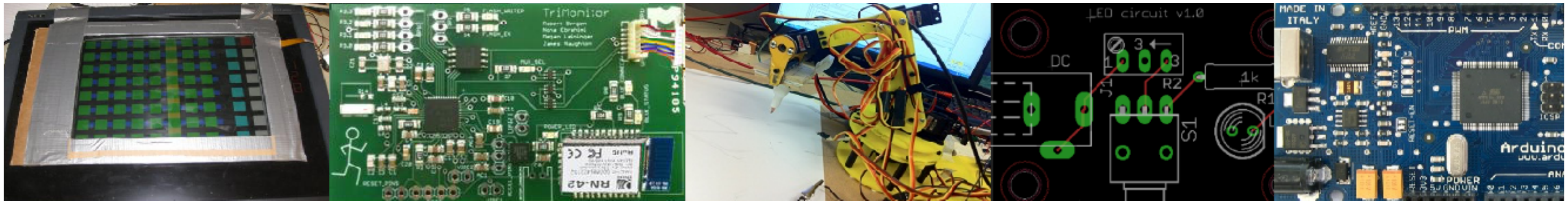


EECS 373

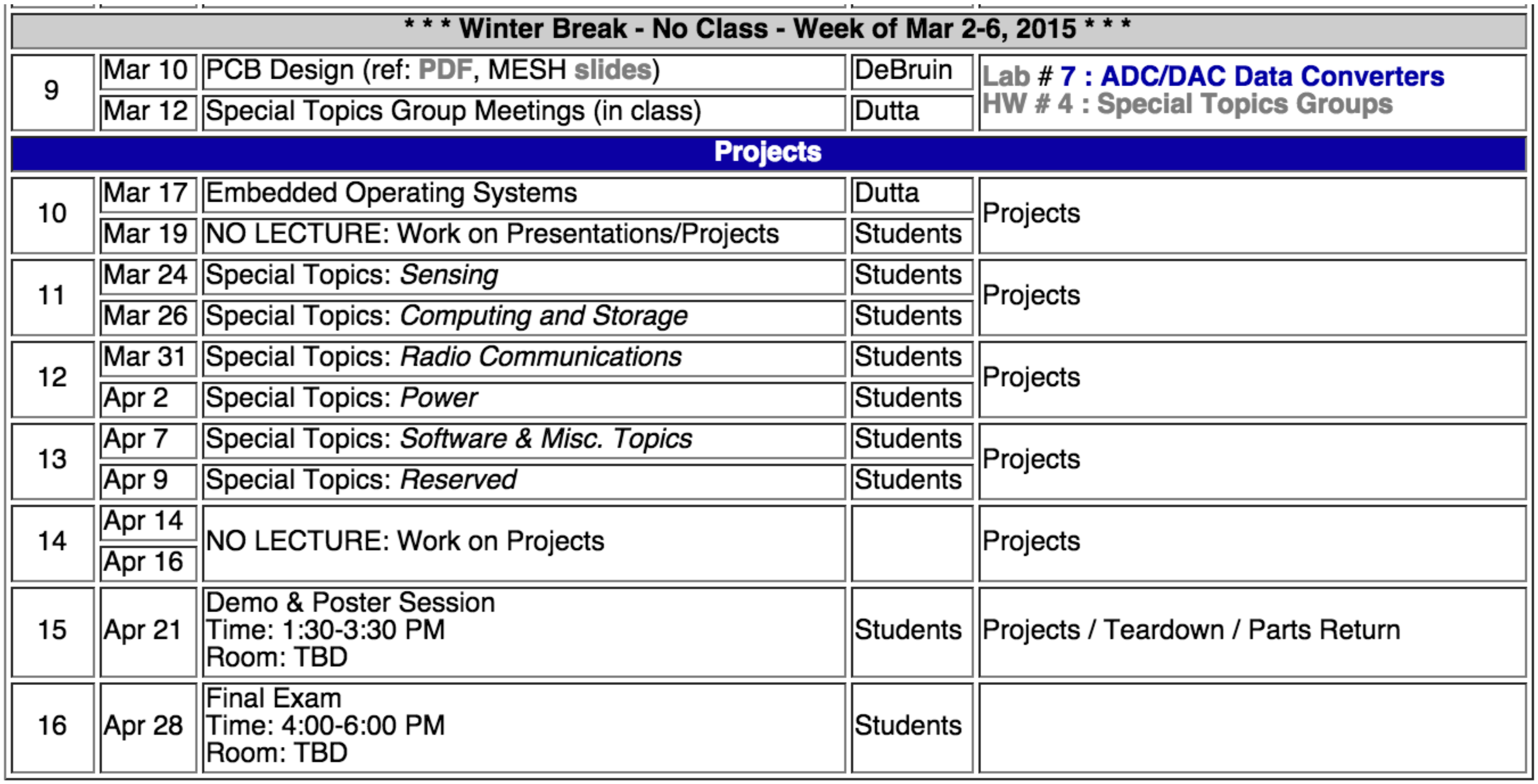
An Introduction to Real Time Oses

Slides originally created by Mark Brehob

Things

- Should be working on your project at this point
  - Your group should have something to work on as of today.
  - Your group should target having all components be "mostly working" by the second week of April
- We're ending the "regular" lectures & labs
  - Focus on special topics
    - Shared slides, spreadsheets
  - More time for project work

What’s left



* * * Winter Break - No Class - Week of Mar 2-6, 2015 * * *				
9	Mar 10	PCB Design (ref: **PDF**, MESH **slides**)	DeBruin	**Lab # 7 : ADC/DAC Data Converters** **HW # 4 : Special Topics Groups**
	Mar 12	Special Topics Group Meetings (in class)	Dutta	
Projects				
10	Mar 17	Embedded Operating Systems	Dutta	Projects
	Mar 19	NO LECTURE: Work on Presentations/Projects	Students	
11	Mar 24	Special Topics: *Sensing*	Students	Projects
	Mar 26	Special Topics: *Computing and Storage*	Students	
12	Mar 31	Special Topics: *Radio Communications*	Students	Projects
	Apr 2	Special Topics: *Power*	Students	
13	Apr 7	Special Topics: *Software & Misc. Topics*	Students	Projects
	Apr 9	Special Topics: *Reserved*	Students	
14	Apr 14 Apr 16	NO LECTURE: Work on Projects		Projects
15	Apr 21	Demo & Poster Session Time: 1:30-3:30 PM Room: TBD	Students	Projects / Teardown / Parts Return
16	Apr 28	Final Exam Time: 4:00-6:00 PM Room: TBD	Students	

Outline

- Quick review of real-time systems
- Overview of RTOSes
  – Goals of an RTOS
  – Features you might want in an RTOS
- Learning by example: FreeRTOS
  – Introduction
  – Tasks
  – Interrupts
  – Internals (briefly)
  – What's missing?

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

What is a Real-Time System?

- Real-time systems have been defined as: "those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced";
  - J. Stankovic, "Misconceptions About Real-Time Computing," ***IEEE Computer,*** 21(10), October 1988.

Real-Time Characteristics

- Pretty much your typical embedded system
  - Sensors & actuators all controlled by a processor.
  - The big difference is **timing constraints** (deadlines).
- Those tasks can be broken into two categories[1]
  - **Periodic Tasks**: Time-driven and recurring at regular intervals.
    - A car checking for a wall every 0.1 seconds;
    - An air monitoring system grabbing an air sample every 10 seconds.
  - **Aperiodic**: event-driven
    - That car having to react to a wall it found
    - The loss of network connectivity.

[1]Sporadic tasks are sometimes also discussed as a third category. They are tasks similar to aperiodic tasks but activated with some known bounded rate. The bounded rate is characterized by a minimum interval of time between two successive activations.

Some Definitions

- **Timing constraint:** constraint imposed on timing behavior of a job: hard, firm, or soft.
- **Release Time**: Instant of time job becomes available for execution.
- **Deadline**: Instant of time a job's execution is required to be completed. If deadline is infinity, then job has no deadline.
- **Response time**: Length of time from release time to instant job completes.

Soft, Firm and Hard deadlines

- The instant at which a result is needed is called a deadline.
  - If the result has utility even after the deadline has passed, the deadline is classified as **soft**, otherwise it is **firm**.
  - If a catastrophe ***could*** result if a firm deadline is missed, the deadline is **hard**.
- Examples?

Definitions taken from a paper by Kanaka Juvva, not sure who originated them.

Scheduling algorithms

- A scheduling algorithm is a scheme that selects what job to run next.
  - Can be preemptive or non-preemptive.
  - Dynamic or static priorities
  - Etc.

In general, a RTS will use some scheduling algorithm to meet its deadlines.

Two common scheduling schemes

- **Rate monotonic (RM)**
  - Static priority scheme
  - Preemption required
  - Simple to implement
  - Nice properties

- **Earliest deadline first (EDF)**
  - Dynamic priority scheme
  - Preemption required
  - Harder to implement
  - Very nice properties

We aren't going to worry about the details of either. The point is that we sometimes want static priorities (each task has a fixed priority) and sometimes we want dynamic priorities (priorities change for a task over time).

But tasks don’t operate in a vacuum

- It is generally the case that different tasks might need shared resources
  - For example, multiple tasks might wish to use a UART to print messages
    - You’ve seen this in the lab.
- How can we share resources?
  - Could have task using resource disable interrupts while using resource.
    - But that would mess with interrupts that don’t (or won’t) use the resource.
  - Could disable those that could use the resource
    - But would mess with interrupts that won’t use it this time.

Sharing resources

- Need some kind of a lock on a resource.
  - If a high priority task finds a resource is locked, it goes to sleep until the resource is available.
  - Task is woken up when resource is freed by lower priority task.
  - Sounds reasonable, but leads to problems.

- More formally stated on next slide.

Priority Inversion

- In a preemptive priority based real-time system, sometimes tasks may need to access resources that cannot be shared.
  - The method of ensuring exclusive access is to guard the critical sections with binary semaphores.
  - When a task seeks to enter a critical section, it checks if the corresponding semaphore is locked.
  - If it is not, the task locks the semaphore and enters the critical section.
  - When a task exits the critical section, it unlocks the corresponding semaphore.
- This could cause a high priority task to be waiting on a lower priority one.
  - Even worse, a medium priority task might be running and cause the high priority task to not meet its deadline!

Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005) 14

Example: Priority inversion

- Low priority task “C” locks resource “Z”.
- High priority task “A” preempts “C” then requests resource “Z”
  - Deadlock, but solvable by having “A” sleep until resource is unlocked.
- But if medium priority “B” were to run, it would preempt C, thus effectively making C and A run with a lower priority than B.
  - Thus priority *inversion*.

Solving Priority inversion

- Priority Inheritance
  - When a high priority task sleeps because it is waiting on a lower priority task, have it boost the priority of the blocking task to its own priority.

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Goals of an RTOS?

- Well, to manage to meet RT deadlines (duh).
  - While that's all we ***need*** we'd ***like*** a lot more.
    - After all, we can meet RT deadlines fairly well on the bare metal (no OS)
      - But doing this is time consuming and difficult to get right as the system gets large.
    - We'd ***like*** something that supports us
      - Deadlines met
      - Interrupts just work
      - Tasks stay out of each others way
      - Device drivers already written (and tested!) for us
      - Portable—runs on a huge variety of systems
      - Oh, and nearly no overhead so we can use a small device!
        - » That is a small memory and CPU footprint.

Detailed features we'd like

Deadlines met

- Ability to specify scheduling algorithm
  - We'd like priority inversion dealt with
- Interrupts are fast
  - So tasks with tight deadlines get service as fast as possible
    - Basically—rarely disable interrupts and when doing so only for a short time.

Interrupts just work

- Don't need to worry about saving/restoring registers
  - Which C just generally does for us anyways.
- Interrupt prioritization easy to set.

Detailed features we'd like:
Tasks stay out of each others way

- This is actually remarkably hard
  - Clearly we need to worry about CPU utilization issues
    - That is what our scheduling algorithm discussion was to address
  - But we also need to worry about *memory* problems.
    - One task running awry shouldn't take the rest of the system down.
  - So we want to prevent tasks from harming each other
    - This can be ***key***. If we want mission critical systems sharing the CPU with less important things we have to do this.
    - Alternative it to have separate processors.
      - $$$$
- The standard way to do this is with page protection.
  - If a process tries to access memory that isn't its own, it fails.
    - Probably a fault.
    - This also makes debugging a LOT easier.
- This generally requires a lot of overhead.
  - Need some sense of process number/switching
  - Need some kind of MMU in hardware
    - Most microcontrollers lack this…
    - So we hit some kind of minimum size.

Further reading on page protection (short) http://homepage.cs.uiowa.edu/~jones/security/notes/06.shtml

Aside: What is an MMU?

- Memory Management Unit
  - Tracks what parts of memory a process can access.
    - Actually a bit more complex as it manages this by mapping virtual addresses to physical ones.
    - Keeps processes out of each other's memory.

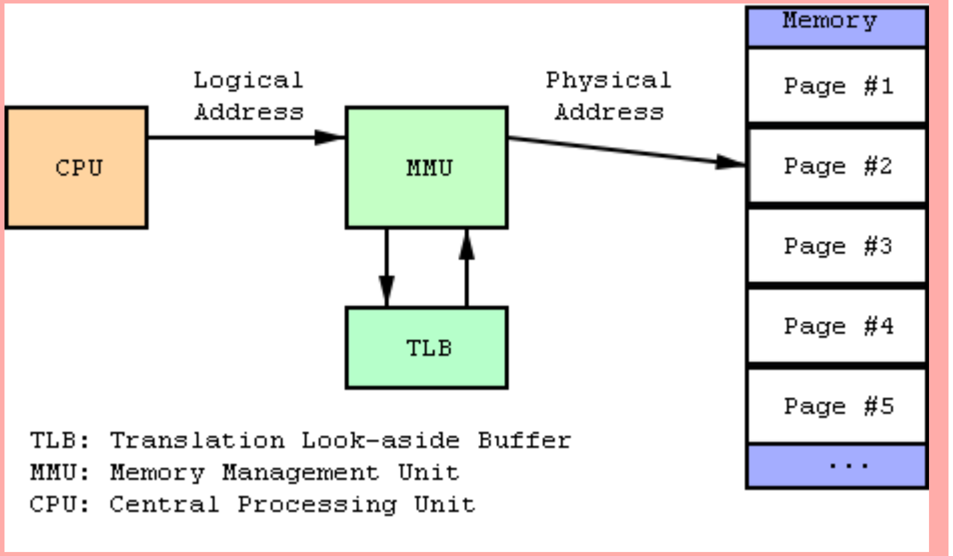

Logical Address
Physical Address
CPU
MMU
TLB
Memory
Page #1
Page #2
Page #3
Page #4
Page #5
...
TLB: Translation Look-aside Buffer
MMU: Memory Management Unit
CPU: Central Processing Unit

Figure from Wikipedia

Device drivers written (and tested!) for us

- Ideally the RTOS has drivers for all the on-board peripherals.
  - It's a lot easier to call a "configure_I2C()" function than to read the details of the device specification and do the memory-mapped work yourself

Portable

- RTOS runs on many platforms.
  - This is potentially incomputable with the previous slide.
  - It's actually darn hard to do even without peripherals
    - For example I spent 10 hours debugging a RTOS that had a pointer problem that only comes up if the pointer type is larger than the int type (20 bit pointers, 16 bit ints, yea!)
    - Things like timers change and we certainly need timers.

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Learning by example: FreeRTOS

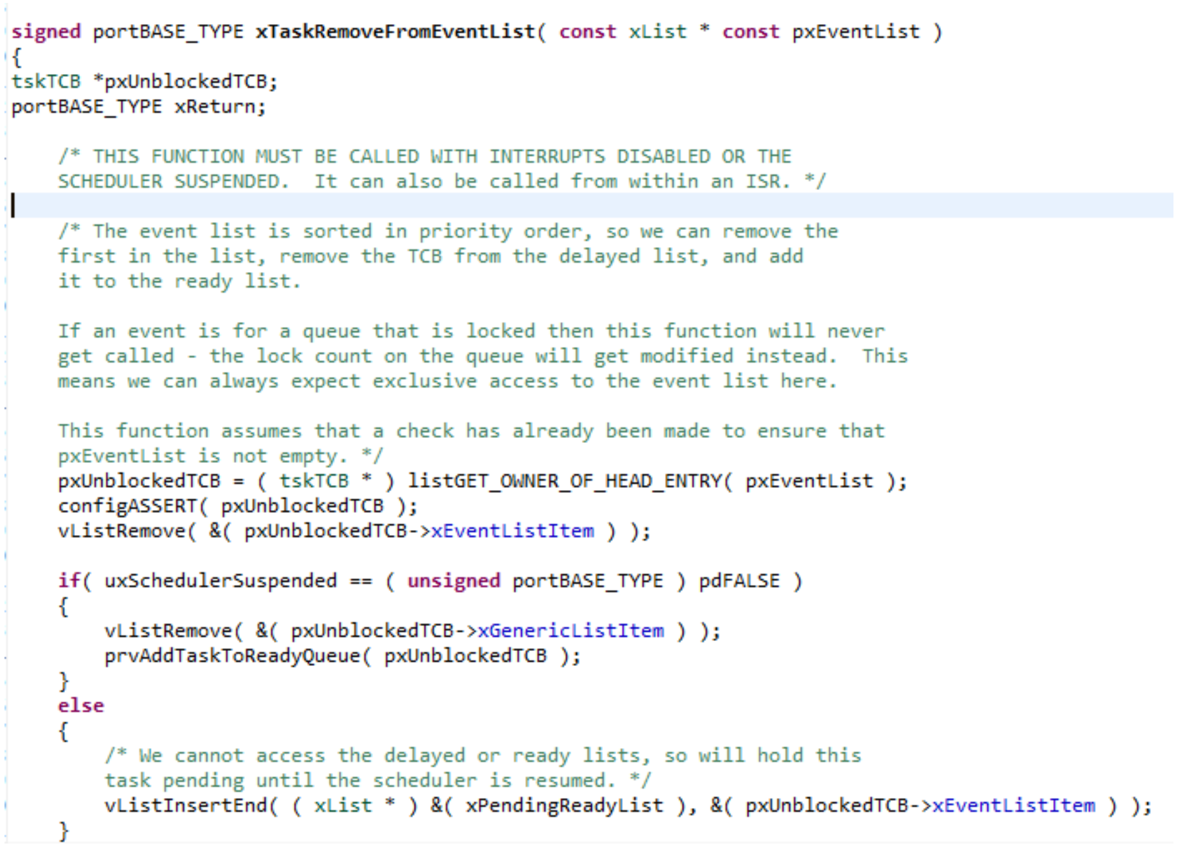
- Introduction taken from Amr Ali Abdel-Naby
  - Nice blog:
    - http://www.embedded-tips.blogspot.com

FreeRTOS Features

- Source code
- Portable
- Scalable
- Preemptive and co-operative scheduling
- Multitasking
- Services
- Interrupt management
- Advanced features

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
- 24 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
- Minimum footprint = 4 KB
- Version in lab is 24 KB including the application (which is fairly large) and data for the OS and application.
  - Pretty darn small for what you get.
  - ~6000 lines of code (including a lot of comments, maybe half that without?)

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

Services

- Queues
- Semaphores
  - Binary and counting
- Mutexes
  - With priority inheritance
  - Support recursion

Interrupts

- An interrupt can suspend a task execution.
- Interrupt mechanism is port dependent.

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

Licensing

- Modified GPL
  - Only FreeRTOS is GPL.
  - Independent modules that communicate with FreeRTOS through APIs can be anything else.
  - FreeRTOS can't be used in any comparisons without the authors' permission.

A bit more

- System runs on "ticks"
  - Every tick the kernel runs and figures out what to do next.
    - Interrupts have a different mechanism
  - Basically hardware timer is set to generate regular interrupts and calls the scheduler.
    - This means the OS eats one of the timers—you can't easily share.

OK, onto tasks!

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Tasks

- Each task is a function that must not return
  - So it's in an infinite loop (just like you'd expect in an embedded system really, think Arduino).
- You inform the scheduler of
  - The task's resource needs (stack space, priority)
  - Any arguments the tasks needs
- All tasks here must be of void return type and take a single void* as an argument.
  - You cast the pointer as needed to get the argument.
    - I'd have preferred var_args, but this makes the common case (one argument) easier (and faster which probably doesn't matter).

Code examples mostly from ***Using the FreeRTOS Real Time Kernel*** (a pdf book), fair use claimed.

Example trivial task with busy wait (bad)

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Task creation

```
portBASE_TYPE xTaskCreate(
  pdTASK_CODE pvTaskCode,
  const char * const pcName,
  unsigned short usStackDepth,
  void *pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pvCreatedTask
  );
```

Create a new task and add it to the list of tasks that are ready to run. **xTaskCreate()** can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using xTaskCreateRestricted().

- **pvTaskCode:** Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- **pcName:** A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by tskMAX_TASK_NAME_LEN – default is 16.
- **usStackDepth:** The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.
- **pvParameters**: Pointer that will be used as the parameter for the task being created.
- **uxPriority:** The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).
- **pvCreatedTask:** Used to pass back a handle by which the created task can be referenced.
- **pdPASS**: If the task was successfully created and added to a ready list, otherwise an error code defined in the file errors.h

From the task.h file in FreeRTOS

Creating a task: example

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
```

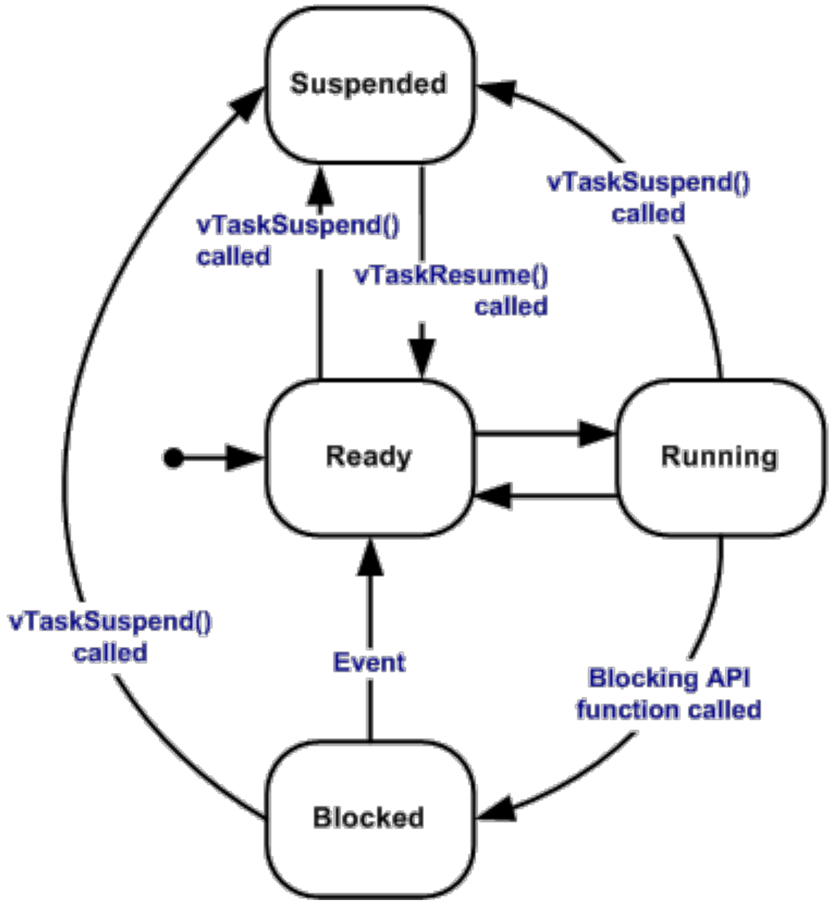
OK, I've created a task, now what?

- Task will run if there are no other tasks of higher priority
  - And if others the same priority will RR.
- But that begs the question: "How do we know if a task wants to do something or not?"
  - The previous example gave *always* wanted to run.
    - Just looping for delay (which we said was bad)
    - Instead should call **`vTaskDelay(x)`**
      - Delays current task for X "ticks" (remember those?)
    - There are a few other APIs for delaying…

Now we need an "under the hood" understanding

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.
- **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.



Mostly from http://www.freertos.org/RTOS-task-states.html

Tasks: there's a lot more

- Can do all sorts of things
  - Change priority of a task
  - Delete a task
  - Suspend a task (mentioned above)
  - Get priority of a task.
- Example on the right
  - But we'll stop here…

```
void
vTaskPrioritySet( xTask
Handle pxTask,
unsigned
uxNewPriority );
```

Set the priority of any task.

- **pxTask:** Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
- **uxNewPriority:** The priority to which the task will be set.

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Interrupts in FreeRTOS

- There is both a lot and a little going on here.
  - The interface mainly uses whatever the native environment uses to handle interrupts
    - This can be **very** port dependent. In Code Composer Studio (TI) you'd set it up as follows:

```
#pragma vector=PORT2_VECTOR
interrupt void prvSelectButtonInterrupt( void )
```

  - That would cause the code to run on the PORT2 interrupt.
    - Need to set that up etc. Very device specific (of course).

More: Deferred Interrupt Processing

- The best way to handle complex events triggered by interrupts is to **not** do the code in the ISR.
  - Rather create a task that is blocking on a semaphore.
    - When the interrupt happens, the ISR just sets the semaphore and exits.
      - Task can now be scheduled like any other. No need to worry about nesting interrupts (and thus interrupt priority).
      - FreeRTOS does support nested interrupts on some platforms though.
  - Semaphores implemented as one/zero-entry queue.

Semaphore example in FreeRTOS

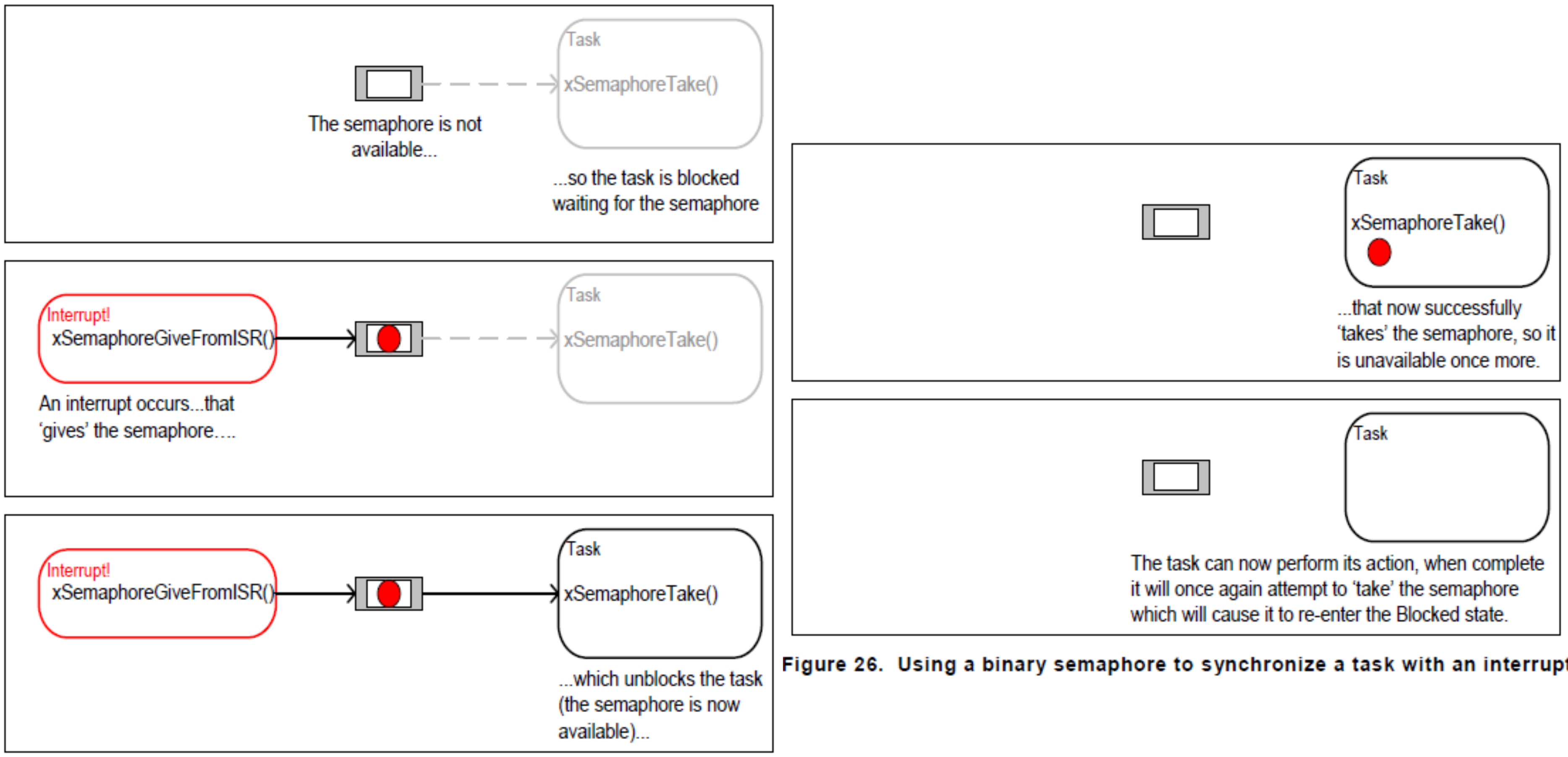

Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
An interrupt occurs...that 'gives' the semaphore....
Task
xSemaphoreTake()
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...
Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.
Task
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

Figure 26. Using a binary semaphore to synchronize a task with an interrupt

Figure from ***Using the FreeRTOS Real Time Kernel*** (a pdf book), fair use claimed.

Semaphore take

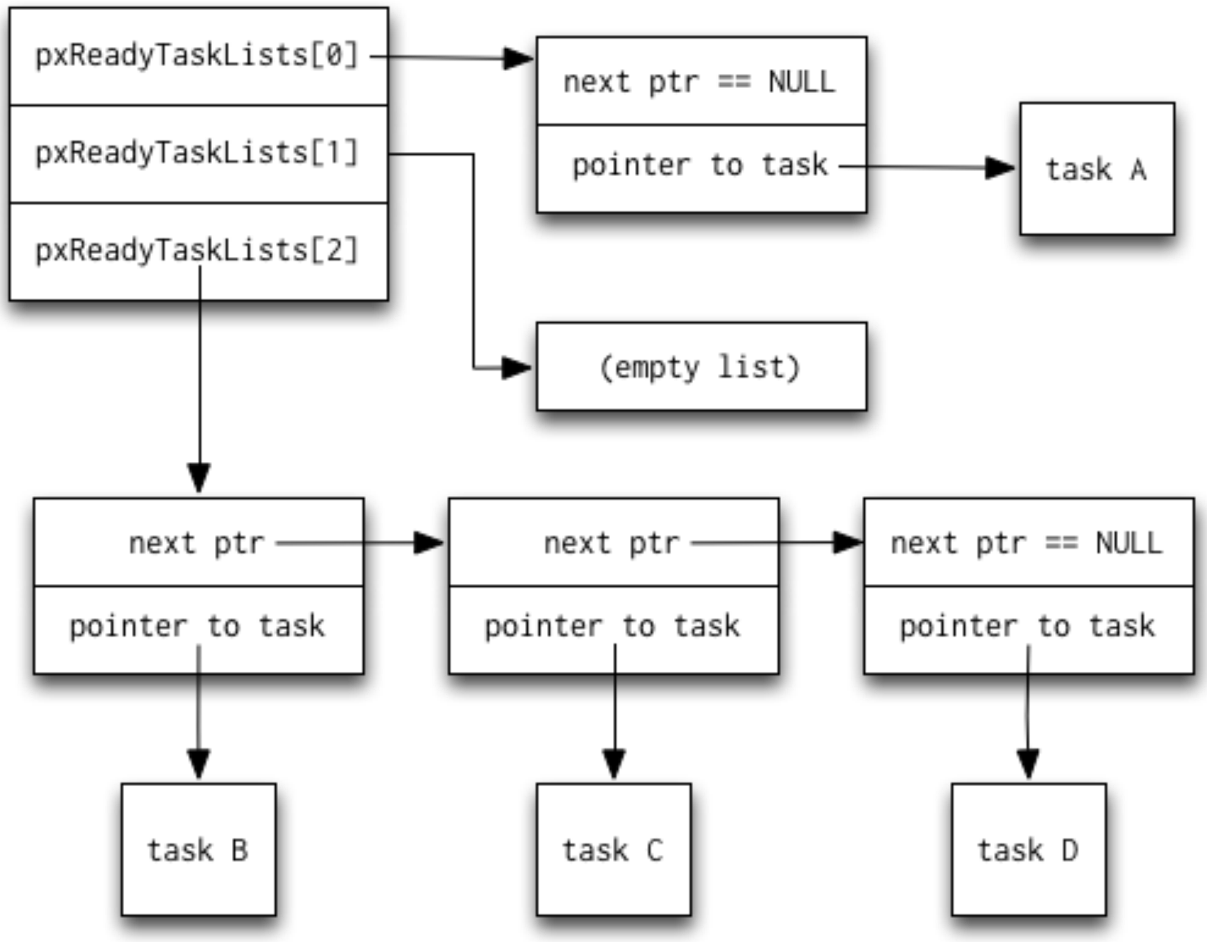
```
xSemaphoreTake(
            xSemaphoreHandle xSemaphore,
            portTickType xBlockTime
             )
```

- *Macro* to obtain a semaphore. The semaphore must have previously been created.
- **xSemaphore** A handle to the semaphore being taken - obtained when the semaphore was created.
- **xBlockTime** The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.|
- TRUE if the semaphore was obtained.
- There are a handful of variations.
  - Faster but more locking version, non-binary version, etc.

Outline

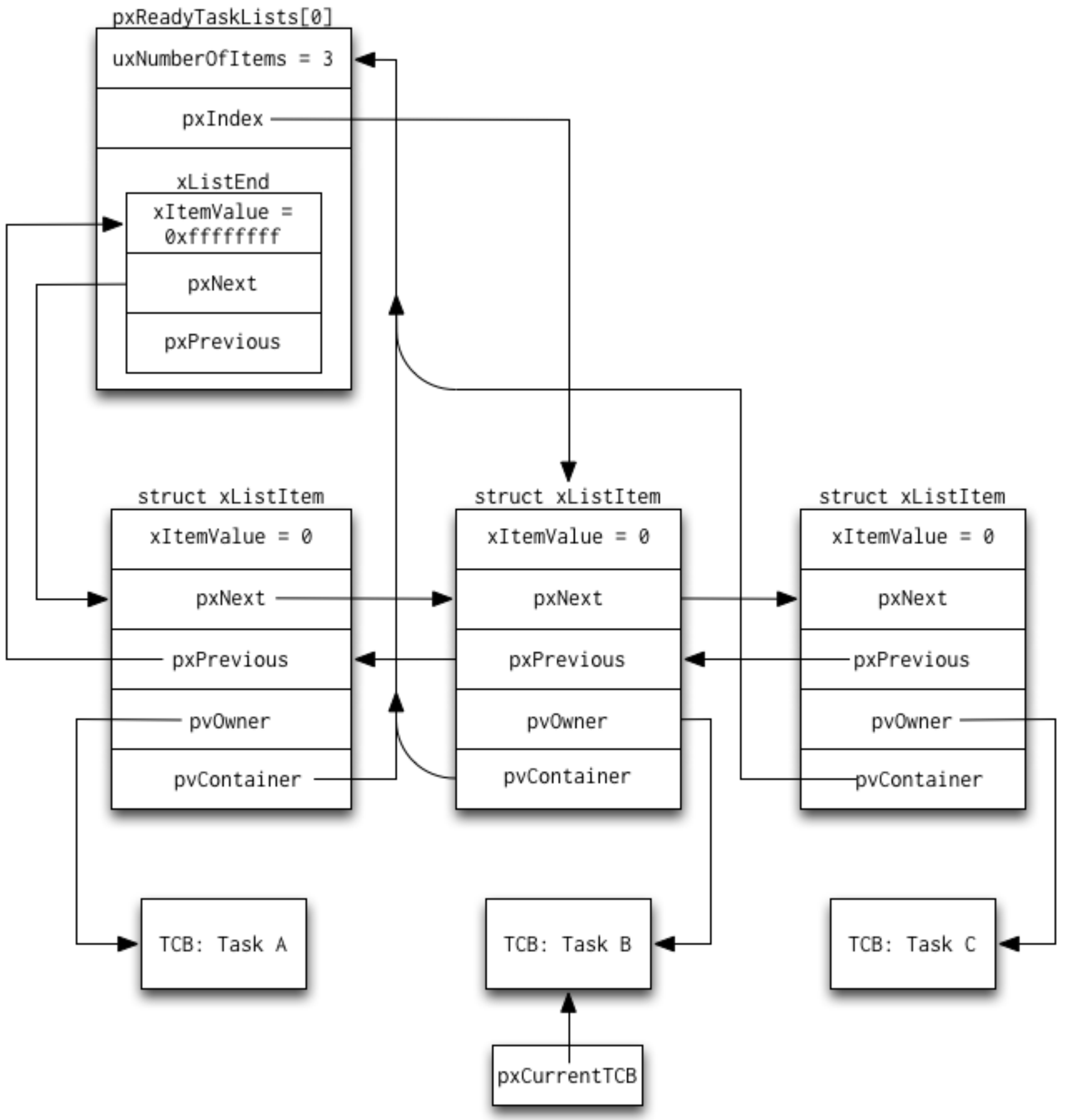
- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Common data structures



This figure and the next are from http://www.aosabook.org/en/freertos.html

FreeRTOS: Internals


pxReadyTaskLists[0]
uxNumberOfItems = 3
pxIndex
xListEnd
xItemValue = 0xffffffff
pxNext
pxPrevious
struct xListItem
xItemValue = 0
pxNext
pxPrevious
pvOwner
pvContainer
struct xListItem
xItemValue = 0
pxNext
pxPrevious
pvOwner
pvContainer
struct xListItem
xItemValue = 0
pxNext
pxPrevious
pvOwner
pvContainer
TCB: Task A
TCB: Task B
TCB: Task C
pxCurrentTCB