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

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’s left

<table>
<thead>
<tr>
<th>Lecture</th>
<th>Topic</th>
<th>Date</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>PCB Design</td>
<td>Mar 10</td>
<td>Class / Lab # 7: AD/DA Converters</td>
</tr>
<tr>
<td>10</td>
<td>Embedded Operating Systems</td>
<td>Mar 17</td>
<td>Projects</td>
</tr>
<tr>
<td>11</td>
<td>Special Topics: Designing</td>
<td>Mar 24</td>
<td>Students</td>
</tr>
<tr>
<td>12</td>
<td>Special Topics: Computer Security</td>
<td>Mar 31</td>
<td>Students</td>
</tr>
<tr>
<td>13</td>
<td>Special Topics: Software &amp; Microcontroller</td>
<td>Apr 7</td>
<td>Projects</td>
</tr>
<tr>
<td>14</td>
<td>Special Topics: Embedded Systems</td>
<td>Apr 9</td>
<td>Students</td>
</tr>
<tr>
<td>15</td>
<td>Special Topics: Embedded Systems</td>
<td>Apr 12</td>
<td>Projects</td>
</tr>
<tr>
<td>16</td>
<td>Final Exam</td>
<td>Apr 22</td>
<td>Room TBD</td>
</tr>
</tbody>
</table>

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";
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 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

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!

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 other’s 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.

```
Further reading on page protection (short): http://www.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.

    ![Figure from Wikipedia](image)

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.

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

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

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

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.

Device support in related products

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

FreeRTOS: Tasks

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).

FreeRTOS: Tasks

Example trivial task with busy wait (bad)

```c
void vTask1( void *pvParameters )
{...
```

FreeRTOS: Tasks

Task creation

```c
portBASE_TYPE xTaskCreate(...
```

Creating a task: example

```c
int main( void )
{...
```
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

FreeRTOS: Tasks

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( xTaskHandle 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?

FreeRTOS: Interrupts

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

Semaphore take

```c
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