real-time operating system (RTOS) ● A real-time operating system (RTOS) is a program that schedules execution in a timely manner, manages system resources, and provides a consistent foundation for developing application code. ● Board Support Package (BSP) is a layer of software that contains hardware-specific drivers and routines (interrupt controller initialization, processor initialization, clock initialization, and initialization of RAM, ...) Defining an RTOS ● For example, in some applications, an RTOS comprises only a kernel, which is the core supervisory software that provides minimal logic, scheduling, and resource-management algorithms. ● Every RTOS has a kernel. ● On the other hand, an RTOS can be a combination of various modules, including the kernel, a file system, networking protocol stacks, and other components required for a particular application. Defining an RTOS Most RTOS kernels contain the following components: ● Scheduler is contained within each kernel and follows a set of algorithms that determines which task executes when. Some common examples of scheduling algorithms include round-robin and preemptive scheduling. ● Objects are special kernel constructs that help developers create applications for real-time embedded systems. Common kernel objects include tasks, semaphores, and message queues. ● Services are operations that the kernel performs on an object or, generally operations such as timing, interrupt handling, and resource management. Defining an RTOS Defining an RTOS Common components in an RTOS kernel that including objects, the scheduler, and some services ● A schedulable entity is a kernel object that can compete for execution time on a system, based on a predefined scheduling algorithm. ● Tasks and processes are all examples of schedulable entities found in most kernels. The Scheduler - Schedulable Entities ● Multitasking is the ability of the operating system to handle multiple activities within set deadlines. ● Many threads of execution appear to be running concurrently; however, the kernel is actually interleaving executions sequentially, based on a preset scheduling algorithm. ● The scheduler must ensure that the appropriate task runs at the right time. The Scheduler - Multitasking ● Each task has its own context, which is the state of the CPU registers required each time it is scheduled to run. ● A context switch occurs when the scheduler switches from one task to another. The Scheduler - The Context Switch ● Every time a new task is created, the kernel also creates and maintains an associated task control block (TCB). ● TCBs are system data structures that the kernel uses to maintain task-specific information. ● TCBs contain everything a kernel needs to know about a particular task. The Scheduler - The Context Switch When the kernel scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps: 1. The kernel saves task 1 context information in its TCB. 2. It loads task 2 s context information from its TCB, which becomes the current thread of execution. The Scheduler - The Context Switch when the kernel s scheduler determines that it needs to stop running task 1 and start running task 2, it takes the following steps: 3. The context of task 1 is frozen while task 2 executes, but if the scheduler needs to run task 1 again, task 1 continues from where it left off just before the context switch. The Scheduler - The Context Switch ● The time it takes for the scheduler to switch from one task to another is the context switch time. ● It is relatively insignificant compared to most operations that a task performs. ● If an application design includes frequent context switching, however, the application can incur unnecessary performance overhead. Therefore, design applications in a way that does not involve excess context switching. The Scheduler -

The Context Switch Every time an application makes a system call, the scheduler has an opportunity to determine if it needs to switch contexts. When the scheduler determines a context switch is necessary, it relies on an associated module, called the dispatcher, to make that switch happen. The Scheduler – The Context Switch ● The dispatcher is the part of the scheduler that performs context switching and changes the flow of execution. ● At any time an RTOS is running, the flow of execution, also known as flow of control, is passing through one of three areas: through an application task, through an ISR, or through the kernel. The Scheduler – The Dispatcher ● The scheduler determines which task runs by following a scheduling algorithm (also known as scheduling policy). Most kernels today support two common scheduling algorithms: preemptive priority-based scheduling, and round-robin scheduling. ● The RTOS manufacturer typically predefines these algorithms; however, in some cases, developers can create and define their own scheduling algorithms. Each algorithm is described next. The Scheduler – Scheduling Algorithms Preemptive Priority-Based Scheduling ● Of the two scheduling algorithms introduced here, most real-time kernels use preemptive priority-based scheduling by default. ● The task that gets to run at any point is the task with the highest priority among all other tasks ready to run in the system. The Scheduler – Scheduling Algorithms Preemptive priority-based scheduling Preemptive Priority-Based Scheduling ● Real-time kernels generally support 256 priority levels, in which 0 is the highest and 255 the lowest. ● With a preemptive priority-based scheduler, each task has a priority, and the highest-priority task runs first. If a task with a priority higher than the current task becomes ready to run, the kernel immediately saves the current task s context in its TCB and switches to the higher priority task. The Scheduler – Scheduling Algorithms Preemptive Priority-Based Scheduling ● Although tasks are assigned a priority when they are created, a task s priority can be changed dynamically using kernel-provided calls. ● The ability to change task priorities dynamically allows an embedded application the flexibility to adjust to external events as they occur, creating a true real-time, responsive system. ● Note, however, that misuse of this capability can lead to priority inversions, deadlock, and eventual system failure. The Scheduler – Scheduling Algorithms Round-Robin Scheduling ● Round-robin scheduling provides each task an equal share of the CPU execution time. ● Pure round-robin scheduling cannot satisfy real-time system requirements because in real-time systems, tasks perform work of ● varying degrees of importance. Instead, preemptive, priority-based scheduling can be augmented with round-robin scheduling which uses time slicing to achieve equal allocation of the CPU for tasks of the same priority. The Scheduler – Scheduling Algorithms The Scheduler – Scheduling Algorithms Round-robin and preemptive scheduling Kernel objects are special constructs that are the building blocks for application development for real-time embedded systems. The most common RTOS kernel objects are: ● Tasks are concurrent and independent threads of execution that can compete for CPU execution time. ● Semaphores are token-like objects that can be incremented or decremented by tasks for synchronization or mutual exclusion. ● Message Queues are buffer-like data structures that can be used for synchronization, mutual exclusion, and data exchange by passing messages between tasks. Objects Along with objects, most kernels provide services that help developers create applications for real-time embedded systems. These services comprise sets of API calls that can be used to perform operations on kernel objects or can be used in general to facilitate timer management, interrupt handling,

.device I/O, and memory management