W

T

U

V

K

L

M

N

O

E

F

G

H

I

Procedure Index

```
PROCEDURE EditAborted (): BOOLEAN;

    (* Checks the next keyboard input.  Returns TRUE for Escape, FALSE  *)
    (* for anything else.  Escape or Carriage Return are consumed, any  *)
    (* other character is returned to the Keyboard module.              *)

PROCEDURE ColourSwap (w: Window; row: RowRange; col: ColumnRange;
                                                    nchar: CARDINAL);

    (* Sets a field of nchar characters, starting at (row,col), to      *)
    (* "reverse video" by swapping the foreground and background        *)
    (* colours.  Notice that the process is reversible: you get back to *)
    (* "normal video" by calling this procedure again.  The location is *)
    (* given in window-relative coordinates, not absolute screen        *)
    (* positions.  NOTE: Do not assume that this procedure can wrap     *)
    (* around to a new line.  It normally cannot.                       *)

PROCEDURE Blink (w: Window; r: RowRange; c: ColumnRange; nchar: CARDINAL);

    (* Toggles the blinking status - that is, turns blinking on if it   *)
    (* was off, and vice versa - for nchar characters, starting at      *)
    (* relative location (r,c) in window w.                             *)
    (* NOTE: This procedure will not wrap around to a new row.          *)

PROCEDURE EraseLine (w: Window; option: CARDINAL);

    (* Erases some or all of the current line (but never the border).   *)
    (* The erased characters are replaced by space characters.  The     *)
    (* window cursor is moved to the location of the first erased       *)
    (* character.  If w is not the currently active window, the changes *)
    (* will not be visible until w is on top again.  The options are:   *)
    (*          0       the whole of the line, except for the border    *)
    (*          1       from the current cursor position onwards        *)
    (*          2       from the start to just before the cursor        *)
    (* If we are inside a scrolling region, then only that part of the  *)
    (* line inside the scrolling region is affected.                    *)

END Windows.
```

```
PROCEDURE WriteChar (w: Window; ch: CHAR);

    (* Write one character.  Control characters are not given special   *)
    (* treatment; they produce something visible just like any other    *)
    (* character.  Wraps to the next line before writing if the write   *)
    (* would put us on or beyond the right border of w.                 *)

PROCEDURE Write (w: Window; ch: CHAR);

    (* Like WriteChar, but codes in the range 0..31 are treated as      *)
    (* control characters.  This procedure is not recommended for       *)
    (* general use, as it leads to obscure programs.  (Instead, do the  *)
    (* control operations by direct calls to the cursor control         *)
    (* procedures which are also supplied in this module).  It is       *)
    (* supplied mainly to help those who are used to the conventions of *)
    (* the "standard" Modula-2 I/O modules such as InOut.               *)

PROCEDURE WriteString (w: Window; text: ARRAY OF CHAR);

    (* Write a string of characters, stopping at the first NUL          *)
    (* character or the end of the array, whichever comes first.        *)

PROCEDURE ReadBack (w: Window;  r: RowRange;  c: ColumnRange): CHAR;

    (* Returns the character which currently occupies relative location *)
    (* (r,c) on the display of window w.                                *)

PROCEDURE ReadChar (w: Window;  VAR (*OUT*) ch: CHAR);

    (* Read one character, and echo it. *)

PROCEDURE LookaheadChar (w: Window): CHAR;

    (* Reads a character without consuming it.  That is, the character  *)
    (* remains available to be read by ReadChar.  This allows the       *)
    (* caller to check whether the character is really wanted.          *)

PROCEDURE ReadCharWithoutEcho (w: Window;  VAR (*OUT*) ch: CHAR);

    (* Read one character, but don't echo it.  However, a blinking      *)
    (* cursor is still displayed to prompt for the character.  (If you  *)
    (* don't want the blinking cursor, use procedure Keyboard.InKey).   *)

PROCEDURE PressAnyKey (w: Window);

    (* Types a "Press any key to continue" message.     *)

PROCEDURE ReadString (w: Window;  VAR (*OUT*) result: ARRAY OF CHAR);

    (* Reads a character string, terminated by carriage return.         *)

PROCEDURE EditString (w: Window;  VAR (*INOUT*) result: ARRAY OF CHAR;
                                                fieldsize: CARDINAL);

    (* Reads a character string, where a default result is supplied by  *)
    (* the caller.  The final result is the state of the string at the  *)
    (* time where the keyboard user types a carriage return or Esc, or  *)
    (* uses a cursor movement key to move out of the displayed field.   *)
    (* The terminating character remains available, via Keyboard.InKey, *)
    (* to the caller.  At most fieldsize characters of the string can   *)
    (* be edited, and perhaps fewer if the result array is smaller or   *)
    (* if there is insufficient space in the window.                    *)
```

```
TYPE CloseHandlerProc = PROCEDURE (Window, DisplayPage);

PROCEDURE InstallCloseHandler (w: Window;  P: CloseHandlerProc);

    (* Sets up P as a procedure to be called when the window is closed. *)
    (* It is legal to define multiple handlers for the same window.     *)

PROCEDURE WindowLocation (w: Window): Rectangle;

    (* Returns the current location of w on the screen. *)

PROCEDURE SetCursor (w: Window; row: RowRange; column: ColumnRange);

    (* Sets the cursor for window w to the given row and column.  The   *)
    (* coordinates are window-relative; that is, they start at (0,0) at *)
    (* the top left of the window.                                      *)

PROCEDURE SaveCursor (w: Window; VAR (*OUT*) row, column: CARDINAL);

    (* Returns the current cursor position.  The coordinates are        *)
    (* window-relative; that is, they start at (0,0) at the top left of *)
    (* the window.                                                      *)

PROCEDURE CursorLeft (w: Window);

    (* Moves the window cursor one position left.  If it falls off the  *)
    (* left edge of the window, it moves to the right edge in the same  *)
    (* row.                                                             *)

PROCEDURE CursorRight (w: Window);

    (* Moves the window cursor one position right.  If it falls off the *)
    (* right edge of the window, it moves to the left edge in the same  *)
    (* row.                                                             *)

PROCEDURE CursorUp (w: Window);

    (* Moves the window cursor one position up.  If it falls off the    *)
    (* top edge of the window, it moves to the bottom edge in the same  *)
    (* column.                                                          *)

PROCEDURE CursorDown (w: Window);

    (* Moves the window cursor one position down.  If it falls off the  *)
    (* bottom edge of the window, it moves to the top edge in the same  *)
    (* column.                                                          *)

PROCEDURE ScrollUp (w: Window);

    (* Scrolls the scrolling region of window w up by one row, filling  *)
    (* the vacated row with spaces.                                     *)

PROCEDURE ScrollDown (w: Window);

    (* Scrolls the scrolling region of window w down by one row,        *)
    (* filling the vacated row with spaces.                             *)

PROCEDURE WriteLn (w: Window);

    (* Go to next line in window, scrolling if necessary.  N.B. The     *)
    (* window does not scroll if you are not in the scrolling region    *)
    (* at the time of the WriteLn.                                      *)
```

```
PROCEDURE ShiftWindowAbs (w: Window;  top: RowRange;  left: ColumnRange);

    (* Like ShiftWindowRel, except that we directly specify the target  *)
    (* position of the top left corner in screen coordinates.           *)

PROCEDURE CloseWindow (w: Window);

    (* Destroys the window.  Note that this could have side-effects     *)
    (* if procedure InstallCloseHandler (see below) has been called.    *)

PROCEDURE Hide (w: Window);

    (* Makes this window invisible on the screen.  It is still possible *)
    (* to write to the window, but the output will not appear until     *)
    (* a PutOnTop(w) is executed.                                       *)

PROCEDURE PutOnPage (w: Window;  p: DisplayPage);

    (* Moves window w to another display page.  The default is to put   *)
    (* every window on page 0 when it is first opened.  To override     *)
    (* the default, call this procedure after opening the window.       *)

PROCEDURE PageOf (w: Window): DisplayPage;

    (* Returns the display page on which window w resides. *)

PROCEDURE PutOnTop (w: Window);

    (* Makes sure that window w is fully visible on the screen.  (This  *)
    (* also cancels the effect of a Hide(w).)  Rarely needed, since     *)
    (* many window operations automatically put their window on top.    *)

PROCEDURE IdentifyTopWindow (VAR (*OUT*) w: Window;
                             VAR (*INOUT*) row: RowRange;
                             VAR (*INOUT*) col: ColumnRange): BOOLEAN;

    (* On entry w is unspecified and (row,col) describes a position on  *)
    (* the screen.  On exit w is equal to the top window containing     *)
    (* this screen location, and (row,col) have been altered to be      *)
    (* window-relative coordinates.  Exception: if there is no visible  *)
    (* window containing the given point, the function result is FALSE, *)
    (* the returned w is meaningless, and row and col are unchanged.    *)

PROCEDURE RefreshDisplay;

    (* Rewrites every open window.  Should not normally be needed, but  *)
    (* available for use in cases the display is corrupted by, for      *)
    (* example, software which bypasses this module and writes directly *)
    (* to the screen.                                                   *)

PROCEDURE SetActivePage (page: DisplayPage);

    (* Changes the active display page.  Not needed unless you are      *)
    (* switching among multiple pages.                                  *)

TYPE PageChangeHandler = PROCEDURE (DisplayPage);

PROCEDURE RequestPageChangeNotification (Proc: PageChangeHandler);

    (* Sets up Proc as a procedure to be called on a page change.       *)
```

```
    Rectangle = RECORD
                    top, bottom: RowRange;
                    left, right: ColumnRange;
                END (*RECORD*);

    FrameType = (noframe, simpleframe, doubleframe);
    DividerType = (nodivider, simpledivider, doubledivider);

    DisplayPage = [0..3];

PROCEDURE OpenWindow (VAR (*OUT*) w: Window;
                      ForegroundColour, BackgroundColour: Colour;
                      firstline, lastline: RowRange;
                      firstcol, lastcol: ColumnRange;
                      FrameDesired: FrameType;
                      DividerDesired: DividerType);

    (* Create a new window.  Note that row and column numbers start     *)
    (* from 0.  NOTE: If the window has a box drawn around it (the case *)
    (* FrameDesired <> noframe), this subtracts from the space          *)
    (* available for text.                                              *)

PROCEDURE OpenSimpleWindow (VAR (*OUT*) w: Window;
                      firstline, lastline: RowRange;
                      firstcol, lastcol: ColumnRange);

    (* Identical to OpenWindow, except that you don't get any choice    *)
    (* about the colours or frame.  The window is white-on-black with   *)
    (* a simple frame and no dividers for the scrolling region.  This   *)
    (* version of OpenWindow is useful for those with monochrome        *)
    (* displays who don't want to be bothered with importing the types  *)
    (* Colour, FrameType, and DividerType.                              *)

PROCEDURE ChangeScrollingRegion (w: Window; firstline, lastline: RowRange);

    (* Changes the scrolling region of window w to the new line         *)
    (* boundaries given, and sets the cursor of this window to the      *)
    (* start of the scrolling region.  Row numbers are window-relative; *)
    (* that is, line 0 is the top line of the window (which is where    *)
    (* the border is, unless you have no border).                       *)

PROCEDURE NewScrollingRegion (w: Window;  firstline, lastline: RowRange;
                              firstcolumn, lastcolumn: ColumnRange);

    (* Changes the scrolling region of w to be the specified rectangle, *)
    (* but unlike ChangeScrollingRegion this procedure does not redraw  *)
    (* the dividers.  Furthermore the old scrolling region set by       *)
    (* ChangeScrollingRegion is remembered and may be restored by a     *)
    (* call to ResetScrollingRegion.                                    *)

PROCEDURE ResetScrollingRegion (w: Window);

    (* Changes the scrolling region of w back to what it was the last   *)
    (* time ChangeScrollingRegion was called.  If ChangeScrollingRegion *)
    (* was never called, the scrolling region goes back to being the    *)
    (* entire window minus the frame (if any).                          *)

PROCEDURE ShiftWindowRel (w: Window;  rowchange, columnchange: INTEGER);

    (* Moves w on the screen.  The second and third arguments may be    *)
    (* negative.  The amount of move may be reduced to prevent a move   *)
    (* off the edge of the screen.                                      *)
```

```
(*************************************************************************)
(*                                                                       *)
(* NOTE ON CRITICAL SECTIONS: When using this module in a multitasking  *)
(* environment, there are numerous critical sections because several    *)
(* tasks may be using the same physical screen.  These sections are     *)
(* protected by semaphores.  (For the case where we are not using the   *)
(* multitasking kernel, there is a version of module Semaphores which   *)
(* contains dummy procedures).  The general philosophy is to assume     *)
(* that critical section protection is needed for inter-window          *)
(* conflicts, but that no protection is needed for operations on a      *)
(* single window because the most common situation is that each window  *)
(* is used by only one task.  If there happen to be windows which are   *)
(* shared by two or more tasks, the associated synchronization problems *)
(* must be resolved by the caller; they are not resolved in module      *)
(* Windows, on the grounds that the extra overhead is not justified     *)
(* just to support a case which rarely happens in practice.             *)
(*                                                                       *)
(*                                                                       *)
(*                       LATEST DEVELOPMENT                              *)
(*                                                                       *)
(* Because of the addition of the MouseControl module, it can happen in *)
(* practice, so I've added partial protection.  The following           *)
(* procedures are indivisible with respect to one another, as far as    *)
(* shared uses of a window are concerned:                               *)
(*      OpenWindow, CloseWindow, ShiftWindowRel, ScrollUp, ScrollDown,  *)
(*      WriteChar, WriteLn, SetCursor, Hide, PutOnTop.                  *)
(* Minor technicality: in some cases there is an implied PutOnTop, or   *)
(* similar side-effect, and this is not always part of the indivisible  *)
(* section.  I can't think of any case where this would matter to the   *)
(* user.  I don't like making protected sections too large, since this  *)
(* can degrade overall system responsiveness.                           *)
(*                                                                       *)
(* Note: procedures WriteString and Write, while not indivisible in     *)
(* the above sense, are protected at the "write a character" level.     *)
(*                                                                       *)
(*************************************************************************)

CONST
    MaxRowNumber = 24;
    MaxColumnNumber = 79;

TYPE
    Window;     (* is private *)
    Colour = (black, blue, green, cyan, red, magenta, brown, white,
                darkgrey, lightblue, lightgreen, lightcyan, lightred,
                lightmagenta, yellow, intensewhite);

    (* Note: Any of these colours may be used as a foreground colour,   *)
    (* but in most applications only [black..white] are suitable as     *)
    (* background colours.  The others may be used, but they can give   *)
    (* strange effects such as a blinking display.  Use with care!      *)

    RowRange = [0..MaxRowNumber];
    ColumnRange = [0..MaxColumnNumber];
```

Windows

```
DEFINITION MODULE Windows;

        (****************************************************************)
        (*                                                              *)
        (*      A simple implementation of screen windows.              *)
        (*                                                              *)
        (*  Programmer:         P. Moylan                               *)
        (*  Last edited:        10 February 1994                        *)
        (*  Status:             Working                                 *)
        (*                                                              *)
        (****************************************************************)

(************************************************************************)
(*                                                                      *)
(* REMARK: There are two versions of the basic 'write' procedure.       *)
(* Procedure WriteChar treats every character as a printable character, *)
(* so it can write any character in the standard character set.  This   *)
(* is the output procedure recommended for general use.  Control        *)
(* operations are supplied by separate procedures such as WriteLn,      *)
(* SetCursor, etc., so there is no need to resort to the BASIC          *)
(* programmers' trick of writing long strings of obscure control codes. *)
(* But for people who can't break this bad habit, procedure Write is    *)
(* supplied; this treats character codes 0..31 as control codes.        *)
(* Its main intended uses are to echo keyboard input, and to print a    *)
(* file which contains embedded control characters; but it may be       *)
(* used for other purposes if desired.                                  *)
(*                                                                      *)
(* Warning: I have not bothered to look after all the control codes,    *)
(* just the ones I found a use for.  Most of them print as a            *)
(* two-character sequence ^<letter>.                                    *)
(*                                                                      *)
(************************************************************************)
```

```
PROCEDURE DrawXAxis (G: Graph;  xinc: LONGREAL;
                     yoffset, DecPlaces: CARDINAL;  numbering: BOOLEAN);

    (* Draws the x-axis for a graph; xinc is the distance between tick  *)
    (* marks, and yoffset gives the distance that the axis will lie     *)
    (* below the line y=ys (ys is defined in OpenGraph).  The option    *)
    (* yoffset <> 0 allows for the case where more than one graph       *)
    (* occupies the same window, since the graphs may have different    *)
    (* scales.  DecPlaces gives the number of decimal places that will  *)
    (* be used.  If numbering is FALSE, the axis is drawn but not       *)
    (* labelled.                                                        *)

PROCEDURE DrawYAxis (G: Graph;  yinc: LONGREAL;
                     xoffset, DecPlaces: CARDINAL;  numbering: BOOLEAN);

    (* Draws the y-axis for a graph; yinc is the space between tick     *)
    (* marks, and xoffset gives the distance that the axis will lie to  *)
    (* the left of the line x=xs (xs is defined in OpenGraph).          *)
    (* DecPlaces gives the number of decimal places that will be used.  *)
    (* If numbering is FALSE, the axis is drawn but not labelled.       *)

PROCEDURE DrawAxes (G: Graph;
        xinc: LONGREAL;  yoffset, xdecplaces: CARDINAL;  numberx: BOOLEAN;
        yinc: LONGREAL;  xoffset, ydecplaces: CARDINAL;  numbery: BOOLEAN);

    (* Draws both the X and Y axes.  See the definitions of             *)
    (* DrawXAxis and DrawYAxis above.                                   *)

PROCEDURE AddPoint (G: Graph;  x, y: LONGREAL);

    (* For lines of type                                                *)
    (*          'dots'  : only the marker will be displayed.            *)
    (*          'joined': draws a line from the last point and marker.  *)
    (*          'vertical' : draws a vertical line and a marker.        *)
    (* NOTE: lines of type 'normal' are used for discontinuous graphs   *)
    (* and are not drawn using this procedure.                          *)
    (* Points outside the region (xmin <= x <= xmax, ymin <= y <= ymax  *)
    (* will be clipped.                                                 *)

PROCEDURE DrawLines (G: Graph;  x, y: ARRAY OF LONGREAL);

    (* Allows vectors of x and y points to be drawn.  Refer to          *)
    (* procedure AddPoint for specifications of the line types.         *)

PROCEDURE DrawLine (G: Graph;  x0, y0, x1, y1: LONGREAL);

    (* Used for drawing discontinuous lines.  The line is drawn between *)
    (* (x0,y0) and (x1,y1) relative to graph G.                         *)

PROCEDURE BarGraph (G: Graph;  y: ARRAY OF LONGREAL;  del: CARDINAL;  bt:
BarType);

    (* Displays a BarGraph of the vector y.  Parameter del gives the    *)
    (* number of stripes per bar, in the case of striped fill patterns. *)

END WGraph.
```

WGraph

```
DEFINITION MODULE WGraph;

        (*********************************************************)
        (*                                                       *)
        (*              Graph module For GWindows.               *)
        (*                                                       *)
        (*  Programmer:         S.P.Lontis, P. Moylan.           *)
        (*  Last edited:        15 February 1993                 *)
        (*  Status:                                              *)
        (*      Basically working, but still fiddling with       *)
        (*      the details.                                     *)
        (*                                                       *)
        (*      It's not clear whether OpenGraph needs so many   *)
        (*      parameters.  Also not clear how bar graphs       *)
        (*      fit into the overall picture.                    *)
        (*                                                       *)
        (*********************************************************)

FROM GWindows IMPORT
     (* type *)   Window;

(*************************************************************************)

TYPE
    Graph;              (* is private *)

    (* Types of graphs that can be drawn.       *)

    GraphType = (linear, loglinear, loglog, bar);

    (* Line Types that can be used.     *)

    LineType  = (dots, joined, vertical, normal);

    (* Marker Types for individual points plotted.      *)

    MarkType  = (none, square, cross, plus, diamond);

    BarType = (One, Two, Three, Four, Five);

(*************************************************************************)

PROCEDURE OpenGraph (VAR (*OUT*) G: Graph;  w: Window;  GrphType: GraphType;
                  LnType: LineType; xst, yst, xen, yen: CARDINAL;
                  xmin, ymin, xmax, ymax: LONGREAL;  Mark: MarkType);

    (* Creates graph G in window w.  The Graph will occupy a rectangle  *)
    (* with a bottom left corner of (xs,ys) and top right corner of     *)
    (* (xe,ye) relative to the window. The range of values which can be *)
    (* plotted is from (xmin,ymin) to (xmax,ymax).  NOTE: for a bar     *)
    (* graph, xmin gives the width of the bars and xmax gives the       *)
    (* distance between different bars.                                 *)
```

```
    Capability = (wshow, wmove, whide, wescape);
    CapabilitySet = SET OF Capability;

    (* This module also gives a special meaning to a right mouse click  *)
    (* on a blank area of the screen: it brings up a list of windows    *)
    (* for which mouse control is enabled, and you can click on a       *)
    (* window name to bring that window to the top of the display.      *)
    (* This is a way of getting back hidden windows.                    *)

    (* An Action procedure is one to be called on a mouse click in a    *)
    (* defined Active Region.  The parameters identify the window       *)
    (* which was clicked on, and the row and column within that window. *)

    Action = PROCEDURE (Window, RowRange, ColumnRange);

(************************************************************************)
(*              ACTIVATING MOUSE CONTROL FOR A WINDOW                   *)
(************************************************************************)

PROCEDURE AllowMouseControl (w: Window;  Title: ARRAY OF CHAR;
                             OptionsEnabled: CapabilitySet): UIWindow;

    (* Adds w to the set of windows which this module is allowed to     *)
    (* manipulate.  The window should already be open (but not          *)
    (* necessarily visible).  The Title parameter gives a name to be    *)
    (* used when a list of windows is displayed.                        *)

(************************************************************************)
(*                      DEFINING ACTIVE REGIONS                         *)
(************************************************************************)

PROCEDURE AddActiveRegion (UIW: UIWindow; Top, Bottom: RowRange;
                       Left, Right: ColumnRange;  ButtonsEnabled: ButtonSet;
                       ActionProc: Action);

    (* After a call to this procedure, any mouse click on a button in   *)
    (* ButtonsEnabled, and in the rectangle defined by the other        *)
    (* parameters, will cause ActionProc to be called.                  *)
    (* Note: Active regions may overlap.  In the event of an ambiguity, *)
    (* the more recently defined active region takes precedence.        *)

END UserInterface.
```

UserInterface

```
DEFINITION MODULE UserInterface;

        (****************************************************************)
        (*                                                              *)
        (*              Text User Interface for PMOS                    *)
        (*                                                              *)
        (*      This module allows you to put selected text windows     *)
        (*      under the control of the mouse.  In the present         *)
        (*      version, the controls available are for repositioning   *)
        (*      and for hiding windows.  The module also lets you       *)
        (*      define "active regions" within windows, i.e. regions    *)
        (*      within which a mouse click will cause a caller-supplied *)
        (*      procedure to be executed.                               *)
        (*                                                              *)
        (*              Original version by M.Walsh.                    *)
        (*                This version by P.Moylan                      *)
        (*                                                              *)
        (*      Last Edited:    25 February 1994                        *)
        (*      Status:         OK                                      *)
        (*                                                              *)
        (****************************************************************)

FROM Mouse IMPORT
    (* type *)  ButtonSet;

FROM Windows IMPORT
    (* type *)  RowRange, ColumnRange, Window;

(************************************************************************)

TYPE
    (* A UIWindow is like a Window from module Windows, but it has      *)
    (* a few extra attributes added for the purposes of this module.    *)

    UIWindow;   (* is private *)

    (* The capabilities which you can add to a Window by making it into *)
    (* a UIWindow are:                                                  *)
    (*  wshow   left click anywhere on visible part of window (except,  *)
    (*          of course, regions for which other click actions are    *)
    (*          defined) brings it to the top of the stack of windows.  *)
    (*          This will also change the input focus in cases where    *)
    (*          a task is waiting for keyboard input in this window.    *)
    (*  whide   window gets a "hide button", and left clicking on this  *)
    (*          makes the window invisible.                             *)
    (*  wmove   window gets a "move button", and you can drag the       *)
    (*          window around the screen with the left or right button. *)
    (*  wescape window gets a "hide button", and left clicking on this  *)
    (*          simulates an Esc from the keyboard.  Note that this     *)
    (*          option supersedes the operation of whide, since the     *)
    (*          same button location is used for both.                  *)
```

Trace

```
DEFINITION MODULE Trace;

        (*********************************************)
        (*                                           *)
        (*          Trace routines for Modula-2      *)
        (*              program development.         *)
        (*                                           *)
        (*  Programmer:         P. Moylan            *)
        (*  Last edited:        16 August 1992       *)
        (*  Status:             Working on TS conversion*)
        (*                                           *)
        (*********************************************)

FROM Windows IMPORT
    (* type *)  RowRange, ColumnRange;

PROCEDURE NYI (name: ARRAY OF CHAR);

        (* Types a "not yet implemented" message.       *)

PROCEDURE Pause;

        (* "Press any key to continue."                 *)

PROCEDURE InTrace (name: ARRAY OF CHAR);

        (* Types "Entering 'name'".     *)

PROCEDURE OutTrace (name: ARRAY OF CHAR);

        (* Types "Leaving 'name'".      *)

PROCEDURE TraceOn (firstrow, lastrow: RowRange;
                       firstcol, lastcol: ColumnRange;
                       SlowDownFactor: CARDINAL);

        (* Turns on tracing.  The first four parameters specify the     *)
        (* size of a screen window to be used for trace messages.  The  *)
        (* size of the final parameter governs how slowly the program   *)
        (* will run - it is needed because in many cases the trace      *)
        (* messages appear on the screen faster than you can read them. *)

PROCEDURE TraceOff;

        (* Turns off tracing.           *)

PROCEDURE TraceStatus(): BOOLEAN;

        (* Says whether tracing is currently on.        *)

END Trace.
```

Timer

```
DEFINITION MODULE Timer;

        (*********************************************************)
        (*                                                       *)
        (*      Clock handler for operating system.              *)
        (*                                                       *)
        (*      Author:         P. Moylan                        *)
        (*      Last edited:    27 May 1989                      *)
        (*      Description:                                     *)
        (*              This module contains the clock           *)
        (*              interrupt routine.  It checks whether    *)
        (*              the current task has used its time       *)
        (*              quota, also keeps track of sleeping      *)
        (*              tasks.                                   *)
        (*                                                       *)
        (*      Status: Seems to be working.                     *)
        (*                                                       *)
        (*********************************************************)

FROM Semaphores IMPORT
    (* type *)  Semaphore;

PROCEDURE Sleep (milliseconds: CARDINAL);

    (* Puts the caller to sleep for approximately the given number of   *)
    (* milliseconds.  The time is not guaranteed to be precise, because *)
    (* (a) after the specified time has expired, the caller is made     *)
    (* ready, but will not run immediately unless it has a higher       *)
    (* priority than the task which is running at that time, and (b) we *)
    (* do not necessarily run the hardware timer with millisecond       *)
    (* resolution anyway.  High resolution just adds to the system      *)
    (* overhead created by the timer interrupts.  For an application    *)
    (* which genuinely needs high-precision delays, it makes more sense *)
    (* to have a separate hardware timer dedicated just to that job.    *)

PROCEDURE TimedWait (VAR (*INOUT*) s: Semaphore;  TimeLimit: INTEGER;
                                    VAR (*OUT*) TimedOut: BOOLEAN);

    (* Like a semaphore Wait, except that it returns with TimedOut TRUE *)
    (* if the corresponding Signal does not occur within TimeLimit      *)
    (* milliseconds.                                                    *)

END Timer.
```

TimeOfDay

```
DEFINITION MODULE TimeOfDay;

        (*****************************************************************)
        (*                                                               *)
        (*                       Time-of-day Module                      *)
        (*                                                               *)
        (*  This module keeps a time-of-day record, to the nearest 16th  *)
        (*  of a second, by using the CMOS clock hardware.               *)
        (*                                                               *)
        (*  Programmer:          P. Moylan                               *)
        (*  Last edited:         3 July 1993                             *)
        (*  Status:              OK                                      *)
        (*                                                               *)
        (*****************************************************************)

TYPE TimeRecord=RECORD
                    ticks,
                    seconds, minutes, hours,
                    dayofweek, dayofmonth,
                    month, year, century        : SHORTCARD
                END (*RECORD*);

(*************************************************************************)

PROCEDURE ReadClock (VAR (*OUT*) result: TimeRecord);

    (* Returns the current time and date.       *)

PROCEDURE SetClock (VAR (*IN*) newtime: TimeRecord);

    (* Modifies the current time and date.      *)

PROCEDURE SetBinaryMode (NewMode: BOOLEAN);

    (* Makes the clock run in BCD if NewMode is FALSE, or in binary if  *)
    (* NewMode is TRUE.  If this requires a mode change, adjusts the    *)
    (* current date/time values held by the hardware.                   *)

END TimeOfDay.
```

```
PROCEDURE TileSetMemory (T: TileSet;  memory: BOOLEAN);

    (* Specifying a FALSE value for the memory parameter means that     *)
    (* subsequent data sent to this TileSet will be written to the      *)
    (* screen but not remembered.  This saves time and memory, the only *)
    (* penalty being that data covered by an overlapping TileSet will   *)
    (* be lost.  Specifying TRUE restores the default condition, where  *)
    (* all data are retained for refreshing the screen when necessary.  *)

PROCEDURE AddPoint (T: TileSet;  p: Point;  colour: ColourType);

    (* Adds a new point to TileSet T, and displays it on the screen.    *)

PROCEDURE AddLine (T: TileSet;  start, end: Point;  colour: ColourType);

    (* Adds a new line to TileSet T, and displays it on the screen.     *)

PROCEDURE AddRectangle (T: TileSet;  R: Rectangle;  colour: ColourType);

    (* Draws a rectangular shape.  A shorthand for four AddLine calls.  *)

PROCEDURE AddString (T: TileSet;  place: Point;
                     VAR (*IN*) text: ARRAY OF CHAR;
                     count: CARDINAL;  colour: ColourType;  R: Rectangle);

    (* Adds a string of count characters to tileset T, and displays it. *)
    (* Points outside rectangle R are not displayed.                    *)

PROCEDURE AddRotatedString (T: TileSet;  place: Point;
                     VAR (*IN*) text: ARRAY OF CHAR;
                     count: CARDINAL;  colour: ColourType;  R: Rectangle);

    (* Like AddString, but writes in the +Y direction.  *)

PROCEDURE MoveText (T: TileSet;  amount, limit: INTEGER);

    (* Moves all character strings up by "amount" rows, discarding what *)
    (* falls above - or below, if amount is negative - the limit.       *)

END Tiles.
```

Tiles

```
DEFINITION MODULE Tiles;

        (*********************************************************)
        (*                                                       *)
        (*          Support module for screen graphics           *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        23 November 1993                 *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*      NOTE: This is a support module for GWindows,     *)
        (*      and is not intended to be called directly by     *)
        (*      applications programs.  It does not contain      *)
        (*      all of the data validity checks which the        *)
        (*      end-user procedures perform.                     *)
        (*                                                       *)
        (*********************************************************)

FROM ScreenGeometry IMPORT
    (* type *)  Point, Rectangle;

FROM Graphics IMPORT
    (* type *)  ColourType;

TYPE
    TileSet;            (* is private *)

PROCEDURE CreateTileSet (border: Rectangle;  background: ColourType): TileSet;

    (* Creates a TileSet which covers the given rectangular region.     *)
    (* The second parameter specifies the background colour.            *)
    (* This will usually require breaking up tiles of previously        *)
    (* created TileSets, but since the caller does not have access to   *)
    (* the internal structure of a TileSet this restructuring is        *)
    (* transparent to the caller.                                       *)

PROCEDURE DiscardTileSet (VAR (*INOUT*) T: TileSet);

    (* Destroys TileSet T.  This too might involve restructuring the    *)
    (* tiling of other TileSets, but again the caller need not know     *)
    (* the details.                                                     *)

PROCEDURE ClearTileSet (T: TileSet);

    (* Removes all points, lines, and text from T, and re-displays      *)
    (* the visible parts of T.                                          *)
```

```
(*                                                                        *)
(**************************************************************************)

PROCEDURE VideoKind (VAR (*OUT*) ScreenSegment: CARDINAL;
                                 VAR (*OUT*) BlackAndWhite: BOOLEAN);

    (* Returns the segment of the screen memory, and a flag saying      *)
    (* whether we are using a monochrome adaptor.                       *)

PROCEDURE SetVideoMode (newmode: CARDINAL);

    (* Sets the video mode, as defined in the BIOS. At present only     *)
    (* modes 3 (25*80 colour) and 7 (25*80 monochrome) are supported.   *)

PROCEDURE RestoreOriginalMode;

    (* Sets the video mode back to what it was before this program ran. *)

PROCEDURE SetTextPage (page: SHORTCARD);

    (* Changes the active display page. *)

PROCEDURE PositionCursor (visible: BOOLEAN;  position: CARDINAL;
                                       blockcursor: BOOLEAN);

    (* Displays a blinking screen cursor at the specified position.     *)

END TextVideo.
```

TextVideo

```
DEFINITION MODULE TextVideo;

        (*******************************************************)
        (*                                                     *)
        (*              Low-level screen functions             *)
        (*                                                     *)
        (*      The function of this module is to look after   *)
        (*      the low-level screen initialisation needed     *)
        (*      by the text-mode modules GlassTTY and Windows. *)
        (*      This reduces the likelihood of conflicts       *)
        (*      arising in programs which import both of those *)
        (*      modules.                                       *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        25 October 1993                *)
        (*  Status:             OK                             *)
        (*                                                     *)
        (*******************************************************)

(*************************************************************************)
(*                                                                       *)
(*              WARNING         WARNING         WARNING                  *)
(*                                                                       *)
(*  The operations performed by this module are among the rare examples *)
(*  where a programming error can cause physical damage to the          *)
(*  hardware.  Do NOT attempt to modify anything in this module unless  *)
(*  you have a detailed understanding of how the video adaptors work.   *)
(*  In particular, beware of putting inappropriate values into the      *)
(*  registers of the 6845 video controller.                             *)
(*                                                                      *)
(*  It is normal for the screen picture to "bloom" or "bounce" after    *)
(*  a mode change (e.g. changing from a text mode to a graphics mode).  *)
(*  Some monitors are worse than others for this, but all should        *)
(*  regain synchronism after a second or so.  A bright dot at the       *)
(*  centre of the screen or a "tearing" effect are NOT normal; if the   *)
(*  screen display goes crazy you should turn off the monitor and/or    *)
(*  reset the computer immediately, since it probably means that the    *)
(*  software is not compatible with your hardware.                      *)
(*                                                                      *)
(*  The procedures in this module are for the use of other library      *)
(*  modules such as GlassTTY and Windows.  They should not normally     *)
(*  be called directly from applications programs.                      *)
(*                                                                      *)
(*  DISCLAIMER: This software is supplied "as is" with no warranty as   *)
(*  to its compatibility with your hardware.  It has been tested on a   *)
(*  small sample of computers, but because of the wide variety of       *)
(*  video adaptors on the market it is impossible to be certain that    *)
(*  it will work correctly on all hardware configurations.              *)
```

```
PROCEDURE TerminationMessage (VAR (*OUT*) message: ARRAY OF CHAR): BOOLEAN;

    (* Returns the message supplied by the caller of the Crash          *)
    (* procedure.  The function result is TRUE if such a message        *)
    (* exists, and FALSE if Crash was never called.                     *)

END TerminationControl.
```

TerminationControl

```
DEFINITION MODULE TerminationControl;

        (*****************************************************************)
        (*                                                               *)
        (*           Support for program termination procedures          *)
        (*                                                               *)
        (*  The facility provided is that the caller can specify any     *)
        (*  number of parameterless procedures to be called when the     *)
        (*  program terminates.  This allows each module which needs it  *)
        (*  to have a final cleanup procedure to do things like          *)
        (*  releasing allocated memory, closing open files, etc.  The    *)
        (*  termination procedures are called in a last-in first-out     *)
        (*  order, which generally means that the higher-level modules   *)
        (*  are dealt with before lower-level modules (which is usually  *)
        (*  what we want).                                               *)
        (*                                                               *)
        (*  Multipass termination processing is supported by allowing    *)
        (*  any termination handler to itself install another handler.   *)
        (*  In such a case the new handler is not executing until the    *)
        (*  current list of waiting handlers is exhausted.  Multipass    *)
        (*  processing is needed when, for example, part of a module     *)
        (*  shutdown cannot be completed until it is guaranteed that     *)
        (*  all multitasking has ceased.                                 *)
        (*                                                               *)
        (*      Programmer:     P. Moylan                                *)
        (*      Last edited:    3 February 1994                          *)
        (*      Status:         OK                                       *)
        (*                                                               *)
        (*  WARNING: the implementation of this facility is heavily      *)
        (*  system-dependent; some systems do not provide any way to     *)
        (*  regain control at program termination.                       *)
        (*                                                               *)
        (*****************************************************************)

PROCEDURE SetTerminationProcedure (TP: PROC);

    (* Adds TP to the list of procedures which will be called just      *)
    (* before program termination.  The list is ordered such that the   *)
    (* last procedure added will be the first one called.  Exception:   *)
    (* if termination is already in progress when this procedure is     *)
    (* called, then TP will not be called until all of the existing     *)
    (* termination procedures have been called.  This rule permits      *)
    (* multi-pass termination processing, where necessary, by letting   *)
    (* termination procedures themselves install more termination       *)
    (* procedures.                                                      *)

PROCEDURE Crash (message: ARRAY OF CHAR);

    (* Terminates the program with an error report.     *)
```

```
PROCEDURE CreateQueue (VAR (*OUT*) KQ: TaskQueue);

    (* Creates an initially empty queue.                                  *)

PROCEDURE MarkAsReady (VAR (*INOUT*) FromQ: TaskQueue);

    (* Takes the first task from queue FromQ and puts it on the ready     *)
    (* list, or runs it immediately if it has higher priority than the    *)
    (* task which called MarkAsReady.                                     *)

PROCEDURE QueueAndSwitchTasks (VAR (*INOUT*) KQ: TaskQueue);

    (* Puts the current task onto list KQ, and switches to the            *)
    (* highest-priority ready task.                                       *)

(************************************************************************)
(*                                                                      *)
(*                           TIMER OPERATIONS                           *)
(*                                                                      *)
(*   These procedures are called by the Timer and Semaphores modules    *)
(*                                                                      *)
(************************************************************************)

PROCEDURE Delay (sleeptime: INTEGER);

    (* Puts the calling task to sleep for the specified number of clock   *)
    (* ticks.                                                             *)

PROCEDURE QueueWithTimeout (VAR (*INOUT*) KQ: TaskQueue;
                                    TimeLimit: INTEGER): BOOLEAN;

    (* Like procedure QueueAndSwitchTasks, this procedure puts the        *)
    (* current task on the tail of list KQ, and gives control to the      *)
    (* highest-priority ready task.  The difference is that we allow      *)
    (* this task to remain on KQ for at most TimeLimit timer ticks.       *)
    (* If the task is removed from KQ before the time limit expires,      *)
    (* we return a result of FALSE.  If the time limit expires first,     *)
    (* we remove the queued task from KQ anyway, and make it runnable,    *)
    (* and return a result of TRUE when it does run.                      *)
    (* Note: this procedure may be called only from inside the kernel.    *)

PROCEDURE CheckSleepers;

    (* Called from the timer interrupt routine, to deal with sleeping     *)
    (* tasks.                                                             *)

PROCEDURE TimeSliceCheck;

    (* Called from the timer interrupt routine, to check whether the      *)
    (* current task has used up its time slice, and to perform a task     *)
    (* switch if so.                                                      *)

    (* This procedure does nothing if time-slicing is disabled.           *)
    (* Time-slicing is controlled by a constant TimeSlicingEnabled        *)
    (* inside the implementation module.                                  *)

END TaskControl.
```

```
(*************************************************************************)
(*                                                                       *)
(*                  SUPPORT FOR INTERRUPT HANDLERS                       *)
(*                                                                       *)
(*************************************************************************)

PROCEDURE CreateInterruptTask (InterruptNumber: CARDINAL;
                               StartAddress: PROC;  taskname: NameString);

    (* Introduces an interrupt task to the system.  The first parameter *)
    (* is the hardware-defined interrupt number, and the second is the  *)
    (* address of the procedure whose code is the interrupt handler.    *)
    (* An interrupt task differs from an ordinary task in that, when it *)
    (* is not running, it is idle rather than ready, and the dispatcher *)
    (* does not consider it to be among the tasks eligible to run.      *)
    (* Rather, it is run by a task switch which is made directly by the *)
    (* assembly language routine which fields the interrupt.  When the  *)
    (* interrupt task has responded to the interrupt, it must call      *)
    (* procedure WaitForInterrupt to put itself back in the idle state. *)
    (* On the next interrupt, it will continue from just after the call *)
    (* to WaitForInterrupt.  Normally, therefore, the interrupt task    *)
    (* will be written as an infinite loop.  If for any reason the      *)
    (* interrupt task exits by falling out of the bottom of its code,   *)
    (* it will be destroyed in the same way as a normal task which      *)
    (* terminates.  That could be fatal, unless steps have been taken   *)
    (* to reset the interrupt vector.                                   *)

(*************************************************************************)
(*                                                                       *)
(*                      DEVICE DRIVER SUPPORT                            *)
(*                                                                       *)
(*  The following procedure may be called only by an interrupt task.     *)
(*                                                                       *)
(*************************************************************************)

PROCEDURE WaitForInterrupt;

    (* Called by an interrupt task, to make itself dormant until the    *)
    (* next interrupt comes along.  It is not necessary to specify      *)
    (* the interrupt number, since this was fixed at the time the       *)
    (* interrupt task was created.                                      *)

    (* Warning: this procedure should never be called by a task which   *)
    (* is not an interrupt task.                                    *)

(*************************************************************************)
(*                                                                       *)
(*                  PROCEDURES PRIVATE TO THE KERNEL                     *)
(*                                                                       *)
(*  The remaining declarations in this module are needed because the     *)
(*  kernel is made up of several modules.  (Unfortunately, there is no   *)
(*  way in Modula-2 to export something to a separately compiled module  *)
(*  without making it visible to everyone; so please close your eyes     *)
(*  at this point.)  The procedures declared here should be called only  *)
(*  from the innermost parts of the operating system.                    *)
(*                                                                       *)
(*************************************************************************)

TYPE TaskQueue; (* is private *)
```

```
(*************************************************************************)
(*                                                                       *)
(*                LOCKS FOR CRITICAL SECTION PROTECTION                  *)
(*                                                                       *)
(*  Note that we distinguish between a Lock and a Semaphore.             *)
(*  A Semaphore is a general semaphore - whose operations are defined    *)
(*  in module Semaphores - which can be used for general inter-task      *)
(*  interlocking.  A Lock is similar to a binary semaphore (with a       *)
(*  more efficient implementation than a Semaphore), but may be used     *)
(*  only in a strictly nested fashion and is therefore useful only       *)
(*  for critical section protection.  No task should perform a           *)
(*  semaphore Wait while it holds a Lock.  Priority inheritance is       *)
(*  used for Locks - that is, a task holding a Lock will have its        *)
(*  priority temporarily increased as long as it is blocking another     *)
(*  task of higher priority - but not for Semaphores.                    *)
(*                                                                       *)
(*************************************************************************)

TYPE Lock;      (* is private *)

PROCEDURE CreateLock (VAR (*OUT*) L: Lock);

    (* Creates a new lock. *)

PROCEDURE DestroyLock (VAR (*INOUT*) L: Lock);

    (* Disposes of a lock. *)

PROCEDURE Obtain (L: Lock);

    (* Obtains lock L, waiting if necessary. *)

PROCEDURE Release (L: Lock);

    (* Releases lock L - which might unblock some other task. *)

PROCEDURE ReleaseAllLocks;

    (* Releases all locks held by the current task.  Application-level  *)
    (* tasks normally won't need to call this procedure; it is          *)
    (* provided to support the system shutdown function and for things  *)
    (* like "emergency abort" operations.                               *)
```

```
PROCEDURE CreateTask (StartAddress: PROC;  taskpriority: PriorityLevel;
                                           taskname: NameString);

    (* Must be called to introduce a task to the system. The first      *)
    (* parameter, which should be the name of a procedure containing    *)
    (* the task code, gives the starting address.  The second parameter *)
    (* is the task's base priority.  If this task has a higher priority *)
    (* than its creator, it will run immediately.  Otherwise, it        *)
    (* becomes ready.                                                   *)

    (* The effective priority of a task can be higher than its base     *)
    (* priority, as the result of priority inheritance.  This happens   *)
    (* when the task holds a lock on which a higher-priority task is    *)
    (* blocked.                                                         *)

    (* NOTE: If time-slicing is enabled, tasks of equal priority share  *)
    (* processor time on a round-robin basis.  To disable this feature, *)
    (* set the constant TimeSlicingEnabled (inside the implementation   *)
    (* module) to FALSE.                                                *)

    (* Tasks of different priorities never share time.  When a          *)
    (* high-priority task becomes able to run, there is an immediate    *)
    (* task switch.                                                     *)

    (* A task terminates itself either by an explicit call to TaskExit, *)
    (* or simply by falling out of the bottom of its code.              *)
    (* There is no provision for tasks to kill other tasks.  Suicide    *)
    (* is legal, but murder is not.                                     *)

PROCEDURE UsingFloatingPoint;

    (* Tells the kernel that this task is one which performs floating   *)
    (* point operations.  The consequence is that the state of the      *)
    (* (physical or emulated, as applicable) floating point processor   *)
    (* is saved on a task switch.  This call is usually unnecessary;    *)
    (* the default assumption is that interrupt tasks do not perform    *)
    (* floating point arithmetic but that all other tasks may.          *)
    (* NOTE: It is never acceptable for an interrupt task to call this  *)
    (* procedure.                                                       *)

PROCEDURE NotUsingFloatingPoint;

    (* Tells the kernel that this task does not perform any floating    *)
    (* point operations.  Calling this (optional) procedure speeds up   *)
    (* task switching slightly, but it does put the onus on the caller  *)
    (* to be certain that it does no floating point operations.  If you *)
    (* call this procedure and then perform floating point arithmetic   *)
    (* anyway, you can get severe and erratic floating point errors.    *)

PROCEDURE TaskExit;

    (* Removes the currently running task from the system, and performs *)
    (* a task switch to the next ready task.                            *)

    (* There is normally no need for a task to call this procedure,     *)
    (* because it is automatically called when the task code "falls out *)
    (* the bottom" by executing its final procedure return.  The stack  *)
    (* is set up, at the time a task is created, in such a way that     *)
    (* TaskExit will be entered at that time.                           *)
```

TaskControl

```
DEFINITION MODULE TaskControl;

        (*****************************************************************)
        (*                                                               *)
        (*   Data structures internal to the kernel of the operating     *)
        (*     system; the dispatcher of the operating system; and       *)
        (*                  related procedures.                          *)
        (*                                                               *)
        (*      This version supports priority inheritance.              *)
        (*                                                               *)
        (*      Programmer:     P. Moylan                                *)
        (*      Last edited:    15 September 1993                        *)
        (*      Status:         OK                                       *)
        (*                                                               *)
        (*****************************************************************)

FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

(*************************************************************************)
(*                                                                       *)
(*                         END-USER PROCEDURES                           *)
(*                                                                       *)
(*************************************************************************)

CONST MaxPriority = 15;

TYPE
    PriorityLevel = [0..MaxPriority];
    NameString = ARRAY [0..15] OF CHAR;
```

```
PROCEDURE Beep;
    (* Produces a short ”beep” noise.   *)
END SoundEffects.
```

SoundEffects

```
DEFINITION MODULE SoundEffects;

        (*****************************************************************)
        (*                                                               *)
        (*           Procedures to produce audible output.               *)
        (*                                                               *)
        (*  Note that this is a low-level output module, in that the     *)
        (*  method of specifying note frequencies is defined in terms    *)
        (*  of what is convenient for the hardware rather than what is   *)
        (*  convenient for the user.  The higher-level features are      *)
        (*  provided in module Music.                                    *)
        (*                                                               *)
        (*      Programmer:     P. Moylan                                *)
        (*      Last edited:    28 May 1989                              *)
        (*                                                               *)
        (*    Status:                                                    *)
        (*      Seems to be working.                                     *)
        (*                                                               *)
        (*****************************************************************)

FROM Semaphores IMPORT
    (* type *)  Semaphore;

TYPE Note = RECORD
                period, duration: CARDINAL;
            END (*RECORD*);

    (* The "period" field of a note record specifies the note frequency *)
    (* indirectly, by specifying a divisor for the 1.193 MHz main clock *)
    (* frequency.  The "duration" is measured in milliseconds.          *)

PROCEDURE Play (VAR (*IN*) playdata: ARRAY OF Note;
                        VAR (*INOUT*) done: Semaphore);

    (* Adds the array to the list of music queued up waiting to be      *)
    (* played.  The actual playing occurs automatically, after the end  *)
    (* of anything earlier in the queue.  The output is asynchronous,   *)
    (* in the sense that we return from this procedure before the       *)
    (* playing is over, and perhaps even before it has started.  The    *)
    (* array playdata must remain undisturbed until the caller receives *)
    (* a Signal on semaphore "done".  That is, the caller must perform  *)
    (* a Wait(done) before re-using or destroying the playdata.  Note   *)
    (* that a procedure return will destroy playdata if it happens to   *)
    (* be a local variable of that procedure.                           *)

    (* A duration code of 0 indicates the end of the data, in cases     *)
    (* where the data do not fill the entire array.                     *)

    (* A period code of 1, with a nonzero duration, produces silence    *)
    (* for the requested duration.                                      *)
```

Skeleton

```
DEFINITION MODULE Skeleton;

        (*********************************************************)
        (*                                                       *)
        (*                                                       *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        27 February 1993                 *)
        (*  Status:                                              *)
        (*                                                       *)
        (*********************************************************)

(*************************************************************************)

PROCEDURE P;

END Skeleton.
```

```
PROCEDURE GetPosBut (VAR (*OUT*) buttons: ButtonSet;
                                VAR (*OUT*) Xposition, Yposition: CARDINAL);

    (* Returns the current mouse position and the state of the buttons. *)
    (* Note: the units here are not the same as for procedure           *)
    (* GetTextMousePosition.  In both this procedure and in the event   *)
    (* handlers the position is presented in units of 1/8th of a        *)
    (* character width or height.                                       *)

PROCEDURE SetPage (page: CARDINAL);

    (* Sets the hardware screen page where the mouse is visible. *)

PROCEDURE SetHorizontalLimits (MinX, MaxX : CARDINAL);
PROCEDURE SetVerticalLimits (MinY, MaxY : CARDINAL);

    (* Sets the cursor limits. *)

PROCEDURE ShowCursor;

    (* Makes the mouse cursor visible on the screen.  Note: we allow    *)
    (* nesting in ShowCursor/HideCursor calls, so that for example if   *)
    (* you've called HideCursor twice then you need to call ShowCursor  *)
    (* twice to make the cursor reappear.                               *)

PROCEDURE HideCursor;

    (* Makes the mouse cursor invisible. *)

PROCEDURE SetEventHandler (DetectedEvents: EventSet;
                                        Handler: EventHandler);

    (* Nominates the procedure to be called whenever an event in the    *)
    (* set DetectedEvents occurs.  Note: the Handler is like an         *)
    (* interrupt procedure, in that it is executing in the context of   *)
    (* an unknown task; typically it should probably restrict its       *)
    (* actions to fairly elementary things, like a Signal to wake up    *)
    (* the task that really wants to know about the event.              *)

END SerialMouse.
```

SerialMouse

```
DEFINITION MODULE SerialMouse;

        (*******************************************************)
        (*                                                     *)
        (*                Serial mouse driver                  *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        22 February 1994               *)
        (*  Status:             OK                             *)
        (*                                                     *)
        (*      This module provides support for a mouse on    *)
        (*      COM1.  Note that this is not the only mouse    *)
        (*      driver in PMOS.  Module Mouse chooses which    *)
        (*      mouse driver to use based on the settings      *)
        (*      in ConfigurationOptions.DEF.                   *)
        (*                                                     *)
        (*******************************************************)

TYPE
    Buttons = (LeftButton, RightButton, MiddleButton);

    ButtonSet = SET OF Buttons;

    Events =  (Motion, LeftDown, LeftUp, RightDown, RightUp, MiddleDown,
              MiddleUp);

    EventSet = SET OF Events;

    EventHandler = PROCEDURE (EventSet,         (* condition mask *)
                              ButtonSet,        (* Button state *)
                              CARDINAL,         (* horizontal cursor position
*)
                              CARDINAL);        (* vertical cursor position *)

CONST DriverInstalled = TRUE;

PROCEDURE ResetMouse (VAR (*OUT*) MousePresent: BOOLEAN;
                      VAR (*OUT*) NumberOfButtons: CARDINAL);

    (* Initializes mouse, returning MousePresent as FALSE if no mouse   *)
    (* available and as TRUE if it is, and NumberOfButtons as the       *)
    (* number of buttons for the mouse if installed.                    *)

PROCEDURE SetCursorPos (X, Y : CARDINAL);

    (* Sets the mouse cursor position. *)
```

SerialIO

```
DEFINITION MODULE SerialIO;

        (*********************************************************)
        (*                                                       *)
        (*           Serial I/O through the COM ports            *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        7 March 1993                     *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*********************************************************)

TYPE
    SerialChannelNumber = [1..4];
    WordLength = SHORTCARD [5..8];
    Parity = (NoParity, OddParity, EvenParity, Always0, Always1);
    StopBits = SHORTCARD [1..2];

(*************************************************************************)

PROCEDURE InitSerialChannel (chan: SerialChannelNumber;  baud: CARDINAL;
                             wordlength: WordLength;
                             parity: Parity; stopbits: StopBits);

    (* Performs initialisation on channel "chan".                        *)

PROCEDURE ReadSerial (chan: SerialChannelNumber;  VAR (*OUT*) value: BYTE);

    (* Reads one byte from a serial channel.    *)

PROCEDURE WriteSerial (chan: SerialChannelNumber;  value: BYTE);

    (* Sends one byte to an output channel.     *)

END SerialIO.
```

Semaphores

```
DEFINITION MODULE Semaphores;

        (*******************************************************)
        (*                                                     *)
        (*      Defines the semaphore data type, and the two   *)
        (*      basic operations on a semaphore.        *)
        (*                                                     *)
        (*      Programmer:     P. Moylan                      *)
        (*      Last edited:    16 August 1993                 *)
        (*      Status:         OK                             *)
        (*                                                     *)
        (*******************************************************)

TYPE Semaphore;         (* is private *)

PROCEDURE CreateSemaphore (VAR (*OUT*) s: Semaphore; InitialValue: CARDINAL);

    (* Creates semaphore s, with the given initial value and an empty   *)
    (* queue.                                                           *)

PROCEDURE DestroySemaphore (VAR (*INOUT*) s: Semaphore);

    (* Reclaims any space used by semaphore s.  Remark:  It is not at   *)
    (* all obvious what should be done with any tasks which happen to   *)
    (* be blocked on this semaphore (should they be unblocked, or       *)
    (* killed?).  At present we take the easy way out and assume that   *)
    (* there are no pending operations on s at the time that it is      *)
    (* destroyed.                                                       *)

PROCEDURE Wait (VAR (*INOUT*) s: Semaphore);

    (* Decrements the semaphore value.  If the value goes negative,     *)
    (* the calling task is blocked and there is a task switch.          *)

PROCEDURE TimedWaitT (VAR (*INOUT*) s: Semaphore;
                      TimeLimit: INTEGER;  VAR (*OUT*) TimedOut: BOOLEAN);

    (* Like procedure Wait, except that it returns with TimedOut TRUE   *)
    (* if the corresponding Signal does not occur within TimeLimit      *)
    (* clock ticks.  Note that this procedure is not recommended for    *)
    (* general use, because "clock ticks" is not a convenient unit of   *)
    (* time for most callers.  For a more useful version, see procedure *)
    (* TimedWait in module Timer.                                       *)

PROCEDURE Signal (VAR (*INOUT*) s: Semaphore);

    (* Increments the semaphore value.  Unblocks one waiting task,      *)
    (* if there was one.                                                *)

END Semaphores.
```

ScreenGeometry

```
DEFINITION MODULE ScreenGeometry;

        (*********************************************************)
        (*                                                       *)
        (*          Support module for screen graphics           *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        6 October 1993                   *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

TYPE
    Point = RECORD
                x, y: INTEGER;
            END (*RECORD*);

    Rectangle = RECORD
                    top, bottom, left, right: INTEGER;
                END (*RECORD*);

(************************************************************************)

PROCEDURE Inside (x, y: INTEGER;  R: Rectangle): BOOLEAN;

    (* Returns TRUE iff point (x,y) is in (or on the border of) R.      *)

PROCEDURE Adjacent (R1, R2: Rectangle;
                                VAR (*OUT*) union: Rectangle): BOOLEAN;

    (* If the union of R1 and R2 is itself a rectangle, returns TRUE    *)
    (* and sets "union" to be the combined rectangle.  Otherwise        *)
    (* returns FALSE, and the "union" result is meaningless.            *)

PROCEDURE TrimLine (VAR (*INOUT*) end1, end2: Point;  R: Rectangle): BOOLEAN;

    (* Modifies end1 and end2, if necessary, to cut off the ends of     *)
    (* the line from end1 to end2 which do not fit in R.                *)
    (* Returns FALSE if none of the line passes through the rectangle.  *)

END ScreenGeometry.
```

```
(*************************************************************************)
(*                          CLOSING A STRUCTURE                          *)
(*************************************************************************)

PROCEDURE DeleteStructure (VAR (*INOUT*) S: Structure);

    (* Deletes structure S.  Calling this procedure is optional, but is *)
    (* recommended in order to reclaim memory space when S is no longer *)
    (* needed (and it makes it clearer in the program listing that S    *)
    (* will no longer be used).  Note that this procedure does NOT      *)
    (* delete the variables to which S gives access; if, for example,   *)
    (* you were working with lists and menus then those lists and menus *)
    (* continue to exist.  DeleteStructure simply deletes the overhead  *)
    (* data which was originally allocated by this module for its own   *)
    (* purposes.                                                        *)

END ScreenEditor.
```

```
(*************************************************************************)
(*                          FOR ADVANCED USERS                           *)
(*************************************************************************)

PROCEDURE MenuField (VAR (*IN*) variable: CARDINAL;
                screenrow, screencolumn, lines, width: CARDINAL;
                M: Menu): Structure;

    (* Creates a one-field structure for editing a cardinal variable    *)
    (* via menu selection.  The caller must ensure that M has already   *)
    (* been defined by a call to Menus.                                 *)

PROCEDURE ListField (VAR (*IN*) variable: List;
                               screenrow, screencolumn: CARDINAL;
                                       f: ListFormat): Structure;

    (* Creates a structure for editing a linear list.  The caller must  *)
    (* ensure that f has been defined by a call to module ListEditor.   *)
    (* This procedure does not add any features beyond what ListEditor  *)
    (* provides, but by returning a result of type Structure it allows  *)
    (* lists and scalars to be mixed in the same editing window.        *)

PROCEDURE CreateField (VariableAddress: ADDRESS;  ftype: FieldType;
                       screenrow, screencolumn, width: CARDINAL): Structure;

    (* Creates a new structure consisting of a single field.  Before    *)
    (* calling this procedure, the caller should make sure, by calling  *)
    (* FieldEditor.DefineFieldType if necessary, that ftype is a type   *)
    (* already known to module FieldEditor.                             *)

(*************************************************************************)
(*              CREATING MULTI-FIELD EDITING STRUCTURES                  *)
(*************************************************************************)

PROCEDURE Combine (VAR (*INOUT*) A: Structure;  B: Structure);

    (* Strips all of the fields from B and adds them to the existing    *)
    (* fields of A.  Note that B is destroyed in the process.           *)

PROCEDURE MakeArray (VAR (*INOUT*) S: Structure;  count: CARDINAL;
                addroffset, rowoffset, coloffset: CARDINAL);

    (* Creates a structure for an array of count elements, where on     *)
    (* entry S is a structure already created for the first array       *)
    (* element.  Parameter addroffset is the difference between         *)
    (* adjacent array elements.  The remaining two parameters give the  *)
    (* offset on the screen between the starting positions of adjacent  *)
    (* array elements.                                                  *)

(*************************************************************************)
(*                               EDITING                                 *)
(*************************************************************************)

PROCEDURE ScreenEdit (w: Window;  S: Structure;  VAR (*OUT*) abort: BOOLEAN);

    (* Displays structure S in window w, and allows the keyboard user   *)
    (* to edit the components of S.  It is assumed that w is already    *)
    (* open and that S has already been fully defined.  Returns         *)
    (* abort=TRUE if user aborted the editing with the Esc key.         *)
```

ScreenEditor

```
DEFINITION MODULE ScreenEditor;

        (*********************************************************)
        (*                                                       *)
        (*              Screen data capture                      *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        28 April 1993                    *)
        (*  Status:                                              *)
        (*      Basic features working, but see faults in        *)
        (*      module RowEditor.                                *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

FROM Windows IMPORT
    (* type *)  Window;

FROM ListEditor IMPORT
    (* type *)  List, ListFormat;

FROM Menus IMPORT
    (* type *)  Menu;

FROM FieldEditor IMPORT
    (* type *)  FieldType;

(************************************************************************)

TYPE
    Structure;          (* is private *)

(************************************************************************)
(*                  INTRODUCING A NEW FIELD TO THE SYSTEM               *)
(************************************************************************)

PROCEDURE ByteField (VAR (*IN*) variable: BYTE;
                        screenrow, screencolumn, width: CARDINAL): Structure;

    (* Creates a one-field structure for editing a BYTE variable.       *)

PROCEDURE CardinalField (VAR (*IN*) variable: CARDINAL;
                        screenrow, screencolumn, width: CARDINAL): Structure;

    (* Creates a one-field structure for editing the given CARDINAL     *)
    (* variable.                                                        *)

PROCEDURE RealField (VAR (*IN*) variable: REAL;
                        screenrow, screencolumn, width: CARDINAL): Structure;

    (* Creates a one-field structure for editing a REAL variable.       *)
```

```
(************************************************************************)

PROCEDURE VideoKind (): VideoAdaptorType;

    (* Returns the display adaptor type.  This is a best guess, and it  *)
    (* is possible that some adaptor types will be misclassified.       *)
    (* In the present version, most SVGA adaptors will be reported as   *)
    (* VESA, and no distinction is drawn between the "ordinary"         *)
    (* Hercules adaptor and the Hercules Plus or Hercules InColor.      *)

PROCEDURE Supported (mode: CARDINAL): BOOLEAN;

    (* Returns TRUE iff the specified mode is a mode supported  *)
    (* by the hardware and by this module.                      *)

PROCEDURE SetVideoMode (newmode: CARDINAL;  ClearScreen: BOOLEAN): BOOLEAN;

    (* Sets the video mode.  The mode numbers are as defined in the     *)
    (* BIOS, plus HercGraphics to denote the Hercules graphics mode,    *)
    (* plus whatever the VESA BIOS (if present) will support.           *)
    (* Returns TRUE iff the mode change was successful.                 *)

PROCEDURE RestoreOriginalMode;

    (* Sets the video mode back to what it was before this program ran. *)

PROCEDURE SelectReadBank (bank: CARDINAL);

    (* Switches to a new bank of screen memory for reading.  Should be  *)
    (* used only with the adaptors which support the high-resolution    *)
    (* modes using bank switching.                                      *)

PROCEDURE SelectWriteBank (bank: CARDINAL);

    (* Switches to a new bank of screen memory for writing.  Should be  *)
    (* used only with the adaptors which support the high-resolution    *)
    (* modes using bank switching.                                      *)

PROCEDURE GetModeData (VAR (*OUT*) ScreenSegment, IObase: CARDINAL);

    (* Returns the segment of the screen memory and the port number     *)
    (* of the video controller.                                         *)

PROCEDURE GetExtendedModeInformation (
                      VAR (*INOUT*) BytesPerRow,
                              MaxX, MaxY, MaxColour: CARDINAL;
                      VAR (*INOUT*) BitsPerPixel: SHORTCARD;
                      VAR (*INOUT*) MultiBank: BOOLEAN);

    (* Returns information supplied by a VESA driver (if present) for   *)
    (* the current mode.  If the information is not available, the      *)
    (* parameter values are left unchanged.                             *)

PROCEDURE WaitForVerticalRetrace;

    (* Busy wait until we reach the vertical retrace period.            *)
    (* Warning: I wrote this quickly for one specific application, and  *)
    (* haven't gotten around to getting it right for the general case.  *)

END Screen.
```

Screen

```
DEFINITION MODULE Screen;

        (*******************************************************)
        (*                                                     *)
        (*              Low-level screen functions             *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        30 January 1994                *)
        (*  Status:             Working                        *)
        (*                                                     *)
        (*******************************************************)

(*********************************************************************)
(*                                                                   *)
(*              WARNING         WARNING         WARNING              *)
(*                                                                   *)
(*  The operations performed by this module are among the rare examples *)
(*  where a programming error can cause physical damage to the       *)
(*  hardware.  Do NOT attempt to modify anything in this module unless  *)
(*  you have a detailed understanding of how the video adaptors work.   *)
(*  In particular, beware of putting inappropriate values into the      *)
(*  registers of the 6845 video controller.                          *)
(*                                                                   *)
(*  It is normal for the screen picture to "bloom" or "bounce" after    *)
(*  a mode change (e.g. changing from a text mode to a graphics mode).  *)
(*  Some monitors are worse than others for this, but all should     *)
(*  regain synchronism after a second or so.  A bright dot at the       *)
(*  centre of the screen or a "tearing" effect are NOT normal; if the   *)
(*  screen display goes crazy you should turn off the monitor and/or    *)
(*  reset the computer immediately, since it probably means that the    *)
(*  software is not compatible with your hardware.                   *)
(*                                                                   *)
(*  The procedures in this module are for the use of other library      *)
(*  modules such as Graphics and Windows.  They should not normally     *)
(*  be called directly from applications programs.                   *)
(*                                                                   *)
(*  DISCLAIMER: This software is supplied "as is" with no warranty as   *)
(*  to its compatibility with your hardware.  It has been tested on a   *)
(*  small sample of computers, but because of the wide variety of       *)
(*  video adaptors on the market it is impossible to be certain that    *)
(*  it will work correctly on all hardware configurations.           *)
(*                                                                   *)
(*********************************************************************)

CONST HercGraphics = 128+7;

TYPE
    (* This is a list of all the adaptors this module can recognise.    *)

    VideoAdaptorType = (MDA, Hercules, CGA, EGA, VGA, VESA, Trident);
```

```
(*************************************************************************)
(*              CREATING MULTI-FIELD EDITING STRUCTURES                  *)
(*************************************************************************)

PROCEDURE NewRow (VariableAddress: ADDRESS;  ftype: FieldType;
                        screencolumn, width: CARDINAL): StructureRow;

    (* Creates a new row containing a single field.     *)

PROCEDURE NewMenu (VAR (*IN*) variable: CARDINAL;  M: Menu;
                        screencolumn, rows, columns: CARDINAL): StructureRow;

    (* Creates a new row containing a menu field.  The screencolumn     *)
    (* field specifies the leftmost column within the screen window,    *)
    (* the rows and columns fields give the size on the screen.         *)

PROCEDURE NewList (VAR (*IN*) variable: List;  f: ListFormat;
                                screencolumn: CARDINAL): StructureRow;

    (* Creates a new row containing a list field.       *)

PROCEDURE CombineRows (VAR (*INOUT*) A: StructureRow;  B: StructureRow);

    (* Strips all of the fields from B and adds them to the existing    *)
    (* fields of A.  Note that B is destroyed in the process.           *)

(*************************************************************************)
(*      The next few procedures are to support array operations.         *)
(*************************************************************************)

PROCEDURE CopyOfRow (R: StructureRow): StructureRow;

    (* Returns a duplicate copy of R.  Note that the variables to be    *)
    (* edited are not duplicated - only the editor structure which      *)
    (* keeps track of what is being edited.                             *)

PROCEDURE AdjustRow (R: StructureRow;  addroffset, columnoffset: CARDINAL);

    (* Modifies every entry in R by adding addroffset to the variable   *)
    (* address and columnoffset to the screen column.                   *)

PROCEDURE DeleteRow (R: StructureRow);

    (* Deallocates the storage which was used in setting up row R.      *)
    (* Note that this has nothing to do with the space used by          *)
    (* variables to which R gives access; we delete only the overhead   *)
    (* space which was originally allocated by this module.             *)

(*************************************************************************)
(*                               EDITING                                 *)
(*************************************************************************)

PROCEDURE EditRow (w: Window;  R: StructureRow;  screenrow: CARDINAL);

    (* Displays structure R in window w, and allows the keyboard user   *)
    (* to edit the components of R.  It is assumed that w is already    *)
    (* open and that R has already been fully defined.                  *)

END RowEditor.
```

RowEditor

```
DEFINITION MODULE RowEditor;

        (*********************************************************)
        (*                                                       *)
        (*        Screen data capture, for a single row          *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        6 October 1990                   *)
        (*  Status:                                              *)
        (*      Basic features working.  Known faults are:       *)
        (*        1.    (fixed)                                  *)
        (*        2.    The criterion for deciding in which      *)
        (*              field to start editing could be better.  *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

FROM Windows IMPORT
    (* type *)  Window;

FROM FieldEditor IMPORT
    (* type *)  FieldType;

FROM ListEditor IMPORT
    (* type *)  List, ListFormat;

FROM Menus IMPORT
    (* type *)  Menu;

(*************************************************************************)

TYPE
    StructureRow;               (* is private *)

(*************************************************************************)
(*                            SCREEN OUTPUT                              *)
(*************************************************************************)

PROCEDURE WriteRow (w: Window;  R: StructureRow;  line: CARDINAL);

    (* Writes R on row "line" of window w.      *)

PROCEDURE StartColumn (R: StructureRow): CARDINAL;

    (* Returns the screen column of the first field in R.       *)

PROCEDURE DumpRow (w: Window;  R: StructureRow);

    (* For debugging: writes a representation of R to the screen.       *)
```

RealIO

```
DEFINITION MODULE RealIO;

        (*********************************************************)
        (*                                                       *)
        (*              Real I/O using windows.                  *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        10 October 1992                  *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*      NOTE: several procedures which used to be        *)
        (*      in this module have now been moved to            *)
        (*      module Conversions.                              *)
        (*                                                       *)
        (*********************************************************)

FROM Windows IMPORT
    (* type *)  Window;

PROCEDURE WriteReal (w: Window;  number: REAL;  places: CARDINAL);
PROCEDURE WriteLongReal (w: Window;  number: LONGREAL;  places: CARDINAL);

    (* Writes the second argument as a decimal number, right-justified  *)
    (* in a field of "places" places.                                   *)

PROCEDURE ReadReal (w: Window): REAL;
PROCEDURE ReadLongReal (w: Window): LONGREAL;

    (* Reads and converts a numeric string from the keyboard.   *)

PROCEDURE ReadBufferedReal (w: Window;  fieldsize: CARDINAL): REAL;
PROCEDURE ReadBufferedLongReal (w: Window;  fieldsize: CARDINAL): LONGREAL;

    (* Like ReadReal, but allows the user to edit within a field of     *)
    (* the specified size.                                              *)

PROCEDURE EditReal (w: Window;  VAR (*INOUT*) variable: REAL;
                                                    width: CARDINAL);
PROCEDURE EditLongReal (w: Window;  VAR (*INOUT*) variable: LONGREAL;
                                                    width: CARDINAL);

    (* Displays the current value of "variable" at the current cursor   *)
    (* position in window w, using a field width of "width" characters, *)
    (* and gives the user the option of altering the value.             *)

END RealIO.
```

Random

```
DEFINITION MODULE Random;

        (*********************************************************)
        (*                                                       *)
        (*              Random number generator                  *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        12 February 1993                 *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

PROCEDURE RANDOM(): REAL;

    (* Returns a random number from a uniform (0.0, 1.0) distribution.  *)

PROCEDURE Randomize (newseed: LONGCARD);

    (* Resets the seed of the random number generator.  Optional, but   *)
    (* useful for either (a) obtaining a different random number        *)
    (* sequence on each run of a program, or (b) conversely, obtaining  *)
    (* a repeatable experiment.                                         *)

END Random.
```

RandCard

```
DEFINITION MODULE RandCard;

        (*********************************************************)
        (*                                                       *)
        (*              Random number generator                  *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        12 February 1993                 *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

CONST modulus = 2147483647;     (*  2^31 - 1  *)

VAR seed: LONGCARD;     (*   visible in case you want to re-randomize   *)

PROCEDURE RandCardinal (): LONGCARD;

    (* Returns a random number in the range [1..modulus-1], with a      *)
    (* uniform distribution over that range.                            *)

END RandCard.
```

QuickSortModule

```
DEFINITION MODULE QuickSortModule;

        (*******************************************************)
        (*                                                     *)
        (*      In-memory sort using the QuickSort method      *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        4 August 1993                  *)
        (*  Status:             OK                             *)
        (*                                                     *)
        (*******************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

TYPE CompareProc = PROCEDURE (ADDRESS, ADDRESS): BOOLEAN;

    (* A "CompareProc" procedure accepts the addresses of two data      *)
    (* elements, and returns TRUE iff the first is greater than or      *)
    (* equal to the second.  It is at the caller's discretion to define *)
    (* the meaning of "greater or equal" for his or her application.    *)

PROCEDURE QuickSort (VAR (*INOUT*) data: ARRAY OF BYTE;
                                N, EltSize: CARDINAL;  GE: CompareProc);

    (* In-place sort of array data[0..N].  EltSize is the element size, *)
    (* and GE is a user-supplied function to compare elements at two    *)
    (* specified addresses.                                             *)

END QuickSortModule.
```

```
FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

TYPE Queue;     (* is private *)

PROCEDURE CreateQueue (VAR (*OUT*) Q: Queue);

    (* Creates a new queue, initially empty.    *)

PROCEDURE DestroyQueue (Q: Queue);

    (* Destroys queue Q, thus freeing up the space it occupied.  Any    *)
    (* data still on the queue are lost.  After this call, no further   *)
    (* operations should be performed on the queue.                     *)

PROCEDURE AddToQueue (Q: Queue;  DataPointer: ADDRESS);

    (* Places a new element at the tail of queue Q.  The caller has an  *)
    (* obligation to ensure that DataPointer^ remains in existence      *)
    (* for as long as it remains on the queue.                          *)

PROCEDURE TakeFromQueue (Q: Queue): ADDRESS;

    (* Removes and returns a pointer to the datum at the head of the    *)
    (* queue.  If the queue is empty, waits until data available.       *)

PROCEDURE Empty (Q: Queue): BOOLEAN;

    (* Returns TRUE iff Q is empty.  Warning: if more than one task is  *)
    (* using this queue, there is no guarantee as to how long the queue *)
    (* will remain empty.                                               *)

END Queues.
```

Queues

```
DEFINITION MODULE Queues;

        (*********************************************************)
        (*                                                       *)
        (*              Generic queue module.                    *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        12 November 1991                 *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

(*************************************************************************)
(*                                                                       *)
(*  A non-obvious decision to be made in the design of a module like     *)
(*  this is whether to work directly with the caller's data, or with     *)
(*  pointers to the data.  In the present case, this means deciding      *)
(*  whether to implement queues of user data or queues of pointers.      *)
(*  The former choice is superior in terms of clarity and ease of use,   *)
(*  but requires the physical copying of data between the queue and      *)
(*  the caller's data structure.  The latter choice is more efficient,   *)
(*  but requires the caller to be concerned with allocation and          *)
(*  deallocation of data space.  With some languages we would not be     *)
(*  faced with this delicate choice, but Modula-2 has relatively poor    *)
(*  support for generic data structures.                                 *)
(*                                                                       *)
(*  For this module, the decision has been to support the more           *)
(*  efficient but less elegant arrangement: to add something to the      *)
(*  queue, the caller supplies a pointer to the data, which might        *)
(*  require that the caller allocate some space for that data.  When     *)
(*  an item is removed from the queue, what is returned is again a       *)
(*  pointer to the data.  That is, the actual data live in a data        *)
(*  space which is under the control of the user.  This implies that     *)
(*  the caller can - but should not - modify queued data.  This          *)
(*  solution is not as clean as it might have been, but is justified     *)
(*  by the need of some callers of this module for a low-overhead        *)
(*  solution.                                                            *)
(*                                                                       *)
(*  Note, however, that the caller is not required to supply space for   *)
(*  the "bookkeeping" information such as the pointers which link the    *)
(*  queue elements together.  That level of detail is hidden inside      *)
(*  this module, as it should be.                                        *)
(*                                                                       *)
(*  Critical section protection is also provided; that is, a queue       *)
(*  may safely be used by multiple tasks.                                *)
(*                                                                       *)
(*************************************************************************)
```

Printer

```
DEFINITION MODULE Printer;

        (*********************************************************)
        (*                                                       *)
        (*          Device driver for the printer.               *)
        (*                                                       *)
        (*      Author:         P. Moylan                        *)
        (*      Last edited:    21 August 1989                   *)
        (*                                                       *)
        (*      Status:         Working.                         *)
        (*                                                       *)
        (*********************************************************)

PROCEDURE PrintChar (ch: CHAR);

    (* Sends one character to the printer.  NOTE: Many printers have    *)
    (* an internal buffer which saves characters until a whole line of  *)
    (* characters has been received.  Thus, the printing of the current *)
    (* character might be delayed until a carriage return is received.  *)

END Printer.
```

Piano

```
DEFINITION MODULE Piano;

        (*********************************************************)
        (*                                                       *)
        (*              Play notes from the keyboard             *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        5 October 1992                   *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

(*************************************************************************)

PROCEDURE PlayPiano;

    (* Turns the home row of the keyboard into a music keyboard, with   *)
    (* the row above being the black keys.  Type Esc to exit.           *)

END Piano.
```

```
PROCEDURE WriteInt (w: Window;  number: INTEGER);

    (* Writes the second argument as a decimal number.  *)

PROCEDURE WriteRJCard (w: Window;  number, fieldsize: CARDINAL);

    (* Like WriteCard, but the result is right justified in a field     *)
    (* of fieldsize characters.                                         *)

PROCEDURE WriteRJShortCard (w: Window;  number: SHORTCARD;
                                        fieldsize: CARDINAL);

    (* Like WriteShortCard, but the result is right justified in a      *)
    (* field of fieldsize characters.                                   *)

PROCEDURE WriteRJLongCard (w: Window;  number: LONGCARD; fieldsize: CARDINAL);

    (* Like WriteLongCard, but the result is right justified in a field *)
    (* of fieldsize characters.                                         *)

PROCEDURE ReadCard (w: Window;  VAR (*OUT*) number: CARDINAL);

    (* Reads a decimal number.  *)

PROCEDURE ReadBufferedCardinal (w: Window;  fieldsize: CARDINAL): CARDINAL;

    (* Reads a decimal number.  The difference between this and         *)
    (* ReadCard is that the user is given a reverse-video field of a    *)
    (* fixed width to work in, and is able to use the cursor control    *)
    (* keys to edit within that field.                                  *)

PROCEDURE EditCardinal (w: Window;  VAR (*INOUT*) value: CARDINAL;
                                                  fieldsize: CARDINAL);

    (* Screen editing of a decimal number. *)

END NumericIO.
```

NumericIO

```
DEFINITION MODULE NumericIO;

        (*****************************************************************)
        (*                                                               *)
        (*              Numeric I/O using windows.                       *)
        (*                                                               *)
        (*  Programmer:         P. Moylan                                *)
        (*  Last edited:        18 February 1994                         *)
        (*  Status:             OK                                       *)
        (*                                                               *)
        (*****************************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

FROM Windows IMPORT
    (* type *)  Window;

PROCEDURE WriteHexByte (w: Window;  number: BYTE);

    (* Writes the second argument as a two-digit hexadecimal number.    *)

PROCEDURE WriteHexWord (w: Window;  number: CARDINAL);

    (* Writes the second argument as a four-digit hexadecimal number.   *)

PROCEDURE WriteHexLongword (w: Window;  number: LONGCARD);

    (* Writes the second argument as an eight-digit hexadecimal number. *)

PROCEDURE EditHexByte (w: Window;  VAR (*INOUT*) value: BYTE);

    (* Screen editing of a 2-digit hexadecimal value *)

PROCEDURE EditHexWord (w: Window;  VAR (*INOUT*) value: CARDINAL);

    (* Screen editing of a 4-digit hexadecimal value *)

PROCEDURE EditHexLongword (w: Window;  VAR (*INOUT*) value: LONGCARD);

    (* Screen editing of an 8-digit hexadecimal value *)

PROCEDURE WriteAddress (w: Window;  addr: ADDRESS);

    (* Writes a segmented address to the screen.        *)

PROCEDURE WriteCard (w: Window;  number: CARDINAL);

    (* Writes the second argument as a decimal number.  *)

PROCEDURE WriteShortCard (w: Window;  number: SHORTCARD);

    (* Writes the second argument as a decimal number.  *)

PROCEDURE WriteLongCard (w: Window;  number: LONGCARD);

    (* Writes the second argument as a decimal number.  *)
```

MusicDemonstration

```
DEFINITION MODULE MusicDemonstration;

        (*********************************************************)
        (*                                                       *)
        (*              Test of module Music.                    *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        24 June 1989                     *)
        (*  Status:                                              *)
        (*      Working.  Musically, it could be polished up a   *)
        (*      little, but from the software development        *)
        (*      viewpoint it is complete - module Music does     *)
        (*      indeed work as desired.                          *)
        (*                                                       *)
        (*********************************************************)

PROCEDURE WaitForEndOfMusic;

    (* Synchronization procedure: does not return until the music       *)
    (* demonstration is over.                                           *)

END MusicDemonstration.
```

```
(*************************************************************************)
(*                                                                       *)
(*  In the "notes" argument to PlayMusic, the following options are      *)
(*  accepted:                                                            *)
(*      C D E F G A B   the usual notation for a note.                   *)
(*      b #             flat or sharp, modifying the previously given    *)
(*                       note.  There is no legality checking; for       *)
(*                       example, B# is illegal but the software does    *)
(*                       check for this.                                 *)
(*      R               a rest.                                          *)
(*      *               double the duration of the following notes.      *)
(*      /               halve the duration of the following notes.       *)
(*      3               divide the duration of the following notes       *)
(*                       by 3.  The *, /, and 3 options take effect for  *)
(*                       all following notes, until the duration is      *)
(*                       modified again by one of these options or by    *)
(*                       an explicit call to SetNoteDuration.            *)
(*      u               go up one octave.                                *)
(*      d               go down one octave.  In many cases the u and d   *)
(*                       options are not needed because PlayMusic        *)
(*                       chooses the appropriate octave based on the     *)
(*                       assumption that successive notes will be        *)
(*                       close to each other; but u and d can override   *)
(*                       this assumption.                                *)
(*                                                                       *)
(*************************************************************************)

END Music.
```

Music

```
DEFINITION MODULE Music;

        (*********************************************************)
        (*                                                       *)
        (*              Module to play music.                    *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        28 May 1989                      *)
        (*  Status:                                              *)
        (*      Initial version working well.  Still testing.    *)
        (*                                                       *)
        (*********************************************************)

PROCEDURE SetNoteDuration (D: CARDINAL);

    (* Sets the duration of each of the following notes, until further  *)
    (* notice, to D milliseconds.  The precision of this setting is     *)
    (* limited by the clock interrupt frequency used in module Timer;   *)
    (* the resolution can be as poor as 1/9 second.  The duration can   *)
    (* subsequently be modified by the * and / options (see below), or  *)
    (* by another call to SetNoteDuration.                              *)

PROCEDURE PlayMusic (notes: ARRAY OF CHAR);

    (* Plays the tune specified in array "notes".  The playing is done  *)
    (* asynchronously; that is, this procedure returns before the music *)
    (* is over.  However, a return from this procedure does imply that  *)
    (* array "notes" can be re-used or destroyed; the notes might not   *)
    (* yet have been played, but the data necessary to play them have   *)
    (* been processed and the necessary information stored.             *)

PROCEDURE WaitForMusicFinished;

    (* Blocks the calling task until there is no more music playing.    *)
    (* This is a guard against things like premature task termination.  *)
```

```
PROCEDURE MapToVirtualScreen (w: Window;  screen: VirtualScreen);

    (* Before calling this procedure, both w and screen must have been  *)
    (* created.  This procedure ensures that window w is visible on     *)
    (* the screen only when the given virtual screen page is active.    *)
    (* The association lasts until the window is closed or the virtual  *)
    (* screen is removed.                                               *)

PROCEDURE RemoveVirtualScreen (VAR (*INOUT*) screen: VirtualScreen);

    (* Destroys all associations between the given virtual screen and   *)
    (* its windows (but does not close the windows), and permanently    *)
    (* removes this screen from the collection of virtual screens.      *)

PROCEDURE RemoveScreenGroup (VAR (*INOUT*) group: ScreenGroup);

    (* As above, but removes an entire group.   *)

END MultiScreen.
```

MultiScreen

```
DEFINITION MODULE MultiScreen;

        (*********************************************************)
        (*                                                       *)
        (*      This module allows the creation of multiple      *)
        (*      virtual screens - complete screens, not just     *)
        (*      windows - with two keyboard "hot keys" used      *)
        (*              to navigate around the screens.          *)
        (*                                                       *)
        (*      We support a two-level hierarchy: a virtual      *)
        (*      screen is a collection of windows, and each      *)
        (*      virtual screen is a member of a group.  (In      *)
        (*      addition, there is a pseudo-group made up of     *)
        (*      all windows which are not associated with        *)
        (*      any virtual screen, to handle the "normal        *)
        (*      output" which bypasses this module.)  One        *)
        (*      hot key cycles through the groups, and the       *)
        (*      second cycles through the virtual screens of     *)
        (*      the currently active group.                      *)
        (*                                                       *)
        (*      In the present version, Ctrl/P is the hot        *)
        (*      key used to select another group, and F6         *)
        (*      cycles within a group.                           *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        12 March 1993                    *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM Windows IMPORT
    (* type *)  Window, DisplayPage;

TYPE ScreenGroup;       (* is private *)
     VirtualScreen;     (* is private *)

PROCEDURE CreateScreenGroup (hardwarepage: DisplayPage): ScreenGroup;

    (* Creates a new screen group, and maps it to the specified display *)
    (* page in the screen hardware.  It is permissible to map more than *)
    (* one group to the same hardware page.  Note that any group on     *)
    (* hardware page 0 shares the screen with "normal output" which     *)
    (* does not belong to any virtual page.  This is permitted, but     *)
    (* on aesthetic grounds is usually not a good idea.                 *)

PROCEDURE CreateVirtualScreen (group: ScreenGroup): VirtualScreen;

    (* Adds a new virtual screen to the specified group.*)
```

```
PROCEDURE LightPenOff;
    (* Sets light pen off. *)
PROCEDURE SetMickeysPerPixel (HorMPP, VertMPP : CARDINAL);
    (* Sets the Mickey / Pixel ratio. *)
PROCEDURE ConditionalOff (Left, Top, Right, Bottom: CARDINAL);
    (* Sets limits within which cursor is on. *)
PROCEDURE SetSpeedThreshold (Threshold : CARDINAL);
    (* Sets mimimum speed Threshold. *)
PROCEDURE SetPage (page: CARDINAL);
    (* Sets the hardware screen page where the mouse is visible. *)
END Mouse33.
```

```
PROCEDURE GetPosBut (VAR (*OUT*) Buttons: ButtonSet;
                                VAR (*OUT*) X, Y : CARDINAL);

    (* Gets position and Button state. *)

PROCEDURE SetCursorPos (X, Y : CARDINAL);

    (* Sets the mouse cursor position. *)

PROCEDURE GetButPress (Button: Buttons;  VAR (*OUT*) Status: ButtonSet;
               VAR (*OUT*) Count: CARDINAL;  VAR (*OUT*) X, Y: CARDINAL);

    (* Gets Button press information. *)

PROCEDURE GetButRelease (Button: Buttons;  VAR (*OUT*) Status : ButtonSet;
                                VAR (*OUT*) Count: CARDINAL;
                                VAR (*OUT*) X, Y: CARDINAL);

    (* Gets Button release information. *)

PROCEDURE SetHorizontalLimits (MinX, MaxX : CARDINAL);
PROCEDURE SetVerticalLimits (MinY, MaxY : CARDINAL);

    (* Sets the cursor limits. *)

TYPE
    GraphicCursor = RECORD
                        ScreenMask,
                        CursorMask : ARRAY [0 .. 15] OF BITSET;
                        HotX, HotY : INTEGER [-16 .. 16];
                    END (*RECORD*);

PROCEDURE SetGraphicsCursor (Cursor : GraphicCursor);

    (* Sets the graphics cursor shape.  The ScreenMask is first ANDed   *)
    (* into the display, then the CursorMask is XORed into the display. *)
    (* The hot spot coordinates are relative to the upper-left corner   *)
    (* of the cursor image, and define where the cursor actually        *)
    (* 'points to'.                                                     *)

PROCEDURE SetTextCursor (HardWare: BOOLEAN;  Start, Stop: CARDINAL);

    (* Sets the text cursor.  For the software text cursor, the second  *)
    (* two parameters specify the screen and cursor masks.  The screen  *)
    (* mask is first ANDed into the display, then the cursor mask is    *)
    (* XORed into the display.  For the hardware text cursor, the       *)
    (* second two parameters contain the line numbers of the first and  *)
    (* last scan line in the cursor to be shown on the screen.          *)

PROCEDURE ReadMotionCounters (VAR (*OUT*) X, Y: CARDINAL);

    (* Read motion counters. *)

PROCEDURE SetEventHandler (Mask: EventSet;  Handler: EventHandler);

    (* Establish conditions and Handler for mouse events.  After this,  *)
    (* when an event occurs that is in the Mask, the Handler is called  *)
    (* with the event set that actually happened, the current Button    *)
    (* status, and the cursor X and Y.                                  *)

PROCEDURE LightPenOn;

    (* Sets light pen on. *)
```

Mouse33

```
DEFINITION MODULE Mouse33;

        (*************************************************************)
        (*                                                           *)
        (*                 Mouse driver using INT 33H                *)
        (*             to call a resident mouse driver               *)
        (*                                                           *)
        (*      Originally developed by Roger Carvalho and Pat Terry.*)
        (*      The present version is maintained by Peter Moylan.   *)
        (*                                                           *)
        (*      Last edited:    7 March 1994                         *)
        (*                                                           *)
        (*************************************************************)

TYPE
    Buttons = (LeftButton, RightButton, MiddleButton);

    ButtonSet = SET OF Buttons;

    Events =  (Motion, LeftDown, LeftUp, RightDown, RightUp, MiddleDown,
              MiddleUp);

    EventSet = SET OF Events;

    (* Note: the present version of this module relies explicitly on    *)
    (* the fact that a procedure of type EventHandler receives its      *)
    (* parameters in registers AX, BX, CX, DX.  Do not use compiler     *)
    (* pragmas to alter this arrangement.                               *)

    EventHandler = PROCEDURE (EventSet,         (* condition mask *)
                              ButtonSet,        (* Button state *)
                              CARDINAL,         (* horizontal cursor position
*)
                              CARDINAL);        (* vertical cursor position *)

VAR DriverInstalled : BOOLEAN;

    (* Flag that indicates whether a mouse driver is loaded or not.  If *)
    (* its value is FALSE, none of the following functions will work.   *)

PROCEDURE ResetMouse (VAR (*OUT*) MousePresent: BOOLEAN;
                      VAR (*OUT*) NumberOfButtons: CARDINAL);

    (* Initializes mouse, returning MousePresent as FALSE if no mouse   *)
    (* available and as TRUE if it is, and NumberOfButtons as the       *)
    (* number of buttons for the mouse if installed.                    *)

PROCEDURE ShowCursor;

    (* Turns mouse cursor on. *)

PROCEDURE HideCursor;

    (* Turns mouse cursor off. *)
```

```
PROCEDURE SetTextMousePage (page: CARDINAL);

    (* Sets the hardware screen page where the mouse is visible. *)

PROCEDURE SetMouseCursorLimits (top, bottom: RowRange;
                                left, right: ColumnRange);

    (* Specifies a rectangular region outside which the mouse cursor    *)
    (* may not go.                                                      *)

PROCEDURE ShowMouseCursor;

    (* Makes the mouse cursor visible on the screen. *)

PROCEDURE HideMouseCursor;

    (* Makes the mouse cursor invisible. *)

PROCEDURE InstallEventHandler (DetectedEvents: EventSet;
                               Handler: EventHandler);

    (* Nominates the procedure to be called whenever an event in the    *)
    (* set DetectedEvents occurs.  Note: the Handler is like an         *)
    (* interrupt procedure, in that it is executing in the context of   *)
    (* an unknown task; typically it should probably restrict its       *)
    (* actions to fairly elementary things, like a Signal to wake up    *)
    (* the task that really wants to know about the event.              *)

END Mouse.
```

Mouse

```
DEFINITION MODULE Mouse;

        (*********************************************************)
        (*                                                       *)
        (*                  Mouse driver                         *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        21 February 1994                 *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*      This module is actually an intermediary          *)
        (*      between the user and the low-level mouse*)
        (*      driver; this is to make it easy to change        *)
        (*      mouse drivers.                                   *)
        (*                                                       *)
        (*********************************************************)

FROM Windows IMPORT
    (* type *)  RowRange, ColumnRange;

FROM ConfigurationOptions IMPORT
    (* const*)  UseSerialDriver;

(*%T UseSerialDriver *)
FROM SerialMouse IMPORT
(*%E *)
(*%F UseSerialDriver *)
FROM Mouse33 IMPORT
(*%E *)
    (* type *)  Buttons, ButtonSet, Events, EventSet, EventHandler,
    (* proc *)  ResetMouse;

(************************************************************************)

PROCEDURE MouseAvailable (): BOOLEAN;

    (* Returns TRUE iff a mouse driver is loaded, a mouse exists, and   *)
    (* mouse operation is permitted in module ConfigurationOptions.     *)

PROCEDURE GetTextMousePosition (VAR (*OUT*) Xposition: ColumnRange;
                                VAR (*OUT*) Yposition: RowRange);

    (* Returns the current position of the mouse cursor. *)

PROCEDURE GetTextMouseStatus (VAR (*OUT*) buttons: ButtonSet;
                              VAR (*OUT*) Xposition: ColumnRange;
                              VAR (*OUT*) Yposition: RowRange);

    (* Returns the current mouse position and state of the buttons.     *)

PROCEDURE SetTextMousePosition (Xposition: ColumnRange; Yposition: RowRange);

    (* Initialises the mouse position. *)
```

```
PROCEDURE ReadCMOS (location: CMOSaddress): BYTE;

    (* Returns the value at the given CMOS location.    *)

PROCEDURE WriteCMOS (location: CMOSaddress;  value: BYTE);

    (* Stores a value at the given CMOS location.       *)

(************************************************************************)
(*                          BIOS/MS-DOS CALLS                           *)
(************************************************************************)

PROCEDURE BIOS (InterruptNumber: BYTE;
                        VAR (*INOUT*) Registers: RegisterPacket);

    (* Performs a software interrupt, with the given interrupt number,  *)
    (* after loading the components of variable "Registers" into the    *)
    (* machine registers.  After the handler returns, the updated       *)
    (* register values are put back into variable "Registers".          *)

(************************************************************************)
(*                   MISCELLANEOUS LOW-LEVEL OPERATIONS                 *)
(************************************************************************)

PROCEDURE EnterCriticalSection(): CARDINAL;

    (* Saves the processor flags word, including the current "interrupt *)
    (* enable" status, on the caller's stack, and returns with          *)
    (* interrupts disabled.   NOTE: this procedure and the following    *)
    (* one should be used as a matched pair.                            *)

PROCEDURE LeaveCriticalSection (savedPSW: CARDINAL);

    (* Restores the processor flags word, including the "interrupt      *)
    (* enable" status, from the stack.  NOTE: this procedure and the    *)
    (* one above should be used as a matched pair.                      *)

PROCEDURE ProcessorStatus(): CARDINAL;

    (* Returns the current value of the processor flags word.   *)

PROCEDURE ShortDelay (amount: CARDINAL);

    (* Provides a time delay for those cases where the required delay   *)
    (* is not long enough to justify a Sleep() operation.               *)
    (* The present version is not entirely satisfactory - needs to be   *)
    (* re-tuned for different compiler options, different processor     *)
    (* models, etc.  This should be seen as an interim solution only.   *)

END MiscPMOS.
```

MiscPMOS

```
DEFINITION MODULE MiscPMOS;

        (*********************************************)
        (*                                           *)
        (*      Miscellaneous PMOS procedures        *)
        (*                                           *)
        (*  Programmer:         P. Moylan            *)
        (*  Last edited:        22 January 1994      *)
        (*  Status:             OK                   *)
        (*                                           *)
        (*********************************************)

IMPORT SYSTEM;

TYPE RegisterPacket = SYSTEM.Registers;
     CMOSaddress = SHORTCARD [0..63];

(*************************************************************************)
(*                              STRING COPY                              *)
(*************************************************************************)

PROCEDURE CopyString (source: ARRAY OF CHAR;  VAR (*OUT*) dest: ARRAY OF
CHAR);

    (* Copies a string, with truncation or null termination as needed.  *)
    (* This function is provided in order to help software portability, *)
    (* i.e. to avoid having to rewrite code for no reason other than    *)
    (* a change of compilers.                                           *)

(*************************************************************************)
(*                      PROCEDURES TO ACCESS CMOS                        *)
(*************************************************************************)
(*                                                                       *)
(*  The first 14 bytes of CMOS are used for the real-time clock - see    *)
(*  module TimeOfDay for details.                                        *)
(*                                                                       *)
(*  The contents of the remaining bytes would take too long to describe *)
(*  here, so only the ones of current interest are mentioned:            *)
(*                                                                       *)
(*      10H     Diskette drive type.  The high order 4 bits describe     *)
(*              drive A, the lower order 4 bits describe drive B.        *)
(*              The encoding is:                                         *)
(*                      0       no drive present                         *)
(*                      1       double sided drive, 48 TPI               *)
(*                      2       high capacity drive, 96 TPI              *)
(*                      4       3.5" 1.44MB drive                        *)
(*                  (other values are reserved).                         *)
(*                                                                       *)
(*************************************************************************)
```

```
(*************************************************************************)
(*                        NORMAL MODE PROCEDURE                          *)
(*************************************************************************)

PROCEDURE PositionMenu (M: Menu; ForegroundColour, BackgroundColour: Colour;
                        startline, endline: RowRange;
                        leftcol, rightcol: ColumnRange);

    (* Sets the screen position, size, and colour to be used when this  *)
    (* menu is displayed by a call to SelectFromMenu.  This procedure   *)
    (* sets the menu to Normal Mode; therefore, the menu will not be    *)
    (* displayed until SelectFromMenu is called.                        *)

(*************************************************************************)
(*                        SPECIAL MODE PROCEDURE                         *)
(*************************************************************************)

PROCEDURE DisplayMenu (w: Window;  M: Menu;
                               rows, columns, initialvalue: CARDINAL);

    (* Displays menu M at the current cursor position in window w,      *)
    (* with initialvalue specifying a field to highlight.  The space    *)
    (* reserved on the screen is "rows" screen rows in height and       *)
    (* "columns" character positions wide.  (The remainder of window w  *)
    (* may of course be used for other purposes, including other        *)
    (* menus.)  This call selects Special Mode.  When SelectFromMenu is *)
    (* called, it will use window w.                                    *)

END Menus.
```

```
(*                                                                        *)
(**************************************************************************)

FROM Windows IMPORT
    (* type *)  Window, Colour, RowRange, ColumnRange;

TYPE
    Menu;               (* is private *)
    ItemText = ARRAY ColumnRange OF CHAR;
    MenuColumn = CARDINAL;

    (* Note the dual use of the word "column".  A variable of type      *)
    (* ColumnRange refers to a horizontal screen position.  But in      *)
    (* menus we use "column" to a column of character strings, which    *)
    (* is why we have the separate type MenuColumn.  The type           *)
    (* MenuColumn is defined primarily to make it clear which type of   *)
    (* column is being referred to in every case.                       *)

(**************************************************************************)
(*                           COMMON PROCEDURES                            *)
(**************************************************************************)

PROCEDURE CreateMenu (VAR (*OUT*) M: Menu;  columns: MenuColumn;
                      VAR (*IN*) Messages: ARRAY OF ItemText;
                      NumberOfItems: CARDINAL);

    (* Introduces a menu into the system, but does not display it yet.  *)
    (* For a simple vertical menu, columns = 1.  Use columns > 1 for    *)
    (* shorter and wider menus.  Messages[0] is the label to put into   *)
    (* the menu header (ignored in Special Mode).  The entries in       *)
    (* Messages[1..NumberOfItems] are the items displayed when the menu *)
    (* is put on the screen.  Special case: if you specify              *)
    (* NumberOfItems = 0 then the whole of array Messages is used.      *)

PROCEDURE SelectFromMenu (M: Menu): CARDINAL;

    (* Displays menu M on the screen, allows terminal user to use       *)
    (* cursor keys to move about the menu and the ENTER key to select   *)
    (* an item.  (The space bar is also accepted, as an alternative to  *)
    (* the ENTER key, to select an item.)  An item may also be selected *)
    (* by typing its initial letter, followed by space or ENTER.        *)
    (* Returns the number of the item which was selected.               *)
    (* (Item numbers start from 1).  An answer of 0 indicates that the  *)
    (* user typed the ESC key to return without selecting anything.     *)

    (* Note: in Special Mode, another way to return from this procedure *)
    (* is for the keyboard user to move the cursor off the edge of the  *)
    (* menu.  This is done to support the case where there are several  *)
    (* menus in the same window.  Note also that in Special Mode the    *)
    (* cursor key which causes the return (Enter, space, Esc, or cursor *)
    (* movement key) remains available to the caller via InKey(), but   *)
    (* in Normal Mode this key is consumed by the procedure.            *)

PROCEDURE DestroyMenu (M: Menu);

    (* Removes a menu from the system, freeing up the space it used.    *)
```

Menus

```
DEFINITION MODULE Menus;

        (****************************************************************)
        (*                                                              *)
        (*      Displays menus on screen, allows terminal user to       *)
        (*                       select from them.                      *)
        (*                                                              *)
        (*  Programmer:         P. Moylan                               *)
        (*  Last edited:        8 May 1993                              *)
        (*  Status:             OK                                      *)
        (*                                                              *)
        (****************************************************************)

(************************************************************************)
(*                                                                      *)
(*  NOTE: This module provides two ways to work with menus; for the     *)
(*  sake of giving them a name, let us call them "Normal Mode" and      *)
(*  "Special Mode".  In either mode, you use CreateMenu to define a     *)
(*  menu initially, SelectFromMenu to prompt the user to choose from    *)
(*  the menu, and DestroyMenu when you have finished using the menu.    *)
(*                                                                      *)
(*  In Normal Mode - the mode most users will want to use - you use     *)
(*  PositionMenu to define where on the screen the menu will be         *)
(*  displayed.  The menu does not appear on the screen until            *)
(*  SelectFromMenu is called, and it disappears from the screen as      *)
(*  soon as the keyboard user has made a selection.  Note that in       *)
(*  Normal Mode you never have to open a screen window to hold the      *)
(*  menu, because SelectFromMenu opens and closes the window for you.   *)
(*  You may, if you wish, use a new call to PositionMenu before each    *)
(*  call to SelectFromMenu, although the more common case would be to   *)
(*  call PositionMenu just once, after the call to CreateMenu.          *)
(*                                                                      *)
(*  In Special Mode, you have to provide your own screen window, and    *)
(*  you have to call DisplayMenu to show the menu on the screen; this   *)
(*  also means that DisplayMenu defines the location of the menu on     *)
(*  the screen.  After that, you call SelectFromMenu to prompt the user *)
(*  to choose an item.  Special Mode is for use when you want the menu  *)
(*  to remain displayed even after the selection has been made, and for *)
(*  things like multiple menus within the same screen window.  It is    *)
(*  suggested that Special Mode should only be used in cases where it   *)
(*  is found that Normal Mode is not flexible enough to do the job.     *)
(*                                                                      *)
(*  Note that the criterion for deciding which mode the menu is in      *)
(*  depends entirely on whether PositionMenu or DisplayMenu is called.  *)
(*  Since you may call these procedures more than once, there is        *)
(*  nothing to stop you from switching from Special Mode to Normal Mode *)
(*  or vice versa.  (I can't think of an application where you would    *)
(*  want to do this, but on the other hand I couldn't think of any      *)
(*  good reason to make it illegal.)                                    *)
```

MaintenancePages

```
DEFINITION MODULE MaintenancePages;

        (*******************************************************)
        (*                                                     *)
        (*      Support for "maintenance page" screen output   *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        12 March 1993                  *)
        (*  Status:             OK                             *)
        (*                                                     *)
        (*******************************************************)

FROM Windows IMPORT
    (* type *)  Window;

TYPE MaintenancePage;   (* is private *)

PROCEDURE CreateMaintenancePage (VAR (*OUT*) page: MaintenancePage);

    (* Creates a new maintenance page.  *)

PROCEDURE Associate (w: Window;  page: MaintenancePage);

    (* Before calling this procedure, both w and page must have been    *)
    (* created.  This procedure ensures that window w is visible on     *)
    (* the screen only when the given maintenance page is active.       *)
    (* The association lasts until the window is closed or the page     *)
    (* is removed.                                                      *)

PROCEDURE RemoveMaintenancePage (VAR (*INOUT*) page: MaintenancePage);

    (* Destroys all associations between the given page and its screen  *)
    (* windows (but does not close the windows), and permanently        *)
    (* removes this page from the collection of maintenance pages.      *)

END MaintenancePages.
```

Mailboxes

```
DEFINITION MODULE Mailboxes;

        (*********************************************************)
        (*                                                       *)
        (*      Mailboxes for intertask communication            *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        8 June 1993                      *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*********************************************************)

TYPE Mailbox;           (* is private *)

PROCEDURE CreateMailbox (LengthLimit: CARDINAL): Mailbox;

    (* Creates a new mailbox.  LengthLimit is the maximum number of     *)
    (* characters in a single message.  (A limit is needed so that a    *)
    (* task reading the mailbox knows how much space to allocate.)      *)

PROCEDURE SendMessage (MB: Mailbox;  messageptr: ADDRESS;
                                     length: CARDINAL): BOOLEAN;

    (* Copies a string, specified by its address and length, into the   *)
    (* specified mailbox.  Returns TRUE if successful, and FALSE if the *)
    (* message is too long or the mailbox does not exist.               *)

PROCEDURE ReceiveMessage (MB: Mailbox;  VAR (*OUT*) message: ARRAY OF CHAR;
                                     TimeLimit: CARDINAL): CARDINAL;

    (* Returns the next message (after waiting if necessary) from       *)
    (* mailbox MB.  TimeLimit is a timeout value in milliseconds.       *)
    (* (Specify TimeLimit=0 for infinite patience.)  The function       *)
    (* return value is the message length; this is zero if no message   *)
    (* was obtained, either because of a faulty mailbox or because of   *)
    (* timeout.  Note: it is also possible to have a genuine message of *)
    (* zero length.                                                     *)

END Mailboxes.
```

```
INLINE
PROCEDURE BlockFillWord (destination: ADDRESS;  wordcount: CARDINAL;
                                                        value: WORD)

    (* Fills the destination array with the given value.*)

     = Code3 (0F3H, 0ABH, Ret);         (* rep stosw *)

    (*# restore *)

(************************************************************************)
(*                           INPUT AND OUTPUT                           *)
(************************************************************************)

    (*# save, call(inline => on, reg_param => (dx, ax)) *)

INLINE
PROCEDURE OutByte (port: CARDINAL; value: BYTE)

    (* Puts the value out to an output port.    *)

      = Code2 (0EEH, Ret);

(************************************************************************)

INLINE
PROCEDURE InByte (port: CARDINAL): BYTE

    (* Reads a byte from an input port. *)

     = Code2 (0ECH, Ret);

    (*# restore *)

(************************************************************************)

    (*# call(inline => off) *)
    (*# save, call(reg_param => (dx, di, es, cx)) *)

PROCEDURE InStringWord (port: CARDINAL;  BufferAddress: ADDRESS;
                                                count: CARDINAL);

    (* Reads count words from an input port.    *)

    (*# restore *)

(************************************************************************)

    (*# save, call(reg_param => (dx, si, ds, cx)) *)

PROCEDURE OutStringWord (port: CARDINAL;  BufferAddress: ADDRESS;
                                                count: CARDINAL);

    (* Writes count words to an output port.    *)

    (*# restore *)

(************************************************************************)

END LowLevel.
```

```
INLINE
PROCEDURE Mul (A, B: CARDINAL): LONGCARD

    (* Same as A*B, except for the type of the result.  We provide this *)
    (* as a general-purpose function since this combination of operands *)
    (* is often precisely what is wanted.                               *)

     = Code3 (0F7H, 0E3H, Ret); (* mul bx *)

    (*# restore *)

(************************************************************************)

    (*# save, call(inline => on, reg_param => (ax, dx, bx)) *)

INLINE
PROCEDURE Div (A: LONGCARD;  B: CARDINAL): CARDINAL

    (* Same as A DIV B, except for the type of A.  We provide this as   *)
    (* a general-purpose function since this combination of operands    *)
    (* is often precisely what is wanted.                               *)

     = Code3 (0F7H, 0F3H, Ret); (* div bx *)

    (*# restore *)

(************************************************************************)
(*                            BLOCK MOVES                               *)
(************************************************************************)

    (*# save, call(inline => on, reg_param => (si, ds, di, es, cx)) *)

INLINE
PROCEDURE Copy (source, destination: ADDRESS;  bytecount: CARDINAL)

    (* Copies an array of bytes from the source address to the          *)
    (* destination address.  In the case where the two arrays overlap,  *)
    (* the destination address should be lower in physical memory than  *)
    (* the source address.                                              *)

     = Code3 (0F3H, 0A4H, Ret);         (* rep movsb *)

(************************************************************************)

    (*# call(inline => off) *)

PROCEDURE CopyUp (source, destination: ADDRESS;  bytecount: CARDINAL);

    (* A variant of Copy which does the move backwards, in order        *)
    (* to handle the case where the destination address is inside the   *)
    (* source array.  In this special case Copy cannot be used,         *)
    (* because it would overwrite data it was about to copy.            *)

    (*# restore *)

(************************************************************************)

    (*# save, call(inline => on, reg_param => (di, es, cx, ax)) *)

INLINE
PROCEDURE BlockFill (destination: ADDRESS;  bytecount: CARDINAL;  value: BYTE)

    (* Fills the destination array with the given value.*)

     = Code3 (0F3H, 0AAH, Ret);         (* rep stosb *)
```

```
INLINE
PROCEDURE LowWord (w: LONGWORD): WORD

    (* Returns the low-order word of its argument.      *)

     = Code1 (Ret);

    (*# restore *)

(*************************************************************************)

    (*# save, call(inline => on, reg_param => (bx, ax)) *)

INLINE
PROCEDURE HighWord (w: LONGWORD): CARDINAL

    (* Returns the high-order word of its argument.     *)

     = Code1 (Ret);

    (*# restore *)

(*************************************************************************)

    (*# save, call(inline => on, reg_param => (dx, ax)) *)

INLINE
PROCEDURE MakeLongword (high, low: WORD): LONGCARD

    (* Combines two words into a longword.  The first argument becomes  *)
    (* the most significant word of the result.                         *)

     = Code1 (Ret);     (* mov al, bl *)

    (*# restore *)

(*************************************************************************)
(*                      MISCELLANEOUS ARITHMETIC                         *)
(*************************************************************************)

    (*# save, call(inline => on, reg_param => (ds, si, ax)) *)

INLINE
PROCEDURE INCV (VAR (*INOUT*) dest: CARDINAL;  src: CARDINAL): BOOLEAN

    (* Computes dest := dest + src, and returns TRUE iff the addition   *)
    (* produced a carry.                                                *)

     = Code5 (01H, 04H, 1BH, 0C0H, Ret);    (* add [si], ax; sbb ax, ax *)

INLINE
PROCEDURE DECV (VAR (*INOUT*) dest: CARDINAL;  src: CARDINAL): BOOLEAN

    (* Computes dest := dest - src, and returns TRUE iff the            *)
    (* subtraction produced a borrow.                                   *)

     = Code5 (29H, 04H, 1BH, 0C0H, Ret);    (* sub [si], ax; sbb ax, ax *)

    (*# restore *)

(*************************************************************************)

    (*# save, call(inline => on, reg_param => (ax, bx)) *)
```

```
PROCEDURE Physical (A: ADDRESS): LONGCARD;

    (* Converts a virtual address to a physical address.  Use with care!*)

    (*# restore *)

(*************************************************************************)

    (*# save, call(inline => on, reg_param => (ax, dx, bx)) *)

INLINE
PROCEDURE AddOffset (A: ADDRESS;  increment: CARDINAL): ADDRESS

    (* Returns a pointer to the memory location whose physical address  *)
    (* is Physical(A)+increment.  In the present version, it is assumed *)
    (* that the caller will never try to run off the end of a segment.  *)

     = Code3 (01H, 0D8H, Ret);  (* add ax, bx *)

(*************************************************************************)

INLINE
PROCEDURE SubtractOffset (A: ADDRESS;  decrement: CARDINAL): ADDRESS

    (* Like AddOffset, except that we go backwards in memory.  Running  *)
    (* off the beginning of the segment is an undetected error.         *)

     = Code3 (29H, 0D8H, Ret);  (* sub ax, bx *)

    (*# restore *)

(*************************************************************************)
(*                       BYTE/WORD/LONGWORD CONVERSIONS                  *)
(*************************************************************************)

    (*# save, call(inline => on) *)

INLINE
PROCEDURE LowByte (w: WORD): BYTE

    (* Returns the low-order byte of its argument.      *)

     = Code1 (Ret);

(*************************************************************************)

INLINE
PROCEDURE HighByte (w: WORD): BYTE

    (* Returns the high-order byte of its argument.     *)

     = Code3 (88H, 0E0H, Ret);  (* mov al, ah *)

(*************************************************************************)

INLINE
PROCEDURE MakeWord (high, low: BYTE): WORD

    (* Combines two bytes into a word.  The first argument becomes the  *)
    (* most significant byte of the result.                             *)

     = Code3 (8AH, 0C3H, Ret);  (* mov al, bl *)

(*************************************************************************)
```

```
INLINE
PROCEDURE RSB (value: BYTE;  count: CARDINAL): SHORTCARD

    (* Right shift of "value" by "count" bit positions, with zero fill. *)

     = Code3 (0D2H, 0E8H, Ret);         (* shr al, cl *)

    (*# restore *)

(*************************************************************************)
(*                          POINTER OPERATIONS                           *)
(*************************************************************************)

    (*# save, call(inline => on, reg_param => (dx, ax)) *)

INLINE
PROCEDURE MakePointer (segment, offset: CARDINAL): ADDRESS

    (* Creates a pointer, given the segment and offset within segment.  *)

     = Code1 (Ret);

    (*# restore *)

(*************************************************************************)

    (*# save, call(inline => on, reg_return => (bx)) *)

INLINE
PROCEDURE SEGMENT (A: ADDRESS): CARDINAL

    (* Returns the segment part of an address.  *)

     = Code1 (Ret);

    (*# restore *)

(*************************************************************************)

    (*# save, call(inline => on) *)

INLINE
PROCEDURE OFFSET (A: ADDRESS): CARDINAL

    (* Returns the offset part of an address.   *)

     = Code1 (Ret);

    (*# restore *)

(*************************************************************************)

    (*# call(inline => off) *)
    (*# save, call(reg_param => (ax, dx), reg_return => (dx, ax)) *)

PROCEDURE Virtual (PA: LONGCARD): ADDRESS;

    (* Converts a physical address to a virtual address, if possible.   *)
    (* There are no guarantees in the case where there is no such       *)
    (* virtual address.                                                 *)

    (*# restore *)

(*************************************************************************)

    (*# save, call(reg_return => (ax, bx)) *)
```

```
    (*# restore *)
    (*# save, call(inline => on, reg_param => (ax, cx)) *)

INLINE
PROCEDURE ROL (value: WORD;  count: CARDINAL): CARDINAL

    (* Left rotation of "value" by "count" bit positions.       *)

     = Code3 (0D3H, 0C0H, Ret);         (* rol ax, cl *)

(*************************************************************************)

INLINE
PROCEDURE ROLB (value: BYTE;  count: CARDINAL): SHORTCARD

    (* Left rotation of "value" by "count" bit positions.       *)

     = Code3 (0D2H, 0C0H, Ret);         (* rol al, cl *)

(*************************************************************************)

INLINE
PROCEDURE LS (value: WORD;  count: CARDINAL): CARDINAL

    (* Left shift of "value" by "count" bit positions, with zero fill.  *)

     = Code3 (0D3H, 0E0H, Ret);         (* shl ax, cl *)

(*************************************************************************)

INLINE
PROCEDURE LSB (value: BYTE;  count: CARDINAL): SHORTCARD

    (* Left shift of "value" by "count" bit positions, with zero fill.  *)

     = Code3 (0D2H, 0E0H, Ret);         (* shl al, cl *)

(*************************************************************************)

INLINE
PROCEDURE ROR (value: WORD;  count: CARDINAL): CARDINAL

    (* Right rotation of "value" by "count" bit positions.      *)

     = Code3 (0D3H, 0C8H, Ret);         (* ror ax, cl *)

(*************************************************************************)

INLINE
PROCEDURE RORB (value: BYTE;  count: CARDINAL): SHORTCARD

    (* Right rotation of "value" by "count" bit positions.      *)

     = Code3 (0D2H, 0C8H, Ret);         (* ror al, cl *)

(*************************************************************************)

INLINE
PROCEDURE RS (value: WORD;  count: CARDINAL): CARDINAL

    (* Right shift of "value" by "count" bit positions, with zero fill. *)

     = Code3 (0D3H, 0E8H, Ret);         (* shr ax, cl *)

(*************************************************************************)
```

```
INLINE
PROCEDURE IANDB (first, second: BYTE): SHORTCARD

    (* Bit-by-bit logical AND for bytes.  With most compilers IAND      *)
    (* would be sufficient, I think, so this procedure might be         *)
    (* withdrawn in future versions.  It's present only because of the  *)
    (* extremely frustrating assignment compatibility rules of the      *)
    (* TopSpeed compiler.                                               *)

     = Code3 (21H, 0D8H, Ret);          (* and ax, bx *)

(************************************************************************)

INLINE
PROCEDURE IOR (first, second: WORD): CARDINAL

    (* Bit-by-bit inclusive OR. *)

     = Code3 (09H, 0D8H, Ret);          (* or ax, bx *)

(************************************************************************)

INLINE
PROCEDURE IORB (first, second: BYTE): SHORTCARD

    (* Bit-by-bit inclusive OR. *)

     = Code3 (09H, 0D8H, Ret);          (* or ax, bx *)

(************************************************************************)

INLINE
PROCEDURE IXOR (first, second: WORD): CARDINAL

    (* Bit-by-bit exclusive OR. *)

     = Code3 (31H, 0D8H, Ret);          (* xor ax, bx *)

(************************************************************************)

INLINE
PROCEDURE IXORB (first, second: BYTE): SHORTCARD

    (* Bit-by-bit exclusive OR. *)

     = Code3 (31H, 0D8H, Ret);          (* xor ax, bx *)

(************************************************************************)

INLINE
PROCEDURE INOT (value: WORD): CARDINAL

    (* Bit-by-bit Boolean complement.   *)

     = Code3 (0F7H, 0D0H, Ret);         (* not ax *)

(************************************************************************)

INLINE
PROCEDURE INOTB (value: BYTE): SHORTCARD

    (* Bit-by-bit Boolean complement.   *)

     = Code3 (0F7H, 0D0H, Ret);         (* not ax *)

(************************************************************************)
```

LowLevel

```
DEFINITION MODULE LowLevel;

        (*********************************************************)
        (*                                                       *)
        (*         Miscellaneous low-level procedures            *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        17 January 1994                  *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*      Note that the implementation of this module      *)
        (*      is heavily compiler-dependent.  This version     *)
        (*      is for use with the TopSpeed compiler.           *)
        (*                                                       *)
        (*********************************************************)

(*# module (init_code => off) *)

CONST
    (*%T _fcall *)
        Ret = 0CBH;
    (*%E *)
    (*%F _fcall *)
        Ret = 0C3H;
    (*%E *)

TYPE
    Code1=ARRAY [0..0] OF SHORTCARD;
    Code2=ARRAY [0..1] OF SHORTCARD;
    Code3=ARRAY [0..2] OF SHORTCARD;
    Code5=ARRAY [0..4] OF SHORTCARD;

(************************************************************************)
(*                           BITWISE LOGIC                              *)
(************************************************************************)

    (*# save, call(inline => on) *)

INLINE
PROCEDURE IAND (first, second: WORD): CARDINAL

    (* Bit-by-bit logical AND.  *)

     = Code3 (21H, 0D8H, Ret);          (* and ax, bx *)

(************************************************************************)
```

LossyQueues

```
DEFINITION MODULE LossyQueues;

        (*************************************************************)
        (*                                                           *)
        (*      A lossy queue is a bounded-length queue with the     *)
        (*      property that the PutQueue operation never blocks;   *)
        (*      if space is unavailable, the data to be put are lost.*)
        (*      This is appropriate for real-time applications where *)
        (*      losing data is more acceptable than losing time.     *)
        (*      Of course, it is desirable to make the queue size big*)
        (*      enough to ensure that data loss will be rare.        *)
        (*                                                           *)
        (*      Author:         P. Moylan                            *)
        (*      Last edited:    17 August 1993                       *)
        (*                                                           *)
        (*      Status:         OK.                                  *)
        (*                                                           *)
        (*************************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE;

TYPE LossyQueue;        (* is private *)

PROCEDURE CreateQueue (VAR (*OUT*) B: LossyQueue;
                                    capacity, elementsize: CARDINAL);

    (* Allocates space for a lossy queue, and initializes it.  The      *)
    (* caller specifies how many elements (assumed to be of equal size) *)
    (* the queue will hold, and the size in bytes of each element.      *)

PROCEDURE PutQueue (B: LossyQueue;  item: ARRAY OF BYTE): BOOLEAN;

    (* If space is available, puts item at the tail of the queue and    *)
    (* returns TRUE.  Returns FALSE if the new item would not fit.      *)
    (* Note: it is assumed that SIZE(item) matches the element size     *)
    (* declared when the queue was created.                             *)

PROCEDURE GetQueue (B: LossyQueue;  VAR (*OUT*) item: ARRAY OF BYTE);

    (* Takes one item from the head of the queue, waiting if necessary. *)

END LossyQueues.
```

```
(************************************************************************)

PROCEDURE DefineListFormat (header, separator, trailer: ARRAY OF CHAR;
                            ComponentType: FieldType): ListFormat;

    (* Sets up the output format for a class of lists.  The header is   *)
    (* what is written before the first component; the separator is     *)
    (* what is written between the components; and the trailer is what  *)
    (* is written after the last component.  For an empty list, only    *)
    (* the header and trailer will be written.  ComponentType           *)
    (* implicitly specifies the procedures which will be used to write  *)
    (* and edit the components of the list.                             *)

PROCEDURE DiscardFormat (format: ListFormat);

    (* A notification from the user that this format will not be used   *)
    (* again (unless it is redefined by another call to procedure       *)
    (* DefineListFormat).  Use of this procedure is optional, but is    *)
    (* recommended for the sake of "clean" memory management.           *)

PROCEDURE WriteList (w: Window;  L: List;  format: ListFormat);

    (* Writes L on the screen, including its delimiters.  This          *)
    (* procedure is not actually used in this module, but is provided   *)
    (* as something that a client module may find useful.               *)

PROCEDURE EditList (w: Window;  VAR (*INOUT*) L: List;  format: ListFormat);

    (* Edits a list at the current cursor position in window w.  We     *)
    (* leave this procedure on seeing a keyboard character which does   *)
    (* not belong to us.  The cursor is left just beyond the "trailer"  *)
    (* string which terminates the displayed form of the list.  The     *)
    (* terminating keystroke is returned to the keyboard driver so that *)
    (* it can still be read by the caller.                              *)

END ListEditor.
```

ListEditor

```
DEFINITION MODULE ListEditor;

        (*********************************************************)
        (*                                                       *)
        (*              "Generic" list editor                    *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        13 February 1991                 *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

FROM Windows IMPORT
    (* type *)  Window;

FROM FieldEditor IMPORT
    (* type *)  FieldType, WriteProc, EditProc;

(*************************************************************************)
(*                                                                       *)
(*  The editor in this module is "generic" in a limited sense.  It       *)
(*  performs screen editing of a linear list, but obviously it has to    *)
(*  make some assumptions about the linkage method by which the list     *)
(*  is implemented.  The list must have the general form                 *)
(*                                                                       *)
(*       TYPE List = POINTER TO RECORD                                   *)
(*                                   next: List;                         *)
(*                                   component: some pointer type;       *)
(*                               END (*RECORD*)                          *)
(*                                                                       *)
(*  Notice however that this definition module actually defines lists    *)
(*  and components in terms of type ADDRESS, so that there is no         *)
(*  requirement that the caller's types and field names match the        *)
(*  above definition.                                                    *)
(*                                                                       *)
(*  The caller is required to supply, via FieldEditor.DefineFieldType,   *)
(*  procedures which write and edit list components.                     *)
(*                                                                       *)
(*  The procedures in this module obey the rules set by module           *)
(*  FieldEditor.  In the case where a list of lists is being edited,     *)
(*  the component editor can simply call EditList to do its job for it.  *)
(*                                                                       *)
(*************************************************************************)

TYPE
    ListFormat;         (* is private *)
    List = ADDRESS;
```

KTrace

```
DEFINITION MODULE KTrace;

        (****************************************************************)
        (*                                                              *)
        (*      Trace routines for Modula 2 program development.        *)
        (*                                                              *)
        (*  This is the version which does NOT use windows.  It is      *)
        (*  intended for low-level tracing of the kernel, where a       *)
        (*  window-based tracing facility would be unsuitable because   *)
        (*  of critical section problems.  However, it is quite         *)
        (*  adequate for any application where we don't care too much   *)
        (*  about a pretty screen layout.                               *)
        (*                                                              *)
        (*  Note, however, that this module is missing the "Press any   *)
        (*  key to continue" option which my other trace modules have.  *)
        (*                                                              *)
        (*  Programmer:         P. Moylan                               *)
        (*  Last edited:        21 January 1989                         *)
        (*  Status:             OK                                      *)
        (*                                                              *)
        (****************************************************************)

PROCEDURE NYI (name: ARRAY OF CHAR);

        (* Types a "not yet implemented" message.       *)

PROCEDURE InTrace (name: ARRAY OF CHAR);

        (* Types "Entering 'name'".     *)

PROCEDURE OutTrace (name: ARRAY OF CHAR);

        (* Types "Leaving 'name'".      *)

PROCEDURE TraceOn;

        (* Turns on tracing.    *)

PROCEDURE TraceOff;

        (* Turns off tracing.           *)

PROCEDURE TraceStatus(): BOOLEAN;

        (* Says whether tracing is currently on.        *)

END KTrace.
```

```
PROCEDURE LockStatus (): CARDINAL;

    (* Returns the current state of the caps lock, num lock, and scroll *)
    (* lock conditions, using the code defined in KBDRIVER.DEF.         *)

PROCEDURE HotKey (FunctionKey: BOOLEAN;  code: CHAR;  S: Semaphore);

    (* After this procedure is called, typing the key combination for   *)
    (* 'code' will cause a Signal(S).  Set FunctionKey=TRUE to trap one *)
    (* of the two-character special function keys, and FALSE otherwise. *)
    (* The character is consumed; if it should be passed on, then the   *)
    (* user's hot key handler can do a PutBack().  Note: there is no    *)
    (* provision for having multiple hot key handlers for the same key; *)
    (* any existing hot key mapping will be overridden.                 *)

END Keyboard.
```

Keyboard

```
DEFINITION MODULE Keyboard;

        (*********************************************)
        (*                                           *)
        (*      Module to deal with keyboard input.  *)
        (*                                           *)
        (*  Programmer:         P. Moylan            *)
        (*  Last edited:        8 February 1994      *)
        (*  Status:             OK                   *)
        (*                                           *)
        (*********************************************)

FROM Semaphores IMPORT
    (* type *)  Semaphore;

PROCEDURE KeyPressed(): BOOLEAN;

    (* Returns TRUE iff a character is available. *)

PROCEDURE InKey (): CHAR;

    (* Reads a single character code from the keyboard. *)

PROCEDURE PutBack (ch: CHAR);

    (* This is an "un-read" operation, i.e. the character ch will       *)
    (* re-appear on the next call to InKey.  This facility is provided  *)
    (* for the use of software which can overshoot by one character     *)
    (* when reading its input - a situation which can often occur.      *)
    (* Some versions of this module will allow several calls to PutBack *)
    (* before the next call to InKey, but no guarantee is made in that  *)
    (* case of uniform treatment from version to version.               *)

PROCEDURE StuffKeyboardBuffer (ch: CHAR);

    (* Stores ch as if it had come from the keyboard, so that a         *)
    (* subsequent InKey() will pick it up.                              *)

    (* NOTE: Procedures PutBack and StuffKeyboardBuffer do almost the   *)
    (* same thing, but not quite.  The differences are:                 *)
    (*  1. StuffKeyboardBuffer stores characters in a first-in-first-out*)
    (*     order, whereas PutBack uses last-in-first-out.               *)
    (*  2. StuffKeyboardBuffer is intended for the case of a task       *)
    (*     sending data to another task (where that other task is       *)
    (*     expecting keyboard input), and PutBack is designed for the   *)
    (*     case of a task talking to itself (i.e. the PutBack(ch) and   *)
    (*     the InKey() are in the same task).  If you get this the      *)
    (*     wrong way around then you could have timing-related          *)
    (*     problems, up to and including deadlock.                      *)

PROCEDURE SetLocks (code: CARDINAL);

    (* Set/clear the caps lock, num lock, and scroll lock conditions.   *)
    (* The code is defined in KBDRIVER.DEF.                             *)
```

KBdriver

```
DEFINITION MODULE KBdriver;

        (*********************************************************)
        (*                                                       *)
        (*      Device driver for the keyboard.                  *)
        (*                                                       *)
        (*      Author:         P. Moylan                        *)
        (*      Last edited:    19 August 1991                   *)
        (*                                                       *)
        (*      Status:         OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE;

CONST

    (* Codes to specify the keyboard indicator lights.  *)

    ScrollLockLED = 1;
    NumLockLED = 2;
    CapsLockLED = 4;

PROCEDURE GetScanCode () : BYTE;

    (* Gets one scan code from the keyboard. *)

PROCEDURE CheckScanCode () : BYTE;

    (* Like GetScanCode, but returns 0 immediately if no scan code is   *)
    (* available - i.e. does not wait for a keyboard press.             *)

PROCEDURE PutLEDs (LEDcode: BYTE);

    (* Sets the keyboard lock indicator lights, as specified by         *)
    (* LEDcode.  Unlike the following two procedures, which can affect  *)
    (* one LED without disturbing the others, this procedure alters all *)
    (* three LEDs as a group.                                           *)

PROCEDURE ClearLED (LEDcode: BYTE);

    (* Clears one or more of the keyboard lock indicator lights.        *)

PROCEDURE ToggleLED (LEDcode: BYTE);

    (* Toggles one or more of the keyboard lock indicator lights.       *)

END KBdriver.
```

```
                                (*  to accept a command.                *)
        ControllerOutOfSync,    (* Status information from the disk     *)
                                (*  controller does not make sense.     *)
        TimeoutError,           (* Data request interrupt did not       *)
                                (*  arrive within a reasonable time.    *)
        CalibrationFailure,     (* Did not succeed in driving the head  *)
                                (*  back to its home position.          *)
        SeekFailure,            (* Failed to seek to the desired track. *)
        DriveNotReady,          (* Could be an equipment failure, but   *)
                                (*  more commonly means that the drive  *)
                                (*  door is open or there is no disk in *)
                                (*  the drive.                          *)
        SectorNotFound,         (* Missing sector, cylinder mismatch,   *)
                                (*  and similar errors.  This could     *)
                                (*  mean a badly formatted disk, it     *)
                                (*  could be just one faulty block, or  *)
                                (*  it could mean that a recalibration  *)
                                (*  is needed.                          *)
        BadBlock,               (* The sector has its "bad block"       *)
                                (*  mark set - should not be used.      *)
        BadData,                (* Something on the disk is corrupted.  *)
                                (*  Could be a transient error, e.g.    *)
                                (*  read error caused by a speck of     *)
                                (*  dust, but it could also be true     *)
                                (*  corruption of the medium.           *)
        WriteFault,             (* Occurs only with hard disk, reflects *)
                                (*  a hardware fault signal.            *)
        WriteProtected,         (* Attempted write to a write-protected *)
                                (*  disk.                               *)
        UndiagnosedFailure      (* Miscellaneous error.                 *)

        );              (* End of the list of error codes *)

(*************************************************************************)

PROCEDURE WriteErrorCode (w: Window;  code: ErrorCode);

    (* Writes an error code to the screen.      *)

END IOErrorCodes.
```

IOErrorCodes

```
DEFINITION MODULE IOErrorCodes;

        (*********************************************************)
        (*                                                       *)
        (*              I/O subsystem error codes.               *)
        (*                                                       *)
        (*  This module defines a set of error codes which may   *)
        (*  be used uniformly throughout the file system.        *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        18 July 1993                     *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM Windows IMPORT
    (* type *)  Window;

(************************************************************************)

TYPE
    ErrorCode = (

        OK,                     (* Normal return, no error      *)
        OperationAborted,       (* Operator forced abort        *)

        (* Error codes from module Files.       *)

        FileNotOpen,            (* Should have opened file first  *)

        (* Error codes from module Devices.     *)

        NoSuchDevice,           (* Illegal device *)
        NoSuchUnit,             (* Illegal unit number *)

        (* Error codes from module Directories. *)

        FeatureNotImplemented,  (* Cannot yet handle this case *)
        InvalidFileNameString,  (* Could not parse filename string *)
        DirectoryNotFound,      (* Subdirectory name not found *)
        NotADirectory,          (* Supposed directory name is not a dir *)
        NameNotFound,           (* File name not found in directory *)
        DuplicateFileName,      (* File name already exists *)
        DeviceFull,             (* No room to create/extend file *)
        DirectoryFull,          (* No room to create new directory entry *)

        (* Error codes from the disk device drivers.    *)

        BadDMAAddress,          (* Operation uses a memory buffer which *)
                                (* the DMA hardware cannot handle       *)
        IllegalBlockNumber,     (* The caller requested an operation on *)
                                (*  a non-existent disk block.          *)
        BadCommand,             (* We tried to do something illegal.    *)
        ControllerNotListening, (* Could not get the disk controller    *)
```

```
INLINE
PROCEDURE NPXrestore (VAR (*IN*) Buffer: NPXSaveArea)

    (* The operation complementary to NPXsave.  Restores the previously *)
    (* saved state of the floating point coprocessor.                   *)

     = Code3 (9BH, 0DDH, 24H);  (* wait; frstor [si] *)

    (*# restore *)

END InnerKernel.
```

```
    (*# save, call(reg_param => (si, ds, ax, bx)) *)

PROCEDURE Transfer (VAR (*OUT*) source: TaskSelector;
                                    destination: TaskSelector);

    (* Performs a task switch to the destination task, at the same time *)
    (* saving a selector for the outgoing task in variable "source".    *)
    (* This allows a subsequent call to Transfer to resume the          *)
    (* original task.  By the time this procedure has returned to the   *)
    (* caller, then, we are again executing the calling task.           *)

    (* Special case: if this procedure is called by an interrupt task,  *)
    (* the call is interpreted as a requiring a task switch from the    *)
    (* interrupted task - i.e. the source parameter must specify the    *)
    (* interrupted task - to the destination task.  In this case the    *)
    (* actual switch to the destination task does not happen until the  *)
    (* interrupt task makes its next call to IOTransfer.  The reason    *)
    (* for this interpretation is that task switching to and from       *)
    (* interrupt tasks is managed internally by this module; the        *)
    (* occurrence of an interrupt is not something that can be          *)
    (* controlled by the caller.                                        *)

    (*# restore *)

PROCEDURE IOTransfer;

    (* May be called only from an interrupt task.  Performs a task      *)
    (* switch from the current interrupt task to the task which it      *)
    (* interrupted.  Unlike Transfer, no parameters are required        *)
    (* because (a) the selector for the destination task is already     *)
    (* known to this module, having been saved at the time of the       *)
    (* interrupt; and (b) selectors for interrupt tasks are maintained  *)
    (* directly by this module rather than by the caller.               *)

PROCEDURE StartInterruptTask (TS: TaskSelector; InterruptNumber: CARDINAL);

    (* Starts an interrupt task by running its initialisation section   *)
    (* - i.e. everything up to the first IOTransfer - and arranging     *)
    (* that from then on it will be activated by the given interrupt.   *)

    (*# save, call(reg_param => (ax, ds)) *)

PROCEDURE DisconnectFromInterrupt (TS: TaskSelector);

    (* Restores the interrupt vector to which TS was connected to its   *)
    (* state before TS was established as an interrupt task.  (N.B. The *)
    (* result could be chaotic if there was no previous call to         *)
    (* StartInterruptTask.)                                             *)

    (*# restore *)

    (*# save, call(reg_param => (si, ds)) *)

INLINE
PROCEDURE NPXsave (VAR (*OUT*) Buffer: NPXSaveArea)

    (* Saves the state of the Numeric Processor Extension coprocessor.  *)
    (* (Has no effect, apart from a short time delay, if there is no    *)
    (* coprocessor present.)                                            *)

     = Code3 (0DDH, 34H, 9BH);  (* fnsave [si]; wait *)
```

InnerKernel

```
DEFINITION MODULE InnerKernel;

        (*****************************************************************)
        (*                                                               *)
        (*      This is the nonportable part of the PMOS kernel.         *)
        (*      It contains procedures whose implementation depends      *)
        (*      not only on the processor, but also on compiler          *)
        (*      conventions (which registers are saved, etc.).           *)
        (*                                                               *)
        (*      Programmer:     P. Moylan                                *)
        (*      Last edited:    7 November 1993                          *)
        (*                                                               *)
        (*      Status:         Working                                  *)
        (*                                                               *)
        (*****************************************************************)

TYPE
    Code3 = ARRAY [0..2] OF SHORTCARD;
    TaskSelector = ADDRESS;
    NPXSaveArea = ARRAY [0..46] OF WORD;

(*************************************************************************)

PROCEDURE EnterKernel (): CARDINAL;

    (* Saves the processor flags word, including the current "interrupt *)
    (* enable" status, and returns with interrupts disabled.            *)
    (* NOTE: this procedure and the following one should be used as a   *)
    (* matched pair.                                                    *)

PROCEDURE LeaveKernel (PSW: CARDINAL);

    (* Restores the processor flags word, including the "interrupt      *)
    (* enable" status.  NOTE: this procedure and the one above should   *)
    (* be used as a matched pair.                                       *)

    (*# save, call(reg_return => (bx, es)) *)

PROCEDURE TaskInit (StackBase: ADDRESS;  StackSize: CARDINAL;
                        EnableInterrupts: BOOLEAN;
                        TaskExit, StartAddress: PROC): TaskSelector;

    (* Initialises the stack for a new task.  Parameter StackBase       *)
    (* points to a block of memory which can be used to hold the stack  *)
    (* (note that this is a pointer to the start of the memory block,   *)
    (* not to the bottom of the stack); and StackSize is the size of    *)
    (* this block.  The next parameter specifies whether processor      *)
    (* interrupts should be enabled when the task is started.           *)
    (* StartAddress and TaskExit are the start address of the task code *)
    (* and the start address of the code to execute when the task       *)
    (* terminates.  The value returned is a selector for the new task.  *)

    (*# restore *)
```

HardDisk

```
DEFINITION MODULE HardDisk;

        (*********************************************************)
        (*                                                       *)
        (*              Device driver for hard disk              *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        27 July 1992                     *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*      Note that there are no explicit entry            *)
        (*      points to this module.  To use it, put an        *)
        (*      "IMPORT HardDisk" declaration somewhere in       *)
        (*      your program, and do all I/O through module      *)
        (*      Files (for file-structured I/O), or module       *)
        (*      Devices (for low-level operations).              *)
        (*                                                       *)
        (*      Do not use this module under OS/2; use module    *)
        (*      FileSys instead.                                 *)
        (*                                                       *)
        (*********************************************************)

END HardDisk.
```

```
(*************************************************************************)
(*                          TEXT OPERATIONS                              *)
(*************************************************************************)
(*                                                                       *)
(*  Every open window has a "text cursor" which is used only for text    *)
(*  operations and is independent of any operations on dots and lines.   *)
(*  The text cursor is updated after any text operation in such a way    *)
(*  that the characters follow one another in the way one would expect   *)
(*  for non-graphics windows.                                            *)
(*                                                                       *)
(*************************************************************************)

PROCEDURE SetCursor (w: Window;  row, column: CARDINAL);

    (* Sets the text cursor to the specified row and column.  The row    *)
    (* and column are measured in units of characters (not pixels),      *)
    (* with (0,0) representing the first character position at the       *)
    (* upper left of the window.                                         *)

PROCEDURE WriteChar (w: Window;  ch: CHAR);

    (* Writes a horizontal character at the current text cursor          *)
    (* position for window w.  The text cursor is updated.               *)

PROCEDURE WriteString (w: Window;  text: ARRAY OF CHAR);

    (* Writes a horizontal character string at the current text cursor   *)
    (* position for window w.  Characters which do not fit on the        *)
    (* current line are wrapped around to a new row.                     *)

PROCEDURE WriteLn (w: Window);

    (* Sets the text cursor to the start of the next text line down.     *)
    (* If the cursor reaches the bottom of the window, the text in the   *)
    (* window is scrolled.                                               *)

END GWindows.
```

```
PROCEDURE WindowMemory (w: Window;  memory: BOOLEAN);

    (* Specifying a FALSE value for the memory parameter means that     *)
    (* subsequent data sent to this window will be written to the       *)
    (* screen but not remembered.  This saves time and memory, the only *)
    (* penalty being that data covered by an overlapping window will    *)
    (* be lost.  Specifying TRUE restores the default condition, where  *)
    (* all window data are retained for refreshing the screen when      *)
    (* necessary.                                                       *)

PROCEDURE CloseWindow (VAR (*INOUT*) w: Window);

    (* Destroys the window.     *)

PROCEDURE SetColour (w: Window;  colour: ColourType);

    (* Specifies the foreground colour to be used until further notice. *)

PROCEDURE PutPixel (w: Window;  p: Point);

    (* Plots a dot at the point (x,y) in window w.  The coordinates are *)
    (* relative to the bottom left of the window.  If the dot lies      *)
    (* outside the window it will be ignored.                           *)

PROCEDURE PutPixel2 (w: Window;  x, y: INTEGER);

    (* Same as PutPixel, with a different way of specifying the point.  *)

PROCEDURE PutPixel2C (w: Window;  x, y: INTEGER;  colour: ColourType);

    (* Same as PutPixel2, with the colour explicitly specified. *)

PROCEDURE Line (w: Window;  start, end: Point);

    (* Draws a straight line.  The points are relative to the bottom    *)
    (* left corner of w.  Parts of the line lying outside the window    *)
    (* are clipped.                                                     *)

PROCEDURE Line2 (w: Window;  xstart, ystart, xend, yend: INTEGER);

    (* The same operation as Line, with a different way of specifying   *)
    (* the parameters.                                                  *)

PROCEDURE Line2C (w: Window;  xstart, ystart, xend, yend: INTEGER;
                                                    colour: ColourType);

    (* The same operation as Line2, but with the colour explicitly      *)
    (* specified.                                                       *)

PROCEDURE GString (w: Window;  x, y: CARDINAL;  text: ARRAY OF CHAR);

    (* Writes a horizontal character string at graphics position (x,y)  *)
    (* relative to window w.  Characters which do not fit are not       *)
    (* displayed.  This is not considered to be a text operation since  *)
    (* the text cursor is not affected and there is no line wrap.       *)

PROCEDURE GStringUp (w: Window;  x, y: CARDINAL;  text: ARRAY OF CHAR);

    (* Like GString, but the string is rotated counterclockwise by      *)
    (* 90 degrees, i.e. it is written in the +Y direction.              *)

PROCEDURE ClearWindow (w: Window);

    (* Erases all data from w, but keeps it open.       *)
```

GWindows

```
DEFINITION MODULE GWindows;

        (*********************************************************)
        (*                                                       *)
        (*          Windows module for screen graphics           *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        12 January 1994                  *)
        (*  Status:                                              *)
        (*      Mostly working, still adding features.           *)
        (*                                                       *)
        (*********************************************************)

FROM ScreenGeometry IMPORT
    (* type *)  Point, Rectangle;

FROM Graphics IMPORT
    (* type *)  ColourType;

(*************************************************************************)

TYPE
    Window;     (* is private *)

    BorderType = (single, double);

(*************************************************************************)

PROCEDURE InitGraphics (mode: CARDINAL);

    (* Sets up the Graphics mode.  Optional, since the module starts up *)
    (* with a best estimate of the "best" mode possible on the          *)
    (* available hardware.                                              *)

PROCEDURE OpenWindow (VAR (*OUT*) w: Window;
                              left, bottom, right, top: CARDINAL;
                              Foregrnd, Backgrnd: ColourType;
                              b: BorderType);

    (* Creates a new window.    *)

PROCEDURE OpenWindowR (VAR (*OUT*) w: Window;  location: Rectangle;
                                   Foregrnd, Backgrnd: ColourType;
                                   b: BorderType);

    (* Same as OpenWindow, except for method of specifying location.    *)
```

```
PROCEDURE ClippedUpString (VAR (*IN*) text: ARRAY OF CHAR;
                           x, y, length: CARDINAL;  colour: ColourType;
                           left, right, bottom, top: CARDINAL);

    (* Like ClippedString, but with text written in the +Y direction.   *)

END Graphics.
```

```
PROCEDURE PlotMark (x, y: CARDINAL;
                    colour: ColourType;  pointtype: SHORTCARD);

    (* Writes a mark at screen position (x, y).  Currently, the options *)
    (* for pointtype are:                                               *)
    (*          0       dot                                             *)
    (*          1       X                                               *)
    (*          2       box                                             *)

PROCEDURE PlotLine (x0, y0, x1, y1: CARDINAL;  colour: ColourType);

    (* Plots a straight line from (x0,y0) to (x1,y1).  It is the        *)
    (* caller's responsibility to ensure that the coordinates are in    *)
    (* range for the current video mode.                                *)

PROCEDURE PlotRectangle (R: Rectangle;  colour: ColourType);

    (* Plots a rectangle, with clipping if necessary to keep the        *)
    (* points within the screen boundary.                               *)

PROCEDURE ClippedLine (x0, y0, x1, y1: CARDINAL;  colour: ColourType;
                    left, right, bottom, top: CARDINAL);

    (* Like PlotLine, but plots only that part of the line which lies   *)
    (* in the rectangle (left <= x <= right), (bottom <= y <= top).     *)
    (* The caller is expected to ensure, by appropriate definition of   *)
    (* the rectangle, that all plotted points are in range for the      *)
    (* current video mode.                                              *)

PROCEDURE Fill (x0, y0, x1, y1: CARDINAL;  colour: ColourType);

    (* Fills the rectangle whose bottom left corner is (x0,y0) and      *)
    (* whose top right corner is (x1,y1) with the indicated colour.     *)

PROCEDURE DrawChar (ch: CHAR;  x, y: CARDINAL;  colour: ColourType);

    (* Draws the single character ch.  The coordinates (x,y) are the    *)
    (* location of the bottom left of the character.                    *)

PROCEDURE PlotString (VAR (*IN*) text: ARRAY OF CHAR;
                    x, y, length: CARDINAL;  colour: ColourType);

    (* Draws a string of "length" characters starting at location (x,y) *)
    (* It is the caller's responsibility to ensure that the string will *)
    (* not run off the screen edges.                                    *)

PROCEDURE ClippedString (VAR (*IN*) text: ARRAY OF CHAR;
                    x, y, length: CARDINAL;  colour: ColourType;
                    left, right, bottom, top: CARDINAL);

    (* Like PlotString, but excludes any points which fall outside the  *)
    (* clip rectangle defined by (left,right,bottom,top).               *)

PROCEDURE PlotStringUp (VAR (*IN*) text: ARRAY OF CHAR;
                    x, y, length: CARDINAL;  colour: ColourType);

    (* Like PlotString, but with text written in the +Y direction       *)
```

```
        (*     extent, but the colours are not what one would expect.   *)
        (*                                                              *)
        (* Note that the ATI vesa driver does not support Vesa modes    *)
        (* beyond 266, it doesn't support mode 261, and it fails on     *)
        (* mode 259.  (This last failure could possibly be fixed with   *)
        (* a more up-to-date VESA driver.)                              *)
        (*                                                              *)
        (****************************************************************)

FROM ScreenGeometry IMPORT
    (* type *)  Rectangle;

FROM Screen IMPORT
    (* const*)  HercGraphics;   (* 720x348 monochrome Hercules graphics *)

(************************************************************************)
(*                                                                      *)
(*  NOTE: When a reference is made to (x,y) coordinates in this module, *)
(*  the x value is the horizontal coordinate, with 0 at the left of the *)
(*  the screen, and the y value is the vertical coordinate, with 0 at   *)
(*  the bottom of the screen.  This is upside-down with respect to the  *)
(*  way the hardware works, but I believe it's the "natural" system     *)
(*  for users who are not necessarily familiar with the quirks of the   *)
(*  hardware.                                                           *)
(*                                                                      *)
(************************************************************************)

TYPE ColourType = CARDINAL;

PROCEDURE SetMode (newmode: CARDINAL;  ClearScreen: BOOLEAN);

    (* Sets the video mode. *)

PROCEDURE SetDefaultMode;

    (* Sets the video mode to (our opinion of) the "best" graphics mode *)
    (* supported by the hardware.                                       *)

PROCEDURE GraphicsOff (ClearScreen: BOOLEAN);

    (* Sets the video mode to a default text mode. *)

PROCEDURE GetScreenShape (VAR (*OUT*) xmax, ymax: CARDINAL;
                          VAR (*OUT*) maxcolour: ColourType);

    (* Returns the maximum values permitted by the current mode for     *)
    (* x, y, and colour.                                                *)

PROCEDURE SetPaletteColour (Palette_Index, Red, Green, Blue: SHORTCARD);

    (* Sets the palette register specified by the first parameter to    *)
    (* the colour combination specified by the Red, Green, and Blue     *)
    (* parameters.  These last three are 6-bit numbers.                 *)

PROCEDURE PlotDot (x, y: CARDINAL;  colour: ColourType);

    (* Writes a dot at screen position (x, y).  *)
```

```
(*      266     132x43 text                     Text            *)
(*      267     132x50 text                     Text            *)
(*      268     132x60 text                     Text            *)
(*      269     300x200x32K                     Untested        *)
(*      270     320x200x64K                     Untested        *)
(*      271     320x200x16.8M                   Not supported   *)
(*      272     640x480x32K                     Working         *)
(*      273     640x480x64K                     Working         *)
(*      274     640x480x16.8M                   Not supported   *)
(*      275     800x600x32K                     Working         *)
(*      276     800x600x64K                     Working         *)
(*      277     800x600x16.8M                   Not supported   *)
(*      278     1024x768x32K                    Untested        *)
(*      279     1024x768x64K                    Untested        *)
(*      280     1024x768x16.8M                  Not supported   *)
(*      281     1280x1024x32K                   Untested        *)
(*      282     1280x1024x64K                   Untested        *)
(*      283     1280x1024x16.8M                 Not supported   *)
(* Modes 271,274,277,280, and 283, which use 24-bit colours,    *)
(* are not supported by the current version of module Graphics. *)
(*                                                              *)
(* I now have partial support on the Trident for the            *)
(* following modes:                                             *)
(*      368     512x480x32K                     Working?        *)
(*      369     512x480x64K                     Working?        *)
(* These have poor horizontal position on my monitor, but this  *)
(* is possibly just a monitor limitation; otherwise they        *)
(* work well.                                                   *)
(*                                                              *)
(* Comments on modes which are "almost" working:                *)
(*      106,258,260                                             *)
(*              These modes work with my ATI adaptor, but on    *)
(*              the Trident I get a triple image across the     *)
(*              screen, as if there were a problem with         *)
(*              horizontal frequency applied to the monitor.    *)
(*      259     Works on the Trident, total failure on the      *)
(*              ATI card even though it's supposedly supported. *)
(*      261..263,269..283                                       *)
(*              The hardware I'm using so far doesn't support   *)
(*              these modes, so I can't yet test them.          *)
(*      348..351, 354, 362, 372..375 (plus some text modes)     *)
(*              these are in some sense "supported" on the      *)
(*              Trident; most of them turn out to be duplicates,*)
(*              so I've eliminated them from the list of        *)
(*              supported modes.  The only genuine extras are   *)
(*              272,273,275,276 (which are not listed as        *)
(*              supported, but are available anyway); and       *)
(*              368,369 (which are listed as supported,         *)
(*              although I'm not sure that the Vesa standard    *)
(*              includes them).                                 *)
(*                                                              *)
(* Note that on the Trident:                                    *)
(*   - all the 16-colour modes using bank switching show the    *)
(*     same symptom: a triple image apparently caused by a      *)
(*     clock synch problem;                                     *)
(*   - the 32K and 64K colour modes are working to some         *)
```

Graphics

```
DEFINITION MODULE Graphics;

        (*****************************************************************)
        (*                                                               *)
        (*                    Screen graphics output                     *)
        (*                                                               *)
        (*  Programmer:         P. Moylan                                *)
        (*  Last edited:        6 December 1993                          *)
        (*  Status:             OK                                       *)
        (*                                                               *)
        (*      The procedures in this module assume that the caller     *)
        (*      has control of the entire screen.                        *)
        (*      For multi-window graphics, see module GWindows.          *)
        (*                                                               *)
        (* The support status for the various graphics modes is listed   *)
        (* below.  Text modes are listed for information only; this      *)
        (* module allows you to select a text mode, but does not supply  *)
        (* any support beyond that point.                                *)
        (*                                                               *)
        (*      -       720x348x1 (Hercules only)       Working          *)
        (*      0       40x25 BW text                   Text             *)
        (*      1       40x25 colour text               Text             *)
        (*      2       80x25 BW text                   Text             *)
        (*      3       80x25 colour text               Text             *)
        (*      4       320x200x4 (CGA)                 Working          *)
        (*      5       320x200 BW (CGA)                Working          *)
        (*      6       640x200 BW (CGA)                Working          *)
        (*      7       80x25 mono text (MDA/Hercules)  Text             *)
        (*      8-12    unsure, but not graphics        Not supported    *)
        (*      13      320x200x16 (EGA)                Working          *)
        (*      14      640x200x16 (EGA)                Working          *)
        (*      15      640x350x1  (EGA)                Working          *)
        (*      16      640x350x16 (EGA)                Working          *)
        (*      17      640x480x1  (VGA)                Working          *)
        (*      18      640x480x16 (VGA)                Working          *)
        (*      19      320x200x256 (VGA)               Working          *)
        (*      20-105  not sure                        Not supported    *)
        (*      106     800x600x16                      Working?         *)
        (*      107-255 not sure                        Not supported    *)
        (*      256     640x400x256                     Working          *)
        (*      257     640x480x256                     Working          *)
        (*      258     800x600x16                      Working?         *)
        (*      259     800x600x256                     Working          *)
        (*      260     1024x768x16                     Working?         *)
        (*      261     1024x768x256                    Untested         *)
        (*      262     1280x1024x16                    Untested         *)
        (*      263     1280x1024x256                   Untested         *)
        (*      264     80x60 text                      Text             *)
        (*      265     132x25 text                     Text             *)
```

```
PROCEDURE WriteHexWord (number: CARDINAL);

    (* Writes its argument as a four-digit hexadecimal number.          *)

PROCEDURE WriteInt (number: INTEGER);
PROCEDURE WriteCard (number: CARDINAL);
PROCEDURE WriteLongCard (number: LONGCARD);

    (* Writes a number to the screen.   *)

PROCEDURE WriteAddress (addr: ADDRESS);

    (* Writes a segmented address to the screen.        *)

END GlassTTY.
```

GlassTTY

```
DEFINITION MODULE GlassTY;

        (*******************************************************)
        (*                                                     *)
        (*           Simple screen output routines.            *)
        (*                                                     *)
        (*  This module handles screen output at a very low    *)
        (*  level, without supplying the advanced features     *)
        (*  which may be found in, for example, module Windows. *)
        (*  It is intended for things like error message       *)
        (*  output, and is designed for compactness rather     *)
        (*  than comprehensiveness.                            *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        13 December 1993               *)
        (*  Status:             OK                             *)
        (*                                                     *)
        (*******************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

PROCEDURE WriteChar (ch: CHAR);

    (* Writes one character. *)

PROCEDURE WriteString (text: ARRAY OF CHAR);

    (* Writes a string of characters.   *)

PROCEDURE WriteLn;

    (* Moves the screen cursor to the beginning of the next line,       *)
    (* scrolling if necessary.                                          *)

PROCEDURE SetCursor (row, column: CARDINAL);

    (* Moves the screen cursor to the specified row and column.         *)

PROCEDURE SaveCursor;

    (* Remembers the current cursor position, for use by a subsequent   *)
    (* call to RestoreCursor.  Note that nesting is not supported, i.e. *)
    (* a call to SaveCursor destroys the information saved by any       *)
    (* earlier call to SaveCursor.                                      *)

PROCEDURE RestoreCursor;

    (* Sets the cursor back to where it was at the time of the last     *)
    (* call to SaveCursor.                                              *)

PROCEDURE WriteHexByte (number: BYTE);

    (* Writes its argument as a two-digit hexadecimal number.           *)
```

Floppy

```
DEFINITION MODULE Floppy;

        (*********************************************************)
        (*                                                       *)
        (*          Device driver for floppy disk.               *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        15 June 1992                     *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*      Note that there are no explicit entry            *)
        (*      points to this module.  To use it, put an        *)
        (*      "IMPORT Floppy" declaration somewhere in         *)
        (*      your program, and do all I/O through module      *)
        (*      Files (for file-structured I/O), or module       *)
        (*      Devices (for low-level operations).              *)
        (*                                                       *)
        (*  Remark: This version supports only two drives, even  *)
        (*  though the controller hardware is capable of         *)
        (*  dealing with four drives.  I have actually tested    *)
        (*  the software with four drives, and there would be    *)
        (*  a negligible amount of work involved in modifying    *)
        (*  the module to handle the four-drive case; but I've   *)
        (*  never heard of any installation which has the four   *)
        (*  drives, so there seemed to be little point in        *)
        (*  retaining that feature.                              *)
        (*                                                       *)
        (*********************************************************)

END Floppy.
```

```
PROCEDURE OpenFile (VAR (*OUT*) f: File;  name: ARRAY OF CHAR;
                                    newfile: BOOLEAN): ErrorCode;

    (* Opens the file named by the given character string, and returns  *)
    (* f as the identification to be used when specifying this file in  *)
    (* future.  The caller must specify newfile = TRUE to create a new  *)
    (* file, or newfile = FALSE if the intention is to open an existing *)
    (* file.  It is illegal to open a new file with the same name as an *)
    (* existing file; this is to protect against accidental deletions.  *)
    (* The value returned is an error code (OK if no error).            *)

PROCEDURE CloseFile (VAR (*INOUT*) f: File);

    (* Closes file f.   *)

PROCEDURE EOF (f: File): BOOLEAN;

    (* Returns TRUE iff we are currently at the end of file f.  *)

PROCEDURE ReadByte (f: File): BYTE;

    (* Returns the next byte from the file.     *)

PROCEDURE ReadRecord (f: File;  buffaddr: ADDRESS;  desired: CARDINAL;
                                VAR (*OUT*) actual: CARDINAL): ErrorCode;

    (* Reads up to "desired" bytes from file f to memory location       *)
    (* "buffaddr".  On return, "actual" gives the number of bytes read. *)

PROCEDURE WriteByte (f: File;  value: BYTE): ErrorCode;

    (* Writes one byte to the file.     *)

PROCEDURE WriteRecord (f: File;  buffaddr: ADDRESS;
                                    count: CARDINAL): ErrorCode;

    (* Writes count bytes from memory location buffaddr.*)

PROCEDURE SetPosition (f: File;  position: LONGCARD): ErrorCode;

    (* Ensures that the next read or write on this file will be at      *)
    (* byte number position in the file.  (The first byte in the file   *)
    (* is byte number 0.)  If a position greater than the file size     *)
    (* is specified, the length of the file will increase.              *)

PROCEDURE SavePosition (f: File): LONGCARD;

    (* Returns the current byte number in file f.       *)

PROCEDURE FileSize (f: File): LONGCARD;

    (* Returns the length of the file in bytes. *)

END FileSys.
```

FileSys

```
DEFINITION MODULE FileSys;

        (*****************************************************************)
        (*                                                               *)
        (*                       File operations                         *)
        (*                                                               *)
        (*      This version is functionally equivalent to module        *)
        (*      Files, but it uses standard library calls rather than    *)
        (*      working through the PMOS device drivers.  It is          *)
        (*      recommended for use in cases where the Files module      *)
        (*      is unsuitable for reasons such as (a) I/O on a device    *)
        (*      for which a PMOS device driver does not exist, or        *)
        (*      (b) conflicts caused by a cache system which does not    *)
        (*      recognise changes to directories.                        *)
        (*                                                               *)
        (*      Note, however, that use of this module rather than       *)
        (*      module Files can limit the real-time response and        *)
        (*      cause difficulties in effective multitasking.  In        *)
        (*      particular, if the underlying operating system is        *)
        (*      MS-DOS then multiple tasks should not attempt to         *)
        (*      perform file operations in parallel.                     *)
        (*                                                               *)
        (*      Note too that the error code returned on an I/O          *)
        (*      error is a rough guess, because of the lack of           *)
        (*      correspondence between the library error codes and       *)
        (*      the standard PMOS error codes.                           *)
        (*                                                               *)
        (*      Programmer:     P. Moylan                                *)
        (*      Last edited:    14 April 1993                            *)
        (*      Status:         OK                                       *)
        (*                                                               *)
        (*****************************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

FROM FIO IMPORT
    (* type *)  File;

FROM IOErrorCodes IMPORT
    (* type *)  ErrorCode;
```

FileSort

```
DEFINITION MODULE FileSort;

        (*********************************************************)
        (*                                                       *)
        (*      In-place file sort using the QuickSort method    *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        4 August 1993                    *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

FROM QuickSortModule IMPORT
    (* type *)  CompareProc;

FROM FileSys IMPORT
    (* type *)  File;

TYPE
    RecordNumber = LONGCARD;

PROCEDURE InplaceSort (f: File;  from, to: RecordNumber;
                       EltSize, offset: CARDINAL;  GE: CompareProc);

    (* In-place sort of part of a file.  We sort record numbers         *)
    (* from..to inclusive.  EltSize is the element size; offset is the  *)
    (* number of bytes (zero, in most cases) before record number 0 in  *)
    (* the file; and GE is a user-supplied function to compare elements *)
    (* at two specified addresses.                                      *)

END FileSort.
```

```
PROCEDURE ReadByte (f: File): BYTE;

    (* Returns the next byte from the file.     *)

PROCEDURE ReadRecord (f: File;  buffaddr: ADDRESS;  desired: CARDINAL;
                                VAR (*OUT*) actual: CARDINAL): ErrorCode;

    (* Reads up to "desired" bytes from file f to memory location       *)
    (* "buffaddr".  On return, "actual" gives the number of bytes read. *)

PROCEDURE WriteByte (f: File;  value: BYTE): ErrorCode;

    (* Writes one byte to the file.     *)

PROCEDURE WriteRecord (f: File;  buffaddr: ADDRESS;
                                       count: CARDINAL): ErrorCode;

    (* Writes count bytes from memory location buffaddr.*)

PROCEDURE SetPosition (f: File;  position: LONGCARD): ErrorCode;

    (* Ensures that the next read or write on this file will be at      *)
    (* byte number position in the file.  (The first byte in the file   *)
    (* is byte number 0.)  If a position greater than the file size     *)
    (* is specified, the length of the file will increase.              *)

PROCEDURE SavePosition (f: File): LONGCARD;

    (* Returns the current byte number in file f.       *)

PROCEDURE FileSize (f: File): LONGCARD;

    (* Returns the length of the file in bytes. *)

END Files.
```

Files

```
DEFINITION MODULE Files;

        (*********************************************************)
        (*                                                       *)
        (*                  File operations.                     *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        16 September 1991                *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*      IMPORTANT NOTE: The file system starts with      *)
        (*      no pre-conceived idea of what devices are        *)
        (*      present; it's up to the device drivers           *)
        (*      themselves to "install" themselves at program    *)
        (*      initialisation time.  To make this work, the     *)
        (*      user of this module should import whatever       *)
        (*      device drivers are needed.  Furthermore, this    *)
        (*      IMPORT declaration must come before the          *)
        (*      IMPORT of module Files, to ensure that device    *)
        (*      driver initialisation is complete before the     *)
        (*      file system starts its own initialisation.       *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

FROM IOErrorCodes IMPORT
    (* type *)  ErrorCode;

TYPE
    File;       (* is private *)

PROCEDURE OpenFile (VAR (*OUT*) f: File;  name: ARRAY OF CHAR;
                                    newfile: BOOLEAN): ErrorCode;

    (* Opens the file named by the given character string, and returns  *)
    (* f as the identification to be used when specifying this file in  *)
    (* future.  The caller must specify newfile = TRUE to create a new  *)
    (* file, or newfile = FALSE if the intention is to open an existing *)
    (* file.  It is illegal to open a new file with the same name as an *)
    (* existing file; this is to protect against accidental deletions.  *)
    (* The value returned is an error code (OK if no error).            *)

PROCEDURE CloseFile (VAR (*INOUT*) f: File);

    (* Closes file f.   *)

PROCEDURE EOF (f: File): BOOLEAN;

    (* Returns TRUE iff we are currently at the end of file f.  *)
```

```
    NameList = POINTER TO NameListRecord;
    NameListRecord= RECORD
                        string: FileName;
                        child: NameList;
                    END (*RECORD*);

(************************************************************************)

PROCEDURE WriteFileName (w: Window;  name: FileName);

    (* For testing: writes name in window w.    *)
    (* Probably won't need to export this in final version.     *)

PROCEDURE Parse (name: ARRAY OF CHAR;  VAR (*OUT*) device: Device;
                VAR (*OUT*) unit: CARDINAL;  VAR (*OUT*) result: NameList;
                VAR (*OUT*) StartAtRoot: BOOLEAN);

    (* Translates a text string "name" into NameList form.  If the      *)
    (* string includes a device specification this is returned as       *)
    (* (device, unit); otherwise we return with device = NullDevice.    *)
    (* Output parameter StartAtRoot is TRUE iff the filename string     *)
    (* started with a '\'.                                              *)

PROCEDURE DiscardNameList (VAR (*INOUT*) path: NameList);

    (* Disposes of the storage occupied by a NameList.  Returns NIL.    *)

END FileNames.
```

FileNames

```
DEFINITION MODULE FileNames;

        (*********************************************************)
        (*                                                       *)
        (*              File name parsing                        *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        19 January 1993                  *)
        (*  Status:             Just started                     *)
        (*                                                       *)
        (*      This module, which is part of the file system    *)
        (*      and is not intended for end-user use, looks      *)
        (*      after translating file name strings into a       *)
        (*      more convenient internal form.                   *)
        (*                                                       *)
        (*********************************************************)

FROM Windows IMPORT
    (* type *)  Window;

FROM Devices IMPORT
    (* type *)  Device;

(*************************************************************************)

TYPE

    (* File names (and directory names) are at most eight characters    *)
    (* long.  This archaic restriction is regretted, but we have no     *)
    (* choice if we want to remain compatible with the MS-DOS directory *)
    (* formats.  Actually, longer names are accepted (see procedure     *)
    (* Scan), but all but the first eight characters are skipped.       *)
    (* FileNameExtension refers to the three characters which may       *)
    (* appear after a period in the complete file name.                 *)

    EightChar = ARRAY [0..7] OF CHAR;           (* not exported *)
    FileNameString = EightChar;
    FileNameExtension = ARRAY [0..2] OF CHAR;
    FileName =  RECORD
                    fname: FileNameString;
                    fext: FileNameExtension;
                END (*RECORD*);

    (* The NameList form of a file name is a linked list where the      *)
    (* first entry gives the device name, following entries give the    *)
    (* directory names in order, and the last entry is the name of the  *)
    (* file within its own directory.                                   *)
```

```
(*************************************************************************)
(*                            THE EDITOR                                 *)
(*************************************************************************)

PROCEDURE EditField (w: Window;  VAR (*INOUT*) address: ADDRESS;
                                 type: FieldType;  width: CARDINAL);

    (* Edits the variable at the given address, and of the given type,  *)
    (* at the current cursor position in window w.  The width parameter *)
    (* specifies how many character positions are to be used on the     *)
    (* screen.  Set width=0 for variable-width fields where the editor  *)
    (* must determine the width.  We leave this procedure on seeing a   *)
    (* keyboard character which does not belong to us.  The cursor is   *)
    (* left just beyond the last character of the field as it is        *)
    (* displayed.  The terminating keystroke is returned to the         *)
    (* keyboard driver so that it can still be read by the caller.      *)
    (* Note that the address is an inout parameter because there are    *)
    (* cases where we allow the user to create and delete fields, i.e.  *)
    (* address could be NIL on entry but not on exit, or vice versa.    *)

END FieldEditor.
```

```
(*  avoid duplication of effort.  These properties are used to          *)
(*  advantage in modules ListEditor and ScreenEditor.                   *)
(*                                                                      *)
(*  As an added bonus, this module exports some pre-defined field types *)
(*  for commonly encountered cases.  For those cases, the user does not *)
(*  need to call DefineFieldType, and therefore does not need to supply *)
(*  the procedures for writing and editing variables of those types.    *)
(*                                                                      *)
(************************************************************************)

TYPE
    FieldType;          (* is private *)
    WriteProc = PROCEDURE (Window, ADDRESS, CARDINAL);
    EditProc = PROCEDURE (Window, VAR (*INOUT*) ADDRESS, CARDINAL);

(************************************************************************)
(*                       THE PREDEFINED TYPES                           *)
(************************************************************************)

VAR Byte, Cardinal, Real: FieldType;

(************************************************************************)
(*                       DEFINING A NEW TYPE                            *)
(************************************************************************)

PROCEDURE DefineFieldType (Writer: WriteProc;  Editor: EditProc): FieldType;

    (* Introduces a new field type into the system.  Writer is a        *)
    (* user-supplied procedure to write a variable of the new type.     *)
    (* Editor is the user-supplied procedure for editing a variable of  *)
    (* that type.                                                       *)

PROCEDURE DiscardFieldType (type: FieldType);

    (* A notification from the user that this type will not be used     *)
    (* again (unless it is redefined by another call to procedure       *)
    (* DefineFieldType).  Use of this procedure is optional, but is     *)
    (* recommended for the sake of "clean" memory management.           *)

(************************************************************************)
(*                          COMPARING TYPES                             *)
(************************************************************************)

PROCEDURE SameType (t1, t2: FieldType): BOOLEAN;

    (* Returns TRUE iff t1 = t2.*)

(************************************************************************)
(*                           SCREEN OUTPUT                              *)
(************************************************************************)

PROCEDURE WriteField (w: Window;  address: ADDRESS;  type: FieldType;
                                                    width: CARDINAL);

    (* Writes address^ on the screen at the current cursor position in  *)
    (* window w.  The width parameter specifies how many character      *)
    (* positions to use.  Use width=0 for variable-width fields for     *)
    (* which the write procedure for that type must work out the width. *)
```

FieldEditor

```
DEFINITION MODULE FieldEditor;

        (*********************************************************)
        (*                                                       *)
        (*              Screen editing utilities                 *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        15 May 1991                      *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

FROM Windows IMPORT
    (* type *)  Window;

(************************************************************************)
(*                                                                      *)
(*  The editor in this module is "generic" in a limited sense.  It      *)
(*  performs screen editing of variables of arbitrary types, provided   *)
(*  that those types have been declared by calls to DefineFieldType.    *)
(*                                                                      *)
(*  The caller is required, when calling DefineFieldType, to supply     *)
(*  procedures which write and edit variables of that type.  Each of    *)
(*  the user-supplied procedures has three parameters: a window, a      *)
(*  pointer to the variable to be written or edited, and the number of  *)
(*  character positions to use.  For field types where the number of    *)
(*  character positions cannot be determined in advance, the caller is  *)
(*  expected to supply 0 as the value of the third parameter, and the   *)
(*  user-supplied procedures are expected to be able to work out the    *)
(*  actual width required.  The user-supplied procedures are expected,  *)
(*  in all cases, to leave the screen cursor at the character position  *)
(*  just beyond the written form of the field.  They must be prepared   *)
(*  to deal with NIL addresses.  The user-supplied editor must handle   *)
(*  all keystrokes which belong to it, but leave intact (via            *)
(*  Keyboard.Putback, for example) the keystroke which causes it to     *)
(*  return.  Note that it will be very common for the editor to receive *)
(*  a cursor movement key implying that the user does not want to       *)
(*  modify this field but is simply skipping over it.  In such cases    *)
(*  the editor still has the responsibility for showing the user where  *)
(*  the cursor is, by using blinking, reverse video, etc.               *)
(*                                                                      *)
(*  Given all of these rules, and the fact that all the hard work is to *)
(*  be done by user-supplied procedures, you might by now be wondering  *)
(*  whether there is any point in having this module.  The main point   *)
(*  is that the rules impose some uniform standards which make it       *)
(*  easier to develop readable software for applications which need a   *)
(*  lot of screen editing.  They also help, in some applications, to    *)
```

DMA

```
DEFINITION MODULE DMA;

        (*********************************************************)
        (*                                                       *)
        (*      Procedures to deal with Direct Memory Access     *)
        (*                  input and output.                    *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        15 June 1992                     *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  ADDRESS;

PROCEDURE CheckDMAAddress (Address: ADDRESS;  count: CARDINAL): BOOLEAN;

    (* Returns TRUE iff the given address and count values are          *)
    (* suitable for a DMA transfer.  An unsuitable pair is one where    *)
    (* the transfer would cross a 64K boundary in memory - a case which *)
    (* the DMA hardware cannot handle.                                  *)

PROCEDURE LoadDMAparameters (channel, operation: CARDINAL;
                             Address: ADDRESS;  count: CARDINAL);

    (* Loads the DMA controller with the address and count values       *)
    (* as a preliminary to starting a DMA transfer.                     *)
    (* The code for "operation" is 0 for verify, 1 for read (transfer   *)
    (* from external device to memory), and 2 for write (from memory to *)
    (* external device).  Other values are illegal.                     *)

    (* It is the caller's responsibility to know that the specified DMA *)
    (* channel is not already in use.  This is normally not a problem,  *)
    (* since each channel is permanently dedicated to a single use.     *)

    (* It is also the caller's responsibility to know that the given    *)
    (* address and count do not cause the transfer to cross a 64 Kbyte  *)
    (* boundary in main memory - a case which the DMA hardware cannot   *)
    (* handle.  Beware!                                                 *)

END DMA.
```

```
PROCEDURE SetDefaultDirectory (VAR (*IN*) path: ARRAY OF CHAR): ErrorCode;

    (* Sets the starting point for directory searches in subsequent     *)
    (* file operations.  Each device has a separate default directory.  *)
    (* If parameter path includes a device name, the default is set for *)
    (* that device; otherwise, the default directory is set for the     *)
    (* current default device.                                          *)

PROCEDURE Lookup (newfile: BOOLEAN;  name: ARRAY OF CHAR;
                  VAR (*OUT*) device: Device;
                  VAR (*OUT*) unit: CARDINAL;
                  VAR (*OUT*) fileid: Handle;
                  VAR (*OUT*) StartingBlock: BlockNumberType;
                  VAR (*OUT*) BytesPerCluster: CARDINAL;
                  VAR (*OUT*) BytesInFile: LONGCARD): ErrorCode;

    (* Parses the file name, returns the device code and unit number,   *)
    (* and looks up the device directory (which might involve some      *)
    (* subdirectory searches) to find the location of the directory     *)
    (* entry, the starting block number, cluster size in bytes, and     *)
    (* file size in bytes, for the file.  If newfile=TRUE, we create a  *)
    (* new directory entry for this file (or report an error if the     *)
    (* file already exists).  When creating a new file, we pre-allocate *)
    (* the first cluster; this partially avoids the complication of     *)
    (* having to go back and modify the directory entry when the first  *)
    (* cluster of data is ready to be written (we will still have to    *)
    (* modify the file size part of the entry when the file is closed). *)

PROCEDURE NextBlockNumber (fileid: Handle;
                           currentblock: BlockNumberType): BlockNumberType;

    (* Given the block number of the current cluster in a file, returns *)
    (* the block number of the following cluster.                       *)

PROCEDURE FindRelativeCluster (fileid: Handle;  N: CARDINAL)
                                                      : BlockNumberType;

    (* Returns the block number of the Nth cluster, where N = 0         *)
    (* corresponds to the starting cluster of the file.                 *)

PROCEDURE AllocateBlock (fileid: Handle;
                         currentblock: BlockNumberType): BlockNumberType;

    (* Allocates a new free cluster, and returns its block number.  The *)
    (* variable currentblock shows the block number of the last cluster *)
    (* used by this file - we need this to update the space allocation  *)
    (* chain.                                                           *)

PROCEDURE UpdateFileSize (fileid: Handle;  NewSize: LONGCARD);

    (* Updates a directory entry to show a modified file size.  *)

PROCEDURE DiscardHandle (VAR (*INOUT*) fileid: Handle);

    (* To be called when a file is no longer going to be accessed.      *)

END Directories.
```

Directories

```
DEFINITION MODULE Directories;

        (*******************************************************)
        (*                                                     *)
        (*              Disk directory lookup.                 *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        13 June 1992                   *)
        (*  Status:                                            *)
        (*      Working, but needs more detailed testing.      *)
        (*      FindRelativeCluster not tested.                *)
        (*                                                     *)
        (*******************************************************)

FROM Devices IMPORT
    (* type *)  Device, BlockNumberType;

FROM IOErrorCodes IMPORT
    (* type *)  ErrorCode;

(*************************************************************************)

TYPE Handle;    (* is private *)

TYPE Cluster = CARDINAL;

CONST

    (* NoSuchBlock is, by convention, returned to indicate a           *)
    (* nonexistent block.                                              *)

    NoSuchBlock = 0FFFFFFFFH;

(*************************************************************************)
(*                                                                       *)
(*  Remark: file space is allocated in clusters, where a cluster is      *)
(*  one or more blocks.  Because the mapping from cluster number to      *)
(*  block number varies from one disk to another (it depends on the      *)
(*  disk capacity and format), and because module Devices does not       *)
(*  know about clusters, it is simplest to keep that mapping private     *)
(*  to this module, and return information to the caller in terms of     *)
(*  block numbers.  However, the caller needs to be told the cluster     *)
(*  size in bytes, in order to know how much data to read or write       *)
(*  at a time.                                                           *)
(*                                                                       *)
(*************************************************************************)
```

```
PROCEDURE DeviceName (device: Device;  unit: CARDINAL;
                      name: ARRAY OF CHAR;  size: BlockNumberType;
                      DefaultDirString: ARRAY OF CHAR);

    (* Specifies an external name for a given device and unit number.   *)
    (* Duplicate names are permitted.  The size parameter gives the     *)
    (* number of sectors in the partition if this "device" is actually  *)
    (* a partition on a hard disk; but it is not necessarily meaningful *)
    (* for other device types.  (This parameter is supplied only for    *)
    (* the benefit of the file system when it has to deal with the      *)
    (* special case of a large partition.)  Device drivers which are    *)
    (* not prepared to specify a meaningful size should supply a value  *)
    (* of 0 for the size.                                               *)
    (* DefaultDirString specifies the initial default directory.        *)

PROCEDURE AcceptRequest (device: Device): RequestBlockPointer;

    (* Returns a pointer to the next request enqueued for this device.  *)
    (* If there is no next request, we wait until one appears.          *)

END Devices.
```

```
PROCEDURE IOrequest (VAR (*INOUT*) details: RequestBlock);

    (* Adds the requested I/O operation to the queue of pending         *)
    (* operations on details.device.  The operation might not be done   *)
    (* immediately since the device driver might still be working on an *)
    (* earlier request.  On I/O completion, the device driver performs  *)
    (* a Signal(details.DoneSemaphorePointer^).                         *)

PROCEDURE GetDefaultDirectory (device: Device;  unit: CARDINAL;
                               VAR (*OUT*) DirectoryString: ARRAY OF CHAR);

    (* Produces a string representing the default directory.  The       *)
    (* string includes the character form of the device name.           *)

PROCEDURE VolumeSize (device: Device;  unitnumber: CARDINAL): BlockNumberType;

    (* Returns the number of sectors on (device, unitnumber).  Note     *)
    (* that the result is meaningful only for device drivers which      *)
    (* supply this information (see DeviceName below).                  *)

PROCEDURE BlockRead (d: Device;  unitnumber: CARDINAL;
                     BlockNumber: BlockNumberType;
                     MemoryAddress: ADDRESS;  amount: CARDINAL): ErrorCode;

    (* Reads "amount" bytes into memory, starting at the                *)
    (* beginning of block number "BlockNumber" on the specified device. *)

PROCEDURE BlockWrite (d: Device;  unitnumber: CARDINAL;
                      BlockNumber: BlockNumberType;
                      MemoryAddress: ADDRESS;  amount: CARDINAL): ErrorCode;

    (* Writes "amount" bytes from memory, starting at the beginning of  *)
    (* block number "BlockNumber" on the specified device.              *)

PROCEDURE ReadPhysicalBlock (d: Device;  unitnumber: CARDINAL;
                BlockNo: BlockNumberType;  MemoryAddress: ADDRESS): ErrorCode;

    (* Reads one sector into memory, starting at the beginning of       *)
    (* physical block number "BlockNo" on the specified device.         *)

(************************************************************************)
(*              PROCEDURES CALLED BY THE DEVICE DRIVERS                 *)
(************************************************************************)

PROCEDURE InstallDeviceDriver (MaxUnitNumber: CARDINAL): Device;

    (* Called by the device driver as part of its initialization code,  *)
    (* to make the driver known to this module.  MaxUnitNumber is the   *)
    (* maximum unit number which this device driver will support (0 for *)
    (* a single-unit device driver).  The value returned is the means   *)
    (* by which the device driver will henceforth identify itself to    *)
    (* this module.                                                     *)
```

```
    (* An I/O operation is specified by the details in a request block  *)
    (* whose fields have the meaning:                                   *)
    (*                                                                  *)
    (*          Status          to show the result of the operation     *)
    (*                            (Status = OK if there was no error)   *)
    (*          operation       0 for verify, 1 for read, 2 for write   *)
    (*          device,unit     the device to be used                   *)
    (*          BlockNumber     the block number on the medium, used    *)
    (*                          only for block-oriented devices         *)
    (*          BufferAddress   the address of an array which holds the *)
    (*                          data to be written, or which will       *)
    (*                          receive the input data                  *)
    (*          ByteCount       the number of bytes to transfer         *)
    (*          DoneSemaphorePointer  the address of a semaphore on     *)
    (*                          which the device will perform a Signal  *)
    (*                          to let the user know that the operation *)
    (*                          is complete                             *)
    (*                                                                  *)
    (* The caller must not modify the contents of the request block     *)
    (* until completion of the I/O operation.  Device drivers do not    *)
    (* modify any field except the Status field.                        *)

TYPE
    RequestBlock = RECORD
                       Status: ErrorCode;
                       operation: OperationType;
                       device: Device;
                       unit: CARDINAL;
                       BlockNumber: BlockNumberType;
                       ByteCount: CARDINAL;
                       BufferAddress: ADDRESS;
                       DoneSemaphorePointer: POINTER TO Semaphore;
                   END (*RECORD*);

    RequestBlockPointer = POINTER TO RequestBlock;

(************************************************************************)
(*                 PROCEDURES CALLED BY THE USER TASKS                  *)
(************************************************************************)

PROCEDURE SameDevice (d1, d2: Device): BOOLEAN;

    (* Tests the condition d1 = d2.     *)

PROCEDURE NullDevice (): Device;

    (* Returns the device code which this module uses internally to     *)
    (* mean "nonexistent device" or "unknown device".  I/O operations   *)
    (* on this device are of course impossible (if attempted, they will *)
    (* result in the NoSuchDevice error code), but the device code can  *)
    (* be used by client modules as a marker to indicate that no        *)
    (* genuine device has yet been specified.                           *)

PROCEDURE IdentifyDevice (name: ARRAY OF CHAR;  VAR (*OUT*) device: Device;
                               VAR (*OUT*) unitnumber: CARDINAL);

    (* Given the character string form of a device name, returns the    *)
    (* device code and unit number.                                     *)
```

Devices

```
DEFINITION MODULE Devices;

        (*********************************************************)
        (*                                                       *)
        (*              Support for device drivers.              *)
        (*                                                       *)
        (*  The aim of this module is to a measure of device     *)
        (*  independence in I/O operations.  It provides a       *)
        (*  uniform I/O interface to all device drivers which    *)
        (*  choose to make themselves known to module Devices.   *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        16 June 1992                     *)
        (*  Status:             Working                          *)
        (*                                                       *)
        (*********************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, ADDRESS;

FROM IOErrorCodes IMPORT
    (* type *)  ErrorCode;

FROM Semaphores IMPORT
    (* type *)  Semaphore;

(*************************************************************************)

TYPE
    BlockNumberType = LONGCARD;

    (* Any I/O device supported by this module is specified by a pair   *)
    (* of type (device, unit), where the first component has private    *)
    (* type Device, and the second is a unit number (normally zero, but *)
    (* some device drivers support multi-unit devices).                 *)

    Device;     (* is private *)

    (* The I/O operations dealt with by this module.  Note that "read"  *)
    (* and "physicalread" have the same meaning for most devices - and  *)
    (* similarly for "write" and "physicalwrite" - but the distinction  *)
    (* is important for hard disks, which can be partitioned into       *)
    (* one or more logical disks.                                       *)

    OperationType = (verify, read, write, physicalread, physicalwrite,
                     shutdown);
```

```
PROCEDURE CardinalToString (number: CARDINAL;
                                    VAR (*OUT*) buffer: ARRAY OF CHAR;
                                    fieldsize: CARDINAL);
PROCEDURE LongCardToString (number: LONGCARD;
                                    VAR (*OUT*) buffer: ARRAY OF CHAR;
                                    fieldsize: CARDINAL);
PROCEDURE RealToString (number: REAL;  VAR (*OUT*) buffer: ARRAY OF CHAR;
                                    fieldsize: CARDINAL);
PROCEDURE LongRealToString (number: LONGREAL;
                                    VAR (*OUT*) buffer: ARRAY OF CHAR;
                                    fieldsize: CARDINAL);

    (* Converts the number to a decimal character string in array       *)
    (* "buffer", right-justified in a field of fieldsize characters.    *)
    (* The format depends on the size of the number relative to the     *)
    (* size of the buffer.                                              *)

PROCEDURE RealToF (number: REAL;  VAR (*INOUT*) fieldsize: CARDINAL;
                        decimalplaces: CARDINAL;  LeftJustified: BOOLEAN;
                        VAR (*OUT*) buffer: ARRAY OF CHAR);
PROCEDURE LongRealToF (number: LONGREAL;  VAR (*INOUT*) fieldsize: CARDINAL;
                        decimalplaces: CARDINAL;  LeftJustified: BOOLEAN;
                        VAR (*OUT*) buffer: ARRAY OF CHAR);

    (* Converts the number to an F-format string, of up to fieldsize    *)
    (* characters with decimalplaces digits after the decimal point.    *)
    (* The result is left justified if LeftJustified = TRUE is          *)
    (* specified by the caller, and right justified with space fill     *)
    (* otherwise.  On return fieldsize gives the number of character    *)
    (* positions actually used.  The result string is terminated with   *)
    (* at least one CHR(0) (which is not counted in fieldsize), except  *)
    (* where the result fills the entire buffer.                        *)

END Conversions.
```

Conversions

```
DEFINITION MODULE Conversions;

        (*******************************************************)
        (*                                                     *)
        (*              Miscellaneous type conversions         *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        21 July 1993                   *)
        (*  Status:             Working                        *)
        (*                                                     *)
        (*******************************************************)

TYPE HexDigit = [0..15];
     EightChar = ARRAY [0..7] OF CHAR;

PROCEDURE atoi (a: LONGREAL;  i: CARDINAL): LONGREAL;

    (* Calculates a**i.  This procedure does not really belong in this  *)
    (* module, but it's missing from MATHLIB and there's no more        *)
    (* logical place to put it.                                         *)

PROCEDURE HexToChar (number: HexDigit): CHAR;

    (* Converts a one-digit hexadecimal number to its readable form.    *)

PROCEDURE StringToHex (string: ARRAY OF CHAR): LONGCARD;
PROCEDURE StringToCardinal (string: ARRAY OF CHAR): CARDINAL;
PROCEDURE StringToLongCard (string: ARRAY OF CHAR): LONGCARD;
PROCEDURE StringToReal (string: ARRAY OF CHAR): REAL;
PROCEDURE StringToLongReal (string: ARRAY OF CHAR): LONGREAL;

    (* Converts a text string to numeric.  Leading blanks are ignored.  *)
    (* The conversion stops at the end of the array or at the first     *)
    (* character which cannot be part of the number, and in the         *)
    (* latter case all subsequent characters are ignored.               *)

PROCEDURE HexToString (value: CARDINAL;  VAR (*OUT*) buffer: ARRAY OF CHAR);
PROCEDURE LongHexToString (value: LONGCARD;  VAR (*OUT*) buffer: EightChar);

    (* Converts 'value' to a string in hexadecimal notation.    *)
```

```
(*************************************************************************)
(*                                MOUSE                                  *)
(*************************************************************************)

TYPE
    MouseType = (NoMouse, INT33, MS, Logitech, PC);

CONST

    (* If you are using a mouse, you have a choice of two mouse         *)
    (* drivers.  Setting MouseKind = INT33 makes the mouse software use *)
    (* INT 33 calls to access a pre-loaded driver.  Otherwise it uses   *)
    (* the SerialMouse driver whose source code is part of PMOS.  (You  *)
    (* might need a bit of trial and error to decide which one works    *)
    (* best for you.)  Set MouseKind = NoMouse if you want mouse        *)
    (* support disabled regardless of whether your hardware supports    *)
    (* it.  Some good reasons for disabling mouse support might be      *)
    (* (a) you can't afford the processor time overhead, or             *)
    (* (b) incompatibilities between your hardware and the PMOS mouse   *)
    (*     support are causing programs to malfunction.                 *)

    MouseKind = NoMouse;

    (* Serial channel: 1 for COM1, 2 for COM2.  This definition is      *)
    (* ignored if MouseKind <= INT33.                                   *)

    MouseChannel = 1;

    (* Don't change the following definitions, several modules use      *)
    (* them to control what other modules they import from.             *)

    UseMouse = (MouseKind <> NoMouse);
    UseSerialDriver = (MouseKind > INT33);

END ConfigurationOptions.
```

ConfigurationOptions

```
DEFINITION MODULE ConfigurationOptions;

        (*********************************************************)
        (*                                                       *)
        (*      The function of this file is to collect          *)
        (*      together some key configuration settings         *)
        (*      which govern things like which PMOS              *)
        (*      features are to be disabled.                     *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        8 March 1994                     *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

(*************************************************************************)
(*                       KERNEL SETTINGS                                 *)
(*************************************************************************)

CONST
    (* The following Boolean constant defines whether the PMOS kernel   *)
    (* will permit round-robin time-slicing among tasks of equal        *)
    (* priority.  For hard real-time applications this should typically *)
    (* be set to FALSE, since time-slicing interferes with the          *)
    (* predictability of execution times.  Disabling time-slicing also  *)
    (* reduces kernel overheads because of the following special        *)
    (* property: with time-slicing disallowed, the only tasks which     *)
    (* participate in task switches and in priority inheritance         *)
    (* calculations are those at the head of each ready queue.  (The    *)
    (* key theoretical result is that there is at most one "active"     *)
    (* task per priority level.)  With time-slicing enabled, any ready  *)
    (* task could be an active task.                                    *)
    (* For applications where keyboard/screen interaction is the        *)
    (* dominant factor, setting TimeSlicingEnabled TRUE sometimes gives *)
    (* a better illusion that multiple tasks are running in parallel.   *)

    TimeSlicingEnabled = FALSE;

    (* The next two constants define the maximum number of concurrent   *)
    (* tasks which are permitted, and the stack size for each task.     *)

    MaxTaskNumber = 31;
    StackSize = 2048;
```

CircularBuffers

```
DEFINITION MODULE CircularBuffers;

        (*****************************************************************)
        (*                                                               *)
        (*      Circular Buffer for passing character data               *)
        (*              between a pair of tasks.                         *)
        (*                                                               *)
        (*      Author:         P. Moylan                                *)
        (*      Last edited:    7 December 1993                          *)
        (*                                                               *)
        (*      Status:         OK.                                      *)
        (*                                                               *)
        (*****************************************************************)

TYPE CircularBuffer;    (* is private *)

PROCEDURE CreateBuffer (VAR (*OUT*) B: CircularBuffer;  size: CARDINAL);

    (* Allocates space for a circular buffer, and initializes it.  The  *)
    (* caller specifies how many characters the buffer will hold.       *)

PROCEDURE PutBuffer (B: CircularBuffer; item: CHAR);

    (* Waits for space available, then puts item at the tail of the queue. *)

PROCEDURE PutBufferImpatient (B: CircularBuffer;  item: CHAR;
                                                TimeLimit: CARDINAL);

    (* Like PutBuffer, but waits no longer than TimeLimit milliseconds  *)
    (* for a buffer slot to become available.  If the time limit        *)
    (* expires, the oldest item in the buffer is overwritten by the     *)
    (* new data.                                                        *)

PROCEDURE GetBuffer (B: CircularBuffer) : CHAR;

    (* Takes one character from the head of the queue, waiting if necessary.*)

PROCEDURE BufferEmpty (B: CircularBuffer): BOOLEAN;

    (* Returns TRUE iff the buffer is empty. *)

END CircularBuffers.
```

Calculator

```
DEFINITION MODULE Calculator;

        (*****************************************************************)
        (*                                                               *)
        (*              Simple calculator.                               *)
        (*                                                               *)
        (*  Programmer:         P. Moylan                                *)
        (*  Last edited:        1 September 1993                         *)
        (*  Status:             OK                                       *)
        (*                                                               *)
        (*****************************************************************)

PROCEDURE RunCalculator;

    (* Displays a calculator window on the screen; this can be operated *)
    (* from the numeric keypad.  On exit, the calculator window is      *)
    (* closed.  However, calculation results are saved between          *)
    (* invocations of this procedure.                                   *)

END Calculator.
```

Bounce

```
DEFINITION MODULE Bounce;

        (*********************************************************)
        (*                                                       *)
        (*              Bouncing ball demonstration.             *)
        (*                                                       *)
        (*  This module is not intended to be of any practical   *)
        (*  use, but it tests the time delay functions of the    *)
        (*  kernel and acts as a demonstration that multiple     *)
        (*  tasks really can coexist.                            *)
        (*                                                       *)
        (*  Programmer:         P. Moylan                        *)
        (*  Last edited:        15 August 1993                   *)
        (*  Status:             OK                               *)
        (*                                                       *)
        (*********************************************************)

FROM Windows IMPORT
    (* type *)  RowRange, ColumnRange;

PROCEDURE Bouncing (top, bottom: RowRange;  left, right: ColumnRange);

    (* Creates a screen window, and runs a bouncing ball demonstration  *)
    (* inside this window.                                              *)

END Bounce.
```

```
(*************************************************************************)
(*                  ANALOGUE INPUT - PERIODIC SAMPLING                   *)
(*************************************************************************)

PROCEDURE StartPeriodicSampling (first, last: InputChannelNumber;
                                 SamplingInterval: LONGCARD;
                                 AmplifierGain: GainCode;
                                 VAR (*OUT*) Buffer: ARRAY OF BYTE);

    (* Initiates a mode of operation in which channels first..last,     *)
    (* inclusive, will be sampled every SamplingInterval microseconds,  *)
    (* with the results stored in array Buffer.  At each sampling time, *)
    (* the specified channels are read as nearly simultaneously as the  *)
    (* hardware will allow.  Procedure WaitForNextSample, below, should *)
    (* be called to check when the data are available in array Buffer.  *)
    (* If WaitForNextSample is not called often enough, there can be a  *)
    (* data overrun in which data are overwritten.  We do not signal    *)
    (* this as an error since the only thing which can be done about it *)
    (* is to use the new data and ignore whatever data have been lost.  *)

    (* WARNING: Although the sampling interval is specified in          *)
    (* microseconds, to allow fine tuning of the interval if desired,   *)
    (* it is in general impractical to specify an interval shorter than *)
    (* about one or two milliseconds, because of software overheads.    *)
    (* The precise limit depends on how much computation is done per    *)
    (* sample, what other high-priority tasks are in the system, and so *)
    (* on.  It is advisable to test the system for evidence of data     *)
    (* overrun, for example by looking at analogue inputs and outputs   *)
    (* with a CRO, and to increase the sampling interval if there       *)
    (* appear to be problems.                                           *)

PROCEDURE WaitForNextSample;

    (* Pauses until the buffer specified in the preceding procedure has *)
    (* been filled with valid data.  Notice that this is essentially a  *)
    (* synchronization procedure; the caller does not have to do any    *)
    (* timing operations since a return from this procedure implies     *)
    (* that the next sample time has arrived.                           *)
    (* WARNINGS:                                                        *)
    (*  1. This procedure should not be called unless periodic sampling *)
    (*     is currently in effect.  Otherwise, it might never return.   *)
    (*  2. Periodic sampling implies that the data buffer is re-filled  *)
    (*     regularly, regardless of whether the user code has finished  *)
    (*     with the previous data.  There are no interlocks, and no     *)
    (*     protection against data being updated just as one is reading *)
    (*     it.  (Any such protection could interfere with the precision *)
    (*     of the timing of sampling external data).  The caller is     *)
    (*     advised to move data out of the buffer promptly, especially  *)
    (*     when the sampling rate is high.                              *)

PROCEDURE StopPeriodicSampling;

    (* Turns off the periodic sampling mode of A/D conversion.  *)

END AnalogueIO.
```

```
TYPE
    GainCode = [0..3];

    (* The gain code for the A/D converter has the interpretation:      *)
    (*          0       gain=1          range  -10 V to   +10 V         *)
    (*          1       gain=10         range   -1 V  to   +1 V         *)
    (*          2       gain=100        range -100 mV to +100 mV        *)
    (*          3       gain=500        range  -20 mV to  +20 mV        *)
    (* The ranges are those applicable when the board is jumpered for   *)
    (* 10V bipolar operation.  The modifications for the unipolar and   *)
    (* 5V bipolar cases should be obvious.                              *)

    OutputChannelNumber = [0..1];
    InputChannelNumber = [0..31];

(************************************************************************)
(*                              DIGITAL I/O                             *)
(************************************************************************)

PROCEDURE DigitalOut (value: BYTE);

    (* Sends the given value to the digital output port of the board.   *)

PROCEDURE DigitalInput (): BYTE;

    (* Reads the digital input port of the board.       *)

(************************************************************************)
(*                    ANALOGUE OUTPUT (RTI-815 ONLY)                    *)
(************************************************************************)

PROCEDURE AnalogueOut (channel: OutputChannelNumber;  value: WORD);

    (* Analogue output.  The channel number should be 0 or 1.  Only the *)
    (* least significant 12 bits of the value are used.  The value can  *)
    (* be treated as either a signed or an unsigned 12-bit number,      *)
    (* depending on hardware jumper selections.                         *)

(************************************************************************)
(*                    ANALOGUE INPUT - SINGLE SAMPLE                    *)
(************************************************************************)

PROCEDURE AnalogueInput (channel: InputChannelNumber; gain: GainCode): WORD;

    (* Analogue input.  The value returned can be a signed or unsigned  *)
    (* number, depending on jumper selections on the board.             *)
    (* This procedure picks up a single sample when called.  It does    *)
    (* not use interrupts or DMA.  It is recommended for use only in    *)
    (* those cases (e.g. isolated sample, or aperiodic sampling) where  *)
    (* the caller takes responsibility for timing.  More commonly, the  *)
    (* procedures in the following section will be more appropriate.    *)
    (* This procedure should NOT be called when periodic sampling has   *)
    (* been activated; the results would be unpredictable.              *)
```

AnalogueIO

```
DEFINITION MODULE AnalogueIO;

        (*******************************************************)
        (*                                                     *)
        (*              Analogue Input and Output.             *)
        (*                                                     *)
        (*  Programmer:         P. Moylan                      *)
        (*  Last edited:        13 February 1991               *)
        (*  Status:             Working, more tests desirable. *)
        (*                                                     *)
        (*******************************************************)

(*************************************************************************)
(*                                                                       *)
(* This module supports the RTI-800/815 Multifunction Input/Output       *)
(* Board.  (RTI is a trademark of Analog Devices).  The board has one    *)
(* 8-bit digital output port, one 8-bit digital input port, and between  *)
(* 8 and 32 channels (depending on hardware options) of analogue input.  *)
(* The RTI-815 version has also two channels of analogue output.  The    *)
(* A/D and D/A converters are 12-bit devices.                            *)
(*                                                                       *)
(* Details such as whether the analogue signals are unipolar or bipolar  *)
(* are controlled by hardware jumpers, and not by software, except for   *)
(* a choice of a 1/10/100/500 gain on analogue input.  It is the user's  *)
(* responsibility to know what this means in terms of actual voltages;   *)
(* the software has no way of knowing which hardware options have been   *)
(* selected.                                                             *)
(*                                                                       *)
(* The board also contains an AM9513A Counter/Timer chip (a product of   *)
(* Advanced Micro Devices Inc.) which supplies 5 16-bit counters along   *)
(* with the support logic to use them in many different ways for         *)
(* various counting and timing applications.  This module reserves the   *)
(* use of counters 4 and 5 for its own internal purposes, as a timing    *)
(* source to control the A/D sampling rate.  (The output of timer 5 is   *)
(* connected, on the RTI-800/815 board, to the gate input of timer 4,    *)
(* and the output of timer 4 triggers an A/D conversion).  The first     *)
(* three timers are not used by this module, and are free to be used     *)
(* for any other purpose which does not interfere with timers 4 and 5.   *)
(* (NOTE: the first three timers are called timers 1-3 in the AM9513A    *)
(* data sheets, but are called timers 0-2 in the RTI-800/815 manual).    *)
(* Because of the great variety of operating options for the timers,     *)
(* and the impossibility of predicting what the user will want to do     *)
(* with them, this module does not attempt to provide any support        *)
(* software for using timers 1-3.                                        *)
(*                                                                       *)
(*************************************************************************)

FROM SYSTEM IMPORT
    (* type *)  BYTE, WORD;
```

level version, which will rarely be needed by the average programmer.

The module called **TerminationControl** provides a means by which any user-written module can declare a "clean-up" routine to be invoked when the program terminates (either normally, or on an error). Multiple calls to this module are possible, so it is quite possible for each module to have its own separate termination procedure. If a termination handler itself installs another termination handler, this forces an extra pass through the termination handling.

Module **Bounce** is for demonstration and testing: it shows a text-mode bouncing ball on the screen.

The "module" called **Skeleton** is not in fact a module. It is simply a file which may be copied to simplify the typing effort when creating a new module.

16. Disclaimer

The PMOS system is an experimental package which is under active development, and there is no guarantee of upward compatibility. The information herein is accurate at the time of writing, but changes are likely to occur in future versions. If you run into version compatibility problems, check the source file of the current version in the PMOS library.

While all care has been taken to avoid software errors, no responsibility can be accepted for reliability of the software.

It would be appreciated if any errors found could be reported to the author of this document.

17. Distribution information

PMOS is available for anonymous ftp at address ee.newcastle.edu.au, in directory pub/PMOS. For a list of alternative sites, contact the author.

PMOS is free for non-profit use. Further information may be found in the README file which is distributed with the software.

be a new file or an existing file. Unlike MS-DOS, PMOS does not allow you to open a new file with the same name as an existing file. This helps to avoid accidental file deletions.

The modules called **Directories**, **FileNames**, **IOErrorCodes**, and **Devices** are parts of the file system which are not normally called by the end user.

When using the Files module to access disk files, you must explicitly import modules **HardDisk** and/or **Floppy** (and the import declarations should come before the importation of Files), even though you will not explicitly call procedures in those modules. The file system does not have direct access to the device drivers; it simply makes use of whatever device drivers have (in their initialisation code) called module Devices to declare their presence.

Although the PMOS file system works well with floppy disks, it does not always handle hard disk files satisfactorily. The problem areas are:

(a) The HardDisk driver does not support all known disk interfaces. It does not handle SCSI disks, for example.

(b) OS/2 will not permit hard disk operations, so under OS/2 the HardDisk module acts as if there is no hard disk present.

(c) At program termination, MS-DOS does not detect the fact that it is holding an obsolete copy of the directories and file allocation table, so it will not see newly created files – and it will attempt to write over them – unless the machine is rebooted.

To circumvent these problems, an alternative module **FileSys** provides the same features as the Files module, but it implements them via system calls rather than by dealing directly with the device drivers. This loses some of the advantages of multitasking, but provides greater program portability.

13. Other device drivers

Module **Printer** supports a dot matrix printer connected to the parallel port. At present, this module is not part of the file system, so the PrintChar procedure must be called directly.

For I/O through the serial ports, use module **SerialIO**.

Module **AnalogueIO** provides an interface to analogue-to-digital hardware. Normally the A/D software will run in a "periodic sampling" mode, which is explained in the comments in the definition module. Note that the support is for one specific type of A/D interface board, and will have to be rewritten if the hardware is changed.

Due to the special nature of A/D operations, there is no intention to support analogue operations within the file system.

14. Writing new device drivers

To add a new device driver to the system, it is necessary to understand how to use procedures CreateInterruptTask and WaitForInterrupt in module TaskControl. It will probably also be necessary to use the procedures in modules **LowLevel, DMA**, and **MiscPMOS**. In this connection, it should be noted that the ROM routines in the computer's BIOS (Basic Input/Output System) are sometimes incompatible with multitasking and may have to be avoided. It is advisable to study the listings of existing device drivers, to see what techniques are needed.

To incorporate a new device driver into the file system, use module **Devices**. The file system does not in itself have access to any device driver, but indirectly it can access any driver which makes itself known to module Devices.

15. Other modules

Module **Trace** is an aid to program testing and debugging; its use will be obvious after reading the module listing. Module **KTrace** is a lower-

There is also a module called **UserInterface** which lets you use a mouse to move windows around on the screen.

9. Screen Editing

The design and implementation of operator interfaces can be a tedious and time-consuming part of software development, so PMOS contains some features to ease this problem. Module **Menus** displays user-defined menus on the screen, and looks after displaying a menu, scrolling when necessary, handling the keystrokes by which the keyboard user selects an option, and returning the result to the caller. Module **ScreenEditor** does a similar job where a display of variable values, of a mixture of types, must be created with an option for the keyboard user to modify some or all of the values. For more complex applications, it might be necessary to use the procedures in modules **FieldEditor** and **ListEditor**. The procedures in module **RowEditor**, which works in collaboration with this group, will not normally be needed to be called directly.

There is a module **Calculator** which, when invoked, displays a calculator window on the screen and allows the user to perform simple calculations. (The calculator procedure returns when the user types the Esc key or clicks on the "close" button with the mouse.) Use this when you want the user to be able to drop out of your program temporarily to perform some calculations.

10. Miscellaneous utilities

The module called **LowLevel** provides a variety of low-level functions: bitwise logic operations, shifts and rotations, pointer decomposition, and the like. More low-level procedures can be found in **MiscPMOS**.

Random numbers are available from modules **RandCard** (for a cardinal-valued result) and **Random** (for a real-valued result).

To implement queues, see modules **CircularBuffers**, **Queues**, and **LossyQueues**. Module Queues handles the most general case, but CircularBuffers can be more efficient when it is applicable. LossyQueues is for the situation, very common in real-time applications, where it is better to lose data than to have to wait for buffer space to become available.

Queue-handling operations are also supplied in **Mailboxes**. This module is intended as a convenient mechanism for message-passing between tasks.

Module **Conversions** deals with things like string-to-numeric conversions.

A date and time procedure may be found in module **TimeOfDay**.

11. Sound output

Module **SoundEffects** deals, at a low level, with output to the speaker. You can, for example, call procedure Beep to produce an audible alarm. For more ambitious sound effects, see modules **Music** and **Piano**.

The module called **MusicDemonstration** has as its only function the playing of a demonstration piece of music. If you import it into your program, the music will play in parallel with whatever your program is doing.

Sorting is provided in **QuickSortModule** and **FileSort**. QuickSortModule supplies a procedure for in-memory sorting, and FileSort does in-place sorting of a file.

12. File I/O

The standard operations of opening and closing a file, and of reading and writing, are in module **Files**.

Note: in many operating systems, you have to specify when opening a file whether you are opening it for reading or for writing. PMOS takes a different approach: you can mix read and write operations, but you have to specify when opening a file whether it is supposed to

lowest level, module **TextVideo** performs some of the most primitive hardware operations.

For numeric input and output, use modules **NumericIO** and **RealIO**. The module called **Conversions** might also be of use here.

It is sometimes useful to have a program produce diagnostic information which does not normally appear on the screen, but which can be viewed by the user on demand. This facility is provided by a module called **MaintenancePages**, which demonstrates one way to use multiple virtual screens. Similar applications can be designed around the features provided by **MultiScreen**.

For graphical rather than text modes, use module **GWindows**. Utility procedures to do graph-plotting may be found in module **WGraph**. The low-level graphics functions are in the modules called **Screen** and **Graphics**. There is also a module called **ScreenGeometry** whose primary function is to define data types "Point" and "Rectangle".

Module **Tiles**, which looks after the issues associated with overlapping graphics windows, is not intended to be called by the end user.

7. Keyboard Input

In "normal" applications, the best keyboard input routines are those, such as ReadChar, supplied in module Windows. However, module **Keyboard** provides a way to read a character without going through the Windows module. The essential difference is this: the input procedures in Windows prompt the user, e.g. with a blinking cursor, and may echo the input on the screen if specified. By going directly to module Keyboard, you avoid the prompt. There are some cases where this is a more desirable behaviour.

Both Windows and Keyboard provide a method for reading a character without consuming it – a highly desirable facility, since there are many applications where one has to overshoot by one character before detecting the end of an input string. The Keyboard module does this by providing an "un-read" operation called PutBack. The solution in Windows is slightly different, see function LookaheadCharacter. The reason for using different approaches in the two modules is somewhat technical and not easily explained in an overview such as this, but you will soon find that the conventions are well adapted to a "natural" style of programming.

The keyboard is actually handled by two modules. Module Keyboard, already mentioned, provides the higher-level interface which most users will find appropriate. Those who need more detailed control over keyboard operations can use the lower-level module **KBdriver**, which allows access to such information as whether several keys have been pressed simultaneously.

8. Dealing with a mouse

The module called **Mouse** provides functions for things like getting the current mouse position, and it also provides for installing a user-supplied procedure to be called on any of a defined set of mouse events.

Internally, this module does its job by importing from either **SerialMouse** or **Mouse33**. The reason for having two distinct mouse drivers is that not all mice are the same, and there are situations where one driver will work and the other will not. Mouse33 works by using INT 33H calls to a pre-loaded driver, so this is potentially the more portable version; but the vendor-supplied mouse driver for at least one popular mouse is incompatible with PMOS because of the way it takes over the timer interrupt. Module SerialMouse talks directly to the mouse via a serial port; this makes it less portable in principle, but it does at least bypass the shortcomings of vendor-supplied drivers.

To specify which mouse driver to use, edit the definition module for the module called **ConfigurationOptions**.

```
Obtain (L);
protected section;
Release (L);
```

to ensure that, while the statements in the protected section are being executed, no other task can gain entry to any section of code which is also protected by the Lock called L. This is a little more efficient than using a semaphore, but is more restricted: a Lock may only be used for critical section protection (unlike a general semaphore, which may be used for things like intertask synchronisation); and the Obtain/Release pairs must be used in a properly nested fashion.

In addition, any section protected by a Lock must be free of any operations – except for a nested Obtain/Release on another Lock – which could block the task. That is, it is not legal to have a Sleep or a semaphore Wait inside such a protected section. (Watch out in particular for procedure calls, where the called procedure might perform one of the forbidden operations.) If this condition cannot be satisfied, then you must use a semaphore rather than a Lock to protect your critical section.

A special feature of the Obtain/Release mechanism is that it provides for *priority inheritance*. If a high-priority task is blocked while trying to obtain a Lock, then the holder of that Lock is promoted in priority, to ensure that it quickly reaches the point where it releases the Lock. In the case where a task holds several Locks, its priority becomes the maximum of the priorities of the tasks it is blocking. This promotion is temporary; the task's priority is dropped again when it is no longer blocking a higher-priority task.

5. Timed Operations

Most of the work done by module **Timer** is invisible to the user, since the main function of this module is to look after internal system functions such as time-slicing. There are, however, two useful user-callable procedures. Procedure Sleep provides a timed delay, putting the caller to sleep (and making the processor available to other tasks) for a specified number of milliseconds. Procedure TimedWait adds a time-out facility to semaphore operations, to allow for recovery in the case that an expected event never happened.

Inside the implementation module, there is a constant MillisecondsPerTick which controls the frequency of timer interrupts. Its value can be decreased if it is necessary to increase the precision of timed operations. You are warned, however, that decreasing the value will increase the time spent by the processor on system overheads.

To avoid confusion, it should be mentioned that the hardware provides more than one timer. Module Timer uses one specific hardware timer which simply interrupts at a fixed frequency. There is a separate time-of-day clock, which keeps track of the date and time even when the computer is powered down, and that is looked after by module **TimeOfDay**. (A module called **Directories** uses the time-of-day clock to know when a file is created; and the time-of-day clock is also useful in software testing, when you want to know how long a program section takes to execute. At present, there is no facility for timing one specific task.) Depending on the hardware configuration, there may be other timers in the system which are used for more specialised purposes – see, for example, module AnalogueIO – but these are of no concern to the Timer module.

6. Screen Output

In the majority of applications, the best procedures to use for screen output are those in module **Windows**. These procedures provide for character and string output and input, and a variety of cursor-positioning and similar operations.

At a lower level, module **GlassTTY** provides low-overhead but very crude output procedures. It is used by some system procedures for error message output, but is probably unsuitable for other applications. At the very

eral user. In this section, we focus attention on the procedures which are intended to be user-callable.

When your program first starts running, there are just two tasks in the system: the task which is executing your program, and a "null task" which does nothing and whose only function is to give the processor something to do while no other task is runnable. Your program may then create a new parallel thread of execution by calling procedure CreateTask, in module TaskControl. (Meanwhile, PMOS has itself created a few more tasks, to look after device drivers and the like.) Further tasks may then be created, either by the original task or by any of the newly created tasks.

When you call CreateTask, you give a procedure name, which becomes the starting address for the new task; a priority; and an abbreviated task name which is used only for testing. As a general guideline priority levels in the range 1 to 7 should be used for user-level tasks, and levels 8 to 15 should be reserved for system-level tasks such as device drivers. Typically most tasks should have priority 1 (the lowest level). There may occasionally be a good reason for giving a high priority to a user-defined task of exceptional urgency, but this facility should be used with care. A careless use of high priorities can make your software run inefficiently, by interfering with the ability of the system to do work on your behalf.

It is important to understand the distinction between a *task* and a *procedure*. A task is the actual thread of execution, which may pass through many procedures during its lifetime. More importantly, it is quite possible for several tasks to be executing the same procedure simultaneously. In this connection, it should be noted that the variables declared inside a procedure belong to an individual task, in the sense that a separate copy of those variables is created for each caller of the procedure. The variables declared at module level, on the other hand, are shared by all tasks which enter that module. Thus, you should avoid declaring global variables unless you intend to make them shared variables, in which case you must of course arrange for suitable critical section protection when accessing those variables.

A task normally terminates by reaching the last statement in the procedure in which it started execution. If you want the task to terminate before that, you can call procedure TaskExit in module TaskControl.

When the "main program" part of your Modula-2 program completes its execution, the entire program terminates, even if there are tasks still in execution. This might or might not correspond to what you want; so it may be necessary to insert tests at the end of the main program which wait until all subsidiary tasks have completed their work.

3. Semaphores

Module **Semaphores** contains procedures CreateSemaphore, Wait, and Signal, whose use will be obvious to anyone familiar with semaphores. Those who are unfamiliar with the concept should consult an operating systems textbook, since an adequate description would take more space than is available here.

Given that semaphores are used for inter-task communication and protection, some programmers become confused about who should be the "owner" of a semaphore, and how a semaphore created by one task can be passed to another. In fact, this is the wrong way to look at the idea of ownership. It makes more sense to say that a semaphore belongs to a module (not to a task), that it is created in the initialisation code of that module, and that it is then available for use by tasks which call procedures in that module.

Procedure TimedWaitT in module Semaphores is not intended for general use. A better alternative exists in module Timer.

4. Locks and priority inheritance

Module TaskControl also defines a data type *Lock*. A Lock is like a binary semaphore, and can be used for protecting critical sections. That is, you can use code of the form

PMOS Reference Manual

P.J. MOYLAN

Department of Electrical and Computer Engineering
The University of Newcastle
NSW 2308, Australia

peter@ee.newcastle.edu.au

1. Introduction

This is a new edition of the manual previously published as "The PMOS Definition Modules", Technical Report EE9107, which described PMOS version 1. There have been some significant changes to PMOS from version 1 to version 2.

The PMOS software system is a set of modules, written in Modula-2, for use in real-time applications such as control systems. At the user level, it does not appear as an operating system in the conventional sense of having a command interpreter and facilities for running multiple independent programs in parallel. Rather, it is a set of library routines which can be incorporated into an applications program to allow for multitasking within that program.

PMOS is designed to run on IBM-PC/AT and similar personal computers. One would normally use the computer's original operating system, usually MS-DOS, while performing the program development task of editing, compiling, and linking. Then, when the applications program is run, PMOS takes over the computer by substituting its own device drivers, etc., for those which were originally present. The keyboard controller is designed in such a way that the Ctrl/Alt/Del combination aborts the program and returns control to the original operating system. (Incidentally, a module called TerminationControl is provided to allow you to provide for graceful program termination, for cases where abrupt program abortion could be disastrous.)

When PMOS is run under OS/2, it runs in a virtual DOS session rather than taking over the whole machine. (A complete takeover is neither sensible nor feasible without first shutting down OS/2.) In that case the Ctrl/Alt/Ins keyboard combination should be used, instead of Ctrl/Alt/Del, to terminate the program. When running under OS/2 there are also some restrictions on what the PMOS device drivers can do. In particular the hard disk driver will not work − it will act as if there were no hard disk available − although the floppy disk driver does work.

The purpose of the present document is to act as a reference manual, by listing all of the procedures within PMOS which may be called. A complete source code listing would be unreasonably long, so only the definition modules of PMOS are listed. This is, in principle, all that a user needs to know about the PMOS code.

For ease of reference, the listings are arranged alphabetically by module name. The index gives an alternative ordering, by listing procedure names rather than module names. Finally, the discussion in the following sections is organised by functional groupings.

2. Multiprogramming features

The kernel of the PMOS system is contained in modules **InnerKernel**, **TaskControl**, **Semaphores**, and **Timer**. Many of the procedures in those modules may be considered as internal system details which are irrelevant to the gen-

List of Modules

CENTRE FOR INDUSTRIAL CONTROL SCIENCE

Department of Electrical and Computer Engineering

University of Newcastle
N.S.W. 2308, Australia
Tel. (049) 21 6091 Fax (049) 60 1712

PMOS REFERENCE MANUAL

Version 2

P.J. MOYLAN

Technical Report EE9409
March 1994