Embedded Systems

Real Time Systems (Part I)

Real Time Operating System (RTOS) Definition and Characteristics

• A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests
• A key characteristic of an RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task
  – Variability in this task acceptance and completion is called jitter
  – A hard RTOS has less jitter than a soft RTOS
• The chief design goal is not high throughput, but rather a guarantee of a soft or hard performance category
  – An RTOS that can usually or generally meet a deadline is a soft real-time OS
  – If it can meet a deadline deterministically it is a hard real-time OS
Foreground/Background Systems

- Small systems of low complexity are generally designed as shown
- These systems are called foreground/background systems
- An application consists of an infinite loop that calls modules (functions) to perform desired operations (background)
- Interrupt service routines (ISRs) handle asynchronous events (foreground)
- Foreground is also called interrupt level; background is called task level

Foreground/Background Systems (continued)

- Critical operations must be performed by the ISRs to ensure that they are dealt with in a timely fashion
  - Because of this, ISRs have a tendency to take longer than they should
- Also, information for a background module that an ISR makes available is not processed until the background routine gets its turn to execute, which is called the task-level response
- The worst case task-level response time depends on how long the background loop takes to execute
- Most high-volume microcontroller-based applications (e.g., microwave ovens, telephones, toys, and so on) are designed as foreground/background systems
RTOS Design Philosophies

- The most common designs are:
  - Event-driven which switches tasks only when an event of higher priority needs service, called preemptive priority, or priority scheduling
  - Time-sharing designs switch tasks on a regular clock interrupt, and on events, called round robin
- Time-sharing designs switch tasks more often than strictly needed, but give smoother multitasking, giving the illusion that a process or user has sole use of a machine

RTOS Concepts Definitions

- Critical Sections of Code
  - A critical section of code, also called a critical region, is code that needs to be treated indivisibly
  - After the section of code starts executing, it must not be interrupted
- Resources
  - A resource is any entity used by a task
  - A resource can be an I/O device, a variable, a structure, or an array
- Shared Resources
  - A shared resource is a resource that can be used by more than one task
  - Each task should gain exclusive access to the shared resource to prevent data corruption
    - This process is called mutual exclusion
RTOS Concepts Definitions (continued)

- Multitasking
  - Multitasking is the process of scheduling and switching the central processing unit (CPU) between several tasks; a single CPU switches its attention between several sequential tasks
  - Multitasking is like foreground/background with multiple backgrounds
  - Multitasking maximizes the use of the CPU and also provides for modular construction of applications
  - One of the most important aspects of multitasking is that it allows the application programmer to manage complexity inherent in real-time applications

Tasks

- A task, also called a thread, is a simple program that thinks it has the CPU all to itself
- The design process for a real-time application involves splitting the work to be done into tasks responsible for a portion of the problem
- Each task is assigned a priority, its own set of CPU registers, and its own stack area
Tasks (continued)

- Each task typically is an infinite loop that can be in any one of five states:
  - Dormant
    - Corresponds to a task that resides in memory but has not been made available to the multitasking kernel
  - Ready
    - A task is ready when it can execute but its priority is less than the currently running task
  - Running
    - A task is running when it has control of the CPU
  - Waiting (for an event)
    - A task is waiting when it requires the occurrence of an event (for example, waiting for an I/O operation to complete, a shared resource to be available, etc.
  - ISR (interrupted)
    - A task is in the ISR state when an interrupt has occurred and the CPU is in the process of servicing the interrupt
RTOS Scheduler Basics

- A real-time OS has an associated algorithm for scheduling tasks
- Scheduler flexibility enables a wider, computer-system orchestration of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications
- Key factors in a real-time OS are
  - minimal interrupt latency and
  - minimal thread switching latency
- A real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time

Context Switches (or Task Switches)

- When a multitasking kernel decides to run a different task, it saves the current task's context (CPU registers) in the current task's context storage area - its stack
- After this operation is performed, the new task's context is restored from its storage area and then resumes execution of the new task's code
- This process is called a context switch or a task switch
- Context switching adds overhead to the application
  - The more registers a CPU has, the higher the overhead
  - The time required to perform a context switch is determined by how many registers have to be saved and restored by the CPU
Kernels

- The kernel is the part of a multitasking system responsible for management of tasks (i.e., for managing the CPU's time) and communication between tasks
- The fundamental service provided by the kernel is context switching
- The use of a real-time kernel generally simplifies the design of systems by allowing the application to be divided into multiple tasks that the kernel manages
- A kernel adds overhead to your system because the services provided by the kernel require execution time
  - The amount of overhead depends on how often you invoke these services
  - In a well-designed application, a kernel uses less than 5% of CPU time

Schedulers

- The scheduler is the part of the kernel responsible for determining which task runs next
- Most real-time kernels are priority based
  - Each task is assigned a priority based on its importance
  - The priority for each task is application specific
- In a priority-based kernel, control of the CPU is always given to the highest priority task ready to run
- When the highest priority task gets the CPU, however, is determined by the type of kernel used
- Two types of priority-based kernels exist:
  - Non-preemptive
  - Preemptive
Common RTOS Scheduling Algorithms

• Non-Preemptive (Cooperative) scheduling
• Preemptive scheduling
  – Rate-monotonic scheduling
  – Round-robin scheduling
  – Fixed priority preemptive scheduling, an implementation of preemptive time slicing
  – Fixed-Priority Scheduling with Deferred Preemption
  – Fixed-Priority Non-preemptive Scheduling
  – Critical section preemptive scheduling
  – Static time scheduling
• Earliest Deadline First

Non-Preemptive Kernels

• Non-preemptive kernels require that each task does something to explicitly give up control of the CPU
  – To maintain the illusion of concurrency, this process must be done frequently
• Non-preemptive scheduling is also called *cooperative multitasking*; tasks cooperate with each other to share the CPU
• Asynchronous events are still handled by ISRs
  – An ISR can make a higher priority task ready to run, but the ISR always returns to the interrupted task
  – The new higher priority task gains control of the CPU only when the current task gives up the CPU
Non-Preemptive Kernels (continued)

- One of the advantages of a non-preemptive kernel is that interrupt latency is typically low.
- At the task level, non-preemptive kernels can also use non-reentrant functions:
  - Non-reentrant functions can be used by each task without concern of corruption by another task.
  - This is because each task can run to completion before it relinquishes the CPU.
  - However, non-reentrant functions should not be allowed to give up control of the CPU.
- Task-level response using a non-preemptive kernel can be much lower than with foreground/background systems because task-level response is now given by the time of the longest task.

Non-Preemptive Kernel Execution Profile
Non-Preemptive Kernel Execution Profile

1) A task is executing but is interrupted
2) If interrupts are enabled, the CPU vectors (jumps) to the ISR
3) The ISR handles the event and makes a higher priority task ready to run
4) Upon completion of the ISR the CPU returns to the interrupted task
5) The task code resumes at the instruction following the interrupted instruction
6) When the task code completes, it calls a service that the kernel provides to relinquish the CPU to another task
7) The kernel sees that a higher priority task has been made ready to run (it doesn't necessarily know that it was from an ISR nor does it care), and thus the kernel performs a context switch so that it can run (i.e., execute) the higher priority task to handle the event that the ISR signaled

Non-Preemptive Kernels (continued)

- The most important drawback of a non-preemptive kernel is responsiveness
  - A higher priority task that has been made ready to run might have to wait a long time to run because the current task must give up the CPU when it is ready to do so
- As with background execution in foreground/background systems, task-level response time in a non-preemptive kernel is nondeterministic
  - You never really know when the highest priority task will get control of the CPU
  - It is up to your application to relinquish control of the CPU
Preemptive Kernels

- A preemptive kernel is used when system responsiveness is important
  - uC/OS-II and most commercial real-time kernels are preemptive
- The highest priority task ready to run is always given control of the CPU
  - When a task makes a higher priority task ready to run, the current task is preempted (suspended), and the higher priority task is immediately given control of the CPU
  - If an ISR makes a higher priority task ready, when the ISR completes, the interrupted task is suspended, and the new higher priority task is resumed

Preemptive Kernel Execution Profile
Preemptive Kernel Execution Profile (continued)

1) A task is executing but is interrupted
2) If interrupts are enabled, the CPU vectors (jumps) to the ISR
3) The ISR handles the event and makes a higher priority task ready to run. Upon completion of the ISR, a service provided by the kernel is invoked (i.e., a function that the kernel provides is called)
4) This function knows that a more important task has been made ready to run, and thus, instead of returning to the interrupted task, the kernel performs a context switch and executes the code of the more important task. When the more important task is done, another function that the kernel provides is called to put the task to sleep waiting for another event (i.e. the ISR) to occur
5) The kernel then sees that a lower priority task needs to execute, and another context switch is done to resume execution of the interrupted task

Preemptive Kernels (continued)

- With a preemptive kernel, execution of the highest priority task is deterministic
  - You can determine when it will get control of the CPU
- Task-level response time is thus minimized by using a preemptive kernel
- Application code using a preemptive kernel should not use non-reentrant functions unless exclusive access to these functions is ensured through the use of mutual exclusion semaphores, because both a low and a high priority task can use a common function
  - Corruption of data can occur if the higher priority task preempts a lower priority task that is using the function
Reentrant Functions

- A reentrant function can be used by more than one task without fear of data corruption.
- A reentrant function can be interrupted at any time and resumed at a later time without loss of data.
- Reentrant functions either use:
  - local variables (i.e., CPU registers or variables on the stack) or
  - protect data when global variables are used.
- Example reentrant function:
  ```c
  void strcpy(char *dest, char *src) {
    while (*dest++ = *src++) {
    }
  *dest=NULL;
  }
  ```
- Because copies of the arguments to strcpy() are placed on the task’s stack, strcpy() can be invoked by multiple tasks without concern that the tasks will corrupt each other’s pointers.

Non-reentrant Function Example

- Assume a simple function that swaps the contents of it two arguments:
  ```c
  int Temp;
  void swap(int *x, int *y) {
    Temp=*x;
    *x=*y;
    *y=Temp;
  }
  ```
- Assume:
  - Preemptive kernel
  - Interrupts are enabled
  - Temp declared as a global integer
Non-reentrant Function Example (continued)

- This example is simple, so it is obvious how to make the code reentrant
- You can make `swap()` reentrant with one of the following techniques:
  - Declare Temp local to `swap()`
  - Disable interrupts before the operation and enable them afterwards
  - Use a semaphore