Embedded Software Architecture

EECS 461, Fall 2008*

J. A. Cook
J. S. Freudenberg

1 Introduction

Embedded systems encompass aspects of control (or more broadly, signal processing), computing and communications. In each arena, the embedded system normally manages multiple tasks with hard real-time deadlines[1] and interacts with various sensors and actuators through on-chip or on-board peripheral devices, and often with other processors over one or several networks. Communication may be wired or wireless, such as the remote keyless entry on your car, or a Bluetooth-enabled consumer device. The performance requirements of the embedded control system dictate its computing platform, I/O and software *architecture*. Often, the quality of embedded software is determined by how well the interfaces between communicating tasks, devices and other networked systems are handled. The following sections will discuss the management of *shared data* among cooperating tasks and software architectures for embedded control systems.

2 Shared Data

When data are shared between cooperating tasks that operate at different rates, care must be taken to maintain the integrity of the calculations that use the shared information. For example, consider the situation where an interrupt routine acquires from an A/D converter, data that are used in the main loop of the pseudocode:

```
int ADC_channel[3]

ISR_ReadData(void) {
    Read ADC_channel[0]
    Read ADC_channel[1]
    Read ADC_channel[2]
    }

int delta, offset

void main(void) {
    while(TRUE) {
       ...
       delta  = ADC_channel[0]-ADC_channel[1];
       offset = delta*ADC_channel[2]
       ...
```

*Revised October 29, 2008.

[1] A hard real-time system is one in which a missed deadline results in system failure; in other words, the response to an event must occur within a specified time for the system to work. In soft real-time systems, the response to an event may be specified as a range of acceptable times.

```
        }
    }
```

The interrupt routine can suspend the main loop and execute at any time. Consider an interrupt that occurs between the calculations of `delta` and `offset`: On the return from interrupt, the data `ADC_channel[0-2]` may result in an unintended value being assigned to the calculated variable `offset` if the values have changed from the previous data acquisition. More subtly, the calculation of `delta` may also be affected because, as we'll see, even a single line of code may be interrupted.

2.1 Atomic Code and Critical Sections

We will recall that *assembly language* is the "human readable" form of the binary machine language code that eventually gets executed by the embedded processor. Assembly language, unlike high-level languages such as `C` and `C++`, is very closely associated with the processor hardware. Typical assembly language instructions reference memory locations or special purpose registers. An assembly language instruction typically consists of three components:

Label: Memory address where the code is located (optional).

Op-Code: Mnemonic for the instruction to be executed.

Operands: Registers, addresses or data operated on by the instruction.

The following are examples of assembly instructions for the Freescale MPC 5553 microprocessor:

```
add     r7, r8, r9;   Add the contents of registers 8 and 9, place the result
                      in register 7.
and     r2, r5, r3;   Bitwise AND the contents of registers 5 and 3,
                      place the result in register 2.
lwz     r6, 0x4(r5);  Load the word located at the memory address formed by
                      the sum of 0x4 and the contents of register 5 into register 6.
lwzx    r9, r5, r8;   Load the word located at the memory location formed by
                      the sum of the contents of registers 5 and 8 into register 9.
stwx    r13, r8, r9;  Store the value in register 13 in the memory location formed
                      by the sum of the contents of registers 8 and 9.
```

The important point about assembly instructions with respect to shared data is that they are *atomic*, that is, an assembly instruction, because it is implementing fundamental machine operations (data moves between registers, and memory), cannot be interrupted. Now, consider the following assembler instructions with the equivalent `C` code `temp = temp - offset`:

```
lwz  r5, 0(r10);    Read temp stored at 0(r10) and put it in r5
li   r6, offset;    Put offset value into r6
sub  r4, r5, r6;    Subtract the offset and put the result into r4
stwz r4, 0(r10);    Store the result back in memory
```

Thus, our single line of `C` code gets compiled into multiple lines of assembler. Consequently, whereas a single line of atomic assembler cannot be interrupted, one line of `C` code *can* be. This means that our pseudocode fragment

```
void main(void) {
    while(TRUE) {
       ...
       delta  = ADC_channel[0]-ADC_channel[1];
       offset = delta*ADC_channel[2]
       ...
        }
    }
```

can be interrupted *anywhere*. In particular, it can be interrupted in the middle of the `delta` calculation, with the result that the variable may be determined with one new and one old data value; undoubtedly not what the programmer intended. We shall refer to a section of code that must be atomic to execute correctly as a *critical* section. It is incumbent upon the programmer to protect critical code sections to maintain data coherency. All microprocessors implement instructions to enable and disable interrupts, so the obvious approach is to simply not permit critical sections to be interrupted:

```
void main(void) {
    while(TRUE) {
       ...
       disable()
       delta  = ADC_channel[0]-ADC_channel[1];
       offset = delta*ADC_channel[2]
       enable()
       ...
        }
    }
```

It must be kept in mind that code in the interrupt service routine has high priority for a reason – something needs to be done immediately. Consequently, it's important to disable interrupts sparingly, and only when absolutely necessary (and, naturally, to remember to enable interrupts again after the section of critical code). Other methods of maintaining data coherency will be discussed in the section on real-time operating systems.

3 Software Architectures for Embedded Control Systems

Software architecture, according to ANSI/IEEE Standard 1471-2000, is defined as the "fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." Embedded software, as we've said, 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. Usually, the critical aspect of an embedded control system is its speed of response which is a function of (among other things) the processor speed and the number and complexity of the tasks to be accomplished, as well as the software architecture. Clearly, embedded systems with not much to do, and plenty of time in which to do it, can employ a simple software organization (a vending machine, for example, or the power seat in your car). Systems that must respond rapidly to many different events with hard real-time deadlines generally require a more complex software architecture (the avionics systems in an aircraft, engine and transmission control, traction control and antilock brakes in your car). Most often, the various tasks managed by an embedded system have different *priorities*: Some things have to be done immediately (fire the spark plug *precisely* 20° before the piston reaches top-dead-center in the cylinder), while other tasks may have less severe time constraints (read and store the ambient temperature for use in a calculation to be done later). Reference [1] describes the four software architectures that will be discussed in the following sections:

- Round robin
- Round robin with interrupts
- Function queue scheduling
- Real time operating systems (RTOS)

3.1 Round Robin

The simplest possible software architecture is called "round robin."[2] Round robin architecture has no interrupts; the software organization consists of one main loop wherein the processor simply polls each

[2]Who is "Round Robin" anyway? According to the etymology cited in [2], the expression comes from a "petition signed in circular order so that no signer can be said to head the list." Robin being an alteration of the *ribbon* affixed to official

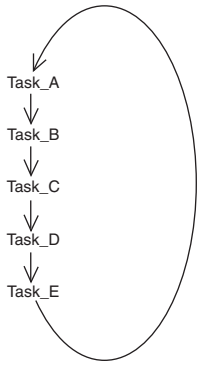

Figure 1: Round Robin Software Architecture

attached device in turn, and provides service if any is required. After all devices have been serviced, start over from the top. Graphically, round robin looks like Figure 1. Round robin pseudocode looks something like this:

```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all devices have been serviced, then start over again
        }
    }
```

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 (usually micro-seconds) between a request for service and the processor response (the time between pushing a button on your microwave and the update of the display, for example). The main advantage to round robin is that it's very simple, and often it's good enough. On the other hand, there are several obvious disadvantages. 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. In fact, the worst case response time for round robin is the sum of the execution times for all of the task code. It's also fragile: suppose you added one more device, or some additional processing to a loop that was almost at its chronometric limit – then you could be in trouble.[3] Some additional performance can be coaxed from the round robin architecture, however. If one or more tasks have more stringent deadlines than the others (they have higher *priority*), they may simply be checked more often:

```
void main(void) {
    while(TRUE) {
        if (device_A requires service)
            service device_A
        if (device_B requires service)
            service device_B
```

documents. The practice said to reflect the propensity of seventeenth century British sea captains to hang as mutineers the initial signers of a grievance.

[3]This is something that happens regularly. Generally referred to as "requirements creep," it occurs whenever the customer (or the marketing department or management) decides to add "just one more feature" after the system specifications have been frozen.

```
        if (device_A requires service)
            service device_A
        if (device_C requires service)
            service device_C
        if (device_A requires service)
            service device_A
        ... and so on, making sure high-priority device_A is always serviced on time
        }
    }
```

3.2 Round Robin with Interrupts

Round robin is simple, but that's pretty much its only advantage. One step up on the performance scale is round robin with interrupts. Here, 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. Possible pseudocode:

```
BOOL flag_A = FALSE; /* Flag for device_A follow-up processing */

/* Interrupt Service Routine for high priority device_A */

ISR_A(void) {
    ... handle urgent requirements for device_A in the ISR,
    then set flag for follow-up processing in the main loop ...
        flag_A = TRUE;
    }

void main(void) {
    while(TRUE) {
        if (flag_A)
            flag_A = FALSE
            ... do follow-up processing with data from device_A
        if (device_B requires service)
            service device_B
        if (device_C requires service)
            service device_C
        ... and so on until all high and low priority devices have been serviced
        }
    }
```

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. With the introduction of interrupts, the problem of shared data may arise: As in the previous example, if the interrupted low priority function is in the middle of a calculation using data that are supplied or modified by the high priority interrupting function, care must be taken that on the return from interrupt the low priority function data are still valid (by disabling interrupts around critical code sections, for example).

3.3 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, and it may be subject to shared data problems.

3.4 Real-time Operating System (RTOS)

The University of Michigan Information Technology Central Services website [3] contains the advisory:

> If your Windows laptop crashes and displays a Blue Screen with an error message, called the Blue Screen of Death (BSOD), and then reboots, when trying to connect to UM Wireless Network, most likely there is a problem ...

A Windows-like BSOD is not something one generally wants to see in an embedded control system (think anti-lock brakes, Strategic Defense Initiative or aircraft flight control). 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 any time 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). In the simplest RTOS, a task can be in one of three states:

Running: The task code is being executed by the processor. Only one task may be running at any time.

Ready: All necessary data are available and the task is prepared to run when the processor is available. Many tasks may be ready at any time, and will run in priority order.

Blocked: A task may be blocked waiting for data or for an event to occur. A task, if it is not preempted, will block after running to completion. Many tasks may be blocked at one time.

The part of the RTOS called a scheduler keeps track of the state of each task, and decides which one should be running. The scheduler is a simple-minded device: It simply looks at all the tasks in the ready state and chooses the one with the highest priority. Tasks can block themselves if they run out of things to do, and they can unblock and become ready if an event occurs, but its the job of the scheduler to move tasks between the ready and running states based on their priorities as shown in Figure 2.

4 The Shared Data Problem Revisited

In an earlier section, the topic of shared data was introduced for systems in which data generated or modified by a high priority, interrupt driven routine were used in a low priority task operating at a slower rate. It was shown that care must be taken to protect critical sections of code in the low priority task so that data are not inadvertently modified by the interrupting routine. This was accomplished by disabling interrupts at the beginning of the critical code section, and enabling them again at the end. A real-time operating system

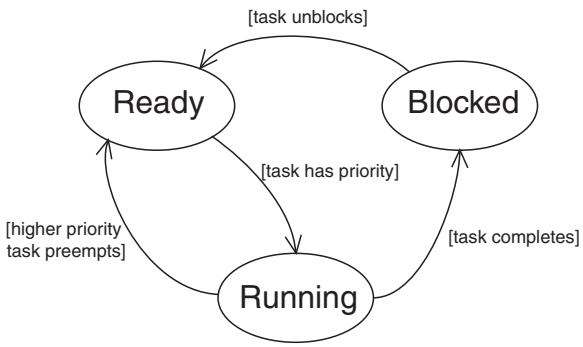

Figure 2: Simple Real-time Scheduler

must also have a mechanism, similar to disabling and enabling interrupts, for managing shared data among tasks with differing priorities. Normally, this mechanism is a *semaphore*. A semaphore may be thought of as an object that is passed among tasks that share data. A task possessing a semaphore locks the scheduler, preventing that task from being preempted until the semaphore is released. Consider again our shared data example. This time, data are read from the A-D converter and processed in a high priority, $10ms$ task. Every $50ms$, a sample of the data are used in a low priority task:

```
int ADC_channel[3]

void function_10ms(void)
    TakeSemaphore() {
    Read ADC_channel[0]
    Read ADC_channel[1]
    Read ADC_channel[2]
    ReleaseSemaphore()
    ...
    do high priority data processing
    ...
    }

int delta, offset
extern int ADC_channel[3]

void function_50ms(void) {
    while(TRUE) {
       ...
       TakeSemaphore()
       delta  = ADC_channel[0]-ADC_channel[1];
       offset = delta*ADC_channel[2]
       ReleaseSemaphore()
       ...
        }
    }
```

Since only one of the tasks can possess the semaphore at any time, coherency is assured by taking and releasing a semaphore around the shared data: If the $10ms$ task attempts to take the semaphore before the $50ms$ task has released it, the faster task will block until the semaphore is available. Problems, however, may arise if care is not taken in the use of semaphores. Specifically, priority inversion and deadlock. Priority inversion, as the name implies, refers to a situation in which a semaphore inadvertently causes a high priority task to block while lower priority tasks run to completion. Consider the case where a high priority task and a

low priority task share a semaphore, and there are tasks of intermediate priority between them (see Figure 3). Initially, the low priority task is running and takes a semaphore; all other tasks are blocked. Should the high priority task unblock and attempt to take the semaphore before the low priority task releases it, it will block again until the semaphore is available. If, in the meantime, intermediate priority tasks have unblocked, the simple-minded RTOS will run each one in priority order, completing all the intermediate priority tasks before finally running the low priority function to the point where it gives up its semaphore, permitting the high priority task to run again. The task priorities have been *inverted*: all of the lower priority tasks have run before the highest priority task gets to complete.

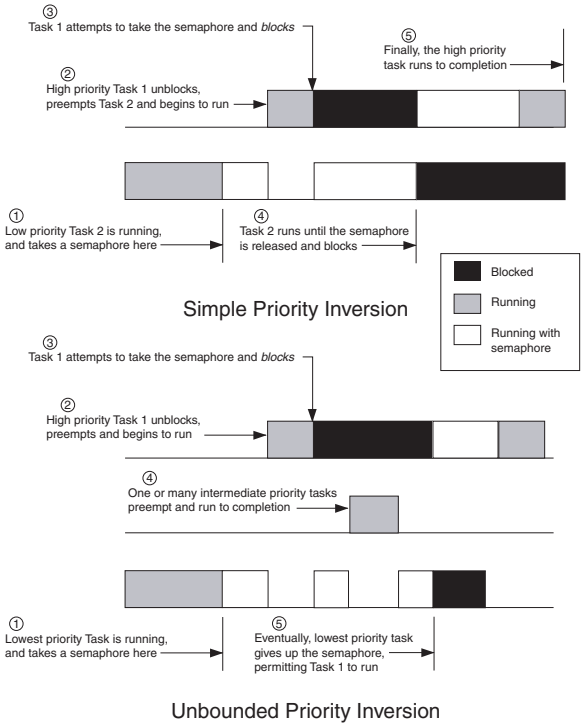

Figure 3: Examples of Priority Inversion

One may, of course, use as many semaphores as necessary to protect shared data in different tasks. If two tasks share more than one semaphore, there is a potential of creating a condition known as deadlock. Consider two tasks sharing nested semaphores as in Figure 4. In this case, cooperating tasks could each take a semaphore; a subsequent task switch could result in each task being blocked (forever), waiting for the other to release. A similar condition, known as live lock, occurs when two tasks continually switch in response to changes in the other task, with the result that neither runs to completion [4].

Different real-time operating systems employ different algorithms, or resource access protocols, to request and release semaphores in order to avoid priority inversion. A common method is called priority inheritance. In this protocol, whenever a lower priority task blocks a higher priority task, it inherits the priority of the blocked task. Reconsider our priority inversion problem, this time with priority inheritance protocol as illustrated in Figure 5. Once again, the low priority task is running and takes a semaphore; all other tasks are blocked, and again the high priority task unblocks and attempts to take the semaphore before the low priority task has released it, blocking again until the semaphore is available. In the meantime the intermediate priority tasks have unblocked, but with priority inheritance, the low priority task has *inherited* the priority of the blocked high priority task. Consequently, the RTOS will schedule the blocking task with its promoted priority first, which runs until the semaphore is released, at which time the high priority task takes the semaphore and runs, and the promoted task is reassigned its initial low priority. Consequently,

[4]This is analogous to the situation of two people approaching each other in a hallway; one moves to his right while the other moves to her left in order to pass, each blocking the other. This dance is usually repeated a couple of times before someone stops and the other, looking sheepish, goes around.

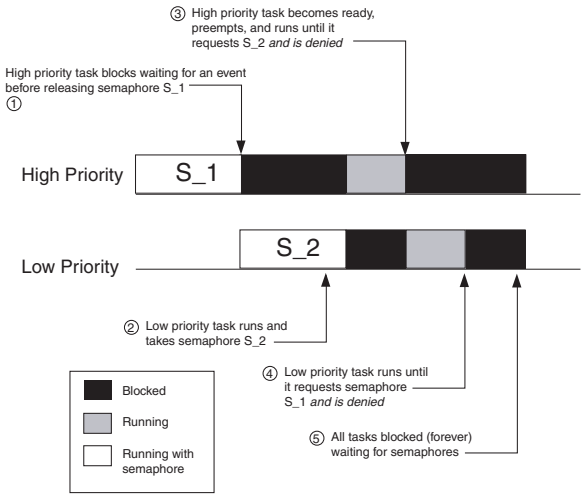

Figure 4: Example of Deadlock

all tasks will run in the correct priority order. Note that if the high priority task accesses multiple shared resources (that is, there is more than one semaphore), it may potentially block as many times as there are semaphores. Furthermore, priority inheritance protocol does nothing to mitigate deadlock. A more complex algorithm is priority ceiling protocol. In priority ceiling protocol, each task is assigned a static priority, and each semaphore, or "resource" is assigned a "ceiling" priority greater than or equal to the maximum priority of all the tasks that use it. At run time, a task assumes a priority equal to the static priority or the ceiling value of its resource, whichever is larger: if a task requires a resource, the priority of the task will be raised to the ceiling priority of the resource; when the task releases the resource, the priority is reset. It can be shown that this scheme minimizes the time that the highest priority task will be blocked, and eliminates the potential of deadlock. An example of priority ceiling protocol is illustrated in Figure 6.

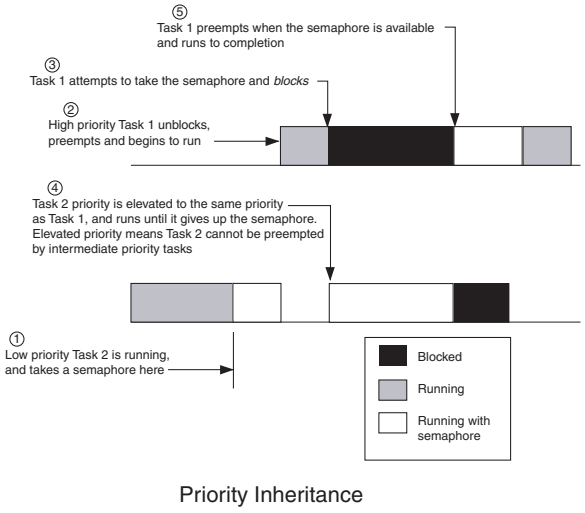

Figure 5: Priority Inheritance Protocol

There are methods other than reliance on RTOS resource access protocols to assure data coherency. Though often used, such methods do not, in general, constitute good programming practice. If the shared data consist of only a single variable, a local copy may be assigned in the low priority task, thus assuring its integrity:

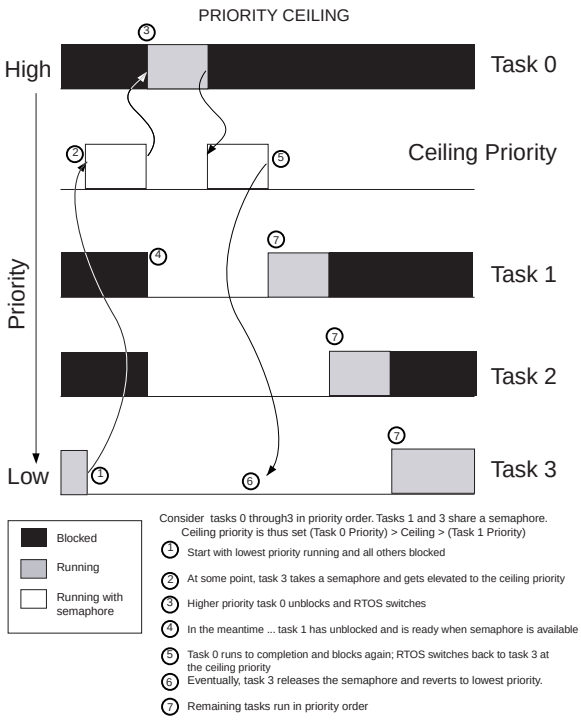

Figure 6: Priority Ceiling Protocol

```
int ADC_channel[3]

void function_10ms(void){
    Read ADC_channel[0]
    Read ADC_channel[1]
    Read ADC_channel[2]
    ...
    do high priority data processing
    ...
    }

int ADC_channel_1
extern int ADC_channel[1]

void function_50ms(void) {
    while(TRUE) {
       ADC_channel_1 = ADC_channel[1];
       ...
       do low priority data processing
       ...
        }
    }
```

Note that this approach requires knowledge of the underlying processor architecture for assurance that the single register read `ADC_channel_1 = ADC_channel[1]` cannot be interrupted (in the PowerPC architecture, for example, even a single register read may be interrupted, but the hardware will restart the read instruction after the interrupt service). Consequently, the code is not *platform independent*, and if for economic or technical reasons the processor changes, the fragile code may no longer work as expected.

5 Scheduling Policies for Real-time Operating Systems

We have said that our real-time operating system examines all of the currently ready tasks, and promotes the highest priority task to the running state. The question arises, "How do we assign priorities to tasks in such a way to assure that each task completes before its deadline?"[5] Consider that in hard real-time systems, there are normally lots of tasks with periodic timing constraints:

- Service the TPU every 10 *ms*.
- Update the PWM output every 50 *ms*.
- Read the A-D converter every 100 *ms*.

The assignment of priorities to tasks is referred to as the scheduling policy. In 1973, C.L. Liu and James Layland analyzed an optimal fixed priority scheduler, and calculated processor utilization bounds for guaranteed service [4]. In other words, optimal refers to a policy that has the highest processor utilization while guaranteeing that each task meets its deadline. Their *rate monotonic scheduling (RMS) policy* assigns fixed priorities in rank order of the task period (that is, the task with the shortest period has the highest priority). Liu and Layland made several assumptions in their analysis:

1. The requests for all tasks for which hard deadlines exist are periodic.
2. Deadlines consist of runability constraints only, that is, each task must be completed before the next request for it occurs.
3. The tasks are independent in that requests for a certain task do not depend on the initiation or completion of requests for other tasks.
4. Task run time is constant.
5. Any non-periodic tasks are special (initialization or failure recovery, for example), and do not have hard critical deadlines.

An example of RMS is shown in the following table and in Figure 7.

Example 1: Rate Monotonic Scheduling			
Task	Execution Time	Period	Priority
T1	1	4	High
T2	2	6	Medium
T3	3	12	Low

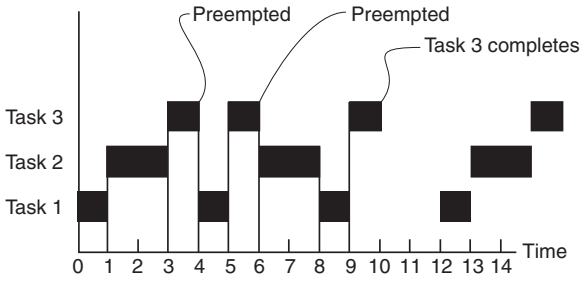

Figure 7: Rate Monotonic Scheduling Example

In Example 1, there are three tasks: Task 1 requires 1 time unit to execute (let's say milliseconds), and has to execute periodically every 4 *ms*; similarly, tasks 2 and 3 require 2 and 3 milliseconds respectively, and

[5]The obvious answer is, "Write the code and try it," and this is how it is often done in practice. Wouldn't it be nice, however, to know beforehand?

have periods of 6 and 12 ms. Consistent with rate monotonic scheduling, task T1 has the highest priority (because it has the shortest period), and tasks T2 and T3 have successively lower priorities. We can see from Figure 7, that all tasks meet their deadlines. Task T1, with highest priority, executes first and runs to completion every four milliseconds. Task T3, the lowest priority task, runs last and is preempted repeatedly by higher priority tasks: After 1 millisecond, T3 is preempted by task T1; when T1 completes, T3 runs for another millisecond before being preempted by task T2. Finally, after T1 executes one more time, T3 runs to completion before its 12 ms deadline.

Example 2: Rate Monotonic Scheduling			
Task	Execution Time	Period	Priority
T1	2	4	High
T2	3	6	Medium
T3	3	12	Low

Consider Example 2. Again we have three tasks with periods of 4, 6 and 12 ms. This time, however, they have execution times of 2, 3 and 3 ms, respectively. In a span of 12 ms (the longest period), T1, (with period equal 4), will execute 3 times; T2, (with period equal 6), will execute twice; and T3 will execute once. The total execution time for each task over the longest period is:

T1: $3 \times 2 = 6\ ms$

T2: $2 \times 3 = 6\ ms$

T3: $1 \times 3 = 3\ ms$

In other words, in a time span of 12 ms, our processor needs to execute 15 ms of code – clearly, this can't be done: the tasks have exceeded the *capacity* of the processor.

5.1 Processor Utilization

Processor utilization for a set of n tasks is defined:

$$U = \sum_{i=1}^{n} \frac{e_i}{P_i} \tag{1}$$

where e_i is the execution time and P_i is the period of each task. Liu and Layland showed that for RMS scheduling with guaranteed deadlines, the upper bound for processor utilization is:

$$U_{RMS} = n(2^{1/n} - 1). \tag{2}$$

As the number of tasks becomes very large, the upper bound processor utilization for guaranteed performance converges to:

$$\lim_{n \to \infty} U_{RMS} \approx 69\%. \tag{3}$$

RMS Processor Utilization	
Number of tasks, n	Upper bound utilization, U
2	82.8%
10	71.8%
100	69.6%

5.2 Schedulability

We can use the concept of processor utilization to establish if a set of tasks can be rate monotonic scheduled, by determining if the total utilization for all tasks is less than the RMS utilization bound defined in (2). That is, if

$$\sum_{i=1}^{n} U_n \leq n(2^{1/n} - 1) \tag{4}$$

then we can say that the set of n tasks is RMS schedulable. Note, however, that this is a *sufficiency* test (it is *sufficient* but not *necessary* that the inequality test be met), and if the utilization sum is greater than the RMS bound, it does *not* mean that the tasks are *not* schedulable. Think about the following examples.

RMS Schedulability Example 1

Consider two tasks, T1 and T2 with the following characteristics:

Task	Execution time	Period	Utilization
T1	20	100	$20/100 = 0.20$
T2	30	145	$30/145 = 0.21$
			$\sum U_n = 0.41$
			RMS bound $= 2(2^{1/2} - 1) = 0.828$

Since the utilization sum for all tasks (0.41) is less than the worst case RMS bound (0.828), the tasks are definitely RMS schedulable.

RMS Schedulability Example 2

Now consider three tasks:

Task	Execution time	Period	Utilization
T1	20	100	$20/100 = 0.20$
T2	30	150	$30/150 = 0.20$
T3	80	210	$80/210 = 0.381$
			$\sum U_n = 0.781$
			RMS bound $= 2(2^{1/3} - 1) = 0.779$

In this case, the utilization sum is greater than the worst case RMS bound. Remember, however, this does not *necessarily* mean that the tasks are not schedulable – it only means that we can't tell from the test. What do we do now? We have to consider the *critical instant* of the task to convince ourselves that these tasks are, indeed, schedulable.

5.3 Critical Instant of a Task

The *critical instant* of a task, according to Liu and Layland, occurs whenever the task is requested simultaneously with the requests of all higher priority tasks. If, at this critical instant, the task can execute before its deadline, then it is schedulable. Another way to say this is that if we can assure ourselves that the lowest priority task will complete before its deadline when all the higher priority tasks have to execute first, then all tasks are schedulable. In general, the time required for a processor to execute a task e_i at its critical instant over a time interval t is:

$$W_i = \sum_{k=1}^{i-1} \left\lceil \frac{t}{P_k} \right\rceil e_k + e_i \tag{5}$$

where $\lceil x \rceil$ denotes the smallest integer greater than or equal to x. Now we can return to our three-task problem and calculate whether or not our lowest priority task will execute in less than its 210 ms deadline:

$$W_3 = \left\lceil \frac{210}{100} \right\rceil 20 + \left\lceil \frac{210}{150} \right\rceil 30 + 80$$
$$W_3 = 3 \times 20 + 2 \times 30 + 80 = 200, \tag{6}$$

and since $200 \leq 210$, which is the deadline for task T3, we know that all tasks are rate monotonic schedulable. What happens if we add one more task? Let's consider the following:

Task	Execution time	Period
T1	20	100
T2	30	150
T3	80	210
T4	100	400

Now calculate the time required for the lowest priority task to execute at its critical instant:

$$W_4 = \left\lceil \frac{400}{100} \right\rceil 20 + \left\lceil \frac{400}{150} \right\rceil 30 + \left\lceil \frac{400}{210} \right\rceil 80 + 100$$
$$W_4 = 4 \times 20 + 3 \times 30 + 2 \times 80 + 100 = 430, \tag{7}$$

and since $430 \geq 400$, the deadline for task T4, the system is *not* schedulable.

Rate monotonic scheduling is a *static* policy: priorities are established at design time and remain fixed. In a *dynamic* scheduling policy, priorities may change during program execution. A deadline driven scheduling algorithm is the earliest deadline first policy; that is, the operating system will promote from ready to running the task with the closest deadline. Liu and Layland established that an earliest deadline first policy may theoretically achieve a processor utilization of 100%. The deadline driven policy is feasible if:

$$\frac{e_1}{P_1} + \frac{e_2}{P_2} + \cdots + \frac{e_n}{P_n} \leq 1 \tag{8}$$

Two of the underlying assumptions for RMS are that all tasks are periodic and that execution times are known and constant. In real systems these assumptions are regularly violated: some tasks are aperiodic (event driven interrupts, for example), execution times are not always constant and are almost never known *a priori*. Nonetheless, rate monotonic scheduling is widely implemented, and provides a method of assessing processor capability.

References

[1] David E. Simon, *An Embedded Software Primer*, Addison-Wesley, 1999.

[2] John Ciardi, *A Browser's Dictionary*, Harper & Row, New York, 1980.

[3] Information Technology Central Services at the University of Michigan, http://www.itcom.itd.umich.edu/wireless/getstarted.html#bsod, viewed October, 2006.

[4] C.L. Liu and James W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the Association of Computing Machinery*, Vol. 20, No. 1, January 1973, pp. 46–61.