Implementation of a Real-Time Kernel

Andersson, Leif

1993

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280–5316
ISRN LUTFD2/TFRT--7511--SE

Implementation of a Real-Time Kernel

Leif Andersson

Department of Automatic Control
Lund Institute of Technology
November 1993

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
Internal Report

Date of issue
November 1993

Document Number
ISRN LUTFD2/TFRT--7511--SE

Author(s)
Leif Andersson

Supervisor

Sponsoring organisation

Title and subtitle
Implementation of a Real-Time Kernel

Abstract

The complete implementation of a Real-Time Kernel is described, including all source code. The implementation is in Modula-2 for MS-DOS machines.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
30

Recipient's notes

Security classification

1. Introduction

This report describes the Real Time Kernel used at the Department of Automatic Control, Lund Institute of Technology, both for courses in Real Time Programming and as a tool for control experiments as part of the research of the department. The report is organized as follows: After a section about background and history follows a short section giving an overview of the kernel together with some timing information. The internal operation of the various parts are then described, starting with the data structures. The later sections of the report describe the hardware interaction, such as clock, keyboard, etc.

2. Background and History

Our department started to use computers in control loops in the early seventies. At that time the computer was a PDP-15 with an RSX-15 operating system. In the late seventies we got a number of PDP-11/03:s and could start experimenting with real time software in high level languages. We ported Concurrent Pascal by Brinch Hansen to this computer and tried to implement a real time kernel in this language. A more complete kernel was written in Pascal with a small nucleus in assembler, and it was used for about five years in project courses. This kernel was ported to IBM-PC clones with Modula-2 as the implementation language. It is this implementation that is described here, although work is in progress to port the kernel to more modern hardware.

It is quite possible to buy commercially available Real-Time kernels or operating systems. The reason we find it worth while write a kernel from scratch, is that we want to be able to discuss it freely with the students. With commercially available kernels it is often not possible to see the source code, much less show it to other people.

3. Hardware and Software

The kernel described here has been implemented and used on IBM-PC clones with 286-processors and math coprocessors, and also on more modern 486-processors. The kernel is implemented in Modula-2, using a compiler from Logitech. The only place where the code is compiler-specific is in the lowest level routines, where we have used special constructs of the Logitech Modula to insert machine code directly in the Modula source. The alternative to using this feature would have been to write the lowest level routines in assembly code.

4. Kernel Overview

Many different concurrency models have been proposed in literature. Among these are the Rendez-Vous model of Ada, the Message Passing model, Semaphores, Monitors etc. We did not want to specify a particular model, rather find and implement a minimal set of basic building blocks such that any concurrency model could be efficiently implemented on top of this base.

One fundamental property of a Real Time Program is that it contains parallel processes. Thus there needs to be a possibility to transfer control between different *threads* or *coroutines* (The author of this report does not know the difference, if any, between these terms). The other fundamental property is that the system can handle external signals, interrupts. The basis for our implementations is the merge of these two properties, i. e. the possibility to transfer control between

coroutines as the result of an interrupt. It should be noted that since the application we have in mind is high-level implementation of control systems, these coroutines will be using floating point computation. It is our impression that few of the commercially available systems takes this into account.

In Modula-2 the required building blocks already exists to a certain extent. There is a TRANSFER call that transfers control from one coroutine to another. There is also an IOTRANSFER call that converts the current process to a interrupt handler process and makes a transfer to another coroutine while waiting for the interrupt. This method means that the handler is a proper process with its own stack context. The disadvantage is that two full context switches are required for every interrupt, which is fairly inefficient.

Therefore we decided that our kernel should use a more efficient mechanism to handle interrupts. Here we let procedures handle interrupts. The advantage of using procedures instead of coroutines, is that the procedure has no context, so neither context restore at entry nor context save at exit is necessary. This means that it is sufficient to save and restore those registers that the interrupt procedure will use. In our current implementation, we let the interrupt procedure live in the stack of the currently executing process, but to use a separate interrupt stack only costs a few instructions, and is worthwhile if we need many small tasks (with small stacks) and/or interrupt handlers that needs lots of stack (not likely).

The scheduler and its queues

All the work of the Real Time Kernel is organized around its two main queues, the Time Queue and the Ready Queue. The former contains processes that have suspended themselves waiting for a specific (future) time instant. The latter contains processes that are ready to run, but compete for the CPU resource. The scheduling policy we have chosen is a strict priority scheme with round robin scheduling among processes of equal priority. Since we keep the Ready Queue sorted in priority order at all times, scheduling is simply achieved by letting the first process in this queue run.

To ensure proper operation of the kernel its important that all queue manipulations are done with interrupts disabled. To make sure that the kernel always is in a consistent state, it is only possible to move a process from one queue to another, not to remove it from one queue without inserting it anywhere else.

The basic scheduling primitives in Modula-2 are: `MovePriority`, which removes a process from its current queue and inserts it in priority order in another (or the same) queue, and `Schedule`, which makes the first process in the ready queue be the running process, subject to interrupt rules described below. There are also some auxiliary routines to disable and enable interrupts.

Time is handled by a clock interrupt driver that is part of the kernel. The driver maintains the Clock Queue and moves processes to the ready queue when appropriate. The basic primitive is a `WaitUntil` procedure that suspends the calling process until a specified time in the future, unlike most kernels, which have a delay statement as the base. It is of course simple to write a delay function given `WaitUntil` and a function that returns the current time, but if a delay statement is the basic primitive then an extra process is needed to wait for a specified time in the future.

Other Primitives

The Semaphore is a simple device used for signaling. It can also be used for data protection, but in our case we chose to implement the more powerful Monitor for this purpose.

A Monitor is an abstract data type with some data and procedures to manipulate this data. The important property is that all Monitor procedures call special

primitives on entry and exit so that at most one process at a time can access the Monitor data.

An Event is a signaling device without memory, i. e. all processes waiting for an event will be released when the event occurs, but if no processes are waiting, the event signaling is a null operation.

Messages and Message passing of various kinds are also important primitives, but in our case we have implemented them on top of the other primitives mentioned here, and they are not described in this report.

Monitor Timing

The times required for the Kernel itself and for some important kernel operations are shown in table 1. The columns are explained below.

Tick As will be explained later, the kernel itself determines a suitable basic tick time based on the speed of the hardware. This column shows the result.

Kernel This is the load the kernel itself puts on the machine. It consists of the clock interrupt every tick.

Cyclic This is the time to switch to a cyclic process, increment a counter in this process, and switch back.

Semaphore This is the time to switch to a process waiting for a semaphore, increment a counter in the process, and wait for the semaphore again.

The lines for the 486, with attributes "cache" and "no cache" respectively, also needs some explanation. Our 486-machines have an external 256 KB cache memory that can be switched on and off. Since the test program consists of loops of rather small pieces of code, it will all fit in the cache memory. A production program is larger, and will therefore not be entirely in the cache. A fair assumption is therefore that the practical times will be between the two values in the table.

5. The data structures

The basic data structures of the Kernel are the Process Records and queues (doubly linked lists) of such Process Records. The process record contains info that the kernel needs to keep separate for each process. Typical examples are process priority and stack address.

The kernel itself has two such queues, the Ready Queue and the Clock Queue. The Ready Queue contains processes that are either running or waiting for the CPU. It is always maintained in priority order so that the process that is first in the queue is the one to run. The Clock Queue contains processes that have suspended themselves waiting for a specific time instant in the future. This queue is maintained in time order so that only the first entry in the queue needs to be checked at each clock tick.

Computer Type	Tick	Kernel %	Cyclic	Semaphore
286, 8 MHz	10 ms	2%	1 ms	1 ms
486, 50 MHz, cache	1 ms	3%	75 μs	75 μs
486, 50 MHz, no cache	1ms	6%	200 μs	200 μs

Table 1. Timing for the kernel and some operations.

Other queues will be created and maintained by other modules. A typical example is that each semaphore has a queue of waiting processes.

The queues are created and manipulated by a module called KernelTypes. The reason to have a separate module for this instead of including it in the Nucleus is that many primitives need special entries in these data structures. It will lead to a simplification in maintenance when a primitive is added if we have a separate module.

The actual data for the kernel is declared in the module Nucleus. This module also contains the procedure Schedule, which uses the Ready Queue to ascertain that the process with the highest priority will get the CPU.

The definition modules for KernelTypes and Nucleus follow here for reference in the following sections. The implementation modules will come later.

```
DEFINITION MODULE KernelTypes;
```
This is the definition of a process record. The reason for separating it from Nucleus, is that many primitives needs some space in this record. It is system and compiler dependent. This version is for IBM-PC.

```
FROM SYSTEM IMPORT ADDRESS;

CONST
  FPsize = 47;
  NameLen = 19;

TYPE
  KernelName = ARRAY [0..NameLen] OF CHAR;
  Time = RECORD hi, lo: CARDINAL; END;
  PROCESS = ADDRESS;
  ProcessRef  = POINTER TO ProcessRec;
  Queue = POINTER TO QueueRec;

  QueueRec  = RECORD
    succ, pred     : ProcessRef;
    priority : CARDINAL;
    nextTime : Time;
    priorityQueue : Queue;
    timeQueue : Queue;
    name : KernelName;
  END;

  ProcessRec  = RECORD
    head : QueueRec;
    Nucleus
    procv : PROCESS;
    timer : CARDINAL;      For time slice.
    FParea : ARRAY [0..FPsize] OF CARDINAL;
    Kernel
    processNr : CARDINAL;
    assignedPriority : CARDINAL;
    stack : ADDRESS;
    stackSize : CARDINAL;
    Monitors
    runningIn : ADDRESS;
    blockedBy : ADDRESS;
  END;

PROCEDURE InitProcessRec(VAR r: ProcessRef);

PROCEDURE InitQueueRec(VAR q: Queue);
```

```
PROCEDURE NewQueue(VAR q : Queue);
```
Creates a new process queue. The queue head is given a priority lower than any proper process and a NextTime as far away as possible in the future.

```
PROCEDURE SetKernelName(VAR kn: KernelName;
                        name: ARRAY OF CHAR);
```
Assigns name to kn. If name is too long it is silently truncated.

```
PROCEDURE MovePriority(P : ProcessRef; q : Queue);
```
Removes P from its queue and then inserts processrecord P in queue Q according to priority.

```
PROCEDURE MoveTime(P : ProcessRef; q : Queue);
```
Removes P from its queue and then inserts processrecord P in queue Q in time order.

```
PROCEDURE CompareTime(VAR t1, t2: Time): INTEGER;
```
See Kernel

```
PROCEDURE IncTime(VAR t: Time; c: CARDINAL);
```
See Kernel

```
END KernelTypes.
```

```
DEFINITION MODULE Nucleus;
```
This is the innermost module of a Real Time Kernel.

```
FROM  KernelTypes IMPORT
  ProcessRef, Queue, Time;

VAR
  Now: Time;
  Running: ProcessRef;
  ReadyQueue     : Queue;
  TimeQueue      : Queue;
```

```
PROCEDURE Init;
```
Initialization. Should only be called by Kernel.

```
PROCEDURE Schedule;
```
Makes the top of the ReadyQueue the running process.

```
PROCEDURE SetEveryTick(TP: PROC);
```
Sets a procedure to be called every clock tick.

```
END Nucleus.
```

6. Semaphores

The Semaphore is the simplest of the Real-Time synchronization primitives. It is described fairly well by its definition module.

Definition Module

```
DEFINITION MODULE Semaphores;
```
Semaphores for the Real Time Kernel. Note that Kernel.init must be called before any of these procedures.

```
TYPE Semaphore;

PROCEDURE InitSem(VAR sem: Semaphore; InitVal: INTEGER;
                  name: ARRAY OF CHAR);
```
Initializes the semaphore sem to InitVal. name is for debugging purposes.

```
PROCEDURE Wait(sem: Semaphore);
```
If the value of the semaphore Sem > 0 then decrement it, else block the calling process. If more than one process is waiting, then queue them first in priority and then in FIFO order.

```
PROCEDURE Signal(sem: Semaphore);
```
If there is one or more processes waiting, then unblock the first one in the queue, else increment the semaphore.

```
END Semaphores.
```

Implementation Module

The data structures for a semaphore contains the semaphore integer and a queue that can hold the processes blocked by the semaphore.

```
IMPLEMENTATION MODULE Semaphores;

FROM Coroutines IMPORT Disable, Reenable, InterruptMask;
FROM Storage IMPORT ALLOCATE;
IMPORT Nucleus;
FROM Nucleus IMPORT
    ReadyQueue, Running, Schedule;
FROM KernelTypes IMPORT
    NewQueue, MovePriority, ProcessRef, Queue;

TYPE
  Semaphore = POINTER TO SemaphoreRec;
  SemaphoreRec = RECORD
     counter: INTEGER;
     waiting: Queue;
  END;
```

The initialization of a semaphore consists of allocating the necessary data structures and setting the semaphore integer to its proper value.

```
PROCEDURE InitSem(VAR sem: Semaphore; InitVal: INTEGER;
name: ARRAY OF CHAR);
BEGIN
  NEW(sem);
  WITH sem^ DO
    counter := InitVal;
    NewQueue(waiting);
  END;
END InitSem;
```

The main semaphore procedures, `Wait` and `Signal`, both follow a similar pattern. All queue manipulations must be done with the interrupts disabled, and therefore the first and last statements are `Disable()` and `Enable()` respectively.

The `Wait` procedure decrements the semaphore integer if possible, otherwise blocks the running process by moving its process record into the semaphore's waiting queue and calling `Schedule`. The call to `Schedule` is really where Real-Time Programming differs most from sequential programming, because this is the point where a process switch takes place. This means that the CPU does not immediately return from the same invocation of `Schedule`, but rather picks up some other execution thread.

```
PROCEDURE Wait(sem: Semaphore);
VAR
  oldDisable: InterruptMask;

BEGIN
  oldDisable := Disable();
  WITH sem^ DO
    IF counter > 0 THEN
      DEC(counter);
    ELSE
      MovePriority(Running, waiting);
      Schedule;
    END;
  END;
  Reenable(oldDisable);
END Wait;
```

The Signal procedure checks if any process is waiting. If so, the waiting process is moved to the Ready Queue, and Schedule is called. If no processes are waiting the only action is to increment the semaphore integer.

```
PROCEDURE Signal(sem: Semaphore);
VAR
  oldDisable: InterruptMask;

BEGIN
  oldDisable := Disable();
  WITH sem^ DO
    IF ProcessRef(waiting) <> waiting^.succ THEN
      MovePriority(waiting^.succ, ReadyQueue);
      Schedule;
    ELSE
      INC(counter);
    END;
  END;
  Reenable(oldDisable);
END Signal;

END Semaphores.
```

7. Events

An event is another simple synchronization primitive, that can be used to let a collection of processes wait for a specific event. When that event occurs all waiting processes are made runnable. In the definition module they are called "Free Events" because the module Monitors, described later, contains a different but related type of events.

```
DEFINITION MODULE Events;
```
Free events for the Real Time Kernel

```
TYPE
  Event;


PROCEDURE InitEvent(VAR ev: Event; name: ARRAY OF CHAR);
```
Initialize the event ev. name is for debugging purposes.

```
PROCEDURE Await(ev: Event);
```
Blocks the current process and places it in the queue associated with ev.

```
PROCEDURE Cause(ev: Event);
```
All processes that are waiting in the event queue associated with ev are unblocked. If no processes are waiting, it is a null operation.

```
END Events.
```

The data structures are simple. They are put in a separate module named `EventInternal` so that it may be possible to access them from special debugging modules separate from the `Events` module itself.

```
DEFINITION MODULE EventInternal;

FROM KernelTypes IMPORT Queue;

TYPE
  Event = POINTER TO EventRec;
  EventRec = RECORD
    waiting    : Queue;
    Debug
    next                : Event;
  END;

VAR
  Initialized : BOOLEAN;
  EventList : Event;

END EventInternal.
```

The main procedures, `Await` and `Cause` are quite similar to the corresponding code in `Semaphores`, with the difference that no integer value is involved.

```
IMPLEMENTATION MODULE Events;

FROM Coroutines IMPORT Disable, Reenable, InterruptMask;
FROM Storage IMPORT ALLOCATE;
FROM KernelTypes IMPORT
   ProcessRef, Queue, NewQueue, MovePriority, SetKernelName;
IMPORT Kernel;
FROM Nucleus IMPORT Running, ReadyQueue, Schedule;
IMPORT EventInternal;
FROM EventInternal IMPORT EventList;

TYPE
  Event = EventInternal.Event;
  EventRec = EventInternal.EventRec;

PROCEDURE InitEvent(VAR ev: Event; name: ARRAY OF CHAR);
BEGIN (* InitEvent *)
  NEW(ev);
  WITH ev^ DO
    NewQueue(waiting);
    next := EventList;
    EventList := ev;
    SetKernelName(waiting^.name,name);
  END (* WITH *)
END InitEvent;

PROCEDURE Await(ev: Event);
VAR oldDisable: InterruptMask;
BEGIN
  oldDisable := Disable();
  MovePriority(Running, ev^.waiting);
```

```
  Schedule;
  Reenable(oldDisable);
END Await;

PROCEDURE Cause(ev: Event);
VAR
  oldDisable: InterruptMask;
BEGIN
  oldDisable := Disable();
  LOOP
    IF ProcessRef(ev^.waiting) = ev^.waiting^.succ THEN
      EXIT
    ELSE
      MovePriority(ev^.waiting^.succ, ReadyQueue);
    END;
  END (* LOOP *);
  Schedule;
  Reenable(oldDisable);
END Cause;

BEGIN (* Events *)
  EventList := NIL;
END Events.
```

8. Monitors

Monitors are used to protect critical regions and guarantee mutual exclusion. They should really be part of a language so that the compiler could automatically insert the lock and unlock code in all procedures accessing the data structure. Since no such language is available to us, we must instead rely on programmer discipline, and simplify the use as much as possible.

Our implementation consists of a data type `MonitorGate` with the operations `EnterMonitor` and `LeaveMonitor`. The `MonitorGate` must then be associated with, or included in, the shared data type, and all procedures operating on it must have `EnterMonitor` as the first and `LeaveMonitor` as the last statement.

Monitors can also have `MonitorEvent` variables associated, similar to the Events described above. The difference is that an `Await` on a `MonitorEvent` will also perform an implicit `LeaveMonitor`. Typcally these events will be used in a producer/consumer situation where the consumer will call `Await` if the buffer is empty, and the producer will call `Cause` every time it enters data into the buffer.

The priority changes mentioned in the definition module will be further explained later in this section.

Monitors Definition Module

```
DEFINITION MODULE Monitors;

TYPE MonitorGate;
TYPE MonitorEvent;

PROCEDURE Init;
```

Initializes the `Monitors` module.

```
PROCEDURE InitMonitor(VAR mon: MonitorGate;
                      name: ARRAY OF CHAR);
```

Initializes the monitor guarded by `mon`. `name` is for debugging purposes.

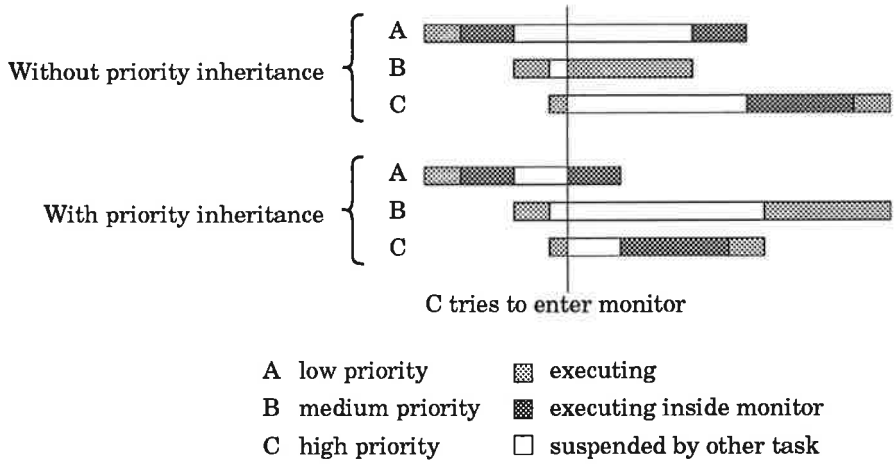

Figure 1. Priority inversion when two processes contend for the same monitor.

```
PROCEDURE EnterMonitor(mon: MonitorGate);
```

Try to enter the monitor `mon`. If no other process is within `mon` then mark the monitor as busy and continue. If the monitor is busy, then block the calling process in a priority queue AND raise the priority of the blocking process to the priority of the blocked process.

```
PROCEDURE LeaveMonitor(mon: MonitorGate);
```

Leave the monitor `mon`. If the priority was raised then lower it to the original value. If there is one or more processes waiting, then unblock the first one in the queue, else mark the monitor as not busy.

```
PROCEDURE InitEvent(VAR ev: MonitorEvent; mon: MonitorGate;
                    name: ARRAY OF CHAR);
```

Initialize the event `ev` and associate it with the monitor `mon`. `name` is for debugging purposes.

```
PROCEDURE Await(ev: MonitorEvent);
```

Blocks the current process and places it in the queue associated with `ev`. Also performs an implicit `LeaveMonitor(mon)`.

```
PROCEDURE Cause(ev: MonitorEvent);
```

All processes that are waiting in the event queue associated with `ev` are moved to the monitor queue associated with `mon`. If no processes are waiting, it is a null operation.

```
END Monitors.
```

Priority Inversion Problem

A possible problem with monitors in general is that a low priority process could unvoluntarily lock out a high priority process for a long time, because another process of intermediate priority prevents the low priority process from finishing its work inside the monitor. In order to prevent this problem, priority inversion, we have implemented a priority inheritance scheme. It means that a process that wants to enter a locked monitor will raise the priority of the locking process to its own priority for the duration of the monitor operation. Figure 1 describes this in some detail.

Monitor Data Structures

The data structures are again put into a separate module for debugging reasons. The records contain the expected queues of blocked processes, and also the variables `blocking` and `priorityDiff`. These variables are used to to implement the priority inheritance mentioned above. The variable `blocking` will contain a reference to the process holding the monitor. `priorityDiff` will indicate how much the priority of the blocking process has been raised. The sections marked `(* Debug *)` contain the name of the monitor, and also a singly linked list of all monitors so that debugging software may find them.

```
DEFINITION MODULE MonitorInternal;

FROM KernelTypes IMPORT Queue, ProcessRef, KernelName;

TYPE
  MonitorGate = POINTER TO MonitorRec;
  MonitorEvent = POINTER TO EventRec;
  MonitorRec = RECORD
    waiting : Queue;
    blocking : ProcessRef;
    priorityDiff    : CARDINAL;
    (* Debug *)
    next       : MonitorGate;
    name : KernelName;
    events : MonitorEvent;
  END;

  EventRec = RECORD
    evMon : MonitorGate;
    waiting     : Queue;
    (* Debug *)
    next       : MonitorEvent;
    name : KernelName;
  END;

VAR
  monitorList    : MonitorGate;

END MonitorInternal.
```

Monitor Implementation

The code for priority inheritance takes up a large part of the routines `EnterMonitor`, `LeaveMonitor` and `Cause`. It has been marked specially in the code to be easily recognized. The reader may notice that apart from the priority inheritance the code is very similar to `Wait` and `Signal` of Semaphores.

```
IMPLEMENTATION MODULE Monitors;

FROM Coroutines IMPORT Disable, Reenable, InterruptMask;
FROM Storage IMPORT ALLOCATE;
FROM KernelTypes IMPORT
   ProcessRef, Queue, NewQueue, MovePriority, SetKernelName;
IMPORT Kernel;
FROM Nucleus IMPORT Running, ReadyQueue, TimeQueue,
    Schedule;
FROM Console IMPORT Trap;
IMPORT MonitorInternal;
FROM MonitorInternal IMPORT MonitorRec, EventRec, monitorList;
```

```
TYPE
  MonitorGate = MonitorInternal.MonitorGate;
  MonitorEvent = MonitorInternal.MonitorEvent;

VAR
  Initialized    : BOOLEAN;

  PROCEDURE EnterMonitor
    (    mon          : MonitorGate);
  VAR
    oldDisable       : InterruptMask;
    runningPriority,
    blockingPriority : CARDINAL;
    blockingQueue    : Queue;
  BEGIN
    oldDisable := Disable();
    WITH mon^ DO
      IF blocking = NIL THEN
        blocking := Running;
      ELSE
        MovePriority(Running,waiting);
        runningPriority := Running^.head.priority;                    □
        blockingPriority := blocking^.head.priority;                  □
        IF runningPriority < blockingPriority THEN                    □
          blocking^.head.priority := runningPriority;                 □
          priorityDiff := priorityDiff + blockingPriority             □
                                       - runningPriority;             □
          blockingQueue := blocking^.head.priorityQueue;              □
          IF blockingQueue <> NIL THEN                                □
            MovePriority(blocking, blockingQueue);                    □
          END;                                                        □
        END;                                                          □
        Schedule;
      END (* IF *);
    END;
    Reenable(oldDisable);
  END EnterMonitor;

  PROCEDURE LeaveMonitor(mon: MonitorGate);
  VAR
    oldDisable  : InterruptMask;
    blockingQueue    : Queue;
  BEGIN
    oldDisable := Disable();
    WITH mon^ DO
      IF blocking <> Running THEN
        Trap("Strange error in Monitors");
      END;
      IF priorityDiff <> 0 THEN                                       □
        INC(Running^.head.priority, priorityDiff);                    □
        blockingQueue := blocking^.head.priorityQueue;                □
        IF blockingQueue <> NIL THEN                                  □
          MovePriority(blocking, blockingQueue);                      □
        END;                                                          □
        priorityDiff := 0;                                            □
      END;                                                            □
      IF ProcessRef(waiting) <> waiting^.succ THEN
        blocking := waiting^.succ;
        MovePriority(blocking, ReadyQueue);
      ELSE
        blocking := NIL;
```

```
    END;
    Schedule;
  END (* WITH *);
  Reenable(oldDisable);
END LeaveMonitor;

PROCEDURE Await(ev: MonitorEvent);
VAR
  oldDisable  : InterruptMask;
BEGIN
  oldDisable := Disable();
  MovePriority(Running, ev^.waiting);
  LeaveMonitor(ev^.evMon);
  Reenable(oldDisable);
END Await;

PROCEDURE Cause(ev: MonitorEvent);
VAR
  oldDisable  : InterruptMask;
  pt : ProcessRef;
  runningPriority, ptPriority: CARDINAL;
BEGIN
  oldDisable := Disable();
  LOOP
    pt := ev^.waiting^.succ;
    IF ProcessRef(ev^.waiting) = pt THEN
      EXIT
    ELSE
      MovePriority(pt, ev^.evMon^.waiting);
      ptPriority := pt^.head.priority;                          □
      runningPriority := Running^.head.priority;                □
      IF ptPriority < runningPriority THEN                      □
        Running^.head.priority := ptPriority;                   □
        ev^.evMon^.priorityDiff :=                              □
                    runningPriority - ptPriority;               □
      END;                                                      □
    END;
  END (* LOOP *);
  Reenable(oldDisable);
END Cause;

PROCEDURE Init;
BEGIN
  IF NOT Initialized THEN
    Initialized := TRUE;
    Kernel.Init;
  END (* IF *);
END Init;

PROCEDURE InitMonitor(
  VAR mon     : MonitorGate;
  name : ARRAY OF CHAR);
VAR
  oldDisable  : InterruptMask;
BEGIN
  IF NOT Initialized THEN
    Init;
  END IF ;

  NEW(mon);
  WITH mon^ DO
    NewQueue(waiting);
```

```
      blocking := NIL;
      priorityDiff := 0;
      events := NIL;
      SetKernelName(waiting^.name,name);
    END (* WITH *);

    (* Debug setup *)
    oldDisable := Disable();
    mon^.next := monitorList;
    monitorList := mon;
    Reenable(oldDisable);
  END InitMonitor;

  PROCEDURE InitEvent
    (VAR ev              : MonitorEvent;
         mon             : MonitorGate;
         name            : ARRAY OF CHAR);
  VAR
    oldDisable           : InterruptMask;
  BEGIN (* InitEvent *)
    NEW(ev);
    WITH ev^ DO
      evMon := mon;
      NewQueue(waiting);
      SetKernelName(waiting^.name, name);
    END;

    (* Debug setup *)
    oldDisable := Disable();
    ev^.next := mon^.events;
    mon^.events := ev;
    Reenable(oldDisable);
  END InitEvent;

BEGIN (* Monitors *)
  Initialized := FALSE;
  monitorList := NIL;
END Monitors.
```

9. Kernel

The Kernel module itself contains mainly primitives for process creation and destruction, and for time handling. An interesting feature is that the internal tick time is determined automatically based on the speed of the hardware. All time specifications are given in milliseconds, and as such independent of the internal tick time.

The main time-handling primitive is `WaitUntil`, which waits until a specified time into the future, rather than the possibly more common Delay-for-x-milliseconds. The reason is that in e.g. a regulator implementation we want to maintain a fixed sampling rate even if the computation time is a considerable, and possibly varying part of the sampling interval. Another possible way to achieve this goal would be to have a primitive for periodic rescheduling, but it has not yet been implemented.

Kernel Definition Module

```
DEFINITION MODULE Kernel;
```

A Real Time Kernel.

```
IMPORT KernelTypes;

TYPE
  Time = KernelTypes.Time;

CONST
  MaxPriority = MAX(CARDINAL);

PROCEDURE Init;
```

Initializes the kernel and makes a process of the main program.

```
PROCEDURE CreateProcess(processa: PROC; memReq: CARDINAL;
                        name: ARRAY OF CHAR);
```

Makes a process of the procedure `processa`. `memReq` is the number of bytes needed for local variables, stack and heap. Typical numbers are in the range 1000..10000. `name` is the name of the process for debugging purposes.

```
PROCEDURE Terminate;
```

Terminates the calling process.

```
PROCEDURE SetPriority(priority: CARDINAL);
```

The priority of the calling process is set to `priority`. High numbers mean low priority. Use numbers in the range 10..1000. Numbers higher than 1000 will cause an error halt. Numbers less than 10 may conflict with predefined internal priorities.

```
PROCEDURE Tick(): CARDINAL;
```

A suitable tick interval is automatically determined based on the speed of the machine we run on. Returns this tick time, in milliseconds.

```
PROCEDURE CurrentTime(VAR t: Time);
```

Returns current time.

```
PROCEDURE IncTime(VAR t : Time; c: CARDINAL);
```

Increments the value of `t` with `c` milliseconds.

```
PROCEDURE CompareTime(VAR t1, t2 : Time): INTEGER;
```

This procedure compares two time-variables. Returns -1 if `t1 < t2`. Returns 0 if `t1 = t2`. Returns +1 if `t1 > t2`. The `VAR`-declaration is for efficiency only; the actual parameters are not touched.

```
PROCEDURE WaitUntil(t: Time);
```

Delays the calling process until the system time >= `t`.

```
PROCEDURE WaitTime(t: CARDINAL);
```

Delays the calling process for `t` milliseconds.

```
END Kernel.
```

Kernel Implementation Module

The implementation module for the kernel is fairly straightforward. It contains very little data of its own, since most of that is in the module `KernelTypes`.

```
IMPLEMENTATION MODULE Kernel;

FROM SYSTEM IMPORT ADDRESS, NEWPROCESS;
FROM Coroutines IMPORT Disable, Reenable, InterruptMask;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;
FROM Console IMPORT Trap;
IMPORT Nucleus;
FROM Nucleus IMPORT
   Running, ReadyQueue, TimeQueue, Schedule, Now;
IMPORT KernelTypes;
FROM KernelTypes IMPORT
   ProcessRef, Queue, InitProcessRec, SetKernelName,
   NewQueue, MovePriority, MoveTime;
IMPORT FindTick;

CONST
  IdleArea = 1000;

VAR
  Initialized    : BOOLEAN;
  NProc : INTEGER;
  Terminated : Queue;

PROCEDURE CreateProcess(processa: PROC; memReq: CARDINAL;
                        name: ARRAY OF CHAR);
VAR
  child : ProcessRef;
  Addr : ADDRESS;
BEGIN
  IF NOT Initialized THEN
    Init;
  END;
  NEW(child);
  InitProcessRec(child);
  NProc := NProc + 1;
  WITH child^ DO
    head.priority := 1;
    assignedPriority := 1;
    processNr := NProc;
    SetKernelName(head.name, name);
(*    GetName(ProcessName); *)
    stackSize := memReq;
    ALLOCATE(stack, stackSize);
    NEWPROCESS(processa, stack, stackSize, procv);
  END;
  MovePriority(child, ReadyQueue);
  Schedule;
END CreateProcess;

PROCEDURE Terminate;
BEGIN
  MovePriority(Running, Terminated);
  Schedule;
  Trap('Kernel -- Terminated process reincarnated');
END Terminate;

PROCEDURE SetPriority(priority : CARDINAL);
BEGIN
  Running^.assignedPriority := priority;
  IF priority < Running^.head.priority THEN
    Running^.head.priority := priority;
  ELSIF ADDRESS(Running^.runningIn) = NIL THEN
```

```
    Running^.head.priority := priority;
    MovePriority(Running, ReadyQueue);
    Schedule;
  END (* IF *);
END SetPriority;

PROCEDURE Tick(): CARDINAL;
BEGIN
  RETURN FindTick.Tick;
END Tick;

PROCEDURE CurrentTime(VAR t: Time);
VAR
  oldDisable : InterruptMask;
BEGIN
  oldDisable := Disable();
  t := Now;
  Reenable(oldDisable);
END CurrentTime;

PROCEDURE IncTime(VAR t : Time; c: CARDINAL);
BEGIN
  KernelTypes.IncTime(t,c);
END IncTime;

PROCEDURE CompareTime(VAR t1, t2 : Time): INTEGER;
BEGIN
  RETURN KernelTypes.CompareTime(t1, t2);
END CompareTime;

PROCEDURE WaitUntil(t: Time);
BEGIN
  Running^.head.nextTime:=t;
  MoveTime(Running,TimeQueue);
  Schedule;
END WaitUntil;

PROCEDURE WaitTime(t: CARDINAL);
VAR
  next : Time;
BEGIN
  CurrentTime(next);
  IncTime(next,t);
  WaitUntil(next);
END WaitTime;

PROCEDURE Idle;
VAR
  P : ProcessRef;
  Q             : Queue;

BEGIN
  SetPriority(MaxPriority - 1);
  NewQueue(Q);

  WHILE TRUE DO
    (* Check for terminated processes *)
    IF ProcessRef(Terminated) <> Terminated^.succ THEN
      P := Terminated^.succ;
      MovePriority(P, Q);
      Q^.succ := ProcessRef(Q);
      Q^.pred := ProcessRef(Q);
```

```
    ELSE
      P := NIL;
    END;
    IF P <> NIL THEN
      DEALLOCATE(P^.stack, P^.stackSize);
      DISPOSE(P);
    END (* IF *);
  END;
END Idle;

PROCEDURE Init;
BEGIN
  IF NOT Initialized THEN
    Initialized := TRUE;
    Nucleus.Init;
    NewQueue(Terminated);
    CreateProcess(Idle, IdleArea, 'Idle');
  END (* IF *);
END Init;

BEGIN (* Kernel *)
  Initialized := FALSE;
  NProc := 1;
END Kernel.
```

10. KernelTypes Implementation

The implementation module for KernelTypes is mostly an exercise in programming of doubly linked lists, and thus fairly repetitive in nature. Since the queues are the basis for the preemptive scheduling and also for the mutual exclusion, all queue handling is done with interrupts disabled.

There are also a couple of routines for time handling, but they are here because time handling is among the machine dependent primitives and they should be collected in as few places as possible.

```
IMPLEMENTATION MODULE KernelTypes;

FROM Coroutines IMPORT Disable, Reenable, InterruptMask;
FROM Storage IMPORT ALLOCATE;

CONST TimeSlice = 1000;

  PROCEDURE InitQueueRec
    (VAR q   : Queue);
  BEGIN InitQueueRec
    WITH q^ DO
      succ := ProcessRef(q);
      pred := ProcessRef(q);
      priority := 0;
      nextTime.hi := 0;
      nextTime.lo := 0;
      priorityQueue := NIL;
      timeQueue := NIL;
    END (* WITH *);
  END InitQueueRec;

  PROCEDURE InitProcessRec
    (VAR r   : ProcessRef);
```

```
  BEGIN
    WITH r^ DO
      head.succ := r;
      head.pred := r;
      head.priority := 0;
      head.nextTime.hi := 0;
      head.nextTime.lo := 0;
      head.name := '';
      head.priorityQueue := NIL;
      head.timeQueue := NIL;

      (* Nucleus *)
      procv := NIL;
      timer := 0;

      (* Kernel *)
      processNr := 0;
      assignedPriority := 0;
      stack := NIL;
      stackSize := 0;

      (* Monitors *)
      runningIn := NIL;
      blockedBy := NIL;
    END WITH ;
  END InitProcessRec;

PROCEDURE SetKernelName(
  VAR kn: KernelName;
      name: ARRAY OF CHAR);
VAR i, ll: CARDINAL; c: CHAR;
BEGIN
  ll := HIGH(name);
  IF NameLen < ll THEN ll := NameLen; END;
  i := 0;
  LOOP
    c := name[i];
    kn[i] := c;
    IF (c = 0C) OR (i >= ll) THEN EXIT END;
    INC(i);
  END;
END SetKernelName;

PROCEDURE NewQueue(VAR q: Queue);
BEGIN
  NEW(q);
  InitQueueRec(q);
  q^.priority := MAX(CARDINAL);
  q^.nextTime.lo := MAX(CARDINAL);
  q^.nextTime.hi := MAX(CARDINAL);
END NewQueue;

PROCEDURE MovePriority(P : ProcessRef; q : Queue);
  Removes P from its queue and then inserts processrecord P in queue Q according to
  priority.

VAR
  oldDisable     : InterruptMask;
  r              : ProcessRef;
  Pri            : CARDINAL;
BEGIN
  IF q <> NIL THEN
    oldDisable := Disable();
```

```
    (* Remove P from old queue *)
    P^.head.pred^.head.succ := P^.head.succ;
    P^.head.succ^.head.pred := P^.head.pred;

    (* Find P's place in the new queue *)
    Pri := P^.head.priority;
    r := q^.succ;
    WHILE Pri >= r^.head.priority DO
      r := r^.head.succ;
    END;

    (* Insert P in the new queue *)
    P^.head.succ := r;
    P^.head.pred := r^.head.pred;
    r^.head.pred^.head.succ := P;
    r^.head.pred := P;
    P^.head.priorityQueue := q;
    P^.head.timeQueue := NIL;

    (* Give P maximum time *)
    P^.timer := TimeSlice;

    Reenable(oldDisable);
  END IF ;
END MovePriority;

PROCEDURE MoveTime(P : ProcessRef; q : Queue);
  Removes P from its queue and then inserts processrecord P in queue Q in time order.
VAR
  oldDisable     : InterruptMask;
  r              : ProcessRef;
  T              : Time;

BEGIN MoveTime
  oldDisable := Disable();

  Remove P from old queue
  P^.head.pred^.head.succ := P^.head.succ;
  P^.head.succ^.head.pred := P^.head.pred;

  Find P's place in the new queue
  T := P^.head.nextTime;
  r := q^.succ;
  WHILE CompareTime(T, r^.head.nextTime) >= 0 DO
    r := r^.head.succ;
  END;

  Insert P in the new queue
  P^.head.succ := r;
  P^.head.pred := r^.head.pred;
  r^.head.pred^.head.succ := P;
  r^.head.pred := P;
  P^.head.priorityQueue := NIL;
  P^.head.timeQueue := q;

  Give P maximum time
  P^.timer := TimeSlice;

  Reenable(oldDisable);
END MoveTime;

PROCEDURE CompareTime(VAR t1, t2: Time): INTEGER;
BEGIN
```

```
  IF t1.hi < t2.hi THEN RETURN -1
  ELSIF t1.hi = t2.hi THEN
    IF t1.lo < t2.lo THEN RETURN -1
    ELSIF t1.lo = t2.lo THEN RETURN 0
    ELSE
      RETURN 1
    END;
  ELSE
    RETURN 1
  END;
END CompareTime;

PROCEDURE IncTime(VAR t: Time; c: CARDINAL);
VAR  P: CARDINAL;
BEGIN
  WITH t DO
    P:=MAX(CARDINAL) - lo;
    IF P >= c THEN
      INC(lo,c);
    ELSE
      lo:= c - P - 1;
      INC(hi);
    END;
  END;
END IncTime;

END KernelTypes.
```

11. FindTick—Finding a Suitable Tick Time

The Kernel described in this report runs on machines of very different speeds. A quick measurement indicated a speed ratio of 30 between the slowest and the fastest machine. It is therefore reasonable to have different basic tick times for the different machines, but we don't want to force the operator to enter it manually every time the kernel starts. The module FindTick performs some typical floating point calculations in a loop, and based on the time for this the Kernel tick time is determined. FindTick is run only once, when the kernel is started, when the program is still in DOS mode, and the only item exported is the cardinal variable tick.

```
DEFINITION MODULE FindTick;
```
  This module performs some computation and times them to find a suitable tick time for the machine we run on.

```
VAR Tick: CARDINAL;

END FindTick.

IMPLEMENTATION MODULE FindTick;
FROM SYSTEM IMPORT DOSCALL;

CONST MAXTICK = 100.0; MINTICK = 1.0;
```
  The maximum and minimum tick times in milliseconds

```
PROCEDURE GetTime(VAR hour, minute, second, csec: CARDINAL);
```
  Returns the DOS calendar time in hours, minutes, seconds and centiseconds

```
VAR hourminute,seccsec: CARDINAL;
BEGIN
  DOSCALL(2CH, hourminute, seccsec);
  hour:=hourminute DIV 256;
  minute := hourminute MOD 256;
```

```
  second := seccsec DIV 256;
  csec := seccsec MOD 256;
END GetTime;

PROCEDURE Compute(turns: CARDINAL);
  The inner computing loops. Performs some simple multiplications and additions.
VAR
  res,x,r,s,t,u: REAL;
  i,j: CARDINAL;
BEGIN
  r:=6.37; s:= -8.93; t:=24.17; u:=3.48;
  res:=r*s+t*u;
  FOR i:=1 TO turns DO
    FOR j:=1 TO 25 DO
      x:=r*s+t*u;
      IF ABS(1.0 - x/res) > 0.0001 THEN
        HALT;
      END;
    END;
  END;
END Compute;

PROCEDURE TimeIt(turns: CARDINAL): CARDINAL;
  Returns the time in milliseconds for Compute.
VAR h1, h2, m1, m2, s1, s2, cs1, cs2: CARDINAL;
BEGIN
  GetTime(h1, m1, s1, cs1);
  Compute(turns);
  GetTime(h2, m2, s2, cs2);
  m2 := m2 + 60*(h2 - h1);
  s2:=s2 + 60*(m2 - m1);
  cs2 := cs2 + 100*(s2 - s1) - cs1;
  RETURN 10*cs2;
END TimeIt;

CONST
  factor = 2.1544347; third root of 10
  span = 1.46780; sixth root of 10

VAR
  turns, time, exponent: CARDINAL;
  rtime, rtick, magnitude: REAL;

BEGIN
  turns:=100;
  LOOP
    time := TimeIt(turns);
    IF (turns >= 64000) OR (time >= 500) THEN EXIT END;
    turns := 2*turns;
  END;
  rtime := FLOAT(time)/FLOAT(turns);

  rtick:=MAXTICK; magnitude := MAXTICK; exponent:=0;
  LOOP
    IF (rtick * span > rtime) AND
       (rtick/span < rtime) THEN
      EXIT;
    END;
    rtick := rtick/factor;
    IF exponent MOD 3 = 0 THEN magnitude := magnitude/10.0; END;
    INC(exponent);
```

```
    IF rtick < MINTICK*span THEN EXIT END;
  END;
  Tick:=TRUNC(FLOAT(TRUNC(rtick/magnitude+0.5))*magnitude);
END FindTick.
```

12. Nucleus Implementation

The Nucleus is the other machine-dependent module of the Real Time Kernel Package. It contains code for three different functions: the transfer of control between processes, the clock interrupt handling, and general initialization of the entire kernel package.

There is also code in the Nucleus to save and restore the floating point registers. These routines need be called only by Schedule, and therefore they don't have to be exported.

The clock interrupt handler checks the Clock Queue to determine if the first process should be made ready to run, and if so moves it to the Ready Queue and similarly checks the new first entry. The clock queue is always maintained in time order, and therefore only the first entry needs be checked

There is also a procedure variable EveryTick, that gets called by the clock interrupt handler. This variable is intended for low-level periodic tasks, such as the handling of a mouse.

The initialization code creates a process record for the main program and enters it in the Ready Queue. It also starts the real time clock via a call to the ClockInterrupt module.

```
IMPLEMENTATION MODULE Nucleus;

IMPORT SYSTEM;
FROM SYSTEM IMPORT
  ADDRESS, ADR, TRANSFER, SETREG, GETREG, DS, BX, CODE;
FROM Coroutines IMPORT Disable, Reenable, InterruptMask;
IMPORT Storage;
FROM Storage IMPORT ALLOCATE;
IMPORT KernelTypes;
FROM KernelTypes IMPORT
   ProcessRef, Queue, InitProcessRec, Time, CompareTime,
   NewQueue, MovePriority;
IMPORT FindTick;
IMPORT ClockInterrupts;
IMPORT Console;

CONST
  MaxLevel = 7;
  TimeSlice = 1000;
VAR
  Initialized : BOOLEAN;
  Tick        : CARDINAL;
  EveryTick   : PROC;

(*$R-*) (*$S-*) (*$T-*)


PROCEDURE Schedule;
VAR
  oldRunning : ProcessRef;
  oldDisable      : InterruptMask;
```

```
BEGIN
  oldDisable := Disable();
  IF ReadyQueue^.succ <> Running THEN
    Fsave;
    oldRunning:=Running;
    Running := ReadyQueue^.succ;
    TRANSFER(oldRunning^.procv, Running^.procv);
    Frestore;
  END;
  Reenable(oldDisable);
END Schedule;

PROCEDURE Fsave;
  Saves the floating point registers

VAR a: ADDRESS;
BEGIN
  a:=ADR(Running^.FParea);
  SETREG(DS,a.SEGMENT);
  SETREG(BX,a.OFFSET);
  (* FSAVE [BX] *)  CODE(0DDH,037H);
END Fsave;

PROCEDURE Frestore;
  Restores the floating point registers

VAR a: ADDRESS;
BEGIN
  a:=ADR(Running^.FParea);
  SETREG(DS,a.SEGMENT);
  SETREG(BX,a.OFFSET);
  (* FRSTOR [BX] *)  CODE(0DDH,027H);
END Frestore;

PROCEDURE Clock;
  The clock interrupt routine

VAR  P: ProcessRef;
BEGIN
  KernelTypes.IncTime(Now, Tick);
  EveryTick;
  LOOP
    P:=TimeQueue^.succ;
    IF CompareTime(P^.head.nextTime,Now) <= 0 THEN
      MovePriority(P, ReadyQueue);
    ELSE
      EXIT;
    END (* IF *);
  END (* LOOP *);

  DEC(Running^.timer);
  IF Running^.timer <= 0 THEN
    MovePriority(Running, ReadyQueue);
  END (* IF *);
  Schedule;
END Clock;


PROCEDURE Init;
BEGIN
  IF NOT Initialized THEN
    NEW(Running);
    InitProcessRec(Running);
    WITH Running^ DO
```

```
      assignedPriority := 0;
    (*   Procv := CurrentProcess(); *)
      processNr := 1;
      head.name := "Main";
    END;
    MovePriority(Running, ReadyQueue);
    ClockInterrupts.Init(Clock,FLOAT(Tick));

    Initialized := TRUE;
  END IF ;
END Init;

PROCEDURE SetEveryTick(TP: PROC);
BEGIN
  EveryTick := TP;
END SetEveryTick;

PROCEDURE Dummy;
BEGIN
END Dummy;

BEGIN
  Initialized := FALSE;
  Now.hi := 0; Now.lo := 0;
  Tick := FindTick.Tick;
  NewQueue(ReadyQueue);
  NewQueue(TimeQueue);
  EveryTick := Dummy;
END Nucleus.
```

13. Clock interrupt driver

This module is on the lowest level of the Kernel package. Its purpose is to intercept the hardware clock interrupts and connect them with the clock routine in the Nucleus. The inner working of the module is described inside the implementation module.

```
DEFINITION MODULE ClockInterrupts;
```

Low level clock interrupt driver.

```
PROCEDURE Init(P: PROC; tick: REAL);
```

Initialization procedure.

`P` the procedure to be called on each clock interrupt.

`tick` the clock interrupt period expressed in ms.

```
END  ClockInterrupts.
```

```
IMPLEMENTATION MODULE ClockInterrupts;
```

The module ClockInterrupts uses the system clock of the computer to give interrupts regularly. The system clock normally interrupts ca. 18 times/second (2^{64} times/hour). The hardware clock registers may be changed to interrupt at a higher rate, which is utilized here. Furthermore, the clock interrupt vector is changed so that a procedure in this module handles the interrupt. In order to maintain the system software clock on time the interrupt routine maintains a counter so that the standard interrupt routine may be called with the correct frequency. In order to call the standard interrupt routine, the original interrupt vector must be copied to an auxiliary software vector. An arbitrary choice of vector 229 has been made. If conflicts should arise, this number appears in one and only one place, in the CONST section below.

```
FROM SYSTEM IMPORT CODE, ADDRESS, OUTBYTE, DISABLE, ENABLE;
FROM Devices IMPORT SaveInterruptVector, RestoreInterruptVector;
FROM RTSMain IMPORT InstallTermProc;
FROM FloatingUtilities IMPORT Round;

CONST
  SavedClockVector = 229;   Auxiliary software interrupt vector
  BaseFrequency = 1193.18;  Frequency driving the counter/timer
  TCC = 043H;               Timer/counter control word
  TC0 = 040H;               Timer 0
  ClockMode = 036H;         Clock Mode 3, 16 bits, binary

VAR
  period: CARDINAL;         The value to set in the hardware counter/timer. Also used to
                            determine when to call the system clock interrupt routine.
                            Set once by Init procedure.
  timer: CARDINAL;          The counter for calling the system clock interrupt routine.
  ClockProcedure: PROC;     The procedure to call on each clock interrupt.

(*$O+*)(*$R-*)(*$S-*)(*$T-*)
PROCEDURE ClockInterrupt;
```

This is the Clock Interrupt Service Routine. Is job is to save the registers and call the higher level clock interrupt handler. It also maintains a counter so that the original Interrupt Service Routine is called at approximately the correct interval.

```
BEGIN
(*      PUSH AX *) CODE(050H);
(*      PUSH CX *) CODE(051H);
(*      PUSH DX *) CODE(052H);
(*      PUSH BX *) CODE(053H);
(*      PUSH SI *) CODE(056H);
(*      PUSH DI *) CODE(057H);
(*      PUSH DS *) CODE(01EH);
(*      PUSH ES *) CODE(006H);
```

At this point all registers are saved. The purpose of the next statement is to increment the counter, but also to set the Carry flag if the increment overflows. The carry is then tested in the next CODE-statement. This is ugly programming, but it works provided there is only MOV-instructions after the ADD-instruction in the Modula-statement. This should be checked with each new version.

```
       timer:=timer+period;
(*     JNC L1 *)                CODE(073H, 004H);
(*     INT SavedClockVector *) CODE(0CDH, SavedClockVector);
(*     JMP L2 *)                CODE(0EBH, 004H);
(* L1: SENDEOI *)               CODE(0B0H, 020H, 0E6H, 020H);
(* L2:         *)
```

All interrupt administration is done. Call the higher level interrupt routine and restore the registers.

```
       ClockProcedure;
(*     POP ES *) CODE(007H);
(*     POP DS *) CODE(01FH);
(*     POP DI *) CODE(05FH);
(*     POP SI *) CODE(05EH);
(*     POP BX *) CODE(05BH);
(*     POP DX *) CODE(05AH);
(*     POP CX *) CODE(059H);
(*     POP AX *) CODE(058H);
(*     LEAVE  *) CODE(0C9H);
(*     IRET   *) CODE(0CFH);
END ClockInterrupt;
```

```
PROCEDURE Init(P: PROC; tick: REAL);
VAR IV: ADDRESS; phigh, plow: CARDINAL;
BEGIN
  InstallTermProc(Stop);
  ClockProcedure:=P;
    Compute the number of clock cycles between each interrupt. We need it in high-
    byte/low-byte form.
  period:=Round(tick * BaseFrequency);
  plow:=period MOD 256;
  phigh := period DIV 256;
    Save the original clock interrupt vector and set the vector to the ClockInterrupt
    procedure of this module. The rest of the initialization is done with interrupts off.
  DISABLE;
  SaveInterruptVector(8,IV);
  RestoreInterruptVector(SavedClockVector,IV);
  RestoreInterruptVector(8,ADDRESS(ClockInterrupt));
    We reprogram the system timer/counter to give interrupts with the rate determined
    by tick. The reason for the do-nothing Delay procedure is that things may
    malfunction if two OUT-instructions are placed too close to each other.
  OUTBYTE(TCC,ClockMode); Delay;
  OUTBYTE(TC0,plow); Delay;
  OUTBYTE(TC0,phigh); Delay;
  ENABLE;
END Init;

PROCEDURE Stop;
VAR IV: ADDRESS;
BEGIN
  DISABLE;
    Reset the clock interrupt vector
  SaveInterruptVector(SavedClockVector,IV);
  RestoreInterruptVector(8,IV);
    Reset the system timer/counter to its normal value of 18 interrupts per second.
  OUTBYTE(TCC,ClockMode); Delay;
  OUTBYTE(TC0,0); Delay;
  OUTBYTE(TC0,0); Delay;
  ENABLE;
END Stop;

PROCEDURE Delay;
  Does nothing
BEGIN
END Delay;

END ClockInterrupts.
```

14. Keyboard Interrupt Module

The main purpose of the Keyboard Interrupt Module is to act as an administrator. The Keyboard interrupts once for each key press and once for each key release. There is an interrupt routine inside the BIOS of the computer that normally handles all these interrupts, decodes the key actions and makes the actual characters available. The purpose of the interrupt handler in this module is to immediately call the standard BIOS interrupt routine, and then on return determine if there is really a character available. If so we call a user supplied Echo procedure to handle the echo, collect characters into line buffers etc.

```
DEFINITION MODULE KBint;
```
  Keyboard interrupt handler module.

```
TYPE EchoProc=PROCEDURE(CHAR);
PROCEDURE Init(ep: EchoProc);
```
  Initialization procedure. The argument ep is the procedure to be called for each keyboard event that means a keyboard character is available. The procedure should handle the echo.

```
END KBint.


IMPLEMENTATION MODULE KBint;
FROM SYSTEM IMPORT CODE,SETREG,GETREG,SWI,AX,ADR,ADDRESS;
FROM Devices IMPORT SaveInterruptVector, RestoreInterruptVector;
FROM RTSMain IMPORT InstallTermProc;
FROM Kernel IMPORT SetPriority, CreateProcess;
FROM Semaphores IMPORT
   Semaphore, InitSem, Wait, Signal;

CONST
  KeyboardInterrupt=9;
  MovedKeyboardInterrupt=0E6H;
VAR
  vector: ADDRESS;
  Echo: EchoProc;
  kbsem: Semaphore;

(*$R-*) (*$S-*) (*$T-*)

PROCEDURE KBintProc;
```
  This is the Interrupt Driver. The basic principle is that for each interrupt we immediately call the normal BIOS interrupt driver to let it do its job. The keyboard makes an interrupt for each key press and each key release, and only some of them mean that a character is available. We therefore check on return from the BIOS if a character really is available.
```
BEGIN
```
  Save some registers.
```
  (*  PUSHA      *)    CODE(060H);
```
  Let the BIOS interrupt handler do its job.
```
      SWI(MovedKeyboardInterrupt);
```
  If there is no character, then exit
```
      SETREG(AX,100H);
      SWI(16H);
  (*  JZ EXIT *)     CODE(074H, 07H);
```
  else save some more registers and signal the handler process
```
  (*  PUSH ES *)  CODE(06H);
  (*  PUSH DS *)  CODE(1EH);
      KBProc;
```
  Restore registers and return.
```
  (*  POP DS  *)     CODE(1FH);
  (*  POP ES  *)     CODE(07H);
  (*EXIT: POPA     *)     CODE(061H);
  (*  LEAVE   *)     CODE(0C9H);
  (*  IRET    *)     CODE(0CFH)
END KBintProc;

PROCEDURE KBProc;
BEGIN
  Signal(kbsem);
END KBProc;

PROCEDURE KeyboardHandler;
```

This is the keyboard process. It has high priority, but spends almost all its time waiting for the semaphore signalled by the interrutpt driver. It then calls the procedure variable Echo and waits again.

```
VAR c: CHAR;
BEGIN
  SetPriority(2);
  LOOP
    Wait(kbsem);
    SETREG(AX,0);
    SWI(16H);
    GETREG(AX,c);
    Echo(c);
  END;
END KeyboardHandler;

PROCEDURE Init(ep: EchoProc);
BEGIN
  InstallTermProc(Stop);
  InitSem(kbsem,0,'kbint.kbsem');
  CreateProcess(KeyboardHandler,1000,'keyboardhandler');
  SaveInterruptVector(KeyboardInterrupt,vector);
  RestoreInterruptVector(MovedKeyboardInterrupt,vector);
  RestoreInterruptVector(KeyboardInterrupt,ADDRESS(KBintProc));
  Echo:=ep;
END Init;

PROCEDURE Stop;
```

This is the Termination procedure, i.e. it gets called when the program terrminates. See the documentation for `Devices.InstallTermProc`. The calls to `PutChar` are just debug printouts still left in.

```
BEGIN
  PutChar('A'); PutChar('B'); PutChar('C');
  RestoreInterruptVector(KeyboardInterrupt,vector);
  PutChar('D'); PutChar('E'); PutChar('F');
END Stop;

END KBint.
```