Traffic Light Implementation Using uC/OS- a project in Real time systems Course

Gerald Jochum, Emmanuel Tuazon and Subramaniam Ganesan
Oakland University, Rochester, MI 48309
gbjochum@oakland.edu, etuazon@oakland.edu, ganesan@oakland.edu,

Abstract

The purpose of this paper is to show a design and implementation of a real time application on a real time operating system. Using an established or proven toolset to do the real time task support provides both speed in developing a totally new design and the trust in a product that is a proven real time manager. The range of possible applications using a real time system could be a house hold light controller, an engine control unit that is the heart of the modern car, or a thermostat controller as well as the traffic light controller[1] which was chosen for this paper. The simple traffic light controller application is written in C for Micrium's µC/OS-II (commonly known as uCOS) and is ported to a HCS12 microprocessor. uCOS is a preemptive real time operating system that is priority driven by using task scheduling. The paper will cover real time scheduling and how it's accomplished by using uCOS. This project is part of a graduate real time systems course and utilizes many of the topics covered in the course.

Introduction --µC/OS-II

Micrium has been around over 15 years, improving and supporting the µC/OS and supporting packages[2]. It allows the new product development to accelerate the phases of scheduling and resource management. The developer has to focus more on the value added function they are to write. There are various additional complimentary modules that can assist in debugging or performance measures of the running µC/OS system.

Micrium's µC/OS-II (uCOS moving forward) is a deterministic pre-emptive real time kernel for microprocessors. uCOS primary audience is aimed at embedded system designers where having an operating system that is easy to implement and portable is a plus. uCOS provides the following features:

- Ported to more than 100 processors
- Supports 8 to 64-bit processors, micro-controllers and Digital Signal Processors (DSPs)
- Interrupt management (nested 255 levels deep)
- Manages up to 64 tasks (8 reserved for the system)
- Provides system services, e.g. semaphores, queues, mailboxes, fix-sized memory partitions, etc.

Missed deadlines are an important issue to prevent regarding any real time system. In a Soft Real Time system you can have failures that may lead to significant loss but regarding hard real time, these losses can be catastrophic. Predicting the outcome to avoid catastrophic events becomes very important regarding hard real time systems. To gain insight into this predictability, using uCOS is ideal. Our traffic light controller is a Hard Real Time system which will use a lot of bandwidth. The operating system must consider which task takes priority. uCOS task management is similar to Rate Monotonic Scheduling (RMS) where the highest priority tasks are executed first. RMS assumes the following:

- All tasks are periodic
- Tasks do not synchronize with other tasks, i.e. share resources, exchanges data
- Preemptive scheduling must be implemented to ensure that the Central Processing Unit (CPU) executes the highest priority tasks that are ready to run

uCOS is a preemptive kernel—this translates to the fact that the highest priority task is always ready to execute. Interrupts or the Interrupt Service Routine (ISR) will preempt tasks when called upon resulting in resuming the highest priority task once it's completed--Figure 1 illustrates this.

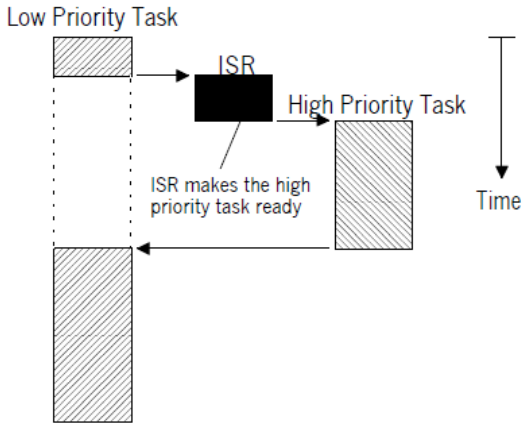

Figure 1: uCOS prioritizes tasks and interrupts

Tasks

Tasks in uCOS are infinite loops (Listing 1).

```
void YourTask (void *pdata)                    (1)
{
   for (;;) {                                  (2)
      /* USER CODE */
      Call one of uC/OS-II's services:
      OSMboxPend();
      OSQPend();
      OSSemPend();
      OSTaskDel(OS_PRIO_SELF);
      OSTaskSuspend(OS_PRIO_SELF);
      OSTimeDly();
      OSTimeDlyHMSM();
      /* USER CODE */
   }
}
```

Listing 1: Task example in uCOS

The uCOS provides a library of functions we can call. This library of functions would best be referred to as the Application Programming Interface (API). An API call appears as a name of the desired function and ends with (). An example would be OSTaskCreate().

Tasks are a continuous sequence of operations that repeat cyclically. Tasks can be deleted when they are not needed. They are more typically set as dormant, if not required to be active for a given time period. uCOS can manage up to 64 tasks, however, 8 are used by the system making a total of 56 user defined tasks. To create a task, either of two arguments could be used: OSTaskCreate() and OSTaskCreateExt(). OSTaskCreateExt is an extended version that provides additional benefits over OSTaskCreate.

Figure 2 is a transition diagram for tasks that are controlled by uCOS.

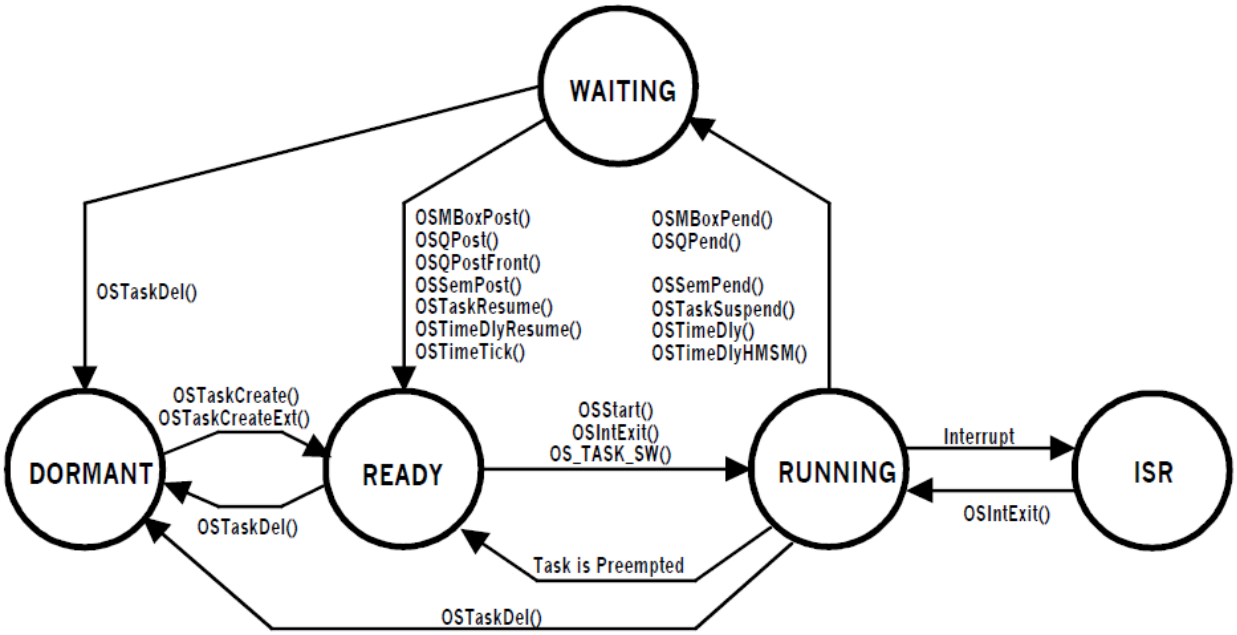

Figure 2: Tasks states

When a task is created by either OSTaskCreate() or OSTaskCreateExt(), it is then in Ready state. If the task is higher in priority, the task is given control of the CPU. Multitasking begins by using OSStart() which is the highest priority and is already in the Running state. Only one task can run one at a time. Ready tasks will only execute when higher priority tasks are in the Waiting state or flagged to remain dormant.

Running state tasks can suspend their execution time by using OSTimeDly() or OSTimeDlyHMSM(). Tasks that use those functions are waiting for their time to expire and the next highest priority task is ready to control the CPU. Using OSTimeTick() allows the task to be delayed until the time delay expires. The Running State of a task may be suspended until an event is called by using OSSemPend(), OSMboxPend() or OSQpend(). This task will wait until those events execute. When a task is pending due to an event, the next highest priority task is given permission to control the CPU. The task will get the Ready State status when the event occurs. Events can be triggered by another task or an ISR.

uCOS has the capability to disable interrupts allowing running critical task sections to not be interrupted. The developer should refrain from disabling interrupts, but rather request the Operating System (OS) to block interrupts for critical sections.

When a task enters ISR state, the interrupt executes and the task is delayed. uCOS will then determine if the returning ISR still has the highest priority. If another task is indeed higher, the

new higher priority task executes, else the ISR can safely resume. When tasks are busy waiting for events or time expirations, uCOS will fire off the idle task, OSTaskIdle().

Task Scheduling

Higher priority tasks are executed first. uCOS determines this by a scheduler. Task scheduler is called by using OSSched(). Listing 2 shows an example of using the task scheduler.

```
void OSSched (void)
{
    INT8U y;


    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {                          (1)
        y             = OSUnMapTbl[OSRdyGrp];                           (2)
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);    (2)
        if (OSPrioHighRdy != OSPrioCur) {                               (3)
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];                 (4)
            OSCtxSwCtr++;                                               (5)
            OS_TASK_SW();                                               (6)
        }
    }
    OS_EXIT_CRITICAL();
}
```

Listing 2: Example of task scheduler

uCOS uses context switching to save the processor registers of the task. The context switching function is OS_TASK_SW(). All code in the OSSched() is in the critical section of the code and interrupts are disabled. The scheduler can be locked or unlock by using functions, OSSchedLock() or OSSchedUnlock(). Listings 3 and 4 show examples of locking and unlocking the scheduler.

```
void OSSchedLock (void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        OSLockNesting++;
        OS_EXIT_CRITICAL();
    }
}
```

Listing 3: Locking scheduler example

```
void OSSchedUnlock (void)
{
    if (OSRunning == TRUE) {
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {
            OSLockNesting--;
            if ((OSLockNesting | OSIntNesting) == 0) {            (1)
                OS_EXIT_CRITICAL();
                OSSched();                                        (2)
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
```

Listing 4: Unlocking scheduler example

DEMONSTRATION DESIGN

Traffic Light Controller

A traffic light control system lies at an intersection with two streets (2 lanes each) that cross at right angles in Figure 2. Direction for one street goes only north and south and the other street can only go east and west.

Traffic lights consist of 3 lights: green, yellow and red. Traffic lights depending on orientation can be viewed from either direction North/South (N/S) and East/West (E/W). Pedestrian crossing signals (walk/don't walk) are positioned in the middle of the street and depending on orientation can be viewed from either direction (N/S and E/W). Pedestrian push buttons are placed at the corners. There are two at each corner so when pushed the request to have the traffic lights cycle so that the pedestrian can cross in a specified direction (depending on where they are situated) is sent:

North to South
South to North
East to West
West to East

The traffic lights cycle by allowing traffic to move on one street to another[3]. The cycle time is explained using Table 1. The traffic lights operate as follows:

If traffic light is green, traffic can continue onto the intersection.
If traffic light is yellow, traffic must stop and wait for the traffic light to turn red.
If traffic light is red, traffic must stop.

Table 1 Traffic Lights and Pedestrian Crossing Timing

State		N/S Lights	E/W Lights	Pedestrian (N/S)	Pedestrian (E/W)	Delay (sec.)
0	ǂ	R	R	Solid Don't Walk	Solid Don't Walk	1
1	↕	G	R	Walk	Solid Don't Walk	25
2	ǂ↕	Y	R	Flashing Don't Walk	Solid Don't Walk	4
3	ǂ	R	R	Solid Don't Walk	Solid Don't Walk	1
4	↔	R	G	Solid Don't Walk	Walk	25
5	ǂ↔	R	Y	Solid Don't Walk	Flashing Don't Walk	4

The pedestrian signals operate as follows[4-5]:

If pedestrian crossing signal is 'WALK', the pedestrian is allowed to walk across the street.
If pedestrian crossing signal is flashing "DON'T WALK", the pedestrian should not enter the intersection or should increase pace to clear the intersection.
If pedestrian crossing signal is "DON'T WALK", the pedestrian should not enter the intersection.
If pedestrian crossing signal is "DON'T WALK", the pedestrian may need to use the button to request the next light change include the “Walk” option.

A demonstration of the traffic light controller system will provide the following:

1. N/S and E/W traffic light timing
2. Push button to simulate pedestrian request to walk
   (It is Ok to Walk when the walk-light is on. Assume Don’t-Walk when light is off)

The above demonstration will showcase some of the highlights of a real time operating system such as critical and non-critical tasks, scheduling tasks and external interrupts (push button for pedestrian walk request).

Intersection Overview

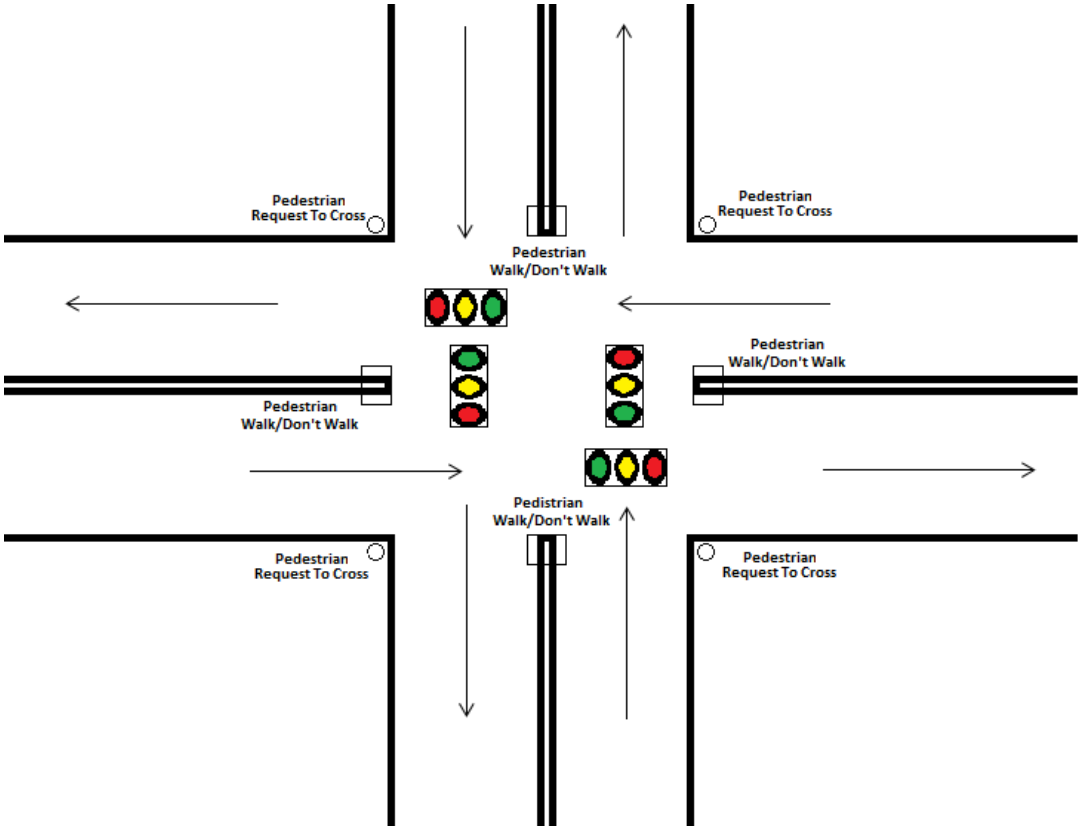

Figure 2: Layout of Intersection with Traffic Light System and Pedestrian Signals and Control

Real-Time Operating System (RTOS)

The choice of uCOS for the traffic light project was modeling a business demand for a quick solution and yet a trustworthy one. The company Micrium has been in business for over fifteen years and they indicate their product has been tested, certified and labeled dependable. The product has been mentioned in some of the universities embedded systems and real time operating systems classes. Prompted by the limited time to make a working model of a traffic light scheduler to complement the report that documented our chosen real time problem and solution, we put the uCOS to the test. Micrium had many premade ports to various common processors and microprocessor demo boards one would expect used as an education tool[6]. With luck, the Dragon 12plus board we had available was among those supported with example code. Looking over the sample module that was free for students to register and download, we compiled it and with minor adjustments were able to run their demo. The minor changes to

compensate our board running a different clock rate were simple. With the examples, most of the logic for making our own task of a traffic light sequencer went quickly.

We needed to add the inter-process communication for detecting a button press. With just a press of a button to simulate the "request to walk" light we needed to trigger a latched flag to the other tasks that sequenced the light changes. Also, the light sequencer task needed to clear the "request to walk" flag when it accepted the request. There were no examples that allowed both tasks to do this interlock and not lock one of the tasks pending the other to provide a release action. With some investigation, the expected API was found. We had the ability to query the flag versus using a wait pending function. There were a few trial runs needed before the query function gave a response. In our next test we got the desired response but the answer was flipped. We now realized the bit test revealed a True/False answer in place of the speculative flag which is one bit. We implemented a minor code change and the sense of flag correctly performed the handshake between tasks. uCOS provided a reliable Rate Monotonic Scheduler that supported priority and context switching among many other features. The uCOS has function calls to request status of other tasks. It also has a port for uProbe, their tool to interactively query the running uCOS kernel via an external serial link. uCOS manages the stack and provides inter-task messaging functions.

The traffic light controller system will use Micriμm μC/OS-II for Wytec MiniDragon-Plus. Figure 3 shows the block diagram of the system. Figure 4 shows the board and the system for demonstration.

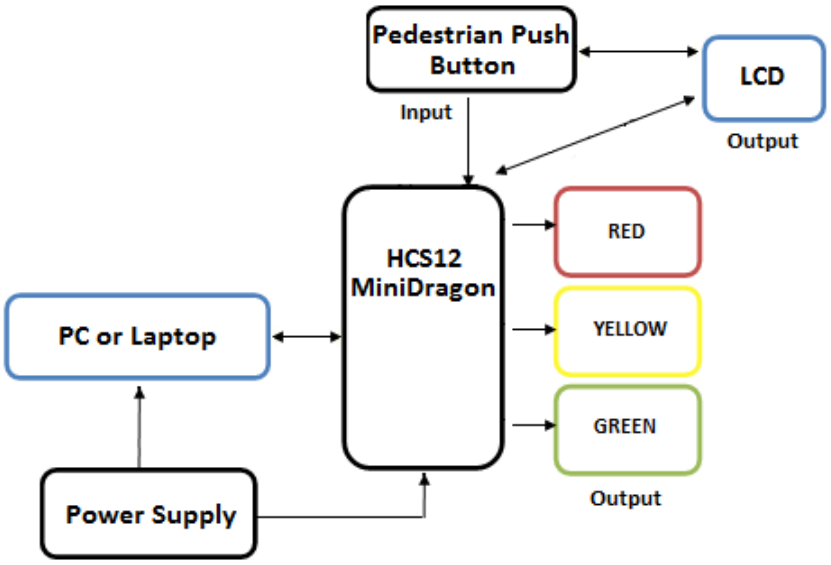

Figure 3: Traffic Light Controller HCS12 with μC/OS-II RTOS Block Diagram

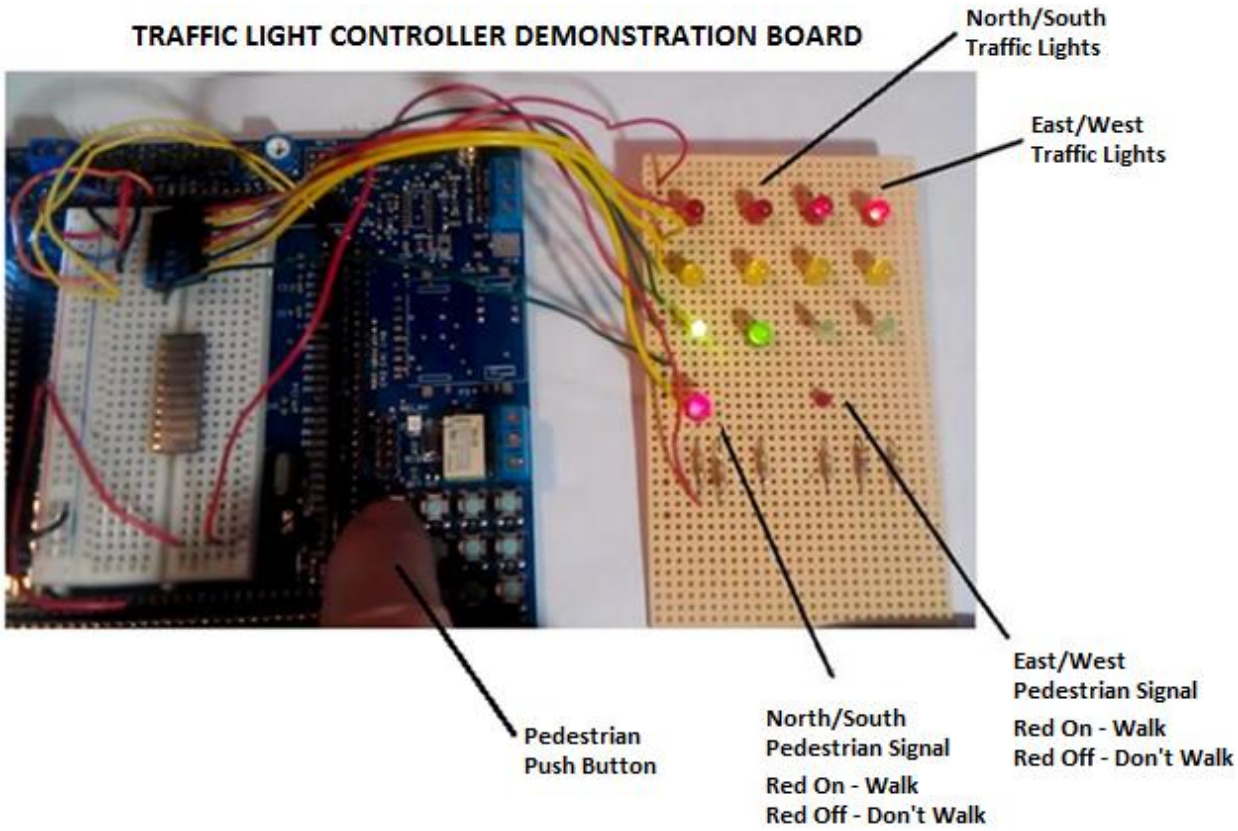

Figure 4. Traffic Light Controller board for demonstration

Application Software

Application software was written in C and Assembly using Freescale CodeWarrior Development Studio. uCOS will achieve the following:

- Initialization
- Control vehicle traffic
- Control pedestrian traffic
- Timed control of traffic lights in the intersection
- Handle pedestrian crossing requests

There is some inter process communication synchronization needed to support "Walk request event."
The light sequencer task would also cycle the walk light with the green light if the inter process flag or trigger is set via other tasks acting on the walk request button press event.

The traffic light controller system processes tasks by using the rate-monotonic approach[7].

- AppStartTaskStk (Setting and starting jobs—one shot)
- LCD_TestTaskStk (LCD task)
- SevenSegTestTaskStk (7 segment task)
- KeypadRDTaskStk (Keypad task)
- TrafficLightTaskStk (Traffic light task)
- TrafficWalkTaskStk (Pedestrian light task)
- OS_Probe_Task (Debugging tool)
- Probe_Comm_RS232_PRIO (Debugging tool)
- OS_Task_Tmr_PRIO (Set the priority of the timer task)

Conclusion

This project shows the use of task scheduling, use of real time operating system, implementation on a microcontroller board and writing real time software for a simple application. A brief video demonstrating the traffic light controller and the ECE 666 Real time system course outline are provided in[8].

REFERENCES

1. *Wilkepedia.com (Traffic Light)*
2. ***MicroC OS II: The Real Time Kernel*** by Jean J. Labrosse
3. SignalTimingManual.com
4. http://www.appropedia.org/LED_traffic_light_FAQ
5. http://blog.motorists.org/6-cities-that-were-caught-shortening-yellow-light-times-for-profit/
6. ***µc/OS-II and The Freescale MC9S12DG256*** (Using the Wytec Dragon12-Plus Evaluation Board - Application Note AN-1456)
7. Real-Time Systems Design and Analysis by Phillip A. Laplante (3rd Edition)
8. *www.secs.oakland.edu/`ganesan*