Abstract

The idea is dedicated to design a simple real time embedded kernel build from scratch. The kernel is cross platform and designed with safety critical code standards taken into account. The operating system allows multi process communication, preemption and incorporates basic priority scheduling.

**Introduction**

1. For security reasons and better debugging a simple

monolithic kernel, which is compiled together with

all user programs, was chosen. This architecture

provides safer operation than OS which runs binary

programs (not compiled together with OS),

because in whole system, there is only one pointer

to user program which is static. This means that

address contained in this pointer is determined dur

ing compilation, and there is no threat of wild

pointer or other hazards.

1. Monolithic kernel- A Monolithic kernel is an OS architecture where the entire operating system (which includes the device drivers, file system, and the application IPC) is working in kernel space.

For eg-linux.

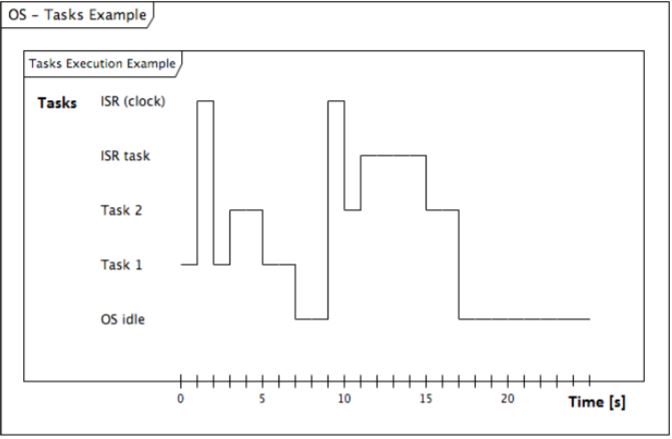
**Real time operating system design**

1.All non time critical tasks will run in cooperative mode and real time tasks will be handled as interrupts. In cooperative multitasking the possible points of tasks preemption in the codeare set by the developer. Thus, the order of execution of given input set of tasks is predictable, allowing for a simpler execution simulation than in the case of preemptive multitasking.

**Example**

Cooperative multitasking will provide better software design, verification and validation than standard preemptive RTOS, because shared resources in the code will be scarce. In the Figure an example of task run and preemption in cooperative mode can be seen. In this

example three tasks are defined. Task1 and Task2 are user defined not time critical programs and ISR task is a hard real time task, which is triggered by ISR clock. OS Idle task means, that there are no tasks in query and the onboard computer together with OS can go to low power mode to save energy.

****

The respective tasks in above figure are as follows:

0: Task 1 is running.

1: Clock ISR fires. Task 2 is made “ready” via OS API. ISR finished.

2: Task 1 continues to execute after ISR.

3: Task 1 voluntarily “yields” execution to the Task 2 (which was made ready by ISR).

5: Task 2 finishes execution and the stack unwinds down to interrupted Task 1stack frame.

7: Task 1 finishes. System goes to idle state.

9: Clock ISR fires. Task 2 is made “ready”. ISR finishes.

10: Since no other task is executing, Task 2 starts to execute immediately.

11: Task 2 is preempted by the “ISR task”. This is the short time-critical task. The ISRs are executing with their own stack; thus, Task 2 stack is not corrupted.

15: “ISR task” finishes and Task 2 continues to execute. Even if Task 2 “yields” execution during its lifetime, it is scheduled again - it is the highest priority task ready.

17: Task 2 finishes. System goes to idle state.

The first part is Application User Interface (API), which enables user to create, make ready, or yield a task, or allows one task to send event or data to another task. The second part deals with the task management and incorporates a task scheduler, which schedules a task flow in

OS. Finally, the third part manages system ISR(s) and real time tasks.