LUND UNIVERSITY

A Real-Time Kernel with Graphics Support Modules

Nielsen, Lars; Andersson, Leif; Andersson, Mats; Årzén, Karl-Erik

1993

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

Link to publication

Citation for published version (APA):
Nielsen, L., Andersson, L., Andersson, M., & Årzén, K.-E. (1993). *A Real-Time Kernel with Graphics Support Modules*. (Technical Reports TFRT-7510). Department of Automatic Control, Lund Institute of Technology (LTH).

Total number of authors:
4

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

ISSN 0280–5316
ISRN: LUTFD2/TFRT--7510--SE

A real-time kernel with graphics support modules

Lars Nielsen, Leif Andersson,
Mats Andersson, and Karl-Erik Årzén

Department of Automatic Control
Lund Institute of Technology
August 1993

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

Document name
Report

Date of issue
August 1993

Document Number
ISRN LUTFD2/TFRT--7510--SE

Author(s)
Lars Nielsen, Leif Andersson, Mats Andersson, and Karl-Erik Årzén

Supervisor

Sponsoring organisation

Title and subtitle
A real-time kernel with graphics support modules

Abstract

This text presents the real-time kernel and the real-time graphics support modules used in the course Real-Time Systems. The majority of the text consists of commented Modula-2 definition modules.

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
53

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Fax +46 46 110019, Telex: 33248 lubbis lund.

Preface

This text presents the real-time kernel and the real-time graphics support modules used in the course Real-Time Systems. The majority of the text consists of commented Modula-2 definition modules. In earlier versions of the course this material has been a part of the course text "Computer Implementation of Control Systems" written by Lars Nielsen.

Contents

1

The Real-Time Kernel Layer

L. Nielsen and L. Andersson

GOAL: To point out some design considerations when implementing and using a real-time kernel.

A general knowledge of Modula-2 and of principles for a real-time kernel layer is assumed. Our specific solution will therefore be briefly presented and commented. The modules presented here also include modules not directly related to real-time, but we have found it conceptually natural to group them in this basic layer of software.

1.1 Function and Implementation of the Modules

The function and implementation of the kernel is very similar to the version presented in the basic course in real-time programming. The kernel uses pre-emptive scheduling, and there is one single ready queue, where all processes ready to run are sorted in priority order. Other queues are associated with semaphores, events, and so on, where processes are waiting. Calls to the primitives, such as `Wait` result in that the process record may be moved from one queue to another. The processes are always sorted on insertion in a queue. The scheduler transfers the first process in the ready queue to running.

Some comments on the use of the primitives

One should distinguish between primitives that implement a concept and primitives that can be used to implement a concept. An example of the first type is the `Semaphores` module, which really implements the semaphore concept. An example of the second type is the module `Monitors`. The monitor concept can be implemented by programming discipline by having a call to `EnterMonitor` first in each monitor procedure and to `LeaveMonitor` last, and by using the `MonitorEvent` to implement conditional critical regions. Note that `EnterMonitor` and `LeaveMonitor` operate on a `MonitorGate`, which is conceptually a semaphore, but it has the added feature that the process occupying the monitor is raised to the priority of the highest of the processes waiting to get in.

Graphics

The module `Graphics` gives an example of the central ideas in real-time graphics. The basic data structure is `VirtualScreen` (accessed via a variable of type `handle`). A virtual screen consist of two objects; a `Window` and a `ViewPort`, both of type `rectangle`. A window is a rectangle where the x- and y-axis represent user variables e.g. physical quantities, whereas a viewport is a rectangle where the x- and y-axis represent coordinates on the computer screen. The graphics system automatically transforms coordinates between the window and the viewport coordinate systems in a virtual screen once they have been defined. The key idea is thus that the user of the module only has to think in user coordinates in the window. Read and write commands are done in window coordinates and the system automatically transforms to viewport coordinates so that the result appears on the computer screen.

Modula-2

Three short comments will be made about developing software in general in Modula-2. A module can be regarded as a language concept, not just as a fix used for separate compilation. Sometimes it may be advantageous to use internal modules in other modules to structure the code. This trend may be compared with the history of the procedure concept. In early Fortran a procedure was a separate compilation unit, but in languages following thereafter like Algol or Pascal, procedures were used in the same compilation unit, even nested etc. The second and third comments are about hiding or leaving implementation details open. The technique to hide information is to use hidden types. See the declarations in `Graphics`, `Monitors`, and `Semaphores` for examples. This means that the internal structure of the these objects are inaccessible to a user of the modules, and that the only way to operate on the objects are via the routines declared in the definition modules. A dual to hiding details is to leave them out of a module. The technique to do this in Modula-2 is to use procedure-type parameters. The general idea is to provide more universal units than if all details were included in the module. One example is hardware dependent procedures in the kernel itself. The machine dependent clock procedure is installed using a procedure-type parameter, and the kernel can be kept clean and easier to port to other computers. Another example, on a higher level, is to write a complete controller framework except for the control algorithm. The control algorithm is then installed by the user without having to change anything else in the system. This idea is extended further in Chapter 6.

Modula-2 provides coroutines and primitives like `TRANSFER` and `IOTRANSFER` to handle concurrent activities. In other languages, like for example Pascal, C, or C++, similar primitives have to be implemented. An earlier version the present kernel was done in Pascal for the LSI-11 computer. In that case, the necessary basic nucleus software for interrupt disabling/enabling, and procedures analogous to `TRANSFER` and `IOTRANSFER` consisted of four pages of assembler code. The fact that Modula-2 already contains such primitives is thus not crucial, since the work to extend other languages with similar capabilities is not overwhelming.

Portability

The real-time kernel modules are one example of a software layer. When designing such a layer one should try to find units or layers that have a long life time. The present kernel was first developed for an LSI-11 computer. It was later transferred to an IBM PC, and has recently been ported to a Sun-VME system. It has also been used on other machines outside the Department. The efforts to transfer the kernel to new machines have been limited. The major part is written in a high level language, and the machine dependent

parts can be well isolated e.g. by the use of procedure parameters as described above. The present kernel has thus survived three hardware generations.

Implementation details

Processes are declared as procedures. Such procedures should not be declared inside other procedures. Processes never terminates, so the last `END` of the procedure should never be reached. There is a minor difference between the IBM PC version and the Sun-VME version in that LONGREAL is used instead of REAL on Sun-VME.

1.2 Research topics

A current trend in the research community is to study what is called hard real-time problems. One example will be presented to give a flavor of that field. Consider the well known dining philosophers problem, which is an idealized problem in scheduling. A solution is feasible if every philosopher eventually will eat. The hard real-time version of this problem is called the dying dining philosophers problem. The new element is to consider time, and to say that a philosopher dies if he is not able to eat within certain time limits. Two main versions of the problem are if there is a waiter or not, i.e. to consider centralized or decentralized scheduling. There are also other extensions treating also two bottles of soy sauce, one curry, and so on. The research field is new and there seems to be few practical results so far, but the questions asked are relevant and one should be aware of this type of work.

1.3 The definition modules

The definition modules are listed on the following pages. One should also remember that the Modula-2 system is delivered with a number of modules e.g. mathlib for numerical functions.

The modules common both to the IBM PC system and to the Sun-VME system are: Console, Conversions, Events, Identifiers, IntConversions, Kernel, LexicalAnalyzer, Messages, Monitors, Semaphores, Strings. Note that LONGREAL is used instead of REAL on Sun-VME.

Modules specific to IBM PC are: AnalogIO, Graphics, RTGraph, RTMouse. Overlapping windows are not supported. The user must check the graphical layout.

Modules specific to Sun-VME are: AnalogIO, DigitalIO, MiscIO, MatComm.

```
DEFINITION MODULE Console;
```
Simple terminal input and output.

```
PROCEDURE GetChar(VAR ch : CHAR);
```
Reads one character from the Console.

```
PROCEDURE GetString(VAR s: ARRAY OF CHAR);
```
Reads a string, up to carriage return or line feed.

```
PROCEDURE CharAvailable(): BOOLEAN;
```
Returns `TRUE` IF a character is available, `FALSE` otherwise.

```
PROCEDURE PutChar(ch : CHAR);
```
Writes one character to the Console.

```
PROCEDURE PutString(s : ARRAY OF CHAR);
```
Writes a string to the Console.

```
PROCEDURE PutLn;
```
Writes a newline to the Console.

```
PROCEDURE Trap(errortext: ARRAY OF CHAR);
```
Writes a string to the Console and halts.

```
END Console.
```

```
DEFINITION MODULE Conversions;
```

Converts between numbers and their string representations. The routines for conversion between strings and integers or cardinals also occur separately in the module `IntConversions`. Only one of the modules `IntConversions` and `Conversions` is thus necessary.

```
PROCEDURE IntToString(VAR string: ARRAY OF CHAR;
                      num: INTEGER;
                      width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE CardToString(VAR string: ARRAY OF CHAR;
                       num: CARDINAL;
                       width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE StringToCard(string: ARRAY OF CHAR): CARDINAL;
```

Converts a string representation to a cardinal. Leading spaces are skipped. A leading + is allowed. If no legal cardinal can be found in the string then `MAX(CARDINAL)` is returned.

```
PROCEDURE StringToInt(string: ARRAY OF CHAR): INTEGER;
```

Converts a string representation to an integer. Leading spaces are skipped. A leading + or − is allowed and interpreted. If no legal integer can be found in the string then `-MAX(INTEGER) - 1` is returned.

```
PROCEDURE StringToReal (string: ARRAY OF CHAR): REAL;
```

Decodes a real value from an input string of characters. The syntax is permissive in the sense that the string '6' is interpreted as 6.0. Leading whitespace is permitted in the string. If no acceptable real number can be found then the value `badreal` (see below) is returned.

```
PROCEDURE RealToString(VAR string: ARRAY OF CHAR;
                       num: REAL;
      width: CARDINAL);
```

Converts a real number to a fixed point or exponent representation with `width` characters. The number is converted such that maximum accuracy is obtained. If there is enough space then a suitable fixed point representation is used. If there is not enough space then an exponent representation is used. If `width > HIGH(s)+1` then s is asterisk filled. If an exponent representation must be used and `width` is too small then the field is asterisk filled.

```
CONST badreal = 10.0E+307;

END Conversions.
```

```
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.
```

```
DEFINITION MODULE Identifiers;
```
Module to decode identifiers.

```
TYPE identset;

PROCEDURE NewIdentSet(VAR id: identset);
```
Initializes an ident set and returns a reference to it.

```
PROCEDURE BuildIdentSet(id: identset; name: ARRAY OF CHAR;
                        key: CARDINAL);
```
Inserts an identifier in an ident set and assigns a key to it.

id The ident set to be used.

name The identifier.

key The key to be associated with the identifier **name**. The value of **key** should be `[1..255]` if `SearchIdentSet` will be used and `[2..255]` if SearchIdentSetAbbrev will be used. See these procedures.

```
PROCEDURE SearchIdentSet(id: identset;
                         name: ARRAY OF CHAR): CARDINAL;
```
Searches for an identifier and returns its key if it is found and 0 otherwise.

id The ident set to be used.

name The identifier

```
PROCEDURE SearchIdentSetAbbrev(id: identset;
                         name: ARRAY OF CHAR): CARDINAL;
```
Searches for an identifier. Any nonambiguous abbreviation of the identifier is acceptable. If the identifier is found, its key is returned. If the abbreviation is ambiguous then 1 is returned and if the identifier is not found then 0 is returned.

id The ident set to be used.

name The identifier

```
END Identifiers.
```

```
DEFINITION MODULE IntConversions;
```

Conversions between strings and cardinals or integers. The routines in this module are duplicated in the module `Conversions`, that also converts between strings and reals. Only one of the modules `IntConversions` and `Conversions` is thus necessary.

```
PROCEDURE IntToString(VAR string: ARRAY OF CHAR;
                      num: INTEGER;
                      width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE CardToString(VAR string: ARRAY OF CHAR;
                       num: CARDINAL;
                       width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE StringToCard(string: ARRAY OF CHAR): CARDINAL;
```

Converts a string representation to a cardinal. Leading spaces are skipped. A leading + is allowed. If no legal cardinal can be found in the string then `MAX(CARDINAL)` is returned.

```
PROCEDURE StringToInt(string: ARRAY OF CHAR): INTEGER;
```

Converts a string representation to an integer. Leading spaces are skipped. A leading + or − is allowed and interpreted. If no legal integer can be found in the string then `-MAX(INTEGER) - 1` is returned.

```
END IntConversions.
```

```
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 TimeToReal(t: Time): REAL;
```
Returns `t` converted to a real number, expressed in milliseconds.

```
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.
```

`DEFINITION MODULE LexicalAnalyzer;`
The routines in this module are used to decode a string. The function `LexScan` decodes the next item in the string and returns a value indicating the type of the decoded item. A call to one of the procedures `LexCardinal` through `LexString` will then return the decoded value.

`TYPE LexHandle;`
A pointer type defined internally in the LexicalAnalyser.

```
TYPE LexTypes =
  (CardLex, CardIntLex, IntLex, RealLex, IdentLex, DelimLex,
   StringLex, EolnLex, EofLex, ErrorLex, RealErrorLex,
   StringErrorLex);
```
The possible results from `LexScan`. `RealErrorLex` corresponds to real overflow or underflow. `StringError` is given when the end of a string is missing.

`PROCEDURE LexInit(VAR lh: LexHandle);`
Initializes the internal data structures and returns a handle.

`PROCEDURE LexInput(lh: LexHandle; s: ARRAY OF CHAR);`
Makes the string `s` ready to be decoded.

`PROCEDURE LexScan(lh: LexHandle): LexTypes;`
Decodes the next item in the string which is connected with the LexHandle `lh`. The items must obey the following syntax.

⟨number⟩ :== [+|-]{⟨digit⟩}*[.{⟨digit⟩}*][⟨exponent⟩]
⟨exponent⟩ :== e|E[+|-]{⟨digit⟩}*
⟨digit⟩ :== 0| .. |9
⟨identifier⟩ :== ⟨letter⟩ {⟨letter⟩|⟨digit⟩}*
⟨letter⟩ :== a| .. |z|A| .. |Z
⟨string⟩ :== '{⟨character⟩}*'|"{⟨character⟩}*"
⟨character⟩ :== ⟨all characters defined in the ASCII table⟩

`PROCEDURE LexCardinal(lh: LexHandle): CARDINAL;`
Returns the decoded value if the result from `LexScan` is either `CardLex` or `CardIntLex`.

`PROCEDURE LexInteger(lh: LexHandle): INTEGER;`
Returns the decoded value if the result from `LexScan` is either `CardIntLex` or `IntLex`.

`PROCEDURE LexReal(lh: LexHandle): REAL;`
Returns the decoded value if the result from `LexScan` is in `CardLex .. RealLex`.

`PROCEDURE LexIdent(lh: LexHandle; VAR s: ARRAY OF CHAR);`
Returns the identifier in `s` if the result from `LexScan` is `IdentLex`. All the letters (`'a'..'z'`) are converted to (`'A'..'Z'`).

`PROCEDURE LexDelim(lh: LexHandle): CHAR;`
Returns the delimiter if the result from `LexScan` is `DelimLex`.

`PROCEDURE LexString(lh: LexHandle; VAR s: ARRAY OF CHAR);`
Returns the delimiter if the result from `LexScan` is `DelimLex`.

`END LexicalAnalyzer.`

```
DEFINITION MODULE Messages;
```
Message Passing Routines.

```
FROM SYSTEM IMPORT ADDRESS;

TYPE
  MailBox;

PROCEDURE InitMailBox(VAR Box: MailBox; maxmessages: CARDINAL;
                      name: ARRAY OF CHAR);
```
Initializes `Box`. The maximum number of messages that the box can contain is `maxmessages`.

```
PROCEDURE SendMessage(Box: MailBox; VAR MessAdr: ADDRESS);
```
Sends the message referenced by `MessAdr` to `Box`. If the mailbox already contains the maximum number of messages then the calling process will wait. On return `MessAdr = NIL`.

```
PROCEDURE ReceiveMessage(Box: MailBox; VAR MessAdr: ADDRESS);
```
Receives a message from `Box`. The calling process is delayed if `Box` is empty.

```
PROCEDURE AcceptMessage(Box : MailBox; VAR MessAdr : ADDRESS);
```
Receives a message from `Box`. If there is a message in the box then `MessAdr` points to the message. If there is no message in the box then `MessAdr = NIL`. `AcceptMessage` does not delay the calling process.

```
END Messages.
```

```
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.

```
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.
```

```
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!*strut*! > 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.
```

```
DEFINITION MODULE Strings;
```
String handling routines. For all these routines the general principle is that if a target string is too short, then the result is silently truncated. There are no run time errors.

```
PROCEDURE Length(str: ARRAY OF CHAR): CARDINAL;
```
Returns the number of characters in `str`.

```
PROCEDURE Compare(astring, bstring: ARRAY OF CHAR): INTEGER;
```
Compares `astring` and `bstring` Returns `-1`, `0` or `+1` indicating less than, equal or greater than.

```
PROCEDURE Position(pattern, source: ARRAY OF CHAR;
                   start: CARDINAL): CARDINAL;
```
Returns the position of `pattern` within `source`. The comparison starts at position `start`

```
PROCEDURE ReversePosition(pattern, source: ARRAY OF CHAR;
                          last: CARDINAL): CARDINAL;
```
Returns the position of the ¡!it last¿ occurence of `pattern` within `source`. The comparison starts `last` characters from the end and proceeds backwards.

```
PROCEDURE Assign(VAR target: ARRAY OF CHAR;
                 source: ARRAY OF CHAR);
```
Assigns `source` to `target`

```
PROCEDURE Insert(VAR target: ARRAY OF CHAR;
                 string: ARRAY OF CHAR;
                 pos: CARDINAL);
```
Inserts `string` into target at position pos

```
PROCEDURE Substring(VAR dest: ARRAY OF CHAR;
                    source: ARRAY OF CHAR;
                    index, len: CARDINAL);
```
Returns in `dest` a substring from `source` starting at `index` and containing `len` characters.

```
PROCEDURE Append(VAR target: ARRAY OF CHAR;
                 string: ARRAY OF CHAR);
```
Appends `string` to `target`.

```
PROCEDURE AppendC(VAR target: ARRAY OF CHAR; c: CHAR);
```
Appends the character `c` to `target`

```
PROCEDURE Delete(VAR target: ARRAY OF CHAR;
                 index, len: CARDINAL);
```
Deletes `len` characters FROM `target`, starting at position `index`

```
PROCEDURE UpperCase(VAR source : ARRAY OF CHAR);
```
Converts `source` to uppercase letters.

```
PROCEDURE LowerCase(VAR source : ARRAY OF CHAR);
```
Converts `source` to lowercase letters.

```
END Strings.
```

```
DEFINITION MODULE AnalogIO;
```
Analog input/output.

```
PROCEDURE ADIn(Channel : CARDINAL) : REAL;
```
Returns a value in the interval [-1.0..1.0], corresponding to [-10.0 V..10.0 V], from channel number Channel. Allowed channel numbers depend on the hardware, but is at least 0–3.

```
PROCEDURE DAOut(Channel : CARDINAL; Value : REAL);
```
Outputs a value in the interval [-1.0..1.0], corresponding to [-10.0 V..10.0 V], to channel number Channel. Allowed channel numbers depend on the hardware, but is at least 0–1.

```
END AnalogIO.
```

```
DEFINITION MODULE Graphics;
```
Warning! Do not use this module together with the module Terminal.

```
TYPE
  handle;
```
A pointer type defined internally in the graphics system

```
  point = RECORD
    h: REAL;     Horizontal coordinate
    v: REAL;     Vertical coordinate
  END;

  rectangle = RECORD
    CASE BOOLEAN OF
    TRUE:
       loleft: point;                Lower left corner
       upright: point;               Upper right corner
    FALSE: xlo,ylo,xhi,yhi: REAL;    Alternate representation
    END;
  END;

  color=(black, blue,green,cyan,red, magenta, brown, white,
         grey, lightblue, lightgreen, lightcyan, lightred,
         lightmagenta, yellow, intensewhite);

  buttontype=(LeftButton, RightButton);   For the mouse buttons

  buttonset = SET OF buttontype;

PROCEDURE VirtualScreen(VAR h: handle);
```
Initializes the data structures for a virtual screen and returns a handle. The default window and viewport are $0.0 < x < 1.5$ and $0.0 < y < 1.0$. The default color for line and text is white; for fill it is black.

```
PROCEDURE SetWindow(h: handle; r: rectangle);
```
Defines the window coordinates.

`h` The virtual screen handle.

`r` The rectangle specifying the window. All real numbers are permitted as window coordinates.

```
PROCEDURE SetViewPort(h: handle; r: rectangle);
```
Positions the viewport on the screen.

`h` The virtual screen handle.

`r` The rectangle specifying the viewport.

The viewport rectangle must satisfy the screen limits $0.0 < x < 1.5$ and $0.0 < y < 1.0$.

```
PROCEDURE SetLineColor(h: handle; c: color);

PROCEDURE SetTextColor(h: handle; c: color);

PROCEDURE SetFillColor(h: handle; c: color);
```

```
PROCEDURE PolyLine(h: handle; VAR polygon: ARRAY OF point;
                   npoint: INTEGER);
```

Draws a polygon.

`h` The virtual screen handle

`polygon` The points of the polygon. The start point is `polygon[0]`. Lines are drawn using the line color of the specified handle. The VAR declaration is for efficiency only, and the actual argument is not changed.

`npoint` The number of points in polygon.

```
PROCEDURE PolyMarker(h: handle; VAR polygon: ARRAY OF point;
                     npoint: INTEGER);
```

Draws markers at the specified points. The markers are plus signs at present.

`h` The virtual screen handle

`polygon` The points where the markers are drawn. The current line color is used. The VAR declaration is for efficiency only, and the actual argument is not changed.

`npoint` The number of points in polygon.

```
PROCEDURE WriteString(h: handle; p :point; s: ARRAY OF CHAR);
```

Writes a string on the screen starting at a specified point.

`h` The virtual screen handle

`p` The starting point of the text.

`s` The text string to be written. If s contains `CHR(0)` then this is considered the end of the string.

The text is written with the text color of the specified handle. Old text is overwritten, not erased. See `EraseChar`.

```
PROCEDURE FillRectangle(h: handle; r: rectangle);
```

Fills a rectangle on the screen.

`h` The virtual screen handle.

`r` The rectangle to be filled.

The rectangle is filled with the fill color of the specified handle.

```
PROCEDURE DrawRectangle(h: handle; r: rectangle);
```

Draws a rectangle on the screen.

`h` The virtual screen handle.

`r` The rectangle to be drawn.

The rectangle is drawn with the line color of the specified handle.

```
PROCEDURE CharacterSize(h: handle; VAR width, height: REAL);
```

Returns the size of a character in the current window coordinate.

`h` The virtual screen handle.

`width` The horizontal size of a character.

`height` The vertical size, i.e. the distance between the baselines of two adjacent text lines.

```
PROCEDURE ReadString(h: handle; p :point; VAR s: ARRAY OF CHAR);
```

Reads a string from the keyboard with echoing. The procedure returns when the user pushes the Return key.

`h` The virtual screen handle.

`p` The point where echoing starts.

`s` The returned string. If it is less than `HIGH(s)` characters long then the string is delimited by `CHR(0)`.

The characters are echoed with the text color of the specified handle. The text background is the fill color of the specified handle.

```
PROCEDURE InputString(h: handle; p :point; VAR s: ARRAY OF CHAR;
                      limit: CARDINAL; VAR complete: BOOLEAN);
```

Reads a string from the keyboard with echoing. The procedure returns when the user pushes the Return key or when another process calls `StopInputString`

`h` The virtual screen handle.

`p` The point where echoing starts.

`s` The returned string. If it is less than `HIGH(s)` characters long then the string is delimited by `CHR(0)`.

`limit` The maximum number of charaters accepted and echoed.

`complete` Returned TRUE if the user pushed Return, FALSE if another process called `StopInputString`

The characters are echoed with the text color of the specified handle. The text background is the fill color of the specified handle.

```
PROCEDURE StopInputString;
```

Stops current reading by InputString, but makes the `complete` parameter return FALSE.

```
PROCEDURE EraseChar(h: handle; p: point; num: CARDINAL);
```

Erases characters, i.e. fills the area with the fill color.

`h` The virtual screen handle.

`p` The starting point of the erase.

`num` A region corresponding to num characters is filled with the fill color.

```
PROCEDURE GetMouse(h: handle; VAR p:point; VAR b:buttonset);
```

Returns the mouse state.

`h` The virtual screen handle.

`p` The mouse position.

`b` The button state. If `LeftButton IN b` then this button is pressed, and conversely for the right button.

```
PROCEDURE WaitForMouse(h: handle; VAR p: point;
                       VAR b: buttonset);
```

Waits until at least one of the buttons get pushed, then returns the mouse state. In a real time situation it is not a busy wait.

`h` The virtual screen handle.

`p` The mouse position.

`b` The button state. If `LeftButton IN b` then this button is pressed, AND conversely for the right button.

```
PROCEDURE SetMouseRectangle(h: handle; r: rectangle;
                            n: CARDINAL);
```

Inserts a rectangle in a list of rectangles to be tested in a WaitMouseRectangle or GetMouseRectangle operation. There is one list for each handle.

`h` The virtual screen handle.

r The rectangle to be inserted.

n The number to be returned if the mouse is inside r.

If the specified number n already exists in the list then no new entry is made, but the old entry gets a new rectangle value. Do not use the number 0, because 0 has a special meaning for **GetMouseRectangle**. No test is made, however. This routine does not draw anything. The drawing should be done with DrawRectangle.

```
PROCEDURE WaitMouseRectangle(h: handle): CARDINAL;
```

Waits until a mouse button is pressed and the cursor is inside one of the rectangles previously specified with SetMouseRectangle. The number associated with that rectangle is then returned. The entries are kept and tested in number order, and the first match found is returned. The cursor hot spot must be strictly inside the rectangle if it is to be considered a match.

```
PROCEDURE GetMouseRectangle(h: handle): CARDINAL;
```

If the cursor hot spot is strictly inside one of the mouse rectangles, then the corresponding number is returned, otherwise 0 is returned. See SetMouseRectangle and WaitMouseRectangle.

```
PROCEDURE HideCursor;
```

The internal hide/show counter is decremented. If the counter is zero after decrementation then the cursor is removed from the screen.

```
PROCEDURE ShowCursor;
```

The internal hide/show counter is incremented. If the counter is 1 after incrementation then the cursor is shown on the screen.

```
PROCEDURE Shutdown;
```

Closes down the entire graphics system and sets the screen in normal text mode.

The following procedure types and procedures are used to define which modules should handle the mouse and the keyboard. In a real time application the modules RTGraph and RTMouse will call these procedures. They should not be referenced directly by user programs.

```
TYPE
  EchoStringProc=PROCEDURE(VAR ARRAY OF CHAR, VAR BOOLEAN);
  MouseProcedureType=
        PROCEDURE(VAR buttonset, VAR INTEGER, VAR INTEGER);

PROCEDURE SetEchoString(p: EchoStringProc; q:PROC);

PROCEDURE SetMouseProcedures(GM,WM: MouseProcedureType;
                             HC, SC: PROC);

END Graphics.
```

`DEFINITION MODULE RTGraph;`
Establishes connections between the Kernel and the graphic modules.

`PROCEDURE Init;`
Initialization. There is an implicit call to `Kernel.Init`.

`END RTGraph.`

`DEFINITION MODULE RTMouse;`
Establishes connections between mouse, Kernel and the graphic modules.

`PROCEDURE Init;`
Initialization. There is an implicit call to `RTGraph.Init` and thus to `Kernel.Init`.

`END RTMouse.`

```
DEFINITION MODULE AnalogIO;

(* Analog IO via the VDAD boards from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the analog ports to operate in
   the range +/- 10V.
*)

FROM VDAD IMPORT NrOfCards;

TYPE CardType     = [1..NrOfCards];
     ChannelType  = [0..15];
     DARange      = [-2048..2047];
     IntArray     = ARRAY ChannelType OF INTEGER;
     ExpGainType  = [0..2];
     OutRangeType = [0..1];

PROCEDURE ADin  (     cardnr  : CardType;
                      channel : ChannelType;
                  VAR value   : INTEGER      );

PROCEDURE DAout (     cardnr  : CardType;
                      channel : ChannelType;
                      value   : DARange      );

PROCEDURE MultiADin (     cardnr      : CardType;
                          LowChannel,
                          HighChannel : ChannelType;
                      VAR value       : IntArray     );

PROCEDURE SetInputGain( cardnr  : CardType;
                        ExpGain : ExpGainType );
  (* ExpGain = 0, 1, 2  =>  Input gain = 1, 10, 100 *)

PROCEDURE SetOutVoltage( cardnr     : CardType;
                         channel    : ChannelType;
                         Gain,
                         UniBiPolar : OutRangeType );
  (* Outvoltage range = Vref * (1 + Gain)  *)
  (* UniBiPolar: 0 = unipolar, 1 = bipolar *)

END AnalogIO.
```

```
DEFINITION MODULE DigitalIO;

(* Digital IO via VDAD boards and the VDIN board from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the digital ports to be output
   on board number one, and input on board number two.

   The VDIN board requires no initialization. To read the 16 bit
   parallell input port, call DigInput.
*)

FROM VDAD   IMPORT NrOfCards;
FROM SYSTEM IMPORT BYTE;

TYPE CardType = [1..NrOfCards];
     WireType = [0..7];
     Bit      = [0..1];
     Byte     = [0..255];

PROCEDURE DigWireOutput (     cardnr : CardType;
                              wire   : WireType;
                              value  : Bit      );

PROCEDURE DigWireInput  (     cardnr : CardType;
                              wire   : WireType;
                        VAR   value  : CARDINAL );

PROCEDURE DigByteOutput (     cardnr : CardType;
                              value  : BYTE     );

PROCEDURE DigByteInput  (     cardnr : CardType;
                        VAR   value  : BYTE );

PROCEDURE DigInput( VAR value : SHORTINT );
  (* value = [-32768, 32767] *)

END DigitalIO.
```

```
DEFINITION MODULE MiscIO;

(* Analog and Digital IO via one VDAD board from PEP computers. *)

FROM SYSTEM IMPORT BYTE;

PROCEDURE Init;
(* Configures the analog ports to operate in the range +/- 10V
   and the digital ports to operate as outputs (TTL-levels)  *)

(* Analog: *)

PROCEDURE ADin  (     channel : CARDINAL;     (* [0..7] *)
                  VAR value   : LONGREAL);    (*  -1.0 <= value <= 1.0 *)

PROCEDURE DAout (     channel : CARDINAL;     (* [0..3] *)
                      value   : LONGREAL);    (*  -1.0 <= value <= 1.0 *)

PROCEDURE VoltIn (    channel : CARDINAL;     (* [0..7] *)
                  VAR voltage : LONGREAL);    (*  -10.0 <= voltage <= 10.0 *)

PROCEDURE VoltOut (   channel : CARDINAL;     (* [0..3] *)
                      voltage : LONGREAL);    (*  -10.0 <= voltage <= 10.0 *)

(* Digital: *)

PROCEDURE DigOutput ( channel : CARDINAL;
                      value   : BOOLEAN );

PROCEDURE DigInput (      channel : CARDINAL;
                    VAR value   : BOOLEAN );

PROCEDURE DigByteOutput ( value  : BYTE);

PROCEDURE DigByteInput ( VAR value  : BYTE );

END MiscIO.
```

```
DEFINITION MODULE MatComm;

IMPORT SYSTEM;
(*$NONSTANDARD*)

CONST
  ProcessNameSignificance = 32;

TYPE
  Socket;
  NameString = ARRAY [1..ProcessNameSignificance] OF CHAR;
  DataType  = (char,real,longreal,complex,longcomplex);
  ErrorType = (OK,notOK,Closed);


PROCEDURE OpenSocket
  (VAR socket    : Socket;
   REF myname : ARRAY OF CHAR);
  (*
  A socket is returned to be used in subsequent calls to the procedures
  below. The 'myname' is required to make the request for a socket unique.
  It can be any string, but is typically the name of the process. The same
  string has to be given as the proc-argument to vmeio in matlab. It does
  not matter if the modula process or the unix process (i.e. matlab) is
  the first one to try to establish a new connection (after Init has been
  called).
  *)

PROCEDURE CloseSocket
  (VAR socket      : Socket);
  (*
  The socket is closed by the caller for further communication. The line
  can also be closed by the remote machine. In both cases, a Send or
  Receive request will return 'Closed'. If so, OpenSocket can be called
  again.
  *)

PROCEDURE Send
  (VAR socket  : Socket;
       nrows : CARDINAL;
       ncols : CARDINAL;
       dtype : DataType;
   REF data : ARRAY OF SYSTEM.BYTE): ErrorType;
  (*
  Send the 'data' on the open socket 'socket'. Proper values for 'nrows',
  'ncols', and 'dtype' have to be supplied (if you don't want a dump of
  the memory following the variable supplied). 'data' is however
  allowed to be bigger than the matrix specified.
  *)

PROCEDURE GetNextType
  (VAR socket       : Socket;
```

```
   VAR nrows : CARDINAL;
   VAR ncols : CARDINAL;
   VAR dtype : DataType): ErrorType;
  (*
  If a new data message is available, the head of the message is read and
  the type of the matrix is returned in nrown, ncols, and dtype. To
  prevent reading the head again in an additional call (without
  Receive in between), and to save some computations, some extra
  information is stored in 'socket'. To allow update this private
  information, 'socket' is also VAR declared. If no data is available,
  an Await for data on 'socket' is performed. If data to be received is
  of fixed type (or size), Receive can be called directly.
  *)

PROCEDURE Receive
  (VAR s    : Socket;
   VAR nrows : CARDINAL;
   VAR ncols : CARDINAL;
   VAR dtype : DataType;
   VAR data : ARRAY OF SYSTEM.BYTE): ErrorType;
  (*
  If not done already for the next message, GetNextType is called.
  This means 'nrows', 'ncols', and 'dtype' will be the same as if
  GetNextType were called. The 'data'-matrix is allowed to be bigger
  then required to store the data. If the 'data'-matrix is to small,
  as much as possible is stored in 'data'. The rest is read in and
  then deallocated. In this case 'notOK' is returned.
  *)

(*PROCEDURE Receive2
  (VAR socket    : Socket;
   VAR nrows : CARDINAL;
   VAR ncols : CARDINAL;
   VAR dtype : DataType;
   VAR data : ARRAY OF SYSTEM.BYTE;
       dsize : CARDINAL): ErrorType;*)
  (*
  As Receive, but the size of the data has to be explicitly given.
  To be used for dynamic variables with size unknown at compile time.
  *)

PROCEDURE Init;

END MatComm.
```

2

Real-Time Graphics Support Modules

M. Andersson

GOAL: To give principles and documentation for a user interface event handler and support modules at a layer on top of the real-time kernel.

An implementation of a control system almost always includes a number of components regarding interaction, such as presenting values, plotting signals, creating signals, and others. Since the functions needed are similar between different applications, it is possible to provide a set of support modules to simplify the use of the basic real-time and graphics primitives, like those in the previous chapter. The advantage of having such support modules can be quantified by comparing the example given in Section 2.2 with a similar program implemented using only the real-time kernel layer. The code size is reduced four times, and the work to develop the program is reduced considerably more. The function and implementation of the library modules are described in Sections 2.1. An example of the use of the routines in a real-time program is given in Section 2.2, and the definition modules of the routines are listed thereafter.

Main principles

A control system can be decomposed in two main parts: the controller and the operator's interface. The controller subsystem includes in this context simple regulators, supervisory control systems, fault diagnosis and any other subsystem interacting with the controlled process. The operator's interface is the subsystem handling all human–machine interaction. It is a desirable to be able to separate the design and definition of the operator's interface from the controller subsystem as far as possible.

The task of the operator's interface is to respond on input events from the operator and to execute them as commands to the controller subsystem. It is also responsible for presenting data from the controller and process to the operator. The program modules presented in this chapter are designed to accept operator commands in form of mouse clicks and keyboard inputs from a single computer or a terminal.

The operator's interface could be designed such that every possible action from the operator has it's own designated process, monitoring that particular event and responding

by an appropriate action. This approach leads to many process. An alternative approach is taking advantage of the facts that the operator is relatively slow in producing events and that commands can be executed comparably quickly. Therefore, it is possible to use a single process, called an event handler, waiting for any possible event, identifying the event and taking appropriate action. This later approach is used in the modules presented in this chapter.

The communication between the operator's interface and the controller subsystems should be done in a uniform way. The method used here is called *callback* procedures. A callback procedure is defined by the client, in this case the controller subsystem, and registered by an interactor object. When the operator performs an input operation the event handler calls the callback procedure connected with the involved interactor. The client can then, by means of the callback procedure, extract useful information from the interactor object which is passed as an argument to the callback procedure.

While a callback procedure is executing the operator's interface is blocked and does not respond to additional events. It is client's responsibility to provide callback procedures which returns without unnecessary delay.

More details about the event handler and about specialized interactor objects are presented in the following sections.

Implementation details

The system is written in Logitech's Modula-2 and is intended to be used on IBM-AT compatible machines with EGA and Microsoft Mouse. The real-time kernel layer described in the previous chapter is used for the implementation.

All coordinates for windows, menus etc. are given in screen coordinates, i.e., $0 \leq x \leq 1.5$ and $0 \leq y \leq 1.0$.

Opaque (hidden) data types have been used to implement buttons, bargraphs, lists, menus, and plot windows. This means that the internal structure of these objects are inaccessible to a user of the modules. The only way to operate on the objects are via the routines declared in the definition modules. This is the Modula-2 way to implement abstract data types.

2.1 The Event Handler

All operator interaction modules described in this chapter are based on two fundamental modules called `MouseEvent` and `TextArea`. The former module is the heart of the operator's interface. It contains a process which waits for the next mouse or keyboard event, issued by the operator, and executes an appropriate event handling procedure.

The Event Handler allows clients to specify click sensitive areas on the screen. The click sensitive area is an object called `MouseArea` defined in module `MouseEvent`. When a mouse area is created, the client specifies, in addition to the screen area, a *callback* procedure. A callback procedure is a procedure defined by the client and invoked by the event handler whenever the operator clicks the mouse within the sensitive area. The callback procedure must be defined with a parameter which is a pointer to a mouse area. When the event handler has detected a mouse event in a sensitive area, it looks up the mouse area object in an internal list of all active mouse areas. Then it invokes its callback procedure with a pointer to the mouse area object itself as argument.

A mouse area object also contains a pointer of type `ADDRESS` to any kind of data specified by the client. The pointer is called *user's pointer* and it can be accessed by the client defined callback procedure and converted to the correct pointer type. This makes it possible to define hierarchical interaction objects. If a higher level interaction object creates one or many mouse areas the user's pointer of these are set to point at the

object itself. The callback procedures are then designed so that they extract the pointer, convert it to a pointer to the high level object and do appropriate manipulations of its data structure.

The interaction between a mouse area object and a client object may appear complex and difficult to understand at first glance but the following example should hopefully throw some light. Refer to the definition of the MouseEvent module given in the end of this chapter. Assume we want to create controller objects where each controller has its own start button on the screen. The controller module defines a callback procedure and a procedure for creating controller objects which also creates the associated start button.

```
MODULE Controller;
TYPE ContrPtr = POINTER TO ContrData;
     ContrData = RECORD ... END;

PROCEDURE CreateController(VAR newContr: ContrPtr);
VAR startButton: MouseAreaPtr;
    area: rectangle;
BEGIN
  NEW(newContr);
  ...
  (* Create interaction object and give a pointer to this controller
     object as the user's pointer:                                    *)
  CreateMouseArea(startButton, area, StartButtonCB, newContr);
  ...
END CreateController;

PROCEDURE StartButtonCB(ma: MouseAreaPtr) (* Callback *)
VAR contr: ContrPtr;
BEGIN
  (* Get the user's pointer and convert it its correct type:    *)
  contr := ContrPtr(GetUsersPtr(ma));
  ...
  (* use contr to access controller data and procedures *)
END StartButtonCB;
```

Mouse areas can be deactivated and activated again. A deactivated mouse area does not respond to mouse clicks. If several mouse areas overlap on the screen it is the last activated area under the mouse which receives a click event.

The event handler is based on two identical processes where one of them is always waiting for a mouse event interrupt. The reason for using two processes is that a process may be blocked waiting for text input. Text input objects, described below, are also handled by the event handler. When one process is blocked by a text input waiting for characters, the other process can still respond on mouse events.

Text input objects

Closely associated with the MouseEvent module is the TextArea module. It defines text area objects which can be used for character inputs from the operator. A text area is an object based on a mouse area. When the operator clicks in an active text area it starts reading and echoing characters from the keyboard. When the operator pushes the Return key, the string is completed and a callback procedure associated with the text area object is invoked by the event handler.

The callback procedure of a text area is invoked with a pointer to the text area object itself as a parameter. The client can then access the completed string and get a pointer

to his own data in the same way as for mouse areas.

Only one text area can do active character reading at a time. If the operator clicks in another text area while one is waiting for characters, the first one will be interrupted and the focus will change to the new text area. The interrupted text area will invoke its callback with the incompleted string as current text. It is possible for the callback to query the text area object and check if it was interrupted or completed by the user.

A special kind of text area called *numerical input* is also defined in the `TextArea` module. A numerical input tries to interpret a completed string as a real number and calls the callback procedure only if this is possible.

Text areas can be deactivated and activated in the same way as mouse areas.

Drawing interaction objects

Neither mouse area nor text area objects draw anything on the screen. The defined mouse sensitive areas are invisible. A special module is available for drawing buttons and frames suitable for giving mouse areas and text input fields an appearance on the screen. The module is called `Draw`.

For example, in order to create a mouse button on the screen, start by defining the screen area where you want the button by defining a `rectangle`. Use `CreateMouseArea` to make the region mouse sensible and give it a suitable callback procedure. Then use `DrawButton` or `DrawDefaultButton` with the same rectangle as parameter. If you later want to get rid of the button you must deactivate the mouse area and fill the rectangle with the background color.

`Draw` does not interact with the event handler or text area modules. This makes it easy for the user to define his own drawing routines and make his own fancy graphical layout.

Easy interactors

A module called `Easy` defines a set of interaction objects designed to be specially easy to use. They are not designed to be extendable and reused in the same way as other interactor modules described below. With `Easy` it is possible to create buttons, text inputs, number inputs, and bargraphs. A single procedure call creates the mouse sensitive area and draws the object on the screen. Easy objects cannot be deactivated or accessed in other way. They only respond to operator actions by calling the specified callback procedure.

Other interactors

A set of modules defining different kinds or interactor objects are available. They are all designed to be useful as stand alone interactors or as parts in other user defined interactors.

The following modules are available.

Bargraph is a device for display and input of numeric values. Bargraphs can be created with a horizontal or a vertical layout.

Menu is a device for doing multiple choice selections. A menu can work as a radio button or as an array of single buttons.

NumMenu is a form with multiple numeric input fields and an enter button. The operator can change individual fields but the callback procedure is not called until the enter button is clicked.

Plotter is a device for plotting up to six signals against a common horizontal time axis.

Other useful modules

`ListHandler` and `Signals` are two modules that are not designed specially for creating operator interfaces but are generally useful. They do not depend on the event handler modules. The `ListHandler` module handles a doubly linked non-circular list with a list head. The list handling is done so that it is independent of the type of the element that is stored in the nodes. This is achieved by using the data type `ADDRESS` in the nodes. This also means that the elements must be referred via a pointer, see the example in the definition module.

The routines in the `Signals` module are used for generating time signals. Any number of signals can be generated, and each signal is identified by a text string. The signals which can be generated are of the types `Sin`, `Step`, `Pulse`, `Ramp`, and `Random`. Signals of different types can be generated at the same time. The user of this module does not have to handle time explicitly. When `GetRefValue` is called the value of the signal at that time is returned. The value of all signals are in the interval [0 1]. There is no way of changing the amplitude and offset of the signal within the module, so that must be taken care of by the user program. The runs a single process generating all signals.

Module `TextWindows` makes it possible to create a window for simple output and input of text and numbers. The text scrolls vertically when lines are added below the last visible line. The number of visible lines and columns depends on the window size, given in screen coordinates.

2.2 An Example

The implementation of a PI controller is used as an example of the use of the modules. The program starts on the next page. It is possible to change the parameters (k and T_i) of the controller. The reference signal is a square wave. The amplitude and frequency of the reference signal can be changed. All parameters are changed by using a numerical menu. The reference value, the control signal, the process value, and the value of the integrator are plotted in the plot window during the operation of the controller. The program halts when the exit button is clicked. A similar program that was implemented using only the real-time kernel layer in the previous chapter was 16 pages long.

```
MODULE Regul;

IMPORT RTMouse;
FROM Semaphores IMPORT Semaphore, InitSem, Wait, Signal;
FROM AnalogIO IMPORT ADIn, DAOut;
FROM Graphics IMPORT point, color, ShowCursor;
FROM Kernel IMPORT Time, CreateProcess, SetPriority, IncTime,
                   WaitUntil, CurrentTime, TimeToReal;
FROM Monitors IMPORT MonitorGate, InitMonitor, EnterMonitor,
                     LeaveMonitor;
FROM Signals IMPORT InitSignals, ChangeOmega, ChangeDelta,
                    ChangeFunction, GetRefValue, FunctionType,
                    MakeRefSignal, ChangeDirection;
FROM Plot IMPORT PlotterPtr, CreatePlotter, SetChannel,
                 SetTime, WriteValue;
FROM NumMenu IMPORT CreateNumMenu, NumMenuPtr, SetEntry, GetValues;

CONST KInit   = 5.0;
      TiInit  = 10.0;
      AmpInit = 0.1;

VAR Exit : Semaphore;
    nm : NumMenuPtr;
    Plt : PlotterPtr;
    Pos : point;
    RegPar : RECORD
               Mutex : MonitorGate;
               K, Ti, Amp : REAL;
             END;
(*-----------------------------------------------------------*)
PROCEDURE GetRegPar(VAR p1, p2, p3 : REAL);
BEGIN
  WITH RegPar DO
    EnterMonitor(Mutex);
    p1 := K;
    p2 := Ti;
    p3 := Amp;
    LeaveMonitor(Mutex);
  END;
END GetRegPar;
(*-----------------------------------------------------------*)
PROCEDURE SetRegPar(p : ARRAY OF REAL);
BEGIN
  WITH RegPar DO
    EnterMonitor(Mutex);
    K := p[0];
    Ti := p[1];
    Amp := p[2];
    LeaveMonitor(Mutex);
  END;
  ChangeOmega("Ref", p[3]);
```

```
END SetRegPar;
(*-----------------------------------------------------------*)
PROCEDURE InitRegPar;
BEGIN
  WITH RegPar DO
    InitMonitor(Mutex, "Mutex");
    K := KInit;
    Ti := TiInit;
    Amp := AmpInit;
  END;
END InitRegPar;
(*-----------------------------------------------------------*)
(* Process *) PROCEDURE RegulProcess;
CONST h = 20; offset = 0.5;
VAR t : Time;
    amp, v, u, y, yref, e, i, k, ti : REAL;
    index : CARDINAL;
BEGIN
  SetPriority(10);
  CurrentTime(t);
  i := 0.0; index := 0;
  LOOP
    GetRefValue("Ref", yref);
    GetRegPar(k, ti, amp);
    yref := 2.0*amp*(yref - 0.5) + offset;
    y := ADIn(1);
    e := yref - y;
    v := k*e + i;
    u := v;
    IF v > 1.0 THEN
      u := 1.0;
    ELSIF v < 0.0 THEN
      u := 0.0;
    END;
    DAOut(1, u);
    i := i + k*e*FLOAT(h)/(1000.0*ti) + FLOAT(h)/(1000.0*ti)*(u - v);
    IF index < 10 THEN
      INC(index);
    ELSE (* plot every 10:th sample *)
      index := 0;
      SetTime(Plt,TimeToReal(t));
      WriteValue(Plt, 1, i);
      WriteValue(Plt, 2, u);
      WriteValue(Plt, 3, yref);
      WriteValue(Plt, 4, y);
    END;
    IncTime(t, h);
    WaitUntil(t);
  END;
END RegulProcess;
(*-----------------------------------------------------------*)
```

```
PROCEDURE ExitButtonCB(p: point); (* Callback for Exit button *)
BEGIN
  Signal(Exit);
END ExitButtonCB;
(*-------------------------------------------------------------------*)
PROCEDURE NumMenuCB(nm: NumMenuPtr); (* Callback for parameter menu*)
VAR data: ARRAY [1..4] OF REAL;
BEGIN
  GetValues(nm,data);
  SetRegPar(data);
END NumMenuCB;
(*-------------------------------------------------------------------*)
BEGIN
  MouseEvent.Init(20);
  InitSem(Exit,0,"ExitSem");

  area.xlo := 0.05; area.xhi := 1.45;
  area.ylo := 0.55; area.yhi := 0.95;
  CreatePlotter(Plt, area, 4, 30.0, white, green);
  SetChannel(Plt, 1, "i", cyan);
  SetChannel(Plt, 2, "u", lightblue);
  SetChannel(Plt, 3, "yref", lightcyan);
  SetChannel(Plt, 4, "y", blue);

  Pos.h := 0.05; Pos.v := 0.2;
  CreateNumMenu(NM, Pos, 4, 10, grey, red, NumMenuCB, NIL);
  SetEntry(NM, 1, "K", KInit);
  SetEntry(NM, 2, "Ti", TiInit);
  SetEntry(NM, 3, "Amplitude", AmpInit);
  SetEntry(NM, 4, "Frequency", 0.5);

  area.xlo := 1.35; area.xhi := 1.45;
  area.ylo := 0.05; area.yhi := 0.15;
  EasyButton(area, red, "Exit", ExitButtonCB);

  InitSignals(20);
  MakeRefSignal("Ref", Step, 0.5);
  ChangeDelta(50);

  InitRegPar;
  CreateProcess(Opcom, 1000, "Opcom");
  CreateProcess(RegulProcess, 1000, "Regul");
  ShowCursor;

  Wait(Exit);
END Regul.
```

```
DEFINITION MODULE MouseEvent;
```

Module MouseEvent is a handler for mouse events based on callback procedures. It serves as the basis for other modules providing specialized objects for user interaction.

This module allows click sensitive regions, called mouse areas, to be defined on the screen. The user should provide a callback procedure for each mouse area. This module reacts on every mouse event in an active region and calls the corresponding callback procedure.

Procedures in this module don't draw anything. Use procedures in Draw, or design your own, and draw things on top of the mouse areas.

Standard screen coordinates used in this moudule are: `0.0 <= x <= 1.5; 0.0 <= y <= 1.0`

The event handler is blocked while a callback procedure is executing. This means that in order to respond on quick consecutive mouse events, callback procedures should be reasonably quick to execute. Callback procedures are executed with the priority given to the Init procedure.

This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle;

TYPE MouseAreaPtr;
     EventProcType  = PROCEDURE(MouseAreaPtr); Callback procedure
     ButtonProcType = PROCEDURE(point);        Callback procedure

PROCEDURE Init(priority: CARDINAL);
```

Inits Kernel, RTMouse, and this event handler. Two identical Eventhandler processes are created so that if one is locked by a reading characters from the keyboard, the other can still handle buttons. The event handler processes will run with the given priority.

```
PROCEDURE CreateMouseArea(VAR newMouseArea : MouseAreaPtr;
                          Area             : rectangle;
                          LeftMouseProc,
                          RightMouseProc   : EventProcType;
                          UsersPtr         : ADDRESS);
```

Creates and activates a click sensitive area on the screen that when clicked will cause Left/RightMouseProc to be called. Returns a pointer to the new object.

```
PROCEDURE Deactivate(ma: MouseAreaPtr);
```

Deactivates a mouse sensitive area.

```
PROCEDURE Activate(ma: MouseAreaPtr);
```

Makes the mouse area sensitive to events.

```
PROCEDURE GetMousePoint(ma: MouseAreaPtr; VAR p: point);
```

Gets point of last mouse click.

```
PROCEDURE GetUsersPtr(ma: MouseAreaPtr) : ADDRESS;
```
Returns the users's pointer of the mouse area

```
PROCEDURE Dispose(ma: MouseAreaPtr);
```
Disposes a mouse area object.

```
PROCEDURE DeactivateArea(InArea: rectangle);
```
Deactivates all mouse sensitive areas inside InArea.

```
END MouseEvent.
```

```
DEFINITION MODULE TextArea;
```

Module TextArea is based on module MouseEvent. It is used for defining text input areas on the screen.

Module MouseEvent must be initialized before any procedure in this module is called.

When a text area is clicked it starts accepting characters from the keyboard. Input can be ended by the Return key or by clicking on another text input. Only one text area at a time can wait for input. A callback procedure is called when input is ended with Return or interrupted by another text input.

This module also supports a related kind of objects specialized for numeric inputs, created by CreateNumInput.

This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;
FROM MouseEvent IMPORT MouseAreaPtr;

CONST MaxTextLength = 80;
      HighlightColor = lightred;

TYPE TextAreaPtr;
     TextColorPtr   = POINTER TO TextColors;

     TextColors = RECORD
       AreaColor,                    Color of the inside area
       TextColor,                    Color of the text
       EditColor   : color;          Color of the text during editing
     END;

     TextProcType = PROCEDURE(TextAreaPtr); Callback

PROCEDURE CreateTextArea(VAR newTextArea: TextAreaPtr;
                         loleft: point;
                         width: CARDINAL;
                         AreaColor, TextColor: color;
                         Callback: TextProcType;
                         UsersPtr: ADDRESS);
```

Creates and activates a click sensitive text input field on the screen. When the area is clicked it will start accepting characters. When editing is finished, the user's callback procedure DoTextProc will be called for further processing. The user may let UsersPtr point to his data. A pointer to the new TextArea object is returned.

```
PROCEDURE Activate(ta: TextAreaPtr);
```

Activates the text area

```
PROCEDURE Deactivate(ta: TextAreaPtr);
```

Deactivates the text area

```
PROCEDURE GetTextColors(ta: TextAreaPtr) : TextColorPtr;
```
Returns a pointer to color data of a text area. Can be used for changing the colors. Changing colors while the text area is doing text input might give strange results.

```
PROCEDURE GetUsersPtr(ta: TextAreaPtr) : ADDRESS;
```
Return the user's pointer.

```
PROCEDURE GetTextPosition(ta: TextAreaPtr; VAR pos: point);
```
Return the point where the text starts.

```
PROCEDURE Interrupted(ta: TextAreaPtr) : BOOLEAN;
```
Returns true if the last text input was interrupted.

```
PROCEDURE IsReading(ta: TextAreaPtr) : BOOLEAN;
```
Returns TRUE is the text area is currently waiting for input.

```
PROCEDURE GetText(ta: TextAreaPtr; VAR text: ARRAY OF CHAR);
```
Returns the current text string.

```
PROCEDURE PutText(ta: TextAreaPtr; VAR text: ARRAY OF CHAR);
```
Sets current string and write it in text area. Writing an empty string will erase the area. The text argument is not changed — VAR is for efficiency only.

```
PROCEDURE Dispose(ta: TextAreaPtr);
```
Dispose the text area.

```
PROCEDURE ActiveTextArea(position: point; width: CARDINAL;
                         VAR area: rectangle;
                         VAR textpoint: point);
```
Given the position of the lower left corner and the width of a text area, compute the active area and the text input position.

```
PROCEDURE CreateNumInput(VAR newTextArea: TextAreaPtr;
                         loleft: point;
                         width: CARDINAL;
                         AreaColor, TextColor: color;
                         Callback: TextProcType;
                         UsersPtr: ADDRESS);
```
Creates a numeric input text area. A numeric input area is similar to a text area with the difference that it only calls the callback procedure when the current text is a valid number.

All procedures valid for text area are also valid for numeric input.

```
PROCEDURE PutNumber(ta: TextAreaPtr; value: REAL);
```
Sets a new number for numeric input object and prints it. This procedure works for objects created by CreateTextArea as well.

```
PROCEDURE GetNumber(ta: TextAreaPtr) : REAL;
```
Gets current number from numeric input object. This procedure should not be used for objects created by CreateTextArea.

```
PROCEDURE StopReading(ma: MouseAreaPtr);
```
Not for public use. Used by module MouseEvent.

```
PROCEDURE Init;
```
Not for public use. Called by MouseEvent.Init

```
END TextArea.
```

```
DEFINITION MODULE Draw;
```

Draw contains graphic routines for drawing buttons and and text input boxes suitable for MouseArea and TextArea objects. The Real-Time Kernel must be initialized before the module is used.

This MODULE is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM Graphics IMPORT point, color, rectangle;

CONST PixWidth     = 1.5/639.0;
      PixHeight    = 1.0/349.0;
      BorderWidth  = PixWidth*2.0;
      BorderHeight = PixHeight*1.0;


PROCEDURE NiceColors(areacolor: color;
                     VAR textcolor,darkcolor,lightcolor: color);
```

Chooses suitable colors for shadowing and text.

```
PROCEDURE DrawButton(
            area       : rectangle;
            buttonText : ARRAY OF CHAR;
            areacolor, textcolor, darkcolor, lightcolor: color);
```

Draws a defaultshaped button on the screen. ButtonText will be truncated to fit into Area.

```
PROCEDURE DrawDefaultButton(
            area       : rectangle;
            buttonText : ARRAY OF CHAR;
            areacolor  : color);
```

Draws a button in default colors

```
PROCEDURE DrawTextArea(
             Position  : point;
             Width : CARDINAL;
             Text : ARRAY OF CHAR;
             DarkColor, LightColor,
             TextColor, BackColor : color);
```

Draws a text area and writes Text into it. The drawn area will be slightly larger than the active area defined by ActiveTextArea in module TextArea. A frame of sizes BorderWidth and BorderHeight is added around the active area. The arguments DarkColor and LightColor are used for the frame.

```
PROCEDURE FillTextArea(position: point; width: CARDINAL;
                       fillcolor: color);
```

Fills the text area with fillcolor. Use color of background for hiding the object.

```
PROCEDURE DrawFrame(area: rectangle; upLeftColor, lowRightColor: color);
```

Draws a shaddow frame around (inside) area. The frame is BorderHeight thick at the top and bottom and BorderWidth thick at the sides.

```
PROCEDURE DrawWindow(Area: rectangle; Title: ARRAY OF CHAR);
```
Draws a predesigned window with Title in window bar

```
END Draw.
```

```
DEFINITION MODULE Easy;
```

Module Easy provides an easy way of creating and drawing simple interaction objects. Easy is based on module MouseEvent.

Initialize module MouseEvent before using Easy procedures.

This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM Graphics IMPORT point, rectangle, color;

CONST BarGraphWidth = 26.0*1.5/80.0;
      BarGraphHeight = 0.14;

TYPE ButtonProc   = PROCEDURE(point);
     InputProc    = PROCEDURE(ARRAY OF CHAR, BOOLEAN);
     NumInputProc = PROCEDURE(REAL);
     BarGraphProc = PROCEDURE(REAL);
```

Callback procedure types

```
PROCEDURE EasyButton(
            Area       : rectangle;
            AreaColor  : color;
            Text       : ARRAY OF CHAR;
            Callback   : ButtonProc);
```

Creates and draws a button. The button will react on mouse clicks by calling the provided callback procedure of type `PROCEDURE(point)`.

```
PROCEDURE EasyInput(
            Position               : point;
            Width                  : CARDINAL;
            FrameColor, BackColor  : color;
            Text                   : ARRAY OF CHAR;
            Callback               : InputProc);
```

Defines and draws a default input field for text and writes Text in it. Give it a callback procedure of type `PROCEDURE(ARRAY OF CHAR, BOOLEAN)`. Input is activated by the operator by a mouse click. When the operator pushes Return or clicks in another text input, Callback will be called with the current string as argument and a boolean argument set to TRUE if input was interrupted, i.e., not ended by Return.

```
PROCEDURE EasyNumInput(
            Position               : point;
            Width                  : CARDINAL;
            FrameColor, BackColor  : color;
            Number                 : REAL;
            Callback               : NumInputProc);
```

Defines and draws a default input field for numbers and writes Number in it. Give it a callback procedure of type `PROCEDURE(REAL)`. Input is activated by the operator by a mouse click. When the operator pushes Return the current string will be interpreted as a real number. If this is possible Callback will be called with the number as argument.

```
PROCEDURE EasyBarGraph (
            LoLeft                      : point;
            AreaColor, FrameColor,
            BackColor, BarColor         : color;
            Value, MinValue, MaxValue : REAL;
            Callback                    : BarGraphProc);
```

Creates and draws a bargraph at position LoLeft. Give it a callback procedure of type `PROCEDURE(REAL)`; When operator changes the value of the bargraph Callback will be called with the new value as argument.

```
END Easy.
```

```
DEFINITION MODULE BarGraph;
```
Module BarGraph is for creating graphical and numerical input devices. This module is based on MouseEvent which must be initialized before any bargraph is created. Bargraphs can have horizontal or vertical layout.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;

TYPE BarGraphPtr;
     BarGraphProc = PROCEDURE(BarGraphPtr);

CONST VBGwidth = 0.3;
      HBGwidth = 0.5;  HBGheight = 0.2;

PROCEDURE CreateHBG(VAR newBG: BarGraphPtr; loleft: point;
                    areaColor, frameColor, barColor: color;
                    value, minValue, maxValue: REAL;
                    callback: BarGraphProc; usersPtr: ADDRESS;
                    title: ARRAY OF CHAR);
```
Creates and activates a horizontal bargraph and returns a pointer to it. The callback procedure is called whenever the operator sets the value.

```
PROCEDURE CreateVBG(VAR newBG: BarGraphPtr; loleft: point; height: REAL;
                    areaColor, frameColor, barColor: color;
                    value, minValue, maxValue: REAL;
                    callback: BarGraphProc; usersPtr: ADDRESS;
                    title: ARRAY OF CHAR);
```
Creates and activates a vertical bargraph and returns a pointer to it. The callback procedure is called whenever the operator sets the value.

```
PROCEDURE GetValue(bg: BarGraphPtr) : REAL;
```
Gets current value of the bargraph.

```
PROCEDURE SetValue(bg: BarGraphPtr; value: REAL);
```
Sets a new value for the bargraph. This will not call the callback procedure.

```
PROCEDURE Activate(bg: BarGraphPtr);
```
Activates and redraws the bargraph.

```
PROCEDURE Deactivate(bg: BarGraphPtr);
```
Deactivates the bargraph.

```
PROCEDURE GetUsersPtr(bg: BarGraphPtr) : ADDRESS;
```
Returns user's pointer.

```
END BarGraph.
```

```
DEFINITION MODULE Menu;
```
Module for defining menus of selections. A menu is vertical list of selectable items. This module is based on Module MouseArea which must be initialized before any procedure is called.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;

TYPE MenuPtr;
     MenuProc = PROCEDURE(MenuPtr);

CONST MaxNoOfItems = 20;

PROCEDURE CreateMenu(newMenu: MenuPtr; loleft: point;
                     items, width: CARDINAL;
                     radio: BOOLEAN; backColor, selectColor: color;
                     callback: MenuProc; usersPtr: ADDRESS;
                     title: ARRAY OF CHAR);
```
Creates and activates a menu. The width is number of characters in each item label. If radio is TRUE, last selected item remains highlighted with selectColor, otherwise selected item is highlighted shortly. If title is given as an empty string no header is drawn on the menu. The callback procedure is called when the operator makes a selection.

```
PROCEDURE GetArea(m: MenuPtr; VAR area: rectangle);
```
Returns the area occupied on the screen by the menu.

```
PROCEDURE SetLabel(m: MenuPtr; item: CARDINAL; label: ARRAY OF CHAR);
```
Sets the label of a menu item.

```
PROCEDURE SetSelection(m: MenuPtr; item: CARDINAL);
```
Sets current selection.

```
PROCEDURE GetSelection(m: MenuPtr) : CARDINAL;
```
Gets current selection.

```
PROCEDURE Activate(m: MenuPtr);
```
Activates and redraws the menu.

```
PROCEDURE Deactivate(m: MenuPtr);
```
Deactivates the menu.

```
PROCEDURE GetUsersPtr(m: MenuPtr) : ADDRESS;
```
Return user's pointer.

```
END Menu.
```

```
DEFINITION MODULE NumMenu;
```
Module NumMenu for creating and operating on numeric menus. A numeric menu is a form with a number of labeled numeric input fields and an enter button. This module is based on module MouseArea which must be initialized before any numeric menu is created.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;

TYPE NumMenuPtr;
     NumMenuProc = PROCEDURE(NumMenuPtr);

CONST MaxNoOfEntries = 12;

PROCEDURE CreateNumMenu(newNM: NumMenuPtr; lowleft: point;
                        noOfEntries, labelWidth: CARDINAL;
                        areaColor, frameColor: color;
                        callback: NumMenuProc; usersPtr: ADDRESS;
                        title: ARRAY OF CHAR);
```
Creates and activates a numeric menu. The callback procedure is called when the operator clicks on the Enter button.

```
PROCEDURE SetEntry(nm: NumMenuPtr; entry: CARDINAL;
                   label: ARRAY OF CHAR;
                   value: REAL);
```
Sets the label string and value of a numeric menu. The entries are numbered 1 to noOfEntries. The label is truncated to the length specified when the menu was created. If this is not called the default label is the empty string and the value is 0.0.

```
PROCEDURE GetArea(nm: NumMenuPtr; area: rectangle);
```
Returns the area occupied on the screen by the numeric menu.

```
PROCEDURE GetValues(nm: NumMenuPtr; VAR value: ARRAY OF REAL);
```
Returns the value of each field of the numeric menu.

```
PROCEDURE Activate(nm: NumMenuPtr);
```
Activates and redraws the numeric menu.

```
PROCEDURE Deactivate(nm: NumMenuPtr);
```
Deactivates the numeric menu.

```
PROCEDURE GetUsersPtr(nm: NumMenuPtr) : ADDRESS;
```
Return user's pointer.

```
END NumMenu.
```

```
DEFINITION MODULE Plotter;
```
Module for plotter objects. A plotter is an area of the screen where up to 6 variables are plotted against a common horizontal axis. The operator can use the mouse to switch the individual channels on or off. The module is based on the module MouseArea and requires that MouseArea is initialized before a plotter is created.

```
FROM Graphics IMPORT rectangle, color;

TYPE PlotterPtr;

CONST MaxNoOfChannels = 6;

PROCEDURE CreatePlotter(VAR newPlotter: PlotterPtr; area: rectangle;
                        noOfChannels: CARDINAL; timeScale: REAL;
                        backColor, frameColor: color;
                        buttons: BOOLEAN; title: ARRAY OF CHAR);
```
Creates and activates a plotter object and returns a pointer to it. Parameters: area is the region the plotter will occupy on the screen, noOfChannels must be a number from 1 to 6, timeScale defines the horizontal axis, backColor defines the background color, frameColor defines the color for the frame. If buttons is TRUE a button for each channel is created where the operator can switch the channel on or off.

```
PROCEDURE SetChannel(pl: PlotterPtr; channel: CARDINAL;
                     name: ARRAY OF CHAR; c: color;
                     minValue, maxValue: REAL);
```
Sets properties of a given channel. Channels are numbered from 1 to noOfChannels. If this is not called, default properties will be used. Default properties are an empty name string, minValue = −1.0, maxValue = 1.0, colors are assigned in sequence: red, green, lightblue, cyan, magenta, lightblue.

```
PROCEDURE SetTime(pl: PlotterPtr; t: REAL);
```
Sets the current time and erases from last time to current time.

```
PROCEDURE WriteValue(pl: PlotterPtr; value: REAL; channnel: CARDINAL);
```
Extends the graph for the given channel to current time.

```
PROCEDURE WriteValues(pl: PlotterPtr; time: REAL; values: ARRAY OF REAL);
```
Sets the current time and draws all graphs.

```
PROCEDURE Activate(pl: PlotterPtr);
```
Activates and redraws the plotter. Old graphs are lost.

```
PROCEDURE Deactivate(pl: PlotterPtr);
```
Deactivates the plotter.

```
END Plotter.
```

```
DEFINITION MODULE ListHandler;

FROM SYSTEM IMPORT ADDRESS;

TYPE
  ListTypePtr; NodeTypePtr;

PROCEDURE NewList() : ListTypePtr;
```
Creates a new empty list.

```
PROCEDURE NewNode(e : ADDRESS) : NodeTypePtr;
```
Create a new node and put a pointer to the element in the node.

```
PROCEDURE InsertFirst(n : NodeTypePtr; VAR l : ListTypePtr);
```
Put node n first in the list l.

```
PROCEDURE InsertLast(n : NodeTypePtr; VAR l : ListTypePtr);
```
Put node n last in the list l.

```
PROCEDURE FirstNode(l : ListTypePtr) : NodeTypePtr;
```
Returns a pointer to the first element in list l.

```
PROCEDURE LastNode(l : ListTypePtr) : NodeTypePtr;
```
Returns a pointer to the last node of the list.

```
PROCEDURE PredNode(n : NodeTypePtr) : NodeTypePtr;
```
Returns a pointer to the preceeding node.

```
PROCEDURE SuccNode(n : NodeTypePtr) : NodeTypePtr;
```
Returns a pointer to the succeeding node.

```
PROCEDURE IsEmptyList(l : ListTypePtr) : BOOLEAN;
```
Returns TRUE if the list is empty.

```
PROCEDURE IsFirstNode(n : NodeTypePtr) : BOOLEAN;
```
Returns TRUE if n points to the first node in a list.

```
PROCEDURE IsLastNode(n : NodeTypePtr) : BOOLEAN;
```
Returns TRUE if n points to the last node in a list.

```
PROCEDURE ElementPtr(n : NodeTypePtr) : ADDRESS;
```
This routine is used to get the actual element from the node. An example:

```
N1 := NewNode(Obj);
InsertFirst(N1, List);
N2 := FirstNode(List);
E1 := ElementPtr(N1);
```

```
E2 := ElementPtr(N2);
```
Now E1 and E2 points to the same element, namely Obj.

```
PROCEDURE RemoveNode(n : NodeTypePtr; l : ListTypePtr);
```
Removes a node from a list.

```
PROCEDURE ClearList(l : ListTypePtr);
```
Deletes an entire list.

```
END ListHandler.
```

```
DEFINITION MODULE Signals;

FROM Graphics IMPORT point;

TYPE
  FunctionType = (Sin, Pulse, Ramp, Step, Random);

PROCEDURE InitSignals(SignalPriority : CARDINAL);
```
Initiates the signal generator module.

```
PROCEDURE MakeRefSignal(SignalName : ARRAY OF CHAR;
                        FunctionName : FunctionType;
                        Omega : REAL);
```
Creates a signal from the generator.

```
PROCEDURE GetRefValue(SignalName : ARRAY OF CHAR;
                      VAR Value : REAL);
```
The value of SignalName gets assignad to Value.

```
PROCEDURE ChangeOmega(SignalName : ARRAY OF CHAR; Omega : REAL);
```
Changes the value of omega (the frequency) of the signal SignalName.

```
PROCEDURE ChangeDelta(Delta : CARDINAL);
```
Changes the frequency of the generation of new values. Delta is given in ms.

```
PROCEDURE ChangeDirection(SignalName : ARRAY OF CHAR;
                          Direction : REAL);
```
Changes the slope of the ramp function.

```
PROCEDURE ChangeFunction(SignalName : ARRAY OF CHAR;
                         Function : FunctionType);
```
Changes the function of the signal belonging to SignalName.

```
END Signals.
```

```
DEFINITION MODULE TextWindows;
```
This module defines a kind of window for simple input and output of text. The text scrolls vertically when lines are added below the last visible line. The number of visible lines and columns depends on the window size, given in screen coordinates.

```
TYPE WindowType;

PROCEDURE MakeTextWindow(xlo, ylo, xhi, yhi: REAL) : WindowType;
```
Creates a new text window

```
PROCEDURE WriteString(W: WindowType; Line: ARRAY OF CHAR);
```
Adds a string to the end of current line.

```
PROCEDURE WriteReal(W: WindowType; Value: REAL; Width: CARDINAL);
```
Prints a real number at the end of current line.

```
PROCEDURE WriteInteger(W: WindowType; Value: INTEGER; Width: CARDINAL);
```
Prints an integer number at the end of current line.

```
PROCEDURE NewLine(W: WindowType);
```
Makes current line visible and start a new one.

```
PROCEDURE WriteLine(W: WindowType; Line: ARRAY OF CHAR);
```
WriteString and NewLine

```
PROCEDURE ReadLine(W: WindowType; Prompt: ARRAY OF CHAR;
                   VAR Result: ARRAY OF CHAR);
```
Prompt user for a string of character. The window becomes blocked for output until thi procedure has finished.

```
PROCEDURE ReadReal(W: WindowType; Prompt: ARRAY OF CHAR;
                   VAR Value: REAL): BOOLEAN;
```
Prompt user for a number. Returns FALSE and Value=0.0 if no valid number was read. The window becomes blocked for output until thi procedure has finished.

```
END TextWindows.
```

ISSN 0280–5316
ISRN: LUTFD2/TFRT--7510--SE

A real-time kernel with graphics support modules

Lars Nielsen, Leif Andersson,
Mats Andersson, and Karl-Erik Årzén

Department of Automatic Control
Lund Institute of Technology
August 1994

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

Document name
Report

Date of issue
August 1994

Document Number
ISRN LUTFD2/TFRT--7510--SE

Author(s)
Lars Nielsen, Leif Andersson, Mats Andersson, and Karl-Erik Årzén

Supervisor

Sponsoring organisation

Title and subtitle
A real-time kernel with graphics support modules

Abstract

This text presents the real-time kernel and the real-time graphics support modules used in the course Real-Time Systems. The majority of the text consists of commented Modula-2 definition modules.

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
85

Recipient's notes

Security classification

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Fax +46 46 110019, Telex: 33248 lubbis lund.

Preface

This text presents the real-time kernel and the real-time graphics support modules used in the course Real-Time Systems. Chapter 1 gives an overview of the real-time kernel used in the course. Chapters 2 and 3 consist mainly of commented Modula-2 definition files for the kernel layer and the graphics support modules. In earlier versions of the course this material has been a part of the course text "Computer Implementation of Control Systems" written by Lars Nielsen.

Contents

1

Implementation of a Real-Time Kernel

L. Andersson

GOAL: To give an overview of the real-time kernel used in the course.

This chapter 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 chapter 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 chapter describe the hardware interaction, such as clock, keyboard, etc.

1.1 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.

1.2 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.

1.3 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 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 chapter.

Monitor Timing

The times required for the Kernel itself and for some important kernel operations are shown in table 1.1. 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.

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 1.1 Timing for the kernel and some operations.

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.

1.4 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.

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.
```

1.5 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.
```

1.6 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.
```

1.7 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 the data type `MonitorGate` together 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.

```
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

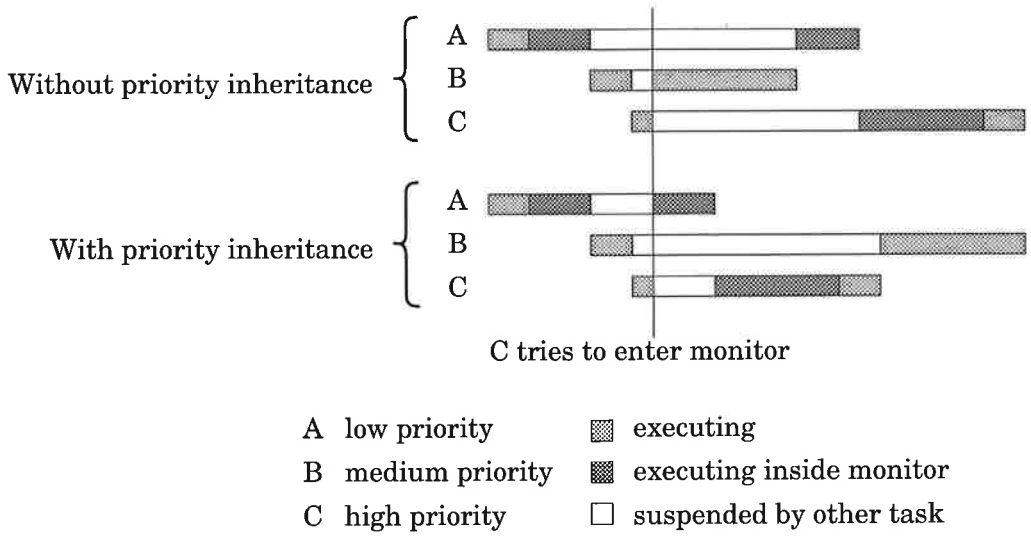

Figure 1.1 Priority inversion when two processes contend for the same monitor.

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 1.1 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.
```

1.8 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.
```

1.9 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.
```

1.10 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.
```

1.11 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.
```

1.12 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.
```

1.13 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.
```

2

The Real-Time Kernel Layer

L. Nielsen and L. Andersson

GOAL: To point out some design considerations when implementing and using a real-time kernel.

A general knowledge of Modula-2 and of principles for a real-time kernel layer is assumed. Our specific solution will therefore be briefly presented and commented. The modules presented here also include modules not directly related to real-time, but we have found it conceptually natural to group them in this basic layer of software.

2.1 Function and Implementation of the Modules

The function and implementation of the kernel is very similar to the version presented in the basic course in real-time programming. The kernel uses pre-emptive scheduling, and there is one single ready queue, where all processes ready to run are sorted in priority order. Other queues are associated with semaphores, events, and so on, where processes are waiting. Calls to the primitives, such as `Wait` result in that the process record may be moved from one queue to another. The processes are always sorted on insertion in a queue. The scheduler transfers the first process in the ready queue to running.

Some comments on the use of the primitives

One should distinguish between primitives that implement a concept and primitives that can be used to implement a concept. An example of the first type is the `Semaphores` module, which really implements the semaphore concept. An example of the second type is the module `Monitors`. The monitor concept can be implemented by programming discipline by having a call to `EnterMonitor` first in each monitor procedure and to `LeaveMonitor` last, and by using the `MonitorEvent` to implement conditional critical regions. Note that `EnterMonitor` and `LeaveMonitor` operate on a `MonitorGate`, which is conceptually a semaphore, but it has the added feature that the process occupying the monitor is raised to the priority of the highest of the processes waiting to get in.

Graphics

The module `Graphics` gives an example of the central ideas in real-time graphics. The basic data structure is `VirtualScreen` (accessed via a variable of type `handle`). A virtual screen consist of two objects; a `Window` and a `ViewPort`, both of type `rectangle`. A window is a rectangle where the x- and y-axis represent user variables e.g. physical quantities, whereas a viewport is a rectangle where the x- and y-axis represent coordinates on the computer screen. The graphics system automatically transforms coordinates between the window and the viewport coordinate systems in a virtual screen once they have been defined. The key idea is thus that the user of the module only has to think in user coordinates in the window. Read and write commands are done in window coordinates and the system automatically transforms to viewport coordinates so that the result appears on the computer screen.

Modula-2

Three short comments will be made about developing software in general in Modula-2. A module can be regarded as a language concept, not just as a fix used for separate compilation. Sometimes it may be advantageous to use internal modules in other modules to structure the code. This trend may be compared with the history of the procedure concept. In early Fortran a procedure was a separate compilation unit, but in languages following thereafter like Algol or Pascal, procedures were used in the same compilation unit, even nested etc. The second and third comments are about hiding or leaving implementation details open. The technique to hide information is to use hidden types. See the declarations in `Graphics`, `Monitors`, and `Semaphores` for examples. This means that the internal structure of the these objects are inaccessible to a user of the modules, and that the only way to operate on the objects are via the routines declared in the definition modules. A dual to hiding details is to leave them out of a module. The technique to do this in Modula-2 is to use procedure-type parameters. The general idea is to provide more universal units than if all details were included in the module. One example is hardware dependent procedures in the kernel itself. The machine dependent clock procedure is installed using a procedure-type parameter, and the kernel can be kept clean and easier to port to other computers. Another example, on a higher level, is to write a complete controller framework except for the control algorithm. The control algorithm is then installed by the user without having to change anything else in the system. This idea is extended further in Chapter 6.
Modula-2 provides coroutines and primitives like `TRANSFER` and `IOTRANSFER` to handle concurrent activities. In other languages, like for example Pascal, C, or C++, similar primitives have to be implemented. An earlier version the present kernel was done in Pascal for the LSI-11 computer. In that case, the necessary basic nucleus software for interrupt disabling/enabling, and procedures analogous to `TRANSFER` and `IOTRANSFER` consisted of four pages of assembler code. The fact that Modula-2 already contains such primitives is thus not crucial, since the work to extend other languages with similar capabilities is not overwhelming.

Portability

The real-time kernel modules are one example of a software layer. When designing such a layer one should try to find units or layers that have a long life time. The present kernel was first developed for an LSI-11 computer. It was later transferred to an IBM PC, and has recently been ported to a Sun-VME system. It has also been used on other machines outside the Department. The efforts to transfer the kernel to new machines have been limited. The major part is written in a high level language, and the machine dependent

parts can be well isolated e.g. by the use of procedure parameters as described above. The present kernel has thus survived three hardware generations.

Implementation details

Processes are declared as procedures. Such procedures should not be declared inside other procedures. Processes never terminates, so the last `END` of the procedure should never be reached. There is a minor difference between the IBM PC version and the Sun-VME version in that LONGREAL is used instead of REAL on Sun-VME.

2.2 Research topics

A current trend in the research community is to study what is called hard real-time problems. One example will be presented to give a flavor of that field. Consider the well known dining philosophers problem, which is an idealized problem in scheduling. A solution is feasible if every philosopher eventually will eat. The hard real-time version of this problem is called the dying dining philosophers problem. The new element is to consider time, and to say that a philosopher dies if he is not able to eat within certain time limits. Two main versions of the problem are if there is a waiter or not, i.e. to consider centralized or decentralized scheduling. There are also other extensions treating also two bottles of soy sauce, one curry, and so on. The research field is new and there seems to be few practical results so far, but the questions asked are relevant and one should be aware of this type of work.

2.3 The definition modules

The definition modules are listed on the following pages. One should also remember that the Modula-2 system is delivered with a number of modules e.g. mathlib for numerical functions.
The modules common both to the IBM PC system and to the Sun-VME system are: Console, Conversions, Events, Identifiers, IntConversions, Kernel, LexicalAnalyzer, Messages, Monitors, Semaphores, Strings. Note that LONGREAL is used instead of REAL on Sun-VME.
Modules specific to IBM PC are: AnalogIO, Graphics, RTGraph, RTMouse. Overlapping windows are not supported. The user must check the graphical layout.
Modules specific to Sun-VME are: AnalogIO, DigitalIO, MiscIO, MatComm.

```
DEFINITION MODULE Console;
```
Simple terminal input and output.

```
PROCEDURE GetChar(VAR ch : CHAR);
```
Reads one character from the Console.

```
PROCEDURE GetString(VAR s: ARRAY OF CHAR);
```
Reads a string, up to carriage return or line feed.

```
PROCEDURE CharAvailable(): BOOLEAN;
```
Returns TRUE IF a character is available, FALSE otherwise.

```
PROCEDURE PutChar(ch : CHAR);
```
Writes one character to the Console.

```
PROCEDURE PutString(s : ARRAY OF CHAR);
```
Writes a string to the Console.

```
PROCEDURE PutLn;
```
Writes a newline to the Console.

```
PROCEDURE Trap(errortext: ARRAY OF CHAR);
```
Writes a string to the Console and halts.

```
END Console.
```

```
DEFINITION MODULE Conversions;
```

Converts between numbers and their string representations. The routines for conversion between strings and integers or cardinals also occur separately in the module `IntConversions`. Only one of the modules `IntConversions` and `Conversions` is thus necessary.

```
PROCEDURE IntToString(VAR string: ARRAY OF CHAR;
                      num: INTEGER;
                      width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE CardToString(VAR string: ARRAY OF CHAR;
                       num: CARDINAL;
                       width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE StringToCard(string: ARRAY OF CHAR): CARDINAL;
```

Converts a string representation to a cardinal. Leading spaces are skipped. A leading + is allowed. If no legal cardinal can be found in the string then `MAX(CARDINAL)` is returned.

```
PROCEDURE StringToInt(string: ARRAY OF CHAR): INTEGER;
```

Converts a string representation to an integer. Leading spaces are skipped. A leading + or − is allowed and interpreted. If no legal integer can be found in the string then `-MAX(INTEGER) - 1` is returned.

```
PROCEDURE StringToReal (string: ARRAY OF CHAR): REAL;
```

Decodes a real value from an input string of characters. The syntax is permissive in the sense that the string '6' is interpreted as 6.0. Leading whitespace is permitted in the string. If no acceptable real number can be found then the value `badreal` (see below) is returned.

```
PROCEDURE RealToString(VAR string: ARRAY OF CHAR;
                       num: REAL;
      width: CARDINAL);
```

Converts a real number to a fixed point or exponent representation with `width` characters. The number is converted such that maximum accuracy is obtained. If there is enough space then a suitable fixed point representation is used. If there is not enough space then an exponent representation is used. If `width > HIGH(s)+1` then s is asterisk filled. If an exponent representation must be used and `width` is too small then the field is asterisk filled.

```
CONST badreal = 10.0E+307;

END Conversions.
```

```
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.
```

```
DEFINITION MODULE Identifiers;
```
Module to decode identifiers.

```
TYPE identset;

PROCEDURE NewIdentSet(VAR id: identset);
```
Initializes an ident set and returns a reference to it.

```
PROCEDURE BuildIdentSet(id: identset; name: ARRAY OF CHAR;
                        key: CARDINAL);
```
Inserts an identifier in an ident set and assigns a key to it.

`id` The ident set to be used.

`name` The identifier.

`key` The key to be associated with the identifier `name`. The value of `key` should be `[2..255]`.

```
PROCEDURE SearchIdentSet(id: identset;
                         name: ARRAY OF CHAR): CARDINAL;
```
Searches for an identifier and returns its key if it is found and 0 otherwise.

`id` The ident set to be used.

`name` The identifier

```
PROCEDURE SearchIdentSetAbbrev(id: identset;
                         name: ARRAY OF CHAR): CARDINAL;
```
Searches for an identifier. Any nonambiguous abbreviation of the identifier is acceptable. If the identifier is found, its key is returned. If the abbreviation is ambiguous then 1 is returned and if the identifier is not found then 0 is returned.

`id` The ident set to be used.

`name` The identifier

```
END Identifiers.
```

```
DEFINITION MODULE IntConversions;
```

Conversions between strings and cardinals or integers. The routines in this module are duplicated in the module `Conversions`, that also converts between strings and reals. Only one of the modules `IntConversions` and `Conversions` is thus necessary.

```
PROCEDURE IntToString(VAR string: ARRAY OF CHAR;
                      num: INTEGER;
                      width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE CardToString(VAR string: ARRAY OF CHAR;
                       num: CARDINAL;
                       width: CARDINAL);
```

Converts `num` to its string representation in `string`, right justified in a field of at least `width` characters. If `string` is too small to hold the representation then it is asterisk filled instead.

```
PROCEDURE StringToCard(string: ARRAY OF CHAR): CARDINAL;
```

Converts a string representation to a cardinal. Leading spaces are skipped. A leading + is allowed. If no legal cardinal can be found in the string then `MAX(CARDINAL)` is returned.

```
PROCEDURE StringToInt(string: ARRAY OF CHAR): INTEGER;
```

Converts a string representation to an integer. Leading spaces are skipped. A leading + or − is allowed and interpreted. If no legal integer can be found in the string then `-MAX(INTEGER) - 1` is returned.

```
END IntConversions.
```

```
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 TimeToReal(t: Time): REAL;
```
Returns `t` converted to a real number, expressed in milliseconds.

```
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.
```

`DEFINITION MODULE LexicalAnalyzer;`
The routines in this module are used to decode a string. The function `LexScan` decodes the next item in the string and returns a value indicating the type of the decoded item. A call to one of the procedures `LexCardinal` through `LexString` will then return the decoded value.

`TYPE LexHandle;`
A pointer type defined internally in the LexicalAnalyser.

```
TYPE LexTypes =
  (CardLex, CardIntLex, IntLex, RealLex, IdentLex, DelimLex,
   StringLex, EolnLex, EofLex, ErrorLex, RealErrorLex,
   StringErrorLex);
```
The possible results from `LexScan`. `RealErrorLex` corresponds to real overflow or underflow. `StringError` is given when the end of a string is missing.

`PROCEDURE LexInit(VAR lh: LexHandle);`
Initializes the internal data structures and returns a handle.

`PROCEDURE LexInput(lh: LexHandle; s: ARRAY OF CHAR);`
Makes the string s ready to be decoded.

`PROCEDURE LexScan(lh: LexHandle): LexTypes;`
Decodes the next item in the string which is connected with the LexHandle `lh`. The items must obey the following syntax.

⟨number⟩ :== [+|-]{⟨digit⟩}*[.{⟨digit⟩}*][⟨exponent⟩]
⟨exponent⟩ :== e|E[+|-]{⟨digit⟩}*
⟨digit⟩ :== 0| .. |9
⟨identifier⟩ :== ⟨letter⟩ {⟨letter⟩|⟨digit⟩}*
⟨letter⟩ :== a| .. |z|A| .. |Z
⟨string⟩ :== '{⟨character⟩}*'|"{⟨character⟩}*"
⟨character⟩ :== ⟨all characters defined in the ASCII table⟩

`PROCEDURE LexCardinal(lh: LexHandle): CARDINAL;`
Returns the decoded value if the result from `LexScan` is either `CardLex` or `CardIntLex`.

`PROCEDURE LexInteger(lh: LexHandle): INTEGER;`
Returns the decoded value if the result from `LexScan` is either `CardIntLex` or `IntLex`.

`PROCEDURE LexReal(lh: LexHandle): REAL;`
Returns the decoded value if the result from `LexScan` is in `CardLex .. RealLex`.

`PROCEDURE LexIdent(lh: LexHandle; VAR s: ARRAY OF CHAR);`
Returns the identifier in s if the result from `LexScan` is `IdentLex`. All the letters `('a'..'z')` are converted to `('A'..'Z')`.

`PROCEDURE LexDelim(lh: LexHandle): CHAR;`
Returns the delimiter if the result from `LexScan` is `DelimLex`.

`PROCEDURE LexString(lh: LexHandle; VAR s: ARRAY OF CHAR);`
Returns the delimiter if the result from `LexScan` is `DelimLex`.

`END LexicalAnalyzer.`

```
DEFINITION MODULE Messages;
```
Message Passing Routines.

```
FROM SYSTEM IMPORT ADDRESS;

TYPE
  MailBox;

PROCEDURE InitMailBox(VAR Box: MailBox; maxmessages: CARDINAL;
                      name: ARRAY OF CHAR);
```
Initializes Box. The maximum number of messages that the box can contain is maxmessages.

```
PROCEDURE SendMessage(Box: MailBox; VAR MessAdr: ADDRESS);
```
Sends the message referenced by MessAdr to Box. If the mailbox already contains the maximum number of messages then the calling process will wait. On return MessAdr = NIL.

```
PROCEDURE ReceiveMessage(Box: MailBox; VAR MessAdr: ADDRESS);
```
Receives a message from Box. The calling process is delayed if Box is empty.

```
PROCEDURE AcceptMessage(Box : MailBox; VAR MessAdr : ADDRESS);
```
Receives a message from Box. If there is a message in the box then MessAdr points to the message. If there is no message in the box then MessAdr = NIL. AcceptMessage does not delay the calling process.

```
END Messages.
```

```
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.

```
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.
```

`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!`*strut*`!` > 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.`

```
DEFINITION MODULE Strings;
```
String handling routines. For all these routines the general principle is that if a target string is too short, then the result is silently truncated. There are no run time errors.

```
PROCEDURE Length(str: ARRAY OF CHAR): CARDINAL;
```
Returns the number of characters in `str`.

```
PROCEDURE Compare(astring, bstring: ARRAY OF CHAR): INTEGER;
```
Compares `astring` and `bstring` Returns `-1`, `0` or `+1` indicating less than, equal or greater than.

```
PROCEDURE Position(pattern, source: ARRAY OF CHAR;
                   start: CARDINAL): CARDINAL;
```
Returns the position of `pattern` within `source`. The comparison starts at position `start`

```
PROCEDURE ReversePosition(pattern, source: ARRAY OF CHAR;
                          last: CARDINAL): CARDINAL;
```
Returns the position of the ¡!it last¿ occurence of `pattern` within `source`. The comparison starts `last` characters from the end and proceeds backwards.

```
PROCEDURE Assign(VAR target: ARRAY OF CHAR;
                 source: ARRAY OF CHAR);
```
Assigns `source` to `target`

```
PROCEDURE Insert(VAR target: ARRAY OF CHAR;
                 string: ARRAY OF CHAR;
                 pos: CARDINAL);
```
Inserts `string` into target at position `pos`

```
PROCEDURE Substring(VAR dest: ARRAY OF CHAR;
                    source: ARRAY OF CHAR;
                    index, len: CARDINAL);
```
Returns in `dest` a substring from `source` starting at `index` and containing `len` characters.

```
PROCEDURE Append(VAR target: ARRAY OF CHAR;
                 string: ARRAY OF CHAR);
```
Appends `string` to `target`.

```
PROCEDURE AppendC(VAR target: ARRAY OF CHAR; c: CHAR);
```
Appends the character `c` to `target`

```
PROCEDURE Delete(VAR target: ARRAY OF CHAR;
                 index, len: CARDINAL);
```
Deletes `len` characters FROM `target`, starting at position `index`

```
PROCEDURE UpperCase(VAR source : ARRAY OF CHAR);
```
Converts `source` to uppercase letters.

```
PROCEDURE LowerCase(VAR source : ARRAY OF CHAR);
```
Converts `source` to lowercase letters.

```
END Strings.
```

```
DEFINITION MODULE AnalogIO;
```
Analog input/output.

```
PROCEDURE ADIn(Channel : CARDINAL) : REAL;
```
Returns a value in the interval [-1.0..1.0], corresponding to [-10.0 V..10.0 V], from channel number Channel. Allowed channel numbers depend on the hardware, but is at least 0–3.

```
PROCEDURE DAOut(Channel : CARDINAL; Value : REAL);
```
Outputs a value in the interval [-1.0..1.0], corresponding to [-10.0 V..10.0 V], to channel number Channel. Allowed channel numbers depend on the hardware, but is at least 0–1.

```
END AnalogIO.
```

```
DEFINITION MODULE Graphics;
```
Warning! Do not use this module together with the module Terminal.

```
TYPE
  handle;
```
A pointer type defined internally in the graphics system

```
  point = RECORD
    h: REAL;     Horizontal coordinate
    v: REAL;     Vertical coordinate
  END;

  rectangle = RECORD
    CASE BOOLEAN OF
    TRUE:
       loleft: point;                Lower left corner
       upright: point;|              Upper right corner
    FALSE: xlo,ylo,xhi,yhi: REAL;    Alternate representation
    END;
  END;

  color=(black, blue,green,cyan,red, magenta, brown, white,
         grey, lightblue, lightgreen, lightcyan, lightred,
         lightmagenta, yellow, intensewhite);

  buttontype=(LeftButton, RightButton);   For the mouse buttons

  buttonset = SET OF buttontype;

PROCEDURE VirtualScreen(VAR h: handle);
```
Initializes the data structures for a virtual screen and returns a handle. The default window and viewport are $0.0 < x < 1.5$ and $0.0 < y < 1.0$. The default color for line and text is white; for fill it is black.

```
PROCEDURE SetWindow(h: handle; r: rectangle);
```
Defines the window coordinates.

h The virtual screen handle.

r The rectangle specifying the window. All real numbers are permitted as window coordinates.

```
PROCEDURE SetViewPort(h: handle; r: rectangle);
```
Positions the viewport on the screen.

h The virtual screen handle.

r The rectangle specifying the viewport.

The viewport rectangle must satisfy the screen limits $0.0 < x < 1.5$ and $0.0 < y < 1.0$.

```
PROCEDURE SetLineColor(h: handle; c: color);

PROCEDURE SetTextColor(h: handle; c: color);

PROCEDURE SetFillColor(h: handle; c: color);
```

```
PROCEDURE PolyLine(h: handle; VAR polygon: ARRAY OF point;
                   npoint: INTEGER);
```

Draws a polygon.

`h` The virtual screen handle

`polygon` The points of the polygon. The start point is `polygon[0]`. Lines are drawn using the line color of the specified handle. The VAR declaration is for efficiency only, and the actual argument is not changed.

`npoint` The number of points in polygon.

```
PROCEDURE PolyMarker(h: handle; VAR polygon: ARRAY OF point;
                     npoint: INTEGER);
```

Draws markers at the specified points. The markers are plus signs at present.

`h` The virtual screen handle

`polygon` The points where the markers are drawn. The current line color is used. The VAR declaration is for efficiency only, and the actual argument is not changed.

`npoint` The number of points in polygon.

```
PROCEDURE WriteString(h: handle; p :point; s: ARRAY OF CHAR);
```

Writes a string on the screen starting at a specified point.

`h` The virtual screen handle

`p` The starting point of the text.

`s` The text string to be written. If s contains `CHR(0)` then this is considered the end of the string.

The text is written with the text color of the specified handle. Old text is overwritten, not erased. See `EraseChar`.

```
PROCEDURE FillRectangle(h: handle; r: rectangle);
```

Fills a rectangle on the screen.

`h` The virtual screen handle.

`r` The rectangle to be filled.

The rectangle is filled with the fill color of the specified handle.

```
PROCEDURE DrawRectangle(h: handle; r: rectangle);
```

Draws a rectangle on the screen.

`h` The virtual screen handle.

`r` The rectangle to be drawn.

The rectangle is drawn with the line color of the specified handle.

```
PROCEDURE CharacterSize(h: handle; VAR width, height: REAL);
```

Returns the size of a character in the current window coordinate.

`h` The virtual screen handle.

`width` The horizontal size of a character.

`height` The vertical size, i.e. the distance between the baselines of two adjacent text lines.

```
PROCEDURE ReadString(h: handle; p :point; VAR s: ARRAY OF CHAR);
```

Reads a string from the keyboard with echoing. The procedure returns when the user pushes the Return key.

`h` The virtual screen handle.

`p` The point where echoing starts.

`s` The returned string. If it is less than `HIGH(s)` characters long then the string is delimited by `CHR(0)`.

The characters are echoed with the text color of the specified handle. The text background is the fill color of the specified handle.

```
PROCEDURE InputString(h: handle; p :point; VAR s: ARRAY OF CHAR;
                      limit: CARDINAL; VAR complete: BOOLEAN);
```

Reads a string from the keyboard with echoing. The procedure returns when the user pushes the Return key or when another process calls `StopInputString`

`h` The virtual screen handle.

`p` The point where echoing starts.

`s` The returned string. If it is less than `HIGH(s)` characters long then the string is delimited by `CHR(0)`.

`limit` The maximum number of charaters accepted and echoed.

`complete` Returned TRUE if the user pushed Return, FALSE if another process called `StopInputString`

The characters are echoed with the text color of the specified handle. The text background is the fill color of the specified handle.

```
PROCEDURE StopInputString;
```

Stops current reading by InputString, but makes the `complete` parameter return FALSE.

```
PROCEDURE EraseChar(h: handle; p: point; num: CARDINAL);
```

Erases characters, i.e. fills the area with the fill color.

`h` The virtual screen handle.

`p` The starting point of the erase.

`num` A region corresponding to num characters is filled with the fill color.

```
PROCEDURE GetMouse(h: handle; VAR p:point; VAR b:buttonset);
```

Returns the mouse state.

`h` The virtual screen handle.

`p` The mouse position.

`b` The button state. If `LeftButton IN b` then this button is pressed, and conversely for the right button.

```
PROCEDURE WaitForMouse(h: handle; VAR p: point;
                       VAR b: buttonset);
```

Waits until at least one of the buttons get pushed, then returns the mouse state. In a real time situation it is not a busy wait.

`h` The virtual screen handle.

`p` The mouse position.

`b` The button state. If `LeftButton IN b` then this button is pressed, AND conversely for the right button.

```
PROCEDURE SetMouseRectangle(h: handle; r: rectangle;
                            n: CARDINAL);
```

Inserts a rectangle in a list of rectangles to be tested in a WaitMouseRectangle or GetMouseRectangle operation. There is one list for each handle.

`h` The virtual screen handle.

r The rectangle to be inserted.

n The number to be returned if the mouse is inside r.

If the specified number n already exists in the list then no new entry is made, but the old entry gets a new rectangle value. Do not use the number 0, because 0 has a special meaning for `GetMouseRectangle`. No test is made, however. This routine does not draw anything. The drawing should be done with DrawRectangle.

`PROCEDURE WaitMouseRectangle(h: handle): CARDINAL;`
Waits until a mouse button is pressed and the cursor is inside one of the rectangles previously specified with SetMouseRectangle. The number associated with that rectangle is then returned. The entries are kept and tested in number order, and the first match found is returned. The cursor hot spot must be strictly inside the rectangle if it is to be considered a match.

`PROCEDURE GetMouseRectangle(h: handle): CARDINAL;`
If the cursor hot spot is strictly inside one of the mouse rectangles, then the corresponding number is returned, otherwise 0 is returned. See SetMouseRectangle and WaitMouseRectangle.

`PROCEDURE HideCursor;`
The internal hide/show counter is decremented. If the counter is zero after decrementation then the cursor is removed from the screen.

`PROCEDURE ShowCursor;`
The internal hide/show counter is incremented. If the counter is 1 after incrementation then the cursor is shown on the screen.

`PROCEDURE Shutdown;`
Closes down the entire graphics system and sets the screen in normal text mode.

The following procedure types and procedures are used to define which modules should handle the mouse and the keyboard. In a real time application the modules RTGraph and RTMouse will call these procedures. They should not be referenced directly by user programs.

```
TYPE
  EchoStringProc=PROCEDURE(VAR ARRAY OF CHAR, VAR BOOLEAN);
  MouseProcedureType=
        PROCEDURE(VAR buttonset, VAR INTEGER, VAR INTEGER);

PROCEDURE SetEchoString(p: EchoStringProc; q:PROC);

PROCEDURE SetMouseProcedures(GM,WM: MouseProcedureType;
                             HC, SC: PROC);

END Graphics.
```

`DEFINITION MODULE RTGraph;`
Establishes connections between the Kernel and the graphic modules.

`PROCEDURE Init;`
Initialization. There is an implicit call to `Kernel.Init`.

`END RTGraph.`

`DEFINITION MODULE RTMouse;`
Establishes connections between mouse, Kernel and the graphic modules.

`PROCEDURE Init;`
Initialization. There is an implicit call to `RTGraph.Init` and thus to `Kernel.Init`.

`END RTMouse.`

```
DEFINITION MODULE AnalogIO;

(* Analog IO via the VDAD boards from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the analog ports to operate in
   the range +/- 10V.
*)

FROM VDAD IMPORT NrOfCards;

TYPE CardType     = [1..NrOfCards];
     ChannelType  = [0..15];
     DARange      = [-2048..2047];
     IntArray     = ARRAY ChannelType OF INTEGER;
     ExpGainType  = [0..2];
     OutRangeType = [0..1];

PROCEDURE ADin  (     cardnr  : CardType;
                      channel : ChannelType;
                  VAR value   : INTEGER      );

PROCEDURE DAout (     cardnr  : CardType;
                      channel : ChannelType;
                      value   : DARange      );

PROCEDURE MultiADin (     cardnr      : CardType;
                          LowChannel,
                          HighChannel : ChannelType;
                      VAR value       : IntArray     );

PROCEDURE SetInputGain( cardnr  : CardType;
                        ExpGain : ExpGainType );
  (* ExpGain = 0, 1, 2  =>  Input gain = 1, 10, 100 *)

PROCEDURE SetOutVoltage( cardnr     : CardType;
                         channel    : ChannelType;
                         Gain,
                         UniBiPolar : OutRangeType );
  (* Outvoltage range = Vref * (1 + Gain)  *)
  (* UniBiPolar: 0 = unipolar, 1 = bipolar *)

END AnalogIO.
```

```
DEFINITION MODULE DigitalIO;

(* Digital IO via VDAD boards and the VDIN board from PEP computers.

   Initialize the VDADs in one and only one of the following ways:
   1. Call InitServoIO in the module ServoIO.
   2. Call InitResolver in the module ResolverIO.
   3. Import VDAD, call InitVDAD, and set up the control registers
      on the board. Refer to VDAD reference manual for details
   Alternatives 1 and 2 configures the digital ports to be output
   on board number one, and input on board number two.

   The VDIN board requires no initialization. To read the 16 bit
   parallell input port, call DigInput.
*)

FROM VDAD   IMPORT NrOfCards;
FROM SYSTEM IMPORT BYTE;

TYPE CardType = [1..NrOfCards];
     WireType = [0..7];
     Bit      = [0..1];
     Byte     = [0..255];

PROCEDURE DigWireOutput (     cardnr : CardType;
                              wire   : WireType;
                              value  : Bit      );

PROCEDURE DigWireInput  (     cardnr : CardType;
                              wire   : WireType;
                          VAR value  : CARDINAL );

PROCEDURE DigByteOutput (     cardnr : CardType;
                              value  : BYTE     );

PROCEDURE DigByteInput  (     cardnr : CardType;
                          VAR value  : BYTE );

PROCEDURE DigInput( VAR value : SHORTINT );
  (* value = [-32768, 32767] *)

END DigitalIO.
```

```
DEFINITION MODULE MiscIO;

(* Analog and Digital IO via one VDAD board from PEP computers. *)

FROM SYSTEM IMPORT BYTE;

PROCEDURE Init;
(* Configures the analog ports to operate in the range +/- 10V
   and the digital ports to operate as outputs (TTL-levels)  *)

(* Analog: *)

PROCEDURE ADin  (     channel : CARDINAL;    (* [0..7] *)
                  VAR value   : LONGREAL);   (*  -1.0 <= value <= 1.0 *)

PROCEDURE DAout (     channel : CARDINAL;    (* [0..3] *)
                      value   : LONGREAL);   (*  -1.0 <= value <= 1.0 *)

PROCEDURE VoltIn (    channel : CARDINAL;    (* [0..7] *)
                  VAR voltage : LONGREAL);   (*  -10.0 <= voltage <= 10.0 *)

PROCEDURE VoltOut (   channel : CARDINAL;    (* [0..3] *)
                      voltage : LONGREAL);   (*  -10.0 <= voltage <= 10.0 *)

(* Digital: *)

PROCEDURE DigOutput ( channel : CARDINAL;
                      value   : BOOLEAN );

PROCEDURE DigInput (     channel : CARDINAL;
                     VAR value   : BOOLEAN );

PROCEDURE DigByteOutput ( value  : BYTE);

PROCEDURE DigByteInput ( VAR value  : BYTE );

END MiscIO.
```

```
DEFINITION MODULE MatComm;

IMPORT SYSTEM;
(*$NONSTANDARD*)

CONST
  ProcessNameSignificance = 32;

TYPE
  Socket;
  NameString = ARRAY [1..ProcessNameSignificance] OF CHAR;
  DataType  = (char,real,longreal,complex,longcomplex);
  ErrorType = (OK,notOK,Closed);


PROCEDURE OpenSocket
  (VAR socket    : Socket;
   REF myname : ARRAY OF CHAR);
  (*
  A socket is returned to be used in subsequent calls to the procedures
  below. The 'myname' is required to make the request for a socket unique.
  It can be any string, but is typically the name of the process. The same
  string has to be given as the proc-argument to vmeio in matlab. It does
  not matter if the modula process or the unix process (i.e. matlab) is
  the first one to try to establish a new connection (after Init has been
  called).
  *)

PROCEDURE CloseSocket
  (VAR socket      : Socket);
  (*
  The socket is closed by the caller for further communication. The line
  can also be closed by the remote machine. In both cases, a Send or
  Receive request will return 'Closed'. If so, OpenSocket can be called
  again.
  *)

PROCEDURE Send
  (VAR socket  : Socket;
       nrows : CARDINAL;
       ncols : CARDINAL;
       dtype : DataType;
   REF data : ARRAY OF SYSTEM.BYTE): ErrorType;
  (*
  Send the 'data' on the open socket 'socket'. Proper values for 'nrows',
  'ncols', and 'dtype' have to be supplied (if you don't want a dump of
  the memory following the variable supplied). 'data' is however
  allowed to be bigger than the matrix specified.
  *)

PROCEDURE GetNextType
```

```
  (VAR socket       : Socket;
   VAR nrows : CARDINAL;
   VAR ncols : CARDINAL;
   VAR dtype : DataType): ErrorType;
  (*
  If a new data message is available, the head of the message is read and
  the type of the matrix is returned in nrown, ncols, and dtype. To
  prevent reading the head again in an additional call (without
  Receive in between), and to save some computations, some extra
  information is stored in 'socket'. To allow update this private
  information, 'socket' is also VAR declared. If no data is available,
  an Await for data on 'socket' is performed. If data to be received is
  of fixed type (or size), Receive can be called directly.
  *)

PROCEDURE Receive
  (VAR s    : Socket;
   VAR nrows : CARDINAL;
   VAR ncols : CARDINAL;
   VAR dtype : DataType;
   VAR data : ARRAY OF SYSTEM.BYTE): ErrorType;
  (*
  If not done already for the next message, GetNextType is called.
  This means 'nrows', 'ncols', and 'dtype' will be the same as if
  GetNextType were called. The 'data'-matrix is allowed to be bigger
  then required to store the data. If the 'data'-matrix is to small,
  as much as possible is stored in 'data'. The rest is read in and
  then deallocated. In this case 'notOK' is returned.
  *)

(*PROCEDURE Receive2
  (VAR socket    : Socket;
   VAR nrows : CARDINAL;
   VAR ncols : CARDINAL;
   VAR dtype : DataType;
   VAR data : ARRAY OF SYSTEM.BYTE;
       dsize : CARDINAL): ErrorType;*)
  (*
  As Receive, but the size of the data has to be explicitly given.
  To be used for dynamic variables with size unknown at compile time.
  *)

PROCEDURE Init;

END MatComm.
```

3

Real-Time Graphics Support Modules

M. Andersson

GOAL: To give principles and documentation for a user interface event handler and support modules at a layer on top of the real-time kernel.

An implementation of a control system almost always includes a number of components regarding interaction, such as presenting values, plotting signals, creating signals, and others. Since the functions needed are similar between different applications, it is possible to provide a set of support modules to simplify the use of the basic real-time and graphics primitives, like those in the previous chapter. The advantage of having such support modules can be quantified by comparing the example given in Section 2.2 with a similar program implemented using only the real-time kernel layer. The code size is reduced four times, and the work to develop the program is reduced considerably more. The function and implementation of the library modules are described in Sections 2.1. An example of the use of the routines in a real-time program is given in Section 2.2, and the definition modules of the routines are listed thereafter.

Main principles

A control system can be decomposed in two main parts: the controller and the operator's interface. The controller subsystem includes in this context simple regulators, supervisory control systems, fault diagnosis and any other subsystem interacting with the controlled process. The operator's interface is the subsystem handling all human–machine interaction. It is desirable to be able to separate the design and definition of the operator's interface from the controller subsystem as far as possible.

The task of the operator's interface is to respond on input events from the operator and to execute them as commands to the controller subsystem. It is also responsible for presenting data from the controller and process to the operator. The program modules presented in this chapter are designed to accept operator commands in form of mouse clicks and keyboard inputs from a single computer or a terminal.

The operator's interface could be designed such that every possible action from the operator has it's own designated process, monitoring that particular event and responding

by an appropriate action. This approach leads to many process. An alternative approach is taking advantage of the facts that the operator is relatively slow in producing events and that commands can be executed comparably quickly. Therefore, it is possible to use a single process, called an event handler, waiting for any possible event, identifying the event and taking appropriate action. This later approach is used in the modules presented in this chapter.
The communication between the operator's interface and the controller subsystems should be done in a uniform way. The method used here is called *callback* procedures. A callback procedure is defined by the client, in this case the controller subsystem, and registered by an interactor object. When the operator performs an input operation the event handler calls the callback procedure connected with the involved interactor. The client can then, by means of the callback procedure, extract useful information from the interactor object which is passed as an argument to the callback procedure.
While a callback procedure is executing the operator's interface is blocked and does not respond to additional events. It is client's responsibility to provide callback procedures which returns without unnecessary delay.
More details about the event handler and about specialized interactor objects are presented in the following sections.

Implementation details

The system is written in Logitech's Modula-2 and is intended to be used on IBM-AT compatible machines with EGA and Microsoft Mouse. The real-time kernel layer described in the previous chapter is used for the implementation.
All coordinates for windows, menus etc. are given in screen coordinates, i.e., $0 \leq x \leq 1.5$ and $0 \leq y \leq 1.0$.
Opaque (hidden) data types have been used to implement buttons, bargraphs, lists, menus, and plot windows. This means that the internal structure of these objects are inaccessible to a user of the modules. The only way to operate on the objects is via the routines declared in the definition modules. This is the Modula-2 way to implement abstract data types.

3.1 The Event Handler

All operator interaction modules described in this chapter are based on two fundamental modules called `MouseEvent` and `TextArea`. The former module is the heart of the operator's interface. It contains a process which waits for the next mouse or keyboard event, issued by the operator, and executes an appropriate event handling procedure.
The Event Handler allows clients to specify click sensitive areas on the screen. The click sensitive area is an object called `MouseArea` defined in module `MouseEvent`. When a mouse area is created, the client specifies, in addition to the screen area, a *callback* procedure. A callback procedure is a procedure defined by the client and invoked by the event handler whenever the operator clicks the mouse within the sensitive area. The callback procedure must be defined with a parameter which is a pointer to a mouse area. When the event handler has detected a mouse event in a sensitive area, it looks up the mouse area object in an internal list of all active mouse areas. Then it invokes its callback procedure with a pointer to the mouse area object itself as argument.
A mouse area object also contains a pointer of type `ADDRESS` to any kind of data specified by the client. The pointer is called the *user's pointer* and it can be accessed by the client defined callback procedure and converted to the correct pointer type. This makes it possible to define hierarchical interaction objects. If a higher level interaction object creates one or many mouse areas the user's pointer of these are set to point at the object

itself. The callback procedures are then designed so that they extract the pointer, convert it to a pointer to the high level object and do appropriate manipulations of its data structure.
The interaction between a mouse area object and a client object may appear complex and difficult to understand at the first glance but the following example should hopefully throw some light. Refer to the definition of the MouseEvent module given in the end of this chapter. Assume we want to create controller objects where each controller has its own start button on the screen. The controller module defines a callback procedure and a procedure for creating controller objects which also creates the associated start button.

```
MODULE Controller;
TYPE ContrPtr = POINTER TO ContrData;
     ContrData = RECORD ... END;

PROCEDURE CreateController(VAR newContr: ContrPtr);
VAR startButton: MouseAreaPtr;
    area: rectangle;
BEGIN
  NEW(newContr);
  ...
  (* Create interaction object and give a pointer to this controller
     object as the user's pointer:                                    *)
  CreateMouseArea(startButton, area, StartButtonCB, newContr);
  ...
END CreateController;

PROCEDURE StartButtonCB(ma: MouseAreaPtr) (* Callback *)
VAR contr: ContrPtr;
BEGIN
  (* Get the user's pointer and convert it to its correct type: *)
  contr := ContrPtr(GetUsersPtr(ma));
  ...
  (* use contr to access controller data and procedures *)
END StartButtonCB;
```

Mouse areas can be deactivated and activated again. A deactivated mouse area does not respond to mouse clicks. If several mouse areas overlap on the screen it is the last activated area under the mouse which receives a click event.
The event handler is based on two identical processes where one of them is always waiting for a mouse event interrupt. The reason for using two processes is that a process may be blocked waiting for text input. Text input objects, described below, are also handled by the event handler. When one process is blocked by a text input waiting for characters, the other process can still respond on mouse events.

Text input objects

The TextArea module is closely related to the MouseEvent module. It defines text area objects which can be used for character inputs from the operator. A text area is an object based on a mouse area. When the operator clicks in an active text area it starts reading and echoing characters from the keyboard. When the operator pushes the Return key, the string is completed and a callback procedure associated with the text area object is invoked by the event handler.
The callback procedure of a text area is invoked with a pointer to the text area object itself as a parameter. The client can then access the completed string and get a pointer

to his own data in the same way as for mouse areas.
Only one text area can do active character reading at a time. If the operator clicks in another text area while one is waiting for characters, the first one will be interrupted and the focus will change to the new text area. The interrupted text area will invoke its callback with the incomplete string as current text. It is possible for the callback to query the text area object and check if it was interrupted or completed by the user.
A special kind of text area called *numerical input* is also defined in the `TextArea` module. A numerical input tries to interpret a completed string as a real number and calls the callback procedure only if this is possible.
Text areas can be deactivated and activated in the same way as mouse areas.

Drawing interaction objects

Neither mouse area nor text area objects draw anything on the screen. The defined mouse sensitive areas are invisible. A special module is available for drawing buttons and frames suitable for giving mouse areas and text input fields an appearance on the screen. The module is called `Draw`.
For example, in order to create a mouse button on the screen, start by defining the screen area where you want the button by defining a `rectangle`. Use `CreateMouseArea` to make the region mouse sensible and give it a suitable callback procedure. Then use `DrawButton` or `DrawDefaultButton` with the same rectangle as parameter. If you later want to get rid of the button you must deactivate the mouse area and fill the rectangle with the background color.
`Draw` does not interact with the event handler or text area modules. This makes it easy for the user to define his own drawing routines and make his own fancy graphical layout.

Easy interactors

A module called `Easy` defines a set of interaction objects designed to be specially easy to use. They are not designed to be extendable and reused in the same way as other interactor modules described below. With `Easy` it is possible to create buttons, text inputs, number inputs, and bargraphs. A single procedure call creates the mouse sensitive area and draws the object on the screen. Easy objects cannot be deactivated or accessed in other way. They only respond to operator actions by calling the specified callback procedure.

Other interactors

A set of modules defining different kinds or interactor objects are available. They are all designed to be useful as stand alone interactors or as parts in other user defined interactors. The following modules are available.

Bargraph is a device for display and input of numeric values. Bargraphs can be created with a horizontal or a vertical layout.

Menu is a device for doing multiple choice selections. A menu can work as a radio button or as an array of single buttons.

NumMenu is a form with multiple numeric input fields and an enter button. The operator can change individual fields but the callback procedure is not called until the enter button is clicked.

Plotter is a device for plotting up to six signals against a common horizontal time axis.

Other useful modules

`ListHandler` and `Signals` are two modules that are not designed specially for creating operator interfaces but are generally useful. They do not depend on the event handler modules. The `ListHandler` module handles a doubly linked non-circular list with a list head. The list handling is done so that it is independent of the type of the element that is stored in the nodes. This is achieved by using the data type `ADDRESS` in the nodes. This also means that the elements must be referred via a pointer, see the example in the definition module.
The routines in the `Signals` module are used for generating time signals. Any number of signals can be generated, and each signal is identified by a text string. The signals which can be generated are of the types `Sin`, `Step`, `Pulse`, `Ramp`, and `Random`. Signals of different types can be generated at the same time. The user of this module does not have to handle time explicitly. When `GetRefValue` is called the value of the signal at that time is returned. The value of all signals are in the interval [0 1]. There is no way of changing the amplitude and offset of the signal within the module, so that must be taken care of by the user program. The runs a single process generating all signals.
Module `TextWindows` makes it possible to create a window for simple output and input of text and numbers. The text scrolls vertically when lines are added below the last visible line. The number of visible lines and columns depends on the window size, given in screen coordinates.

3.2 An Example

The implementation of a PI controller is used as an example of the use of the modules. The program starts on the next page. It is possible to change the parameters (k and T_i) of the controller. The reference signal is a square wave. The amplitude and frequency of the reference signal can be changed. All parameters are changed by using a numerical menu. The reference value, the control signal, the process value, and the value of the integrator are plotted in the plot window during the operation of the controller. The program halts when the exit button is clicked. A similar program, that was implemented using only the real-time kernel layer, was 16 pages long.

```
MODULE Regul;

IMPORT MouseEvent;
FROM Semaphores IMPORT Semaphore, InitSem, Wait, Signal;
FROM AnalogIO IMPORT ADIn, DAOut;
FROM Graphics IMPORT point, rectangle, color, ShowCursor;
FROM Kernel IMPORT Time, CreateProcess, SetPriority, IncTime,
                   WaitUntil, CurrentTime, TimeToReal;
FROM Monitors IMPORT MonitorGate, InitMonitor, EnterMonitor,
                     LeaveMonitor;
FROM Signals IMPORT InitSignals, ChangeOmega, ChangeDelta,
                    ChangeFunction, GetRefValue, FunctionType,
                    MakeRefSignal, ChangeDirection;
FROM Plotter IMPORT PlotterPtr, CreatePlotter, SetChannel,
                    SetTime, WriteValue;
FROM NumMenu IMPORT CreateNumMenu, NumMenuPtr, SetEntry, GetValues;

FROM Easy IMPORT EasyButton;

CONST KInit   = 5.0;
      TiInit  = 10.0;
      AmpInit = 0.1;

VAR Exit : Semaphore;
    nm : NumMenuPtr;
    Plt : PlotterPtr;
    pos : point;
    area: rectangle;
    RegPar : RECORD
               Mutex : MonitorGate;
               K, Ti, Amp : REAL;
             END;
(*-----------------------------------------------------------*)
PROCEDURE GetRegPar(VAR p1, p2, p3 : REAL);
BEGIN
  WITH RegPar DO
    EnterMonitor(Mutex);
    p1 := K;
    p2 := Ti;
    p3 := Amp;
    LeaveMonitor(Mutex);
  END;
END GetRegPar;
(*-----------------------------------------------------------*)
PROCEDURE SetRegPar(p : ARRAY OF REAL);
BEGIN
  WITH RegPar DO
    EnterMonitor(Mutex);
    K := p[0];
    Ti := p[1];
    Amp := p[2];
```

```
    LeaveMonitor(Mutex);
  END;
  ChangeOmega("Ref", p[3]);
END SetRegPar;
(*-----------------------------------------------------------*)
PROCEDURE InitRegPar;
BEGIN
  WITH RegPar DO
    InitMonitor(Mutex, "Mutex");
    K := KInit;
    Ti := TiInit;
    Amp := AmpInit;
  END;
END InitRegPar;
(*-----------------------------------------------------------*)
(* Process *) PROCEDURE RegulProcess;
CONST h = 50; offset = 0.5;
VAR t : Time;
    amp, v, u, y, yref, e, i, k, ti : REAL;
BEGIN
  SetPriority(10);
  CurrentTime(t);
  i := 0.0;
  LOOP
    GetRefValue("Ref", yref);
    GetRegPar(k, ti, amp);
    yref := 2.0*amp*(yref - 0.5) + offset;
    y := ADIn(1);
    e := yref - y;
    v := k*e + i;
    u := v;
    IF v > 1.0 THEN
      u := 1.0;
    ELSIF v < 0.0 THEN
      u := 0.0;
    END;
    DAOut(1, u);
    i := i + k*e*FLOAT(h)/(1000.0*ti) + FLOAT(h)/(1000.0*ti)*(u - v);

    SetTime(Plt,TimeToReal(t)/1000.0);
    WriteValue(Plt, i, 1);
    WriteValue(Plt, u, 2);
    WriteValue(Plt, yref, 3);
    WriteValue(Plt, y, 4);

    IncTime(t, h);
    WaitUntil(t);
  END;
END RegulProcess;
(*-----------------------------------------------------------*)
PROCEDURE ExitButtonCB(p: point); (* Callback for Exit button *)
```

```
BEGIN
  Signal(Exit);
END ExitButtonCB;
(*-----------------------------------------------------------------*)
PROCEDURE NumMenuCB(nm: NumMenuPtr); (* Callback for parameter menu*)
VAR data: ARRAY [1..4] OF REAL;
BEGIN
  GetValues(nm,data);
  SetRegPar(data);
END NumMenuCB;
(*-----------------------------------------------------------------*)
BEGIN
  MouseEvent.Init(20);
  InitSem(Exit,0,"ExitSem");

  area.xlo := 0.05; area.xhi := 1.45;
  area.ylo := 0.55; area.yhi := 0.95;
  CreatePlotter(Plt, area, 4, 30.0, white, green, TRUE, "");
  SetChannel(Plt, 1, "i", cyan, -1.0, 1.0);
  SetChannel(Plt, 2, "u", lightblue, -1.0, 1.0);
  SetChannel(Plt, 3, "yref", lightcyan, -1.0, 1.0);
  SetChannel(Plt, 4, "y", blue, -1.0, 1.0);

  pos.h := 0.05; pos.v := 0.2;
  CreateNumMenu(nm, pos, 4, 10, grey, red, NumMenuCB, NIL, "Parameters");
  SetEntry(nm, 1, "K", KInit);
  SetEntry(nm, 2, "Ti", TiInit);
  SetEntry(nm, 3, "Amplitude", AmpInit);
  SetEntry(nm, 4, "Frequency", 0.5);

  area.xlo := 1.35; area.xhi := 1.45;
  area.ylo := 0.05; area.yhi := 0.15;
  EasyButton(area, red, "Exit", ExitButtonCB);

  InitSignals(20);
  MakeRefSignal("Ref", Step, 0.5);
  ChangeDelta(50);

  InitRegPar;
  CreateProcess(RegulProcess, 1000, "Regul");
  ShowCursor;

  Wait(Exit);
END Regul.
```

3.3 The definition modules

The definition modules are given on the following pages in the following order: MouseEvent, TextArea, Draw, Easy, Bargraph, Plotter, Button, Menu, NumMenu, ListHandler, Signals.

```
DEFINITION MODULE MouseEvent;
```

Module MouseEvent is a handler for mouse events based on callback procedures. It serves as the basis for other modules providing specialized objects for user interaction.

This module allows click sensitive regions, called mouse areas, to be defined on the screen. The user should provide a callback procedure for each mouse area. This module reacts on every mouse event in an active region and calls the corresponding callback procedure.

Procedures in this module don't draw anything. Use procedures in Draw, or design your own, and draw things on top of the mouse areas.

Standard screen coordinates used in this moudule are: `0.0 <= x <= 1.5; 0.0 <= y <= 1.0`

The event handler is blocked while a callback procedure is executing. This means that in order to respond on quick consecutive mouse events, callback procedures should be reasonably quick to execute. Callback procedures are executed with the priority given to the Init procedure.

This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle;

TYPE MouseAreaPtr;
     EventProcType  = PROCEDURE(MouseAreaPtr); Callback procedure

PROCEDURE Init(priority: CARDINAL);
```

Inits Kernel, RTMouse, and this event handler. Two identical Eventhandler processes are created so that if one is locked by a reading characters from the keyboard, the other can still handle buttons. The event handler processes will run with the given priority.

```
PROCEDURE CreateMouseArea(VAR newMouseArea : MouseAreaPtr;
                          Area             : rectangle;
                          LeftMouseProc,
                          RightMouseProc   : EventProcType;
                          UsersPtr         : ADDRESS);
```

Creates and activates a click sensitive area on the screen that when clicked will cause Left/RightMouseProc to be called. Returns a pointer to the new object.

```
PROCEDURE Deactivate(ma: MouseAreaPtr);
```

Deactivates a mouse sensitive area.

```
PROCEDURE Activate(ma: MouseAreaPtr);
```

Makes the mouse area sensitive to events.

```
PROCEDURE GetMousePoint(ma: MouseAreaPtr; VAR p: point);
```

Gets point of last mouse click.

```
PROCEDURE GetUsersPtr(ma: MouseAreaPtr) : ADDRESS;
```

Returns the users's pointer of the mouse area

```
PROCEDURE Dispose(ma: MouseAreaPtr);
```
Disposes a mouse area object.

```
PROCEDURE DeactivateArea(InArea: rectangle);
```
Deactivates all mouse sensitive areas inside InArea.

```
END MouseEvent.
```

```
DEFINITION MODULE TextArea;
```

Module TextArea is based on module MouseEvent. It is used for defining text input areas on the screen.

Module MouseEvent must be initialized before any procedure in this module is called.

When a text area is clicked it starts accepting characters from the keyboard. Input can be ended by the Return key or by clicking on another text input. Only one text area at a time can wait for input. A callback procedure is called when input is ended with Return or interrupted by another text input.

This module also supports a related kind of objects specialized for numeric inputs, created by CreateNumInput.

This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

Modified by Tord Bjorsne E90, 940822.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;
FROM MouseEvent IMPORT MouseAreaPtr;

CONST MaxTextLength = 80;
      HighlightColor = lightred;

TYPE TextAreaPtr;
     TextColorPtr   = POINTER TO TextColors;

     TextColors = RECORD
       AreaColor,                    Color of the inside area
       TextColor,                    Color of the text
       EditColor   : color;          Color of the text during editing
     END;

     TextProcType = PROCEDURE(TextAreaPtr); Callback

PROCEDURE CreateTextArea(VAR newTextArea: TextAreaPtr;
                         loleft: point;
                         width: CARDINAL;
                         AreaColor, TextColor: color;
                         Callback: TextProcType;
                         UsersPtr: ADDRESS);
```

Creates and activates a click sensitive text input field on the screen. When the area is clicked it will start accepting characters. When editing is finished, the user's callback procedure DoTextProc will be called for further processing. The user may let UsersPtr point to his data. A pointer to the new TextArea object is returned.

```
PROCEDURE Activate(ta: TextAreaPtr);
```

Activates the text area

```
PROCEDURE Deactivate(ta: TextAreaPtr);
```

Deactivates the text area

```
PROCEDURE TerminateReading(ta : TextAreaPtr);
```
Terminate input as if the user has selected another TextArea.

```
PROCEDURE GetTextColors(ta: TextAreaPtr) : TextColorPtr;
```
Returns a pointer to color data of a text area. Can be used for changing the colors. Changing colors while the text area is doing text input might give strange results.

```
PROCEDURE GetUsersPtr(ta: TextAreaPtr) : ADDRESS;
```
Return the user's pointer.

```
PROCEDURE GetTextPosition(ta: TextAreaPtr; VAR pos: point);
```
Return the point where the text starts.

```
PROCEDURE Interrupted(ta: TextAreaPtr) : BOOLEAN;
```
Returns true if the last text input was interrupted by user or by TerminateReading.

```
PROCEDURE IsReading(ta: TextAreaPtr) : BOOLEAN;
```
Returns TRUE is the text area is currently waiting for input.

```
PROCEDURE GetText(ta: TextAreaPtr; VAR text: ARRAY OF CHAR);
```
Returns the current text string.

```
PROCEDURE PutText(ta: TextAreaPtr; VAR text: ARRAY OF CHAR);
```
Sets current string and write it in text area. Writing an empty string will erase the area. The text argument is not changed — VAR is for efficiency only.

```
PROCEDURE DrawText(ta : TextAreaPtr);
```
TB 931202 Displays the text in the input buffer.

```
PROCEDURE Dispose(ta: TextAreaPtr);
```
Dispose the text area.

```
PROCEDURE ActiveTextArea(position: point; width: CARDINAL;
                         VAR area: rectangle;
                         VAR textpoint: point);
```
Given the position of the lower left corner and the width of a text area, compute the active area and the text input position.

```
PROCEDURE CreateNumInput(VAR newTextArea: TextAreaPtr;
                         loleft: point;
                         width: CARDINAL;
                         AreaColor, TextColor: color;
                         Callback: TextProcType;
```

```
                                UsersPtr: ADDRESS);
```

Creates a numeric input text area. A numeric input area is similar to a text area with the difference that it only calls the callback procedure when the current text is a valid number.

All procedures valid for text area are also valid for numeric input.

```
PROCEDURE PutNumber(ta: TextAreaPtr; value: REAL);
```

Sets a new number for numeric input object and prints it. This procedure works for objects created by CreateTextArea as well.

```
PROCEDURE GetNumber(ta: TextAreaPtr) : REAL;
```

Gets current number from numeric input object. This procedure should not be used for objects created by CreateTextArea.

```
PROCEDURE InputText(ta : TextAreaPtr);
```

Not for public use.

```
PROCEDURE StopReading(ma: MouseAreaPtr);
```

Not for public use. Used by module MouseEvent.

```
PROCEDURE Init;
```

Not for public use. Called by MouseEvent.Init

```
END TextArea.
```

```
DEFINITION MODULE Draw;
```
Draw contains graphic routines for drawing buttons and and text input boxes suitable for MouseArea and TextArea objects. The Real-Time Kernel must be initialized before the module is used.
This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM Graphics IMPORT point, color, rectangle;

CONST PixWidth     = 1.5/639.0;
      PixHeight    = 1.0/349.0;
      BorderWidth  = PixWidth*2.0;
      BorderHeight = PixHeight*1.0;

PROCEDURE NiceColors(areacolor: color;
                     VAR textcolor,darkcolor,lightcolor: color);
```
Chooses suitable colors for shadowing and text.

```
PROCEDURE DrawButton(
            area       : rectangle;
            buttonText : ARRAY OF CHAR;
            areacolor, textcolor, darkcolor, lightcolor: color);
```
Draws a defaultshaped button on the screen. ButtonText will be truncated to fit into Area.

```
PROCEDURE DrawDefaultButton(
            area       : rectangle;
            buttonText : ARRAY OF CHAR;
            areacolor  : color);
```
Draws a button in default colors

```
PROCEDURE DrawTextArea(
             Position  : point;
             Width : CARDINAL;
             Text : ARRAY OF CHAR;
             DarkColor, LightColor,
             TextColor, BackColor : color);
```
Draws a text area and writes Text into it. The drawn area will be slightly larger than the active area defined by ActiveTextArea in module TextArea. A frame of sizes BorderWidth and BorderHeight is added around the active area. The arguments DarkColor and LightColor are used for the frame.

```
PROCEDURE FillTextArea(position: point; width: CARDINAL;
                       fillcolor: color);
```
Fills the text area with fillcolor. Use color of background for hiding the object.

```
PROCEDURE DrawFrame(area: rectangle; upLeftColor, lowRightColor: color);
```

Draws a shaddow frame around (inside) area. The frame is BorderHeight thick at the top and bottom and BorderWidth thick at the sides.

```
PROCEDURE DrawWindow(Area: rectangle; Title: ARRAY OF CHAR);
```
Draws a predesigned window with Title in window bar

```
END Draw.
```

```
DEFINITION MODULE Easy;
```

Module Easy provides an easy way of creating and drawing simple interaction objects. Easy is based on module MouseEvent.

Initialize module MouseEvent before using Easy procedures.

This module is based on a project in "Realtidssystem" in spring 1993, made by Ola Johansson, E88, and Richard Zembron, D88.

```
FROM Graphics IMPORT point, rectangle, color;

CONST BarGraphWidth = 26.0*1.5/80.0;
      BarGraphHeight = 0.14;

TYPE ButtonProc   = PROCEDURE(point);
     InputProc    = PROCEDURE(ARRAY OF CHAR, BOOLEAN);
     NumInputProc = PROCEDURE(REAL);
     BarGraphProc = PROCEDURE(REAL);
```

Callback procedure types

```
PROCEDURE EasyButton(
            Area       : rectangle;
            AreaColor  : color;
            Text       : ARRAY OF CHAR;
            Callback   : ButtonProc);
```

Creates and draws a button. The button will react on mouse clicks by calling the provided callback procedure of type `PROCEDURE(point)`.

```
PROCEDURE EasyInput(
            Position                : point;
            Width                   : CARDINAL;
            FrameColor, BackColor   : color;
            Text                    : ARRAY OF CHAR;
            Callback                : InputProc);
```

Defines and draws a default input field for text and writes Text in it. Give it a callback procedure of type `PROCEDURE(ARRAY OF CHAR, BOOLEAN)`. Input is activated by the operator by a mouse click. When the operator pushes Return or clicks in another text input, Callback will be called with the current string as argument and a boolean argument set to TRUE if input was interrupted, i.e., not ended by Return.

```
PROCEDURE EasyNumInput(
            Position                : point;
            Width                   : CARDINAL;
            FrameColor, BackColor   : color;
            Number                  : REAL;
            Callback                : NumInputProc);
```

Defines and draws a default input field for numbers and writes Number in it. Give it a callback procedure of type `PROCEDURE(REAL)`. Input is activated by the operator by a mouse click. When the operator pushes Return the current string will be interpreted as a real number. If this is possible Callback will be called with the number as argument.

```
PROCEDURE EasyBarGraph (
            LoLeft                      : point;
            AreaColor, FrameColor,
            BarColor        : color;
            Value, MinValue, MaxValue : REAL;
            Callback                    : BarGraphProc);
```

Creates and draws a bargraph at position LoLeft. Give it a callback procedure of type `PROCEDURE(REAL)`; When operator changes the value of the bargraph Callback will be called with the new value as argument.

```
END Easy.
```

```
DEFINITION MODULE BarGraph;
```
Module BarGraph is for creating graphical and numerical input devices. This module is based on MouseEvent which must be initialized before any bargraph is created. Bargraphs can have horizontal or vertical layout.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;

TYPE BarGraphPtr;
     BarGraphProc = PROCEDURE(BarGraphPtr);

CONST VBGwidth = 0.3;
      HBGwidth = 0.5;  HBGheight = 0.2;

PROCEDURE CreateHBG(VAR newBG: BarGraphPtr; loleft: point;
                    areaColor, frameColor, barColor: color;
                    value, minValue, maxValue: REAL;
                    callback: BarGraphProc; usersPtr: ADDRESS;
                    title: ARRAY OF CHAR);
```
Creates and activates a horizontal bargraph and returns a pointer to it. The callback procedure is called whenever the operator sets the value.

```
PROCEDURE CreateVBG(VAR newBG: BarGraphPtr; loleft: point; height: REAL;
                    areaColor, frameColor, barColor: color;
                    value, minValue, maxValue: REAL;
                    callback: BarGraphProc; usersPtr: ADDRESS;
                    title: ARRAY OF CHAR);
```
Creates and activates a vertical bargraph and returns a pointer to it. The callback procedure is called whenever the operator sets the value.

```
PROCEDURE GetValue(bg: BarGraphPtr) : REAL;
```
Gets current value of the bargraph.

```
PROCEDURE SetValue(bg: BarGraphPtr; value: REAL);
```
Sets a new value for the bargraph. This will not call the callback procedure.

```
PROCEDURE Activate(bg: BarGraphPtr);
```
Activates and redraws the bargraph.

```
PROCEDURE Deactivate(bg: BarGraphPtr);
```
Deactivates the bargraph.

```
PROCEDURE GetUsersPtr(bg: BarGraphPtr) : ADDRESS;
```
Returns user's pointer.

```
END BarGraph.
```

```
DEFINITION MODULE Menu;
```
Module for defining menus of selections. A menu is vertical list of selectable items. This module is based on Module MouseArea which must be initialized before any procedure is called.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;

TYPE MenuPtr;
     MenuProc = PROCEDURE(MenuPtr);

CONST MaxNoOfItems = 20;

PROCEDURE CreateMenu(newMenu: MenuPtr; loleft: point;
                     items, width: CARDINAL;
                     radio: BOOLEAN; backColor, selectColor: color;
                     callback: MenuProc; usersPtr: ADDRESS;
                     title: ARRAY OF CHAR);
```
Creates and activates a menu. The width is number of characters in each item label. If radio is TRUE, last selected item remains highlighted with selectColor, otherwise selected item is highlighted shortly. If title is given as an empty string no header is drawn on the menu. The callback procedure is called when the operator makes a selection.

```
PROCEDURE GetArea(m: MenuPtr; VAR area: rectangle);
```
Returns the area occupied on the screen by the menu.

```
PROCEDURE SetLabel(m: MenuPtr; item: CARDINAL; label: ARRAY OF CHAR);
```
Sets the label of a menu item.

```
PROCEDURE SetSelection(m: MenuPtr; item: CARDINAL);
```
Sets current selection.

```
PROCEDURE GetSelection(m: MenuPtr) : CARDINAL;
```
Gets current selection.

```
PROCEDURE Activate(m: MenuPtr);
```
Activates and redraws the menu.

```
PROCEDURE Deactivate(m: MenuPtr);
```
Deactivates the menu.

```
PROCEDURE GetUsersPtr(m: MenuPtr) : ADDRESS;
```
Return user's pointer.

```
END Menu.
```

```
DEFINITION MODULE NumMenu;
```
Module NumMenu for creating and operating on numeric menus. A numeric menu is a form with a number of labeled numeric input fields and an enter button. This module is based on module MouseArea which must be initialized before any numeric menu is created.

```
FROM SYSTEM IMPORT ADDRESS;
FROM Graphics IMPORT point, rectangle, color;

TYPE NumMenuPtr;
     NumMenuProc = PROCEDURE(NumMenuPtr);

CONST MaxNoOfEntries = 12;

PROCEDURE CreateNumMenu(newNM: NumMenuPtr; lowleft: point;
                        noOfEntries, labelWidth: CARDINAL;
                        areaColor, frameColor: color;
                        callback: NumMenuProc; usersPtr: ADDRESS;
                        title: ARRAY OF CHAR);
```
Creates and activates a numeric menu. The callback procedure is called when the operator clicks on the Enter button.

```
PROCEDURE SetEntry(nm: NumMenuPtr; entry: CARDINAL;
                   label: ARRAY OF CHAR;
                   value: REAL);
```
Sets the label string and value of a numeric menu. The entries are numbered 1 to noOfEntries. The label is truncated to the length specified when the menu was created. If this is not called the default label is the empty string and the value is 0.0.

```
PROCEDURE GetArea(nm: NumMenuPtr; area: rectangle);
```
Returns the area occupied on the screen by the numeric menu.

```
PROCEDURE GetValues(nm: NumMenuPtr; VAR value: ARRAY OF REAL);
```
Returns the value of each field of the numeric menu.

```
PROCEDURE Activate(nm: NumMenuPtr);
```
Activates and redraws the numeric menu.

```
PROCEDURE Deactivate(nm: NumMenuPtr);
```
Deactivates the numeric menu.

```
PROCEDURE GetUsersPtr(nm: NumMenuPtr) : ADDRESS;
```
Return user's pointer.

```
END NumMenu.
```

```
DEFINITION MODULE Plotter;
```
Module for plotter objects. A plotter is an area of the screen where up to 6 variables are plotted against a common horizontal axis. The operator can use the mouse to switch the individual channels on or off. The module is based on the module MouseArea and requires that MouseArea is initialized before a plotter is created.

```
FROM Graphics IMPORT rectangle, color;

TYPE PlotterPtr;

CONST MaxNoOfChannels = 6;

PROCEDURE CreatePlotter(VAR newPlotter: PlotterPtr; area: rectangle;
                        noOfChannels: CARDINAL; timeScale: REAL;
                        backColor, frameColor: color;
                        buttons: BOOLEAN; title: ARRAY OF CHAR);
```
Creates and activates a plotter object and returns a pointer to it. Parameters: area is the region the plotter will occupy on the screen, noOfChannels must be a number from 1 to 6, timeScale defines the horizontal axis, backColor defines the background color, frameColor defines the color for the frame. If buttons is TRUE a button for each channel is created where the operator can switch the channel on or off.

```
PROCEDURE SetChannel(pl: PlotterPtr; channel: CARDINAL;
                     name: ARRAY OF CHAR; c: color;
                     minValue, maxValue: REAL);
```
Sets properties of a given channel. Channels are numbered from 1 to noOfChannels. If this is not called, default properties will be used. Default properties are an empty name string, minValue = −1.0, maxValue = 1.0, colors are assigned in sequence: red, green, lightblue, cyan, magenta, lightblue.

```
PROCEDURE SetTime(pl: PlotterPtr; t: REAL);
```
Sets the current time and erases from last time to current time.

```
PROCEDURE WriteValue(pl: PlotterPtr; value: REAL; channnel: CARDINAL);
```
Extends the graph for the given channel to current time.

```
PROCEDURE WriteValues(pl: PlotterPtr; time: REAL; values: ARRAY OF REAL);
```
Sets the current time and draws all graphs.

```
PROCEDURE Activate(pl: PlotterPtr);
```
Activates and redraws the plotter. Old graphs are lost.

```
PROCEDURE Deactivate(pl: PlotterPtr);
```
Deactivates the plotter.

```
END Plotter.
```

```
DEFINITION MODULE ListHandler;

FROM SYSTEM IMPORT ADDRESS;

TYPE
  ListTypePtr; NodeTypePtr;

PROCEDURE NewList() : ListTypePtr;
```
Creates a new empty list.

```
PROCEDURE NewNode(e : ADDRESS) : NodeTypePtr;
```
Create a new node and put a pointer to the element in the node.

```
PROCEDURE InsertFirst(n : NodeTypePtr; VAR l : ListTypePtr);
```
Put node n first in the list l.

```
PROCEDURE InsertLast(n : NodeTypePtr; VAR l : ListTypePtr);
```
Put node n last in the list l.

```
PROCEDURE FirstNode(l : ListTypePtr) : NodeTypePtr;
```
Returns a pointer to the first element in list l.

```
PROCEDURE LastNode(l : ListTypePtr) : NodeTypePtr;
```
Returns a pointer to the last node of the list.

```
PROCEDURE PredNode(n : NodeTypePtr) : NodeTypePtr;
```
Returns a pointer to the preceeding node.

```
PROCEDURE SuccNode(n : NodeTypePtr) : NodeTypePtr;
```
Returns a pointer to the succeeding node.

```
PROCEDURE IsEmptyList(l : ListTypePtr) : BOOLEAN;
```
Returns TRUE if the list is empty.

```
PROCEDURE IsFirstNode(n : NodeTypePtr) : BOOLEAN;
```
Returns TRUE if n points to the first node in a list.

```
PROCEDURE IsLastNode(n : NodeTypePtr) : BOOLEAN;
```
Returns TRUE if n points to the last node in a list.

```
PROCEDURE ElementPtr(n : NodeTypePtr) : ADDRESS;
```
This routine is used to get the actual element from the node. An example:

```
  N1 := NewNode(Obj);
  InsertFirst(N1, List);
  N2 := FirstNode(List);
  E1 := ElementPtr(N1);
```

```
E2 := ElementPtr(N2);
```
Now E1 and E2 points to the same element, namely Obj.

```
PROCEDURE RemoveNode(n : NodeTypePtr; l : ListTypePtr);
```
Removes a node from a list.

```
PROCEDURE ClearList(l : ListTypePtr);
```
Deletes an entire list.

```
END ListHandler.
```

```
DEFINITION MODULE Signals;

FROM Graphics IMPORT point;

TYPE
  FunctionType = (Sin, Pulse, Ramp, Step, Random);

PROCEDURE InitSignals(SignalPriority : CARDINAL);
```
Initiates the signal generator module.

```
PROCEDURE MakeRefSignal(SignalName : ARRAY OF CHAR;
                        FunctionName : FunctionType;
                        Omega : REAL);
```
Creates a signal from the generator.

```
PROCEDURE GetRefValue(SignalName : ARRAY OF CHAR;
                      VAR Value : REAL);
```
The value of SignalName gets assignad to Value.

```
PROCEDURE ChangeOmega(SignalName : ARRAY OF CHAR; Omega : REAL);
```
Changes the value of omega (the frequency) of the signal SignalName.

```
PROCEDURE ChangeDelta(Delta : CARDINAL);
```
Changes the frequency of the generation of new values. Delta is given in ms.

```
PROCEDURE ChangeDirection(SignalName : ARRAY OF CHAR;
                          Direction : REAL);
```
Changes the slope of the ramp function.

```
PROCEDURE ChangeFunction(SignalName : ARRAY OF CHAR;
                         Function : FunctionType);
```
Changes the function of the signal belonging to SignalName.

```
END Signals.
```

```
DEFINITION MODULE TextWindows;
```
This module defines a kind of window for simple input and output of text. The text scrolls vertically when lines are added below the last visible line. The number of visible lines and columns depends on the window size, given in screen coordinates.

RTMouse or RTGraph must be initialized before this module is used.

```
TYPE WindowType;

PROCEDURE MakeTextWindow(xlo, ylo, xhi, yhi: REAL) : WindowType;
```
Creates a new text window

```
PROCEDURE WriteString(W: WindowType; Line: ARRAY OF CHAR);
```
Adds a string to the end of current line.

```
PROCEDURE WriteReal(W: WindowType; Value: REAL; Width: CARDINAL);
```
Prints a real number at the end of current line.

```
PROCEDURE WriteInteger(W: WindowType; Value: INTEGER; Width: CARDINAL);
```
Prints an integer number at the end of current line.

```
PROCEDURE NewLine(W: WindowType);
```
Makes current line visible and start a new one.

```
PROCEDURE WriteLine(W: WindowType; Line: ARRAY OF CHAR);
```
WriteString and NewLine

```
PROCEDURE ReadLine(W: WindowType; Prompt: ARRAY OF CHAR;
                   VAR Result: ARRAY OF CHAR);
```
Prompt user for a string of character. The window becomes blocked for output until thi procedure has finished.

```
PROCEDURE ReadReal(W: WindowType; Prompt: ARRAY OF CHAR;
                   VAR Value: REAL): BOOLEAN;
```
Prompt user for a number. Returns FALSE and Value=0.0 if no valid number was read. The window becomes blocked for output until thi procedure has finished.

```
END TextWindows.
```