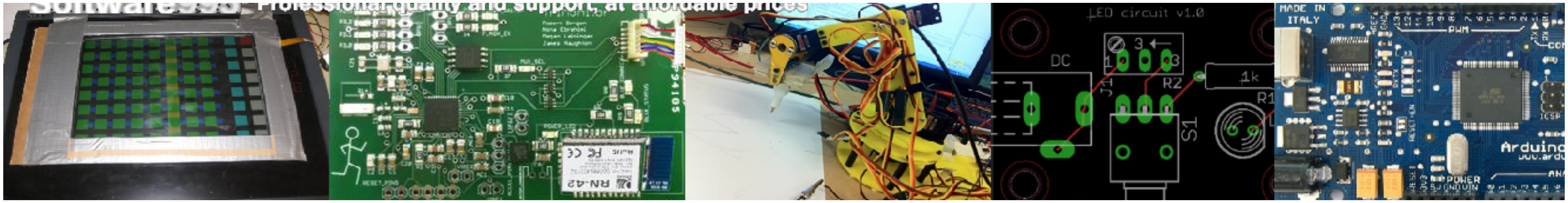


EECS 373

An very brief introduction to

- Real-time systems
- Real-time OSes

Chunks adapted from work by
Dr. Fred Kuhns of Washington University
and Farhan Hormasji

Announcements

- Schedule for the remainder of the semester
  - Today, 12/5, (Last lecture)
    - Use remaining lecture time slots to work on projects
  - 12/13 Design Expo
  - 12/15 Project Write-up Due
  - 12/18 Lab Clean-up complete
  - 12/21 Final Exam
    - 1:30-3:30pm in 1010/1018 DOW
- Please fill out course evaluation online

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.

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.

Why is this hard?

Three major issues

1. We want to use as cheap ($$, power) a processor as possible.
   – Don’t want to overpay
2. There are non-CPU resources to worry about.
   – Say two devices both on an SPI bus.
   – So often can’t treat tasks as independent
3. Validation is hard
   – You’ve got deadlines you ***must*** meet.
      • How do you ***know*** you will?

Let’s discuss that last one a bit more

What is a real-time OS (RTOS)?

- Well, an OS 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

- All the tasks can run and stay out of each other's way.
- 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.

Say you have “tasks” you want to do

- Consider a car driving around by itself. Has a lot of things it needs to do
  - Read sensors
  - Read camera (yes, it’s a sensor, but lots more CPU)
  - Drive motors
  - Make high-level decisions about what actions to take.
- For your project, you have probably found that “integration” is the hard part.
  - That is, each task isn’t so bad, but getting them all working together sucks.

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
    - scheduling algorithm
  - 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

Hardware interfaces written (and tested!) for us

- Ideally the RTOS has an interface 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 than to 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
    - Things like timers change and we certainly need timers.

A specific RTOS: FreeRTOS

- One of the more popular (and free) RTOSes out there.
  - There are many commercial ones out there with lots of support and features.
  - But FreeRTOS is:
    - Free (as in beer and speech), complete with source
    - Well documented (somewhat free)
    - Easy to use
    - Does the basics well

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).

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 taskbeing 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();
```

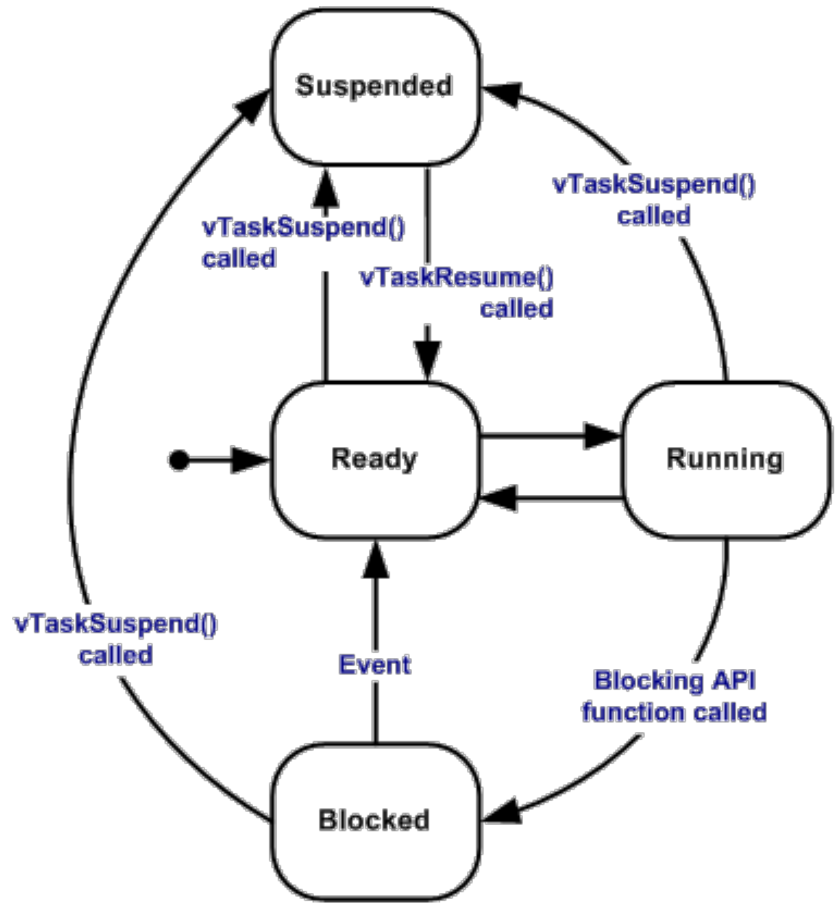
OK, I've created a task, now what?

- 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"
    - 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.

A RTOS needs to do a lot more…

- Interrupts
  - Including deferred interrupts
- Memory management
- Standard I/O interfaces
- Fast context switch
- Locks
  - So only one task can use certain resources at a time.

- FreeRTOS does each of those, some better than others.