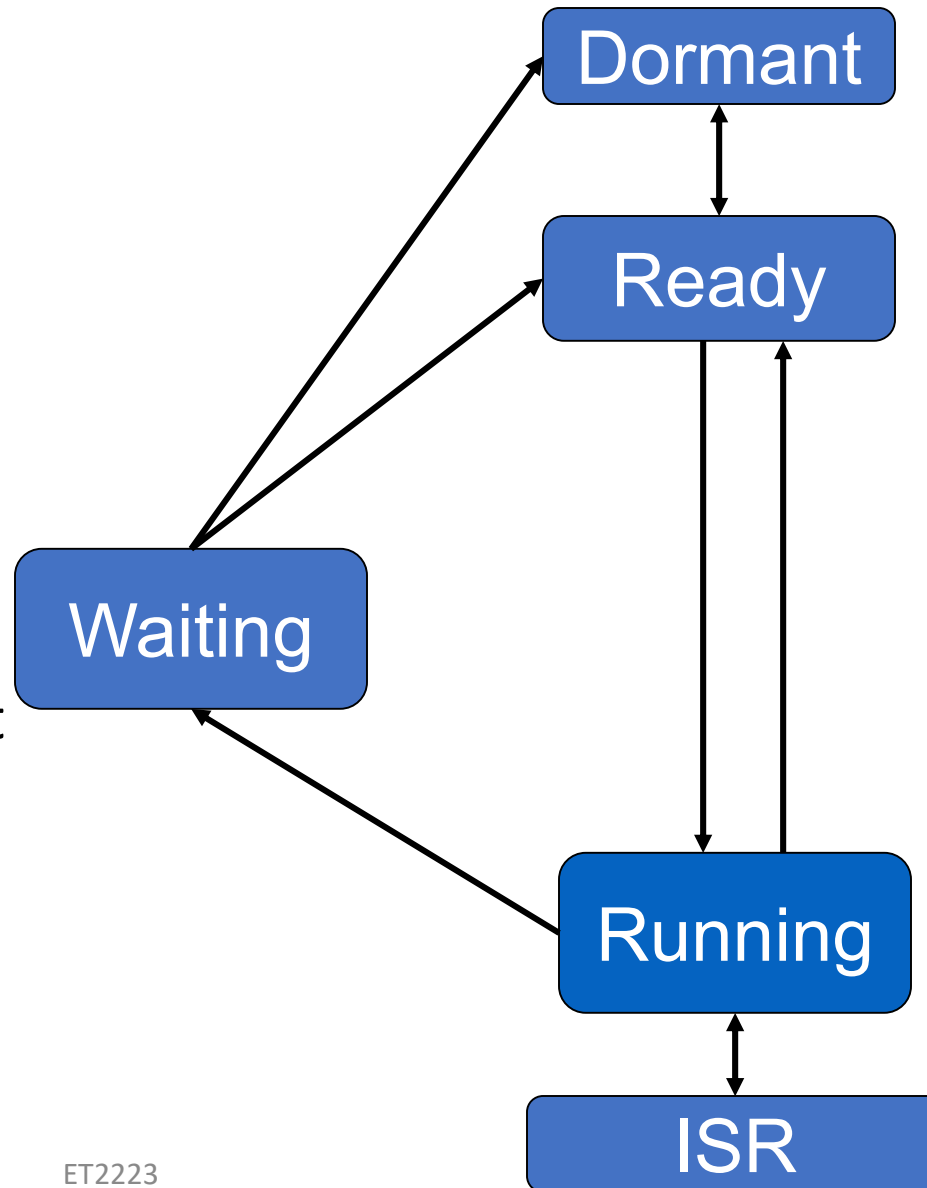
Pronounced “micro C OS”, a full-featured embedded operating system.

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 priority 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
 - ppos: 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( &(amp; pxUnblockedTCB->xEventListItem) );

    if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE )
    {
        vListRemove( &(amp; 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 * ) &(amp; xPendingReadyList), &(amp; 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

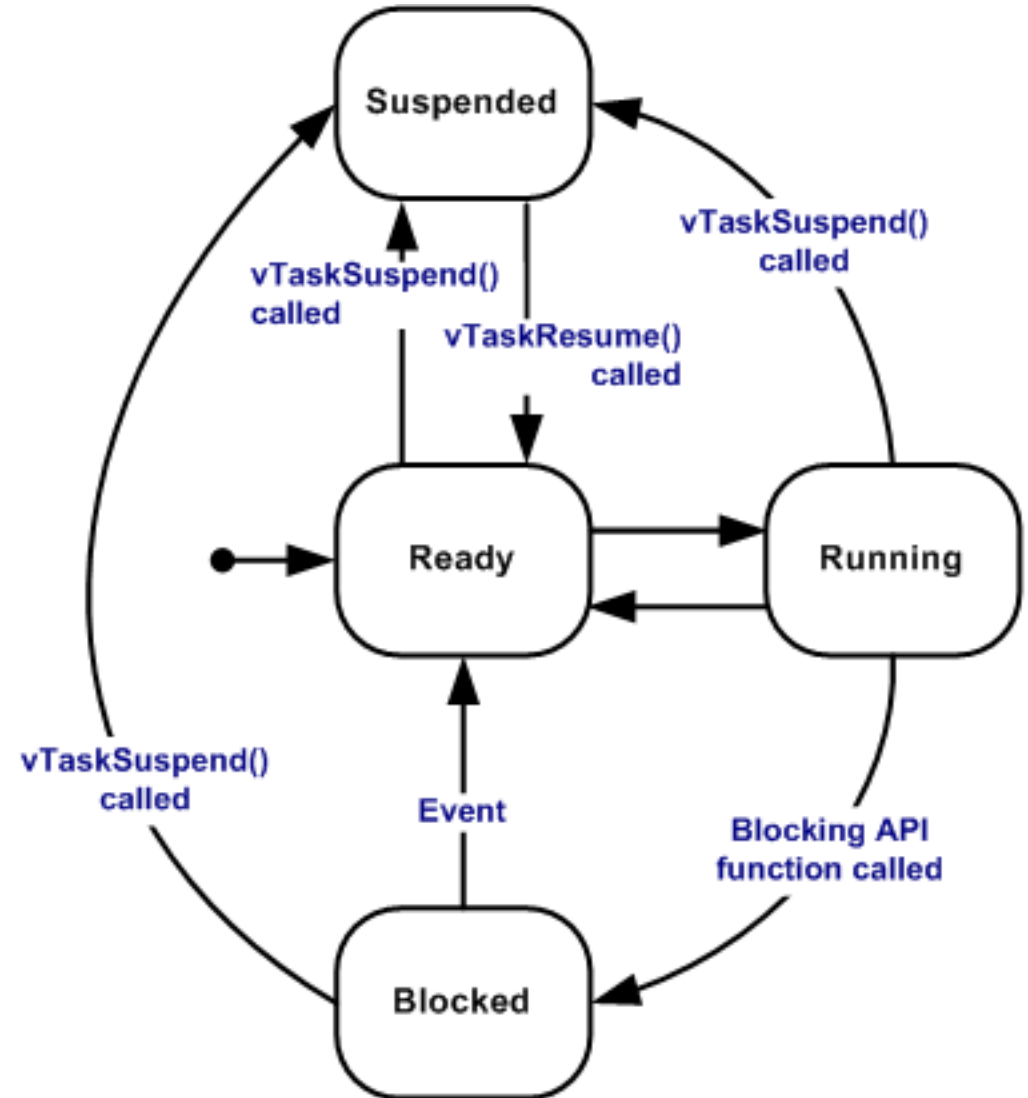
- Execution tracing
- Run time statistics collection
- Memory management
- Memory protection support
- Stack overflow protection

Device support in related products

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

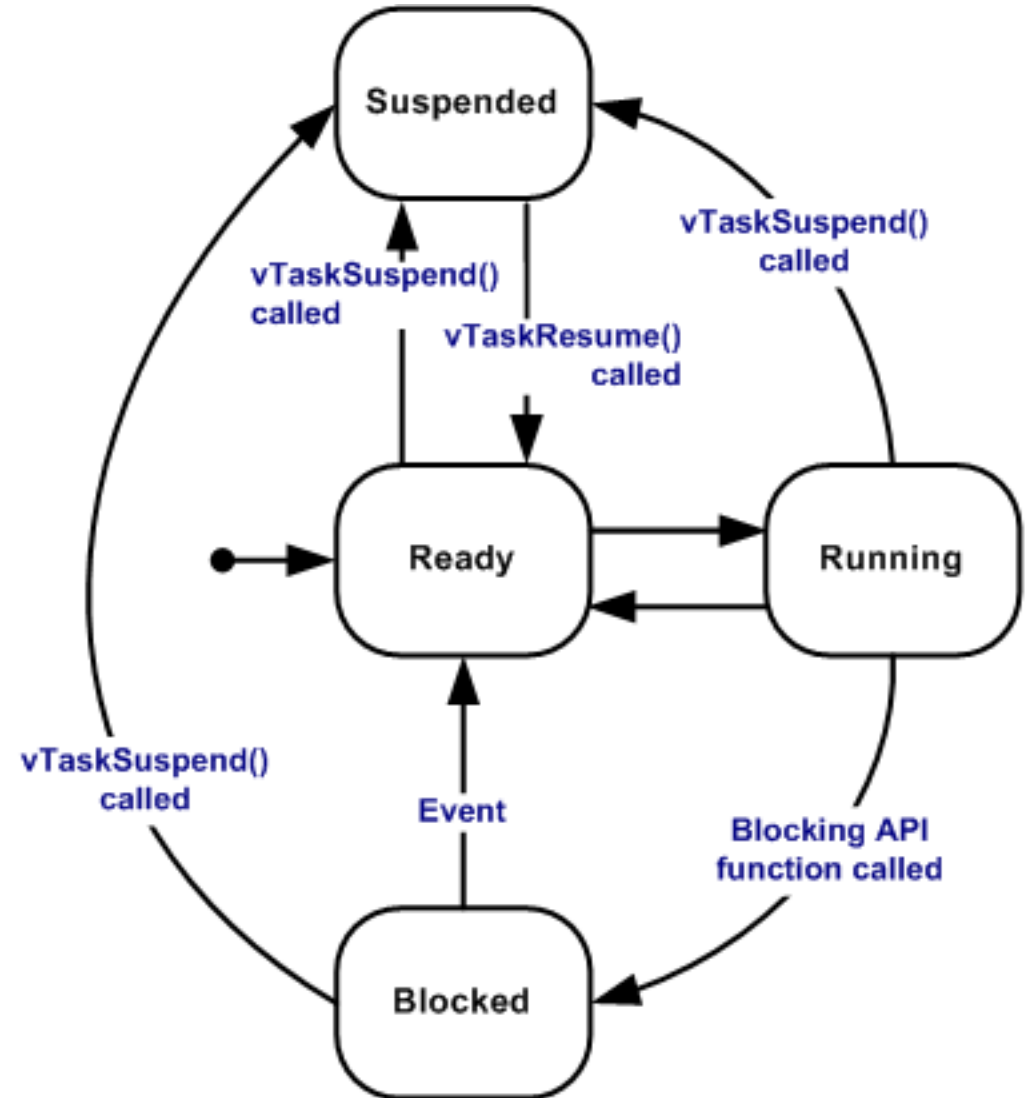
Task status in FreeRTOS

- 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 one 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

Other RTOS's

- LynxOS
 - Microkernel Architecture
 - Kernel provides scheduling/interrupt handling
 - Additional features through Kernel Plug Ins(KPIs)
 - TCP/IP stack , Filesystem
 - KPI's are multithreaded
 - Memory Protection/ Demand Paging Optional
 - Development and Deployment on the same host
 - OS support for compilers/debuggers

Other RTOS's (contd)

- VxWorks
 - Monolithic Architecture
 - Real Time Posix compliant
 - Cross development System
- pSOS
 - Object Oriented OS

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”