# Embedded Software Architecture

Embedded software, must interact with the environment through sensors and actuators, and often has hard, real-time constraints. The organization of the software, or its architecture, must reflect these realities. Four software architectures are:

## Round Robin – Infinite Loop

The simplest possible software architecture is called "round robin." Round robin architecture has no interrupts; the software organization consists of one main loop wherein the processor simply polls each attached device in turn, and provides service if any is required. After all devices have been serviced, start over from the top. One can think of many examples where round robin is a perfectly capable architecture: A vending machine, ATM, or household appliance such as a microwave oven (check for a button push, decrement timer, update display and start over). Basically, anything where the processor has plenty of time to get around the loop, and the user won't notice the delay. The main advantage to round robin is that it's very simple, and often it's good enough. On the other hand, If a device has to be serviced in less time than it takes the processor to get around the loop, then it won't work. The worst case response time for round robin is the sum of the execution times for all of the task code.

## Round Robin with Interrupts

In this, urgent tasks get handled in an interrupt service routine, possibly with a flag set for follow-up processing in the main loop. If nothing urgent happens (emergency stop button pushed, or intruder detected), then the processor continues to operate round robin, managing more mundane tasks in order around the loop. The obvious advantage to round robin with interrupts is that the response time to high-priority tasks is improved, since the ISR always has priority over the main loop (the main loop will always stop whatever it's doing to service the interrupt), and yet it remains fairly simple. The worst case response time for a low priority task is the sum of the execution times for all of the code in the main loop plus all of the interrupt service routines.

## Function Queue Scheduling

Function queue scheduling provides a method of assigning priorities to interrupts. In this architecture, interrupt service routines accomplish urgent processing from interrupting devices, but then put a pointer to a handler function on a queue for follow-up processing. The main loop simply checks the function queue, and if it's not empty, calls the first function on the queue. Priorities are assigned by the order of the function in the queue – there's

no reason that functions have to be placed in the queue in the order in which the interrupt occurred. They may just as easily be placed in the queue in priority order: high priority functions at the top of the queue, and low priority functions at the bottom. The worst case timing for the highest priority function is the execution time of the longest function in the queue (think of the case of the processor just starting to execute the longest function right before an interrupt places a high priority task at the front of the queue). The worst case timing for the lowest priority task is infinite: it may never get executed if higher priority code is always being inserted at the front of the queue. The advantage to function queue scheduling is that priorities can be assigned to tasks; the disadvantages are that it's more complicated than the other architectures discussed previously.

Real-time Operating System (RTOS)

Embedded systems may be so simple that an operating system is not required. When an OS is used, however, it must guarantee certain capabilities within specified time constraints. Such operating systems are referred to as "real-time operating systems" or RTOS. A real time operating system is complicated, potentially expensive, and takes up precious memory in our almost always cost and memory constrained embedded system. Why use one? There are two main reasons: flexibility and response time. The elemental component of a real-time operating system is a task, and it's straightforward to add new tasks or delete obsolete ones because there is no main loop: The RTOS schedules when each task is to run based on its priority. The scheduling of tasks by the RTOS is referred to as multi-tasking. In a preemptive multi-tasking system, the RTOS can suspend a low priority task at anytime to execute a higher priority one, consequently, the worst case response time for a high priority task is almost zero (in a non-preemptive multi-tasking system, the low priority task finishes executing before the high priority task starts).