Automotive & Embedded Info

Never Forget Basics Whether its Life or Anything Else ... Basics are Cores. While seeing a Tree how can we forget Seed...

RTOS

What is Operating System?

We have little misunderstanding about Operating System. When we say operating system then in mind thought come Windows, Linux, Android etc. But actually operating system is software/program code that has a responsibility to provides an environment for execution to other software's/program codes.

By Definition:

An operating system is a program that provides an environment for executing other programs, often providing facilities for I/O, a file system, networking, virtual memory, and multitasking: a way to run multiple program concurrently on a single processor.

What is RTOS?

RTOS is a multitasking operating system intended for real time applications. In the OS, there is a module called the scheduler, which

schedules different tasks and determines when a process will execute on the processor. This way, the multi-tasking achieved. The scheduler in RTOS is designed to provide a predictable execution pattern. In an embedded system, a certain event must be entertained in strictly defined time.

To meet real time requirements, the behaviour of scheduler must be predictable. This type of OS which have a scheduler with predictable execution pattern is called real time OS (RTOS). The features of an RTOS are:

1. Context switching latency should be short.
2. Interrupt latency should be short.
3. Interrupt dispatch latency should be short.
4. Reliable and time bound inter process mechanism.
5. Should support kernel pre-emption.

In general an operating system is responsible for managing hardware resources of computer and hosting application that run on computer. An RTOS perform these tasks but it is also specially designed to run application with very precise timing and high degree of reliability. This can be especially important in measurement and automation systems where downtime is costly or a program could cause safety hazards.

To be considered A"REAL TIME", an operating system must have a known maximum time for each of critical operation that it performs (or at least be able to guaranteed maximum most of time).

Some of these operations include OS calls and interrupt handling. Operating system that can absolutely guarantee a maximum time for these operation are commonly referred to as "hard real time", while operating systems that can only guarantee a maximum most of time are referred to as "SOFT REAL TIME".

EXAMPLE:

Imagine that we are designing an airbag system for a new model of car. In this case a small error in timing (causing the airbag to deploy too early or too late) could be catastrophic and cause injury. Therefore a hard real time system is needed, you need assurance as a system designer no single operation will exceed certain timing constraints. On other hand if we were design a mobile phone that receive streaming video, it may be ok to lose a small amount of data even though on average it is important to keep up with video stream. For this application, a soft real time operating system may suffice.

An RTOS can guarantee that a program will run with very consistent timing.

Real-time operating systems do this by providing programmers with a high degree of control over how tasks are prioritized, and typically also allow checking to make sure that important deadlines are met.

RTOS Responsibility:

RTOS have 4 main responsibilities:

1. Task management and scheduling
2. (Deferred) interrupt servicing
3. Inter-process communication and synchronization
4. Memory management

RTOS Types:

Hard real-time – Systems where it is absolutely imperative that

Soft real-time – Systems where deadlines are important but which will still function correctly if deadlines are occasionally missed. E.g. Data acquisition system.

Real real-time – Systems which are hard real-time and which the response times are very short. E.g. Missile guidance system.

Firm real-time – Systems which are soft real-time but in which there is no benefit from late delivery of service.

A single system may have all hard, soft and real real-time subsystems.

Below are the some RTOS terms explained based on the Automotive operating system OSEK/VDX:

AUTOSAR OS specification uses the industry standard OSEK OS (ISO 17356-3) as the basis for the AUTOSAR OS.

OS Objects:

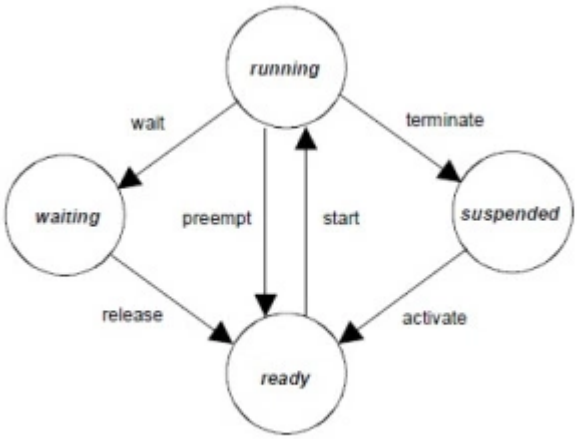
Task, Events, Counter, Scheduler, Resource, Alarm and Hook Functions are OS Objects.

TASK:

A task provides the framework for the execution of functions. Complex control software can conveniently be subdivided in parts executed according to their real-time requirements. Two different task concepts are provided by the OSEK operating system:

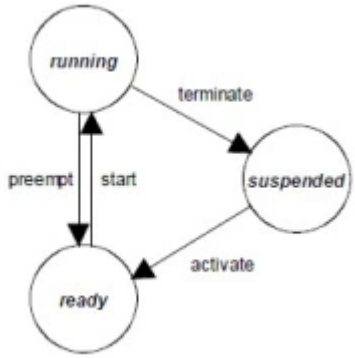
Extended task:

Extended tasks have four task states: 1. Running 2. Ready 3. Waiting 4. Suspended.



Basic Task:

Basic tasks have four task states: 1. Running 2. Ready 3. Suspended. Basic tasks do not have a *waiting* state.



SCHEDULING AND TYPES:

Represent in form of schedule table. With schedule table task and event execute. 4 types of scheduling policy:

1. Pre-emptive

2. Non Pre-emptive
3. Group of Task
4. Mixed Pre-emptive

RTOS uses pre-emptive scheduling. In pre-emptive scheduling, the higher priority task can interrupt a running process and the interrupted process will be resumed later.

EVENTS:

Assigned to Extended Task. Meaning of events is defined by the application, e.g. signalling of an expiring timer, the availability of a resource, the reception of a message, etc.

COUNTER:

Represent count value. Belong to Os-App. Increment by core on Os-App resides. Use to drive scheduler and alarm if they are in same core.

ALARM:

Use to activate task, Set event, increment counter and call-back alarm.

HOOK FUNCTIONS:

Hook routine are only used during debugging and are not used in a final product.

AUTO START OBJECTS:

Before scheduling starts, the Multi-Core OS shall activate all configured auto-start objects on the respective core.

OS Application:

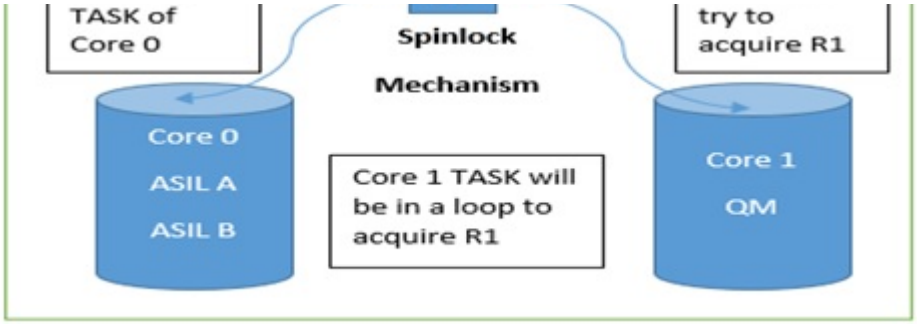
OS must be capable of supporting a collection of Operating System objects (Tasks, ISRs, Alarms, Schedule tables, Counters) that form a cohesive functional unit. This collection of objects is termed an OS-Application.an OS-Application may belong to different AUTOSAR Software Components. Type: Trusted and Non Trusted.

Trusted OS-Applications are allowed to run with monitoring or protection features disabled at runtime. Non-Trusted OS-Applications are not allowed to run with monitoring or protection features disabled at runtime. Trusted OS-Applications can be permitted access to IO space, allowed direct access to the hardware etc.

Spin lock:

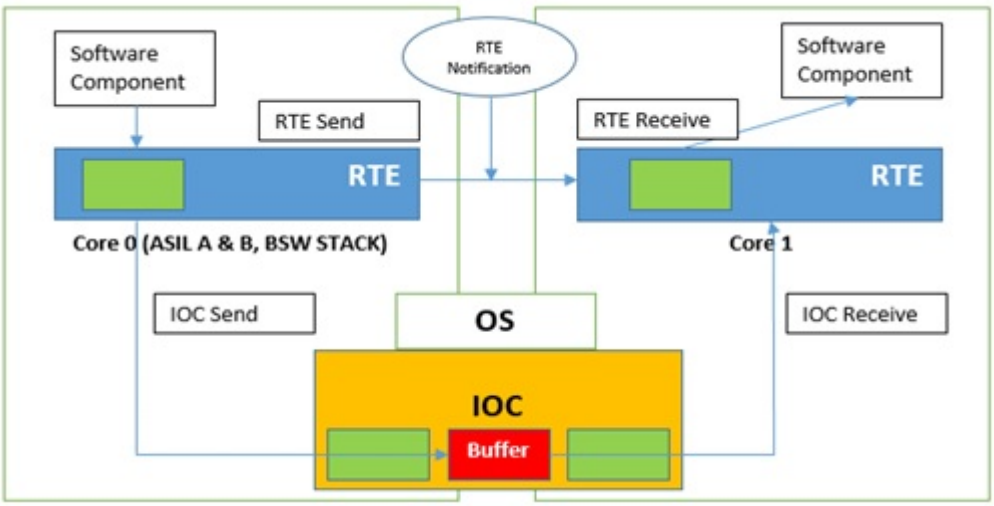
If a resource is locked, a thread that wants to access that resource may repetitively check whether the resource is available. During that time, the thread may loop and check the resource without doing any useful work. Suck a lock is termed as spin lock.

Used in multicore concept. Busy waiting mechanism that polls a (lock) variable until it becomes available. Once a lock variable is occupied by a TASK/ISR, other core shall be unable to occupy the lock variable. Spinlock will not reschedule these other tasks while they poll the lock variable.



IOC (Inter OS Communication):

The "IOC" is responsible for the communication between OS-Applications and in particular for the communication crossing core or memory protection boundaries. Its internal functionality is closely connected to the Operating System.



Scalability Classes:

There are four scalability classes in AUTOSAR OS.

SC1 = OSEK OS+ Counter IF + SWFRT IF + Schedule Table + Stack Monitoring
SC2 = SC1 + Timing Protection + GTS

Trusted Function
SC4 = SC2+ SC3

Protection Mechanism:

1. Timing Protection,
2. Memory Protection,
3. Service Protection,
4. Hardware Protection used by OS

TIMING PROTECTION:

Safe behaviour requires that the systems actions and reactions are performed within the right time. The right time can be described in terms of a set of timing constraints that have to be satisfied. **Fault Models**: Blocking of execution, Deadlocks, Lovelocks, Incorrect allocation of execution time and Incorrect synchronization between software elements.

MEMORY PROTECTION:

Memory partitioning means that OS-Applications reside in different memory areas (partitions) that are protected from each other. Moreover, memory partitioning enables to protect read-only memory segments (e.g. code execution), as well as to protect memory-mapped hardware. **Fault Models**: Corruption of content, Read or write access to memory allocated to another software element. It is assumed that memory partitioning will be implemented on a microcontroller which has an MPU or similar hardware features.

SERVICE PROTECTION:

As OS-Applications can interact with the Operating System module through services, it is essential that the service calls will not corrupt the Operating System module itself. Service Protection guards against such corruption at runtime. **Fault Models**: Invalid Object Parameter or Out of Range Value, Service Calls Made from Wrong Context, Services with Undefined Behaviour, Service Restrictions for Non-Trusted OS-Applications, Service Calls on Objects in Different OS-Application.

HARDWARE PROTECTION USED BY OS:

Executing the Operating System module in privileged mode and Tasks/ISRs in non-privileged mode protects the registers fundamental to Operating System module operation from inadvertent corruption by the objects executing in non-privileged mode.

Stack Monitoring Facilities:

Stack Monitoring is necessary for Processors with no MPU H/W. Stack monitoring will identify where a task or ISR has exceeded a specified stack usage at context switch time.
Note that for systems using a MPU and scalability class 3 or 4 a stack overflow may cause a memory exception before the stack monitoring is able to detect the fault.

Priority Inversion:

If two tasks share a resource, the one with higher priority will run first. However, if the lower-priority task is using the shared resource when the higher-priority task becomes ready, then the higher-priority task must

wait for the lower-priority task to finish. In this scenario, even though the task has higher priority it needs to wait for the completion of the lower-priority task with the shared resource. This is called priority inversion. The following scenario for priority inversion:

Task T4 which has a low priority, occupies the semaphore S1. T1 preempts T4 and requests the same semaphore. As the semaphore S1 is already occupied, T1 enters the waiting state. Now the low-priority T4 is interrupted and preempted by tasks with a priority between those of T1 and T4. T1 can only be executed after all lower-priority tasks have been terminated, and the semaphore S1 has been released again. Although T2 and T3 do not use semaphore S1, they delay T1 with their runtime.

Priority_Inversion.jpg

Priority Inheritance:

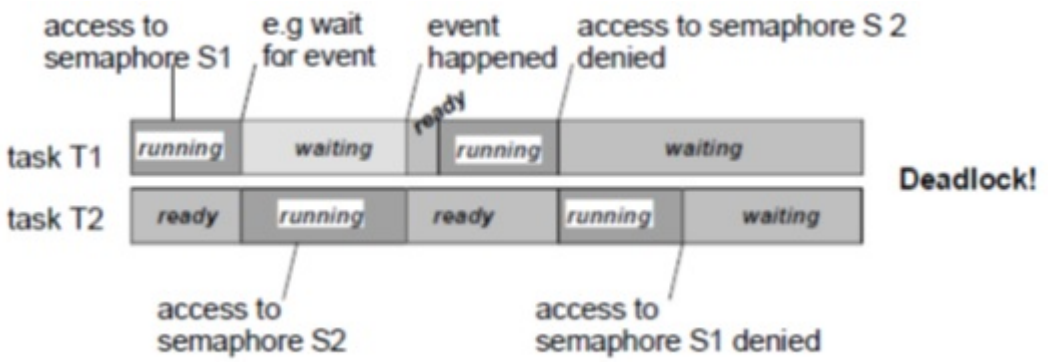
Priority inheritance is a solution to the priority inversion problem. The process waiting for any resource which has a resource lock will have the maximum priority. This is priority inheritance. When one or more high priority jobs are blocked by a job, the original priority assignment is ignored and execution of critical section will be assigned to the job with the highest priority in this elevated scenario. The job returns to the original priority level soon after executing the critical section.

Dead Lock:

Impossibility of task execution due to infinite waiting for mutually locked resources. The following scenario results in a deadlock.

Task T1 occupies the semaphore S1 and subsequently cannot continue

task T2 is transferred into the running state. It occupies the semaphore S2. If T1 gets ready again and tries to occupy semaphore S2, it enters the waiting state again. If now T2 tries to occupy semaphore S1, this results in a deadlock.



Priority ceiling Protocol:

1. At the system generation, to each resource its own ceiling priority is statically assigned. The ceiling priority shall be set at least to the highest priority of all tasks that access a resource or any of the resources linked to this resource. The ceiling priority shall be lower than the lowest priority of all tasks that do not access the resource, and which have priorities higher than the highest priority of all tasks that access the resource.
2. If a task requires a resource, and its current priority is lower than the ceiling priority of the resource, the priority of the task is raised to the ceiling priority of the resource.
3. If the task releases the resource, the priority of this task is reset to the priority which was dynamically assigned before requiring that resource.

Note: Priority ceiling results in a possible time delay for tasks with priorities equal or below the resource priority. This delay is limited by the maximum time the resource is occupied by any lower priority task.

Semaphore:

Semaphore is actually a variable or abstract data type which controls access to a common resource by multiple processes. Semaphores are of two types: –

Binary semaphore – it can have only two values (0 and 1). The semaphore value is set to 1 by the process in charge, when the resource is available.

Binary semaphores have no ownership attribute and can be released by any thread or interrupt handler regardless of who performed the last take operation. Because of this binary semaphores are often used to synchronize threads with external events implemented as ISRs, for example waiting for a packet from a network or waiting that a button is pressed.

Counting semaphore – it can have value greater than one. It is used to control access to a pool of resources.

Mutex:

Note that, unlike semaphores, mutexes do have owners. A mutex can be unlocked only by the thread that owns it, this precludes the use of mutexes from interrupt handles but enables the implementation of the Priority Inheritance protocol, most RTOSs implement this protocol in order to address the Priority Inversion problem.

Mutexes have one single use, Mutual Exclusion, and are optimized for that. Semaphores can also handle mutual exclusion scenarios but are best used as a communication mechanism between threads or between ISRs and threads.

DIFFERENCE BETWEEN SEMAPHORE AND MUTEX?

1. Mutual exclusion and synchronization can be used by binary semaphore while MUTEX is used only for mutual exclusion.
2. A MUTEX can be released by the same thread which acquired it. Semaphore values can be changed by other thread also.
3. From an ISR, a MUTEX cannot be used.
4. The advantage of semaphores is that, they can be used to synchronize two unrelated processes trying to access the same resource.
5. Semaphores can act as MUTEX, but the opposite is not possible.

IPC mechanism(Inter-process mechanism):

1. PIPES
2. FIFO
3. SEMAPHORE
4. SHARED MEMORY
5. MESSAGE QUEUE
6. SOCKET

Automotive & Embedded Info / Powered by WordPress.com.