Design and Implementation of Real-Time Executive (RTDM) for Multitask System

***Dr. Dhuha Basheer Abdullah** ****Dr. Basil Shukur Mahmmod**

Received on: 13/ 2 / 2005

Accepted on: 30/ 6/2005

Abstract

The purpose of this paper is to design an efficient, reliable, fast, easy to extend, and small size experimental real-time operating system kernel for uniprocessor multitask system. It is a kernel with an underlying formal model based on periodic and sporadic tasks with guaranteed response time requirements to tasks and shared software resources. It is distinguished by the programming model it supports and its use of a new processor scheduling and resource allocation.The kernel represents a nucleus of supervisory software to perform the required real–time functions. It logically runs and coordinates concurrent and physically asynchronous activities. It offers appropriate synchronization and task communication mechanism among tasks.

الخلاصة

الغرض من هذا البحث هو تصميم نواة تجريبية لنظام تشغيل زمن حقيقي يستعمل لنظـــام أحـــادي المعـــالج ومتعدد المهام على أن تكون هذه النواة كفوءة وسريعة ومن السهل توسيعها وصغيرة الحجم ويمكن الاعتماد عليها ، وهذه النواة تحتوي على نموذج رسمي مبني على مهام دورية وتعاقبية باستجابة مضمونة لمتطلبات الـــزمن للمهـــام ومصادر برمجية مشتركة . والنواة المقصودة تتميز بنموذج البرمجة الذي تسنده واستعمالها لجدولة جديـــدة للمعـــالج وحجز المصادر .

كما أن هذه النواة تمثل نواة لمراقبة برمجية للقيام بوظائف الزمن الحقيقي اللازمة وهي بذلك تنفـــذ منطقيـــا وتنسق الفعاليات المتزامنة وغير المتزامنة فيزيائيا ، كما تقدم ميكانيكيات مناسبة للتزامن والاتصال بين المهام .

1. Introduction

A real-time computer system is required to provide timely responses to external events occurring in its operating environment. The performance of such a system is directly related to its ability to adhere to timing constraints placed by these external events and the strictness of these constraints[4, 5, 9, 11, 12]. Based on the nature of these constraints, real-time systems can be broadly classified into hard and soft real-time systems. Hard real-time systems are required to meet every timing constraint without fail**[**1, 2, 10**]**. The performance of hard real-time systems must be predictable in a deterministic sense. In contrast, temporal correctness requirements in soft real-time systems

are less stringent (i.e., failure to meet every timing constraint does not affect the correctness of the system) [6, 7]. The scope of this work is soft real-time system.

With the rapid advances in computing and communication technology, software driven computerised control is being widely employed in many real-time systems[8, 11] . Software implementation of such real-time systems offers many advantages - it permits more complex control operations in a responsive manner, and can also be made to dynamically adapt to changes in the operating environment through the use of appropriate control structures. The increasing complexity and the sophisticated demand of such systems in terms of safety, reliability, and performance requires for both functional and temporal (real-time) aspects of the system[3, 9].

The real-time system can be characterised by the following [10]:-

1- A real-time system is any system that responds in a timely manner.
2- Many real-time systems are embedded systems, i.e., they are components of a larger system that interacts with the physical world.
3- Real-time system software must also be concurrent[8].
4- Many real-time systems are dependable.
5- As a result, real-time systems must often be highly reliable (i.e., they must perform correctly), and available (i.e., they must operate continuously).

In this paper, we consider the problem of real-time system design from a temporal perspective, which is what makes real-time system design inherently different from other forms of system design. That is, real-time system design must explicitly consider the timeliness aspects, so that the desired real-time behaviour can be predictably met in the final system. The designed real-time kernel is called Real-time Deadline Multitask (RTDM) and the following sections show the complete RTDM structure.

2. RTDM Layers

The kernel RTDM consists of three layers :-

- application layer
- scheduling layer
- machine layer

The application layer is a layer on which user interaction programs and application are supported and integrated. The upper layer functions are divided into six groups (table 1). These groups of functions handles :
task management, time management, task synchronization, task communication, system initialization, and system management. This layer is written in C++ objected oriented language. The second layer is divided into six groups that represent the main functions used to manage and control the over all system (table 2). It handles and manages lists, queues, software traps, hardware interrupts, task state transition, events, and scheduling and dispatching tasks. This layer is written in C++ object oriented language. While the machine layer is used to carry out low level functions and machine dependent programs (table 3). It consists of three groups that are used to carry out context switching, enable and disable interrupts, clearing and restoring flags, initializing registers and to provide I/O drivers of RTC and DUART. The third layer is written in PentiumIII assembly language.

The data structures of the RTDM contain blocks, tables, lists, and queues. Blocks, lists, and queues are

implemented using "struct" and "typdef" C++ declaration types. Kernel tables are implemented using arrays of integer data type.

The designed kernel (RTDM) in this research has been written using a mix of high level language and assembly language. C++ object oriented programming language is used in the implementation of all machine independent functions. Assembly language is used for writing dependent subroutines and device drivers. C++ is a well-known efficient programming language which combines high level language with factionalism of assembler. Interfacing C++ language to assembly language is implemented either by in line coding in C++ programs using the '#asm' and '#endasm' directive or linking the previously assembled relocatable file (s) with the compiled C++ programs.

Table (1) Application Layer Functions Summary.

Tasks Management RTD_CreateTask() : create a new task. RTD_TerminateTask() : terminate a complete running task. RTD_BlockTask() : block a task and put it in wait state. RTD_ResumeTask():resume a waiting task and put it in ready-to-run list RTD_SleepTask() : put calling task to sleep.
Time Management RTD_GetTime() : get time. RTD_SetTime() : set time. RTD_CreateTimer() : create a logical timer for specified task. RTD_CancelTimer() : delete a logical timer.
Task Synchronization RTD_CreateSem() : create semaphore. RTD_CloseSem() : delete semaphore. RTD_WaitSem() : wait on a semaphore. RTD_SignalSem() : signal a semaphore. RTD_WaitSemTimeout() : wait on a semaphore with timeout.
Task Communication RTD_Send() : send a message to a task queue. RTD_Receive() : obtain a message from one of it's event queue. RTD_Reply() : reply a message to a task queue.
System Initialization RTD_InitTaskTable() : initialize task table. RTD_Initkernel() : initialize RTDM kernel. RTD_InitRTC() : initialize real time clock. DTR_InitLogicalTimer() : initialize logical timer. DTR_InitIDT() : initialize interrupt descriptor table.
System Management RTD_GetTaskState() : get task state. RTD_SetTaskState() : set task state. RTD_GetTaskPriority() : get priority of a specified task. RTD_SetTaskPriority() : set priority of a specified task.

Table (2) Scheduling Layer Functions

Queue Management RTD_Enqueue() : add object to the end of queue. RTD_Dequeue() : remove object from front of the queue. RTD_remove() : remove object from whatever queue it is currently in.
Scheduling and Dispatching RTD_Relinqush() : give up the processor. RTD_Reschedule() : scheduling tasks.
Task Handling RTD_AddReady() : add task p to the list of ready-to run tasks. RTD_RemoveReady() : remove task p from the list of ready-to-run tasks. RTD_addNewtask() : increment the scheduler count of tasks. RTD_RemoveTask() : reduce the count of active tasks by one.
Interrupt Handler RTD_SystemIntHand() : system interrupt handler. RTD_RTCIntHand() : RTC interrupt handler. RTD_IOIntHand() : I/O interrupt handler.

I/O Handler
RTD_WritetoIO() : write to I/O device.
RTD_readIO() : read from I/O device.
RTD_GetSTIO() : get status of a specified I/O device.
Time and event handler
RTD_TimerISR() : timer handler.
RTD_InterHand() : Interrupt handler.

Table (3) Machine Layer Functions Summary.

Initialization and Context Switching
RTD_InitStackBase() : initialize stack. RTD_InitRegs() : initalize registers. RTD_ASMContextSwitch():context switching.
Interrupt Management
RTD_EnableInts() : enable interrupts at the processor. RTD_DisableInts() : disable interrupts at the processor. RTD_RestoreFlages() : update processor flags register.
I/O Driver
RTD_SerialDriver() : serial driver. RTD_RTCDriver() : RTC driver.

3. RTDM Data Structure

Four different data structures are defined in this kernel TD, TB, DD, and IOD :

1- Task Descriptor (TD): Is a template that describes the status of each task within the real-time system. It represents the control point through which the kernel defines the actions of a task. Its fields are:-

Task name;
Task priority;
Task release time;
Task deadline time;
Task state;
Task time out;
Task cost;
Task register
Task list of resources;

2- Time Block (TB) : used to implement software logical timers and time delays, its fields are :-

Associated TD;
Task control flag;
Repetition interval;
Time of expiration;

3- Device Descriptor (DD): It is used for each I/O device. Each DD describes the characteristics of a device in a standard form, its fields are:-

Device name;
Device state;
Vector number;
Read entry;
Write entry;
IOD entry;
Getstate entry;
Set state entry;
Terminate entry;

4- Input/Output Descriptor (IOD) : IOD is used to describe an individual I/O operation. All IODs for a device are linked together in a queue pointed to by the DD for the device, its fields are :-

Associated TD;
Device name;
Data address;
Data count;
Status field;

4. RTDM Program Structure

The general steps of operation of the RTDM system is given by the flowchart shown in figure (1)

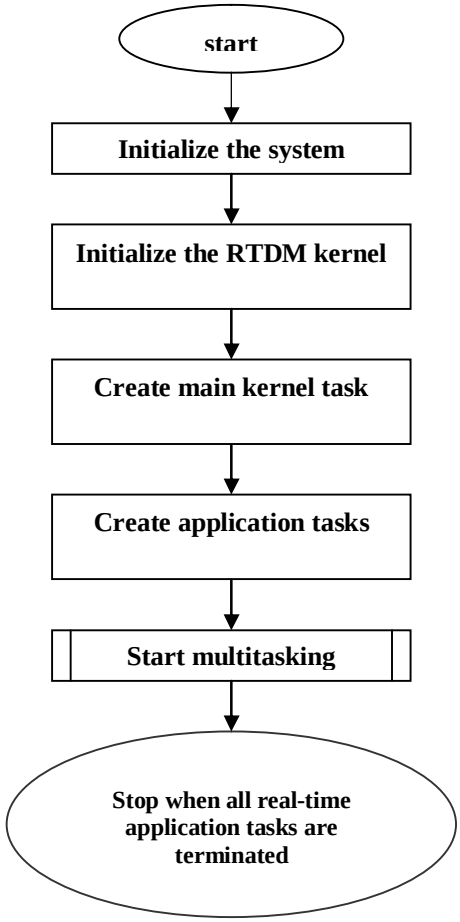

Figure (1) General Steps of the RTDM system.

4.1 System Initialization

The main initialization activities are :

. Defining the allowable task states and priorities.
. Initialization of the data structures.
. Initialization of Queues, lists, and other data types or structures.
. Initialization of real-time clock.
. Initialization of logical software timers.
. Initialization of standard I/O devices.
. Redefining Interrupt Descriptor Table entries.
. Redefining of UART.
. Initializing the kernel RTDM.

The initialization steps of the kernel RTDM are shown in figure(2).

```
RTD_InitializeKernel()
{
  create scheduler instance;
  initailize the sleep queue;
 initialize the stack;
create the idle task;
add idle task to ready
queue; create semaphore
mutex;
}
```

Figure (2) Function to initialize the kernel.

4.2 Start Multitasking

This function makes the idle task active task not by performing a context switch but in such a way specialized for this function. The idle task then relinquishes the processor, and because the main task is ready to run, RTDM kernel performs a context switch to this task. The RTDM program is now a multitasking system. Figure (3) shows the algorithm for this function.

```
RTD_Start_Multitasking()
  { remove the idel task from the
    ready queue;
    change idle task status to active;
    RTD_Relinquish();  }
```

Figure (3) Start Multitasking algorithm.

4.3 Task Management

For any task in the Real-time system, the RTDM kernel defines its state according to the diagram shown in figure (4). The probable state of any task is one of the following :-

. ***Ready :*** the task is ready to run (there may be several tasks in this state).

. ***Active :*** the task is currently running (i.e. the task which has a control on the CPU and it is normally the task that has the highest priority among the ready tasks).

. ***Blocked :*** the task is blocked on a semaphore.

. ***Sleep-Blocked :*** the task is blocked on a sleep queue.

. ***Queue-Blocked :*** the task is blocked on an event queue.

. ***Sem-Timed-Blocked :*** the task is blocked on a semaphore with time out.

. ***Queue-Timed-Blocked :*** the task is blocked on an event queue with time out.

. ***Terminated :*** the task is terminated.

Only one task can be in the active state and other tasks must be in some state other than the active state.

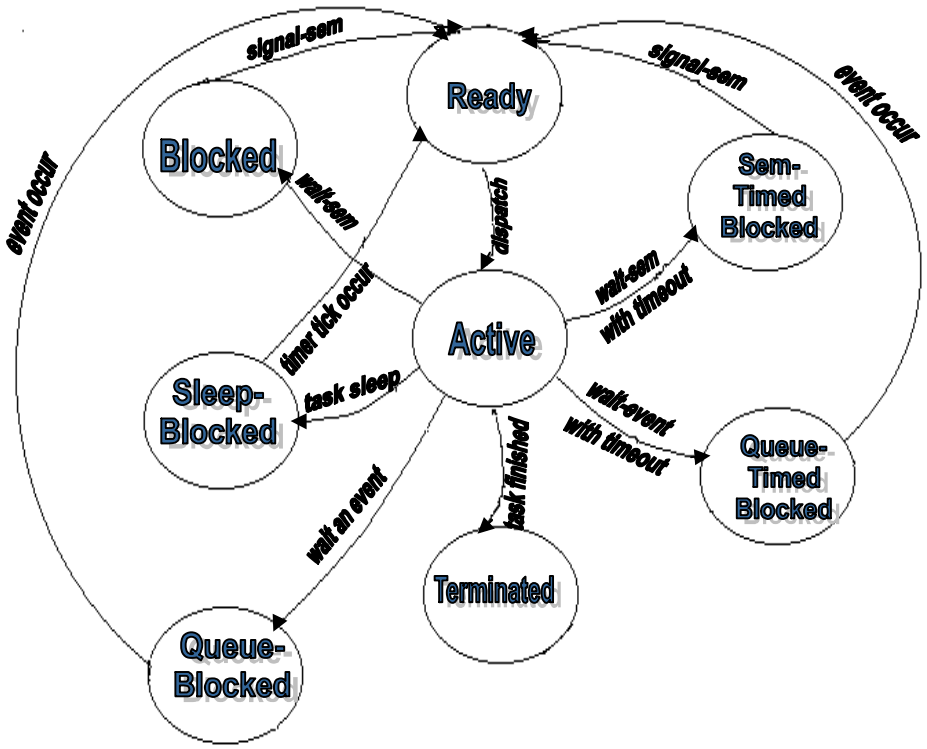

Figure (4) : RTDM's Task State Diagram.

The RTDM kernel calls RTD_CreateTask() function to form a new task. A task descriptor (TD) is being created to store information about the task. Each task has a separate (TD). One task is active at any time, consequently, one TD corresponds to the currently executing task, while all other TDs contain information about the tasks that are either ready to run or blocked. The algorithms that perform task creation, task sleeping, task blocking, task resuming, and task termination are shown in figures (5-9).

```
RTD_CreateTask()
{
   disable interrupts;
   allocate a task descriptor object; // an instance of class task descriptor
   initialize the stack for the task;
   initialize task registers;
   save the task's states;//priority, release time, deadline,…
   initialize the task's event queues;
   change task stater to ready;
  request from scheduler to add this task to its list of   ready-torun task;
  enable interrupts;
  returns a pointer to the new task descriptor object;
 }
```

Figure (5) Task Creation Algorithm.

```
 RTD_SleepTask()
{
  disable interrupts;
  if time tick of the system timer >0 then   {
     change task state to task-sleep-Blooked;
     insert the task in the sleep queue;
     RTD_Reschedule();     }
  enable interrupts;
 }
```

Figure (6) Task Sleeping Algorithm.

```
RTD_BlockTask()
{
 save the current state of the CPU in a TSS table;
 test the action;
if  wait-sem then insert task in the Blocked Queue
else if  wait-sem with timeout then insert task in
    sem-time Blocked Queue
else if   wait-an event then insert task in queue
    Blocked
else if wait-an event with time out then insert task  in queue-
      Timed Blocked
else if task-sleep then insert task into sleep-Blocked  queue;
}
```

Figure (7) Task Blocking Algorithm.

```
RTD_ResumeTask()
{
   remove the task from the Blocking queue;
  insert it in the Ready queue;
}
```

Figure (8) Task Resuming Algorithm.

```
RTD_TerminateTask()
{  reduce the count of active tasks by one;
 change task state to terminated;
 remove logical timer of this task;
 free the memory space allocated to this task;
If count =1 then  terminate multitasking and exit
else   RTD_Reschedule();   }
```

Figure (9) Task Termination Algorithm.

4.4 Time Management

The RTDM kernel uses the Real-time Clock RTC to perform the following functions :

- Returns the current time to the caller.
- Setting the time by passing new values.
- Creating logical timer.
- Cancel logical timer.

The algorithms for these services are shown in figures (10-13).

```
RTD_GetTime()
{
 do {
      check RTC ready bit;
      if RTC is ready then {
     read current time of day from RTC;
     transfer the current time to required format; }
   } while RTC will be ready;
 return current-time;
}
```

Figure (10) Getting Time Algorithm.

```
RTD_SetTime()
{
 set RTC to new passing value;
 call RTD_Gettime();
 if setting value = reading value then
     return message "the setting time is passed"
 else
     return the message "the setting time is failed";
 }
```

Figure (11) Setting Time Algorithm.

```
RTD_CreateTimer()
{
  allocate memory space for new time block;
  if the allocation is succeeded  then
    {
      get the address of current task descriptor;
      store the address of TD, repetition interval,  expiration time,
         and task control flag;
      insert the new time block in the TB queue;
    }
  else    return out of memory;
}
```

Figure (12) Creating Timer Algorithm.

```
RTD_CancelTimer()
{
     remove time block from TB queue;
    cancel all logical operation associated with this time block;
   free memory space allocated with time block;
}
```

Figure (13) Canceling Timer Algorithm.

4.5 Task Synchronization

The RTDM kernel offers a counting semaphore for task synchronization because it is easy to implement, and well understood by most programmers. RTDM semaphore considers : an integer counter, two operations called wait and signal operations, and list of tasks that organized as first-in, first-out queue. When a task requests a wait operation and the semaphore count is 0, the task is blocked (by being placed at the tail of the semaphores' queue). If the semaphore counter is positive, wait operation decrements the counter and the task continues executing. The signal operation complements wait. Signaling a semaphore causes the blocked task at the head of the semaphore queue to be removed from the

list; that task is now eligible to run. The semaphore counter keeps track of a number of signal operations that occur when no task is blocked on the semaphore. If a semaphore's queue is empty when it is signaled, the semaphore counter is incremented by one. In the RTDM kernel five functions calls are available for semaphore, these are :-

RTD_CreateSem : create a semaphore as shown in Fig (14).

RTD_ColseSem : delete semaphore.

RTD_SignalSem : signaling a semaphore as shown in Fig (15).

RTD_WaitSem : waiting on this semaphore as shown in Fig (16).

RTD_WaitSemTimeout : Waiting on this semaphore with time out as shown in Fig(17).

```
RTD_CreateSem()
{ disable interrupts;
 initialize the semaphore-count;
 allocate a new sem object;
 enable interrupts;
 return a pointer to the object;  }
```

Figure (14) Creating a Semaphore Algorithm.

```
RTD_SignalSem()
{
 disable interrupts;
 if semaphore-queue is empty  then
    increase the semaphore-count by one
 else  {
        remove the task from the front of the semaphore queue;
        if the task specified a timeout remove the task from sleep
            queue;
       make the removing task ready;
       RTD_Relinquish();  }
  Enable interrupts;
}
```

Figure (15) Signaling a Semaphore Algorithm.

```
RTD_WaitSem()
{
 disable interrupts;
if semaphore count > 0 then
   decrease count by one
else {
        make the active task blocked moving it to the semaphore
           queue;
       RTD_Reschedule();
enable interrupts;
}
```

Figure (16) Waiting on a Semaphore Algorithm.

```
RTD_WaitSemTimeout()
{
 disable interrupts;
 if count then   decrease count by one
else  if timeout = 0  then  status = Timed_out
     else  {
     add a pointer of calling task to the sleep queue;
    change active task status to SemaphoreTimedBlocked;
   RTD_Reshedule();  }
enable interrupts;
}
```

Figure (17) Waiting on Semaphore with Timeout Algorithm.

4.6 Task Communications

Inter-task communication in the RTDM is fulfilled via message passing. Semaphores are relatively low-level mechanism for coordinating tasks.

Sometimes a task must respond to several different events, but the order in which the events will occur can not be determined a head of time. Allocating a specific semaphore for each event is not the best approach; that is because when a

task is waiting on a semaphore for an event, another event arrives at another semaphore, then the execution of that task will not be resumed.

The RTDM provides an event queue as a facility for task communication. The event queue permits data to be transferred from one task to another. This kind of interprocess communication is known as message passing. A message is sent to a task by depositing the message in one of the tasks's event queues. Sending a message to an event queue is similar to signaling an event queue, in that neither operation requires the sender to wait. A process retrieves a message by removing it from the queue. Receiving a message from an event queue is similar to waiting for a signals in that it may cause the task to wait. If there are no message in the event queues, receiving task will wait until a message arrives. Thus, event queues provide an asynchronous messaging.

Messges are sent to a task by calling RTD_Send() function :

RTD_Send (destination, queue, msg, ack-queue)

The function arguments are as follows :

. destination - the task identifier of the recipient of the message.

. queue – the event queue where the message will be deposited.

. msg – the message.

. ack-queue – the queue where the sending task will expect to receive a reply().

To obtain a message from one of it's event queues, a task calls RTD_Receive() function :

Queue = RTD_Receive (msg, correspondent, ack-queue)

This kernel function searches the task's unmasked event queues for a message and removes it from the head of a queue. If all the queues are empty, the task waits until a message arrives in one of the unmasked queues. Four pieces of information are returned by RTD_Receive() :

. queue – the queue from which the message was removed.

. msg – the received message.

. correspondent – the task identifier of the task that sent the message.

. ack-queue – the queue where the correspondent task will expect to receive a reply.

Figures (18) and (19) show the algorithms used for sending and receiving a message.

```
RTD_Send()
{  if queue < 0  or  queue >= Max-Queues then return
  else  {
     disable interrupts;
     allocate memory for the message;
     // the active task is being now the correspondent task.
    insert message to event queue
    if state = Queue-Blocked or state = Queue-Timed-Blocked  then  {
       make the task ready;
      add this to the ready queue;
      if priority > active task-priority then  {
          make activer task ready;
         insert it to ready queue;
        RTD_Reschedule();  } }
    enable interrupts;    }
```

Figure (18) Sending a message algorithm.

```
RTD_Receive()
{
      disable interrupts;
      for I= 0; I<max-Queue, I+1  {
t-queue[I].mask = unmasked && event-queue[I].empty()
{
          remove a node from event-queue;
          make message node;
          delete node;
         enable interrupts;
         return I;              }   }
```

Figure (19) Receiving a message algorithm.

4.7 Timeout Mechanism

Real-time systems operate in an unreliable world. So in addition to correctly responding to expected events and dealing with anticipated events, real-time programs must be able to cope when expected events fail to occur. In previous sections, a point is being considered that how tasks wait for events to arrive at semaphores or event queues. But because a waiting task remains blocked until an event happens, it will wait for ever if the anticipated event never occur. To avoid this, the RTDM kernel provides a timeout mechanism. Tasks must wait for events to occur, but if a waiting task does not receive an event before a specified length of time has elapsed, it is unblocked with an indication that a timeout happened. The task can then take whatever steps are appropriate to recover.

4.8 RTDM Scheduler

The scheduler represents the master program of the RTDM kernel which is responsible for sharing the processor between tasks. RTDM provides preemptive, dynamic priority-based task scheduling depending on the Earliest Resource Release and Deadline First (ERRDF) algorithm. This object maintains RTDM's queue of ready-to-run tasks. Ready list is an array of queue (one queue for each task priority level); Task priorities range from 0 (lowest priority) to n (the highest priority), where n is 16. Each element queue is a task descriptor object for a task that is eligible to run. When a task becomes eligible to run, its TD object is added to the queue associated with its priority levels and during scheduling by kernel function (RTD_Reschedule()), the task at the head of the highest-priority, nonempty queue is selected for execution. While this task executes, it is referred to as the current task or active task. Figure (20) shows the algorithm of RTD_Reschedule(). The active task continues to run until it relinquishes the processor by calling a kernel function (RTD_Relinquish) that blocks the task (in other words, the task gives up the processor until RTDM is informed that a particular event has occurred and makes the task eligible to resume execution). Another way for an active task to relinquish the processor is to call kernel function that makes a higher-priority task eligible to run. When this happens, the active task remains eligible to run. Finally, an active task will be preempted when a real-time event causes a higher-priority to become eligible to run. Figure (21) shows the algorithm of RTD_Relinquish() function.

```
RTD_Reschedule()
{
  select the next ready-to-run tasks from the ready queue;
  if calling task is ready and is the highest priority then
   {
     make this task active;
     return;
   }
  make the selected task from queue active;
 perform context switch to that task;
}
```

Figure (20) The Scheduling Algorithm.

```
RTD_Relinquish()
{
    disable interrupts;
    change the active task status to ready;
    add this task to ready queue;
    RTD_Reschedule();
    enable interrupts;
}
```

Figure (21) Relinquishing the Processor Algorithm.

When RTDM performs a context switch between two tasks, it must first save enough information about the active task so that its execution can eventually resume exactly where it left off, suspend the active task, and transfer control to the other task. To do this, RTDM first stores copies of the task registers in the TSS table of the active task's TD. The register contents will change when the next task executes, so RTDM must save their current values while the active task is still executing. Next, RTDM loads the registers from the second TSS table of the task's TD, which causes control to transfer to that task.

4.9 Interrupt Handling

The RTDM kernel handles I/O devices interrupt according to their data transfer mode (fast or slow). It provides two possibilities for managing external events. The first strategy uses conventional interrupt services routines. Here the activities initiated by the interrupts are executed with disabled interrupts. Therefore, this method is only appropriate for short and high priority interrupt. Another strategy serviced interrupt in concurrency with other tasks (i.e. consider interrupt service routine as a new task). This strategy is suitable for long task with low interrupt rate and priority. Figure (22) shows the algorithm that explains the main steps which are required to perform interrupt handling.

```
RTD_IntrHandler()
{
  save general registers values in
appropriate area;
  if the required action indicate an I/O
      device Dependent request then {
        locate the DD for the device;
        determine the address of the
service subroutine; }
   else
     determine the address of the
```

Figure (22) Interrupt Handling

4.10 Time Interrupt Handler

Servicing interrupts from RTC and processing the expired logical timers. The time interrupt handler is shown in figure (23).

```
RTD_TimerISR()
 {
  disable interrupts;
  if timeout then  {
    remove a task from the sleep queue;
    wakeup = pointer to this task;
    while wakeup != Null
      {
        if state = Sem-Timed-Blocked  {
           remove the task from the Sem queue;  }
        make task ready;
        add this wakeup task to ready-to-run list
        if wakeup-> priority > active-priority then
              contextswitchneeded = true;
        remove a task from the sleep queue
        wakeup = pointer to this task;
       }
      if (contextswitchneeded)  then
       {
        change the state of active task to ready state;
        add the active task to the list of ready-to-run
task;
         RTD_Reschedule();
       }
    RTD_Restoreflag;
 }
```

Figure (23) Time interrupt handler algorithm.

5. RTDM Test and Evaluation

Three tasks were taken into account with the following parameters in table (4).

Table (4) Parameter for displaying messages application.

Task number	Task name	Release time in microsecond	Cost In microsecond	Relative deadline in microsecond	Priority
1	$Task_1$	1	10	25	3
2	$Task_2$	4	5	20	1
3	$Task_3$	2	1	16	2

The function of $task_1$ is to display the message "$Task_1$ is now active" ten times. $Task_2$ displays the message "$Task_2$ is now active" five times. $Task_3$ displays the message "$task_3$ is now active". It is obvious from table (4) that $task_2$ has the highest priority, $task_3$ has the medium priority, and task1 has the lowest priority.

The source code for this example is illustrated in figure (24). The three tasks (programs) are linked with the kernel RTDM and then the whole project (real-time system) starts executing under Windows Millennium. Practically it is proved that all these three tasks meet their stated deadline.

```
task task1
release  1
cost     10
relative deadline  25
body
       for (i=0, i<10, i+1)
          DH(cout<<"task1 is now active");
endbody
task task2
release  4
cost     5
relative deadline  20
body
       for (i=0, i<5, i+1)
          DH(cout<<"task2 is now active");
endbody
task task3
release  2
cost     1
relative deadline  16
body
          DH(cout<<"task3 is now active");
endbody
```

Figure (24) : The source code of displaying messages application

6. Conclusions

The area of real-time computations has a strong practical grounding in domains like operating systems, databases, and the control of physical tasks. Besides, these practical applications however research in this area is primarily focused on formal methods (specifically, on reasoning about the correctness of programs) and on communication issues in real-time systems. By contrast, little work has been done in the direction of the design and analysis of algorithms for real-time computations.

A real-time multitasking executive RTDM for uniprocessor system has been designed. RTDM kernel is designed to be small in size, fast in speed, modular in structure, reliable in use and adaptable to new requirements. RTDM offers many features :

- System Initialization
- Task management
- Time Management
- Task dispatching and scheduling using ERRDF algorithm.
- Semaphores as a tool for task-to-task signaling, synchronization and mutual exclusion.
- Message passing for task communications.

In this work, a look is being given to the problem of real-time software design. A two design styles (time-driven and event-driven) have been presented and showed how the two issues are addressed in them.

It is concluded that it is not a convenient way for multitask kernels designed for real-time applications to use time slicing. Time slicing means the kernel schedules tasks in a round-robin fashion, allowing each task to run for a fixed amount of time. Instead for these kernels, a suggestion is to rely on real-time events to make tasks ready to run and preemption of lower-priority tasks to ensure that tasks that need processor time urgently get it.

References

[1] Alur, R., Courcoubetis, C., & Dill, D.L, 1990, " Model-Checking for Real-Time Systems". Proc. Symp. on Logic in Comp. Sc., Pages 414–425.

[2] Chak raborty S., Erlebach T., Kunzli S., Thiese L., 2002, "Schedulability of Event-Driven Code Blocks in Real-time Embedded Systems", Computer engineering and networks laboratory Swiss Feederal Institute of Technology (ETH) Zurich.

[3] Fowler S., Wellings A.J., 1996, "Formal Analysis of a Real-Time Kernel Specification", Fourth Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems, Sweden.

[4] Gerber R. and Kang D. , 1995, "End-to-End Design of Real-Time Systems", John Wiley & Sons Ltd.

[5] Jin M., Baker J. W., Meilander W. C., 2002, "The power of SIMDs Vs MIMDs in Real-time Scheduling", http:// www.cs.kent.edu/~ parallel /papers/ jinot.pdf .

[6] Krishna, C. M. and Shin K. G., 1997, "Real Time System", McGraw-Hill Companies, Inc.

[7] Peter A. Nee, 1997, "Experimental Evaluation of Two-Dimensional Media Scaling Techniques for Internet Videoconferencing", Ph.D Thesis, University of North Carolina at Chapel Hill.

[8] Regehr J., Reid A., Webb K., Parker M., Lepreau J., 2003, "Evolving real-time systems using hierarchical Scheduling and Concurrency Analysis", In Proceeding of the 24th IEEE Real-time Symposiam, pp. 25-36.

[9] Saksena M., 1998, "Real-Time Design : A Temporal Perspective", proc. of IEEE Canadian conference on electrical and computer engineering, waterloo, may 1998.

[10] Saksena M. and Selic B., 1999, "Real-Time Software Design – State of the Art and Future Challenges", IEEE Canadian Review, PP. 5-8.

[11] Saksena M. and Wang Y., 2000, "Scalable Multi-Tasking using Preemption Thresholds", IEEE Real-Time Technology and Applications Symposium, work-in-progress session.

[12] Sanfridson M., 2004, "Quality of control and real-time Scheduling-allowing for time-variations in computer control systems", PhD thesis, Department of Machine Design, Royal Institute of Technology, Stockholm, Sweeden. KTH Machine Design.