

Pascal listing: MOUSEP.PAS

```
{*****}
{*               M O U S E P . P A S               *}
{*-----*}
{*   Task           : Demonstrates the different functions available*}
{*                   in mouse programming.                         *}
{*-----*}
{*   Author          : Michael Tischer                           *}
{*   Developed on     : 04/21/89                                   *}
{*   Last update      : 04/07/95                                   *}
{*****}
```

```
uses Dos;                                { Add DOS unit }
```

```
{ $L mousepa }                          { Link assembler module }
                                           { Adjust path to your system needs }
{== Declaration of external functions ==}
```

```
{ $F+ }                                { FAR function }
procedure AssmHand; external ;           { Assembler event handler }
{ $F- }                                { FAR functions no longer accessible }
```

```
{== Constants ==}
```

```
const
```

```
{-- Event codes -----}
```

```
EV_MOU_MOVE      = 1;                    { Mouse movement }
EV_LEFT_PRESS    = 2;                    { Left mouse button pressed }
EV_LEFT_REL      = 4;                    { Left mouse button released }
EV_RIGHT_PRESS   = 8;                    { Right mouse button pressed }
```

```

EV_RIGHT_REL    = 16;           { Right mouse button released }
EV_MOU_ALL      = 31;           { All mouse events }

LBITS           = 6;           { EV_LEFT_PRESS or EV_LEFT_REL }
RBITS           = 24;           { EV_RIGHT_PRESS or EV_RIGHT_REL }

NO_RANGE        = 255;         { Mouse cursor not in xy range }

PtrSameChar     = $00ff;        { Same character }
PtrSameCol      = $00ff;        { Same color }
PtrInvCol       = $7777;        { Inverse color }
PtrSameColB     = $807f;        { Same color, blinking }
PtrInvColB      = $F777;        { Inverse color, blinking }

EAND             = 0;           { Event comparisons for MouEventWait }
EVOR             = 1;

CRLF             = #13#10;      { CR/LF }

```

{== Type declarations =====}

```

type FNCTPTR    = longint;      { Address of a FAR function }
PTRVIEW        = longint;      { Mask for mouse cursor }
RANGE = record                { Describes a mouse range }
    x1,          { Upper left and lower }
    y1,          { right coordinates for }
    x2,          { the specified range }
    y2          : byte;
    PtrMask      : PTRVIEW;     { Mask for mouse cursor }
end;
RNGARRAY = array [0..100] of RANGE;
RNGPTR    = ^RNGARRAY;

```

```

PTRREC    = record                                { Allows access to any }
            Ofs : word;                            { mouse cursor record }
            Seg : word;                            { existing }
        end;

PTRVREC    = record                                { Allows access to }
            ScreenMask : word;                      { PTRVIEW }
            CursorMask : word;
        end;

RNGBUF     = array [0..10000] of byte;              { Range buffer }
BBPTR      = ^RNGBUF;                             { Pointer to a range buffer }

{== Global variables =====}

var  NumRanges,                                { Number of ranges }
    TLine,                                    { Number of text lines }
    TCol      : byte;                        { Number of text columns }
    MouAvail  : boolean;                    { TRUE if mouse is available }
    OldPtr,   { Old mouse cursor appearances }
    StdPtr    : PTRVIEW;                   { Mask for default mouse cursor }
    BufPtr    : BBPTR;                     { Pointer to range recognition buffer }
    ActRngPtr : RNGPTR;                    { Pointer to current range vector }
    BLen      : integer;                   { Range buffer length in bytes }
    ExitOld   : pointer;                   { Pointer to old exit procedure }

{-- Variables which are loaded into mouse handler on every call -----}

    MouRng,                                { Current mouse range }
    MouCol,                                { Mouse column (text screen) }
    MouRow    : byte;                      { Mouse line (text screen) }
    MouEvent  : integer;                   { Event mask }

{-- Variables which load with any occurrence of expected events -----}

```

```

EvRng,                { Range in which the mouse can be found }
EvCol,                { Mouse column }
EvRow : byte;        { Mouse line }

```

```

{*****}
* MouPtrMask: Executes cursor mask and screen mask from a bitmap *
*               containing character and color.                   *
**-----**
* Input  : Chars = Bitmask of character as found in cursor mask *
*               and screen mask                                  *
*          Color  = Bitmask of character color as found in      *
*               cursor mask and screen mask                     *
* Output : Cursor mask and screen mask as a value of typ PtrView *
* Info:   The constants PtrSameChar, PtrSameCol, PtrSameColB,   *
*          PtrInvCol, PtrInvColB, and the results of the PtrDifChar*
*          and PtrDifCol functions also control character & color *
{*****}

```

```
function MouPtrMask( Chars, Color : word ) : PTRVIEW;
```

```
var Mask : PTRVIEW;      { For creating cursor mask and screen mask }
```

```
begin
```

```
    PTRVREC( Mask ).ScreenMask := ( ( Color and $ff ) shl 8 ) +
                                   ( Chars and $ff );
```

```
    PTRVREC( Mask ).CursorMask := ( Color and $ff00 ) + ( Chars shr 8 );
```

```
    MouPtrMask := Mask;      { Return mask to caller }
```

```
end;
```

```

{*****}
* PtrDifChar: Defines character structure of cursor and screen *
{*****}

```

```

{ *                               masks in conjunction with character.                               * }
{ **-----** }
{ *   Input   : ASCII code of the character on which cursor is based   * }
{ *   Output  : Cursor and screen masks for this cursor               * }
{ *   Info:    Function result should be computed with the help of the * }
{ *           MouPtrMask function                                     * }
{ ***** }

```

```
function PtrDifChar( Chars : byte ) : word;
```

```
begin
```

```
    PtrDifChar := Chars shl 8;
```

```
end;
```

```

{ ***** }
{ *   PtrDifCol: Creates the character segment of the cursor and screen* }
{ *                               masks in conjunction with the mouse cursor color.                               * }
{ **-----** }
{ *   Input   : Character color on which the mouse cursor will be based * }
{ *   Output  : cursor and screen masks for this color                 * }
{ *   Info:    The function's result should be computed with the help   * }
{ *           of the MouPtrMask function                               * }
{ ***** }

```

```
function PtrDifCol( Color : byte ) : word;
```

```
begin
```

```
    PtrDifCol := Color shl 8;
```

```
end;
```

```

{ ***** }
{ *   MouDefinePtr: Assigns the mouse driver the cursor mask and       * }

```

```

{ *                               screen mask, from which the driver can create the * }
{ *                               mouse cursor.                               * }
{ **-----** }
{ *   Input   : Mask = The cursor and screen mask as a parameter of         * }
{ *                               type PTRVIEW                               * }
{ *   Info:    - The mask parameter should be created with the help of      * }
{ *                               the MouPtrMask function                    * }
{ *            - Most significant 16 bits represent the screen mask,        * }
{ *            least significant 16 bits represent cursor mask              * }
{ ***** }

```

```

procedure MouDefinePtr( Mask : PTRVIEW );

```

```

var Regs : Registers;           { Processor registers for interrupt call }

```

```

begin
  if OldPtr <> Mask then          { Mask change since last call? }
  begin                          { Yes }
    Regs.AX := $000a;            { Funct. no.: Set text cursor type }
    Regs.BX := 0;                { Create software cursor }
    Regs.CX := PTRVREC( Mask ).ScreenMask; { Low word is AND mask }
    Regs.DX := PTRVREC( Mask ).CursorMask; { High word is XOR mask }
    Intr( $33, Regs);           { Call mouse driver }
    OldPtr := Mask;             { Reserve new bitmask }
  end;
end;

```

```

{ ***** }
{ *   MouEventHandler: Called by the assembler routine AssmHand as soon* }
{ *                               as a mouse event occurs.                * }
{ **-----** }
{ *   Input   : EvFlags = The event mask                                * }

```

```

{ *           ButState = Current mouse button status           * }
{ *           X, Y      = Current coordinates of the mouse cursor on * }
{ *                   the text screen                             * }
{ ***** }

```

```

procedure MouEventHandler( EvFlags, ButState, x, y : integer );

```

```

var NewRng : byte;                                     { Number of new range }

```

```

begin

```

```

    MouEvent := MouEvent and not(1);                    { Bit 0 excluded }
    MouEvent := MouEvent or ( EvFlags and 1 );          { Bit 0 copied }

```

```

    if ( EvFlags and LBITS ) <> 0 then{ Lft button released or pressed? }
    begin                               { Yes }
        MouEvent := MouEvent and not( LBITS ); { Remove previous status }
        MouEvent := MouEvent or ( EvFlags and LBITS ); { Add status }
    end;

```

```

    if ( EvFlags and RBITS ) <> 0 then{ Rgt button released or pressed? }
    begin                               { Yes }
        MouEvent := MouEvent and not( RBITS ); { Remove previous status }
        MouEvent := MouEvent or ( EvFlags and RBITS ); { Add status }
    end;

```

```

    MouCol := x;                                     { Convert columns to text columns }
    MouRow := y;                                     { Convert lines to text lines }

```

```

{-- Determine range in which the mouse should be found and -----}
{-- determine whether range has changes since the previous call ----}
{-- of the handler. If so, the cursor image must be redefined. ----}

```

```

NewRng := BufPtr^[ MouRow * TCol + MouCol ];           { Get range }
if NewRng <> MouRng then                                { New range? }
  begin                                                  { Yes }
    if NewRng = NO_RANGE then                            { Outside of a range? }
      MouDefinePtr( StdPtr )                            { Yes --> Default cursor }
    else                                                { No --> Range recognized }
      MouDefinePtr( ActRngPtr^[ NewRng ].PtrMask );
    end;
  MouRng := NewRng;                                     { Reserve range number in global variable }
end;

```

```

{*****}
{ *   MouIBufFill: Store the code for a mouse range within the   * }
{ *                               modular range memory.           * }
{ **-----** }
{ *   Input   : x1, y1 = Upper left corner of the mouse range   * }
{ *             x2, y2 = Lower right corner of the mouse range  * }
{ *             Code   = Range code                             * }
{*****}

```

```

procedure MouIBufFill( x1, y1, x2, y2, Code : byte );

```

```

var Index      : integer;                                { Points to array }
    Column,    : integer;                                { Loop counter }
    Line       : byte;

begin
  for Line:=y1 to y2 do                                  { Count individual lines }
    begin
      Index := Line * TCol + x1;                        { First line index }
      for Column:=x1 to x2 do { Go through the columns in this line }
        begin

```



```

        BufPtr^[ Index ] := Code;                                { Save code }
        inc( Index );                                           { Set index to next array }
    end;
end;
end;

```

```

{*****}
*  MouDefRange:  Allows the registration of different screen  *
*               ranges, which the mouse recognizes as different *
*               ranges. The mouse cursor's appearance changes  *
*               when it senses each range.                      *
**-----**
*  Input   : Number = Number of screen ranges                  *
*           BPtr   = Pointer to the array in which the individual *
*               ranges are written as a structure of type      *
*               RANGE                                           *
*  Info:    - The free areas of the screen are assigned the code *
*           NO_RANGE                                           *
*           - When the mouse cursor enters one of the ranges,  *
*               the mouse range calls the event handler        *
{*****}

```

```

procedure MouDefRange( Number : byte; BPtr : RNGPTR );

```

```

var ActRng,                               { Number of the current range }
    Range : byte;                         { Loop counter }

```

```

begin
    ActRngPtr := BPtr;                    { Reserve pointer to vector }
    NumRanges := Number;                  { and number of ranges }
    FillChar( BufPtr^, BLen, NO_RANGE ); { All elements=NO_RANGE }
    for Range:=0 to Number-1 do           { Check out different ranges }

```

```

with BPtr^[ Range ] do
  MouIBufFill( x1, y1, x2, y2, Range );

{-- Redefine mouse cursor -----}
ActRng := BufPtr^[ MouRow * TCol + MouCol ];           { Get range }
if ActRng = NO_RANGE then                               { Outside a range? }
  MouDefinePtr( StdPtr )                               { Yes --> Default cursor }
else                                                     { No --> Range recognized }
  MouDefinePtr( BPtr^[ ActRng ].PtrMask );
end;

```

```

{*****}
*  MouEventWait: Waits for a specific mouse event.  *
**-----**
*  Input   : TYP           = Type of comparison between different events*
*            WAIT_EVENT = Bitmask which specifies the awaited event  *
*  Output  : Bitmask of the occurring event                          *
*  Info:    - WAIT_EVENT can be used in conjunction with OR for      *
*            other constants like EV_MOU_MOVE, EV_LEFT_PRESS etc.    *
*            - Comparison types can be given as AND or OR. If AND is *
*            selected, the function returns to the caller if all      *
*            anticipated events occur. OR returns the function to    *
*            the caller if at least one of the events occurs.        *
{*****}

```

```

function MouEventWait( Typ : BYTE; WaitEvent : integer ) : integer;

```

```

var ActEvent : integer;
    Line,
    Column   : byte;
    CEnd     : boolean;

```

```

begin
  Column := MouCol;           { Reserve current mouse position }
  Line := MouRow;
  CEnd := false;

repeat
  {-- Wait for one of the events to occur -----}

  if Typ = EAND then          { AND comparison? }
    repeat                    { Yes --> All events must occur }
      ActEvent := MouEvent;   { Get current event }
    until ActEvent = WaitEvent
  else                          { OR comparison }
    repeat                    { At least one event must occur }
      ActEvent := MouEvent;   { Get current event }
    until ( ActEvent and WaitEvent ) <> 0;

  ActEvent := ActEvent and WaitEvent;      { Check event bits only }

  {-- While waiting for mouse movement, the event is accepted -- }
  {-- only if the mouse cursor moves to another line and/or -- }
  {-- column in the text screen - }

  if ( ( WaitEvent and EV_MOU_MOVE ) <> 0 ) and
    ( Column = MouCol ) and ( Line = MouRow ) ) then
    begin
      { Mouse moved, but still at the same screen position }
      ActEvent := ActEvent and not( EV_MOU_MOVE ); { Move bit out }
      CEnd := ( ActEvent <> 0 ); { Still waiting for events? }
    end
  else
    { Event occurs }
    CEnd := TRUE;
until CEnd;

```

```

EvCol := MouCol;           { Determine current mouse }
EvRow := MouRow;           { position and range in   }
EvRng := MouRng;           { global variables       }

MouEventWait := ActEvent;

end;

{*****}
{ * MouISetEventHandler: Installs an event handler which is called * }
{ * when a particular mouse event occurs. * }
{**-----**}
{ * Input : EVENT = Bitmask which describes the event, called * }
{ * through an event handler * }
{ * FPTR = Pointer to the event handler of type FNCTPTR * }
{ * Info: - EVENT can be used through OR comparisons in conjunc- * }
{ * tion with constants like EV_MOU_MOVE, etc. * }
{ * - The event handler must be a FAR procedure, and change * }
{ * none of the given processor registers * }
{*****}

procedure MouISetEventHandler( Event : integer; FPtr : FNCTPTR );

var Regs : Registers;      { Processor registers for interrupt call }

begin
  Regs.AX := $000C;         { Funct. no.: Set Mouse Handler }
  Regs.CX := event;         { Load event mask }
  Regs.DX := PTRREC( FPtr ).Ofs; { Offset address of handler }
  Regs.ES := PTRREC( FPtr ).Seg; { Segment address of handler }
  Intr( $33, Regs );        { Call mouse driver }
end;

```

```

{*****}
{ *   MouIGetX: Returns the text column in which the mouse cursor can   * }
{ *                               be found                               * }
{ **-----** }
{ *   Output : Mouse column converted to text screen                     * }
{*****}

```

```
function MouIGetX : byte;
```

```
var Regs : Registers;          { Processor registers for interrupt call }
```

```
begin
```

```
    Regs.AX := $0003;           { Funct. no.: Get mouse position }
```

```
    Intr( $33, Regs );         { Call mouse driver }
```

```
    MouIGetX := Regs.CX shr 3;  { Convert column and return new value }
```

```
end;
```

```

{*****}
{ *   MouIGetY: Returns the text line in which the mouse cursor can     * }
{ *                               be found.                               * }
{ **-----** }
{ *   Output : Mouse line converted to text screen                     * }
{*****}

```

```
function MouIGetY : byte;
```

```
var Regs : Registers;          { Processor registers for interrupt call }
```

```
begin
```

```
    Regs.AX := $0003;           { Funct. no.: Get mouse position }
```

```
    Intr( $33, Regs );         { Call mouse driver }
```

```

    MouIGetY := Regs.DX shr 3;          { Convert line and return new value }
end;

```

```

{ ***** }
{ *   MouShowMouse: Show mouse cursor on the screen.   * }
{ **-----** }
{ *   Info: Calls between MouShowMouse and MouHideMouse must be evenly * }
{ *           balanced                                     * }
{ ***** }

```

```

procedure MouShowMouse;

```

```

var Regs : Registers;                { Processor regs for interrupt call }

```

```

begin
    Regs.AX := $0001;                { Funct. no. for "Show Mouse" }
    Intr( $33, Regs );                { Call mouse driver }
end;

```

```

{ ***** }
{ *   MouHideMouse: Hide mouse cursor from the screen   * }
{ **-----** }
{ *   Info: Calls between MouShowMouse and MouHideMouse must be evenly * }
{ *           balanced                                     * }
{ ***** }

```

```

procedure MouHideMouse;

```

```

var Regs : Registers;                { Processor regs for interrupt call }

```

```

begin
    Regs.AX := $0002;                { Funct. no. for "Hide Mouse" }

```

```

    Intr( $33, Regs);                { Call mouse driver }
end;

```

```

{*****}
*  MouSetMoveArea: Specify movement range for mouse cursor.          *
**-----**
*  Input   :  x1, y1 = Coordinates of range's upper left corner      *
*            x2, y2 = Coordinates of range's lower right corner      *
*  Info:    - The coordinates indicate the text screen coordinates,  *
*            and not the virtual graphic screen used by the mouse    *
*            driver                                                    *
{*****}

```

```

procedure MouSetMoveArea( x1, y1, x2, y2 : byte );

```

```

var Regs : Registers;                { Processor regs for interrupt call }

```

```

begin

```

```

    Regs.AX := $0008;                { Funct. no. for "Set vertical limits" }
    Regs.CX := integer( y1 ) shl 3;  { Conversion to virtual          }
    Regs.DX := integer( y2 ) shl 3;  { mouse screen                    }
    Intr( $33, Regs );                { Call mouse driver          }
    Regs.AX := $0007;                { Funct. no. for "Set horizontal limits" }
    Regs.CX := integer( x1 ) shl 3;  { Conversion to virtual          }
    Regs.DX := integer( x2 ) shl 3;  { mouse screen                    }
    Intr( $33, Regs );                { Call mouse driver          }
end;

```

```

{*****}
*  MouSetSpeed: Configures movement speed of mouse cursor          *
**-----**
*  Input   : XSpeed = Speed in X-direction                          *
{*****}

```

```

{*          YSpeed = Speed in Y-direction                      *}
{*  Info:      - Parameters are measured in units of          *}
{*             mickeys (8 per pixel)                          *}
{*****}

```

```

procedure MouSetSpeed( XSpeed, YSpeed : integer );

```

```

var Regs : Registers;           { Processor regs for interrupt call }

```

```

begin
  Regs.AX := $000f;           { Funct. no. for "Set mickeys to pixel ratio" }
  Regs.CX := XSpeed;
  Regs.DX := YSpeed;
  Intr( $33, Regs);           { Call mouse driver }
end;

```

```

{*****}
{*  MouMovePtr: Moves mouse cursor to a specific position on the *}
{*              screen                                           *}
{**-----**}
{*  Input   : COL = New screen column for mouse cursor          *}
{*              ROW = New screen line for mouse cursor          *}
{*  Info:    - The coordinates indicate the text screen, and not the *}
{*              virtual graphic screen used by the mouse driver  *}
{*****}

```

```

procedure MouMovePtr( Col, Row : byte );

```

```

var Regs      : Registers;           { Processor regs for interrupt call }
    NewRng    : byte;               { Range into which the mouse is moved }

```

```

begin

```



```

Regs.AX := $0004;      { Funct. no. for "Set mouse cursor position" }
MouCol := col;          { Store coordinates in }
MouRow := row;          { global variables }
Regs.CX := integer( col ) shl 3;  { Convert coordinates and store }
Regs.DX := integer( row ) shl 3;  { in global variables }
Intr( $33, Regs );      { Call mouse driver }

NewRng := BufPtr^[ Row * TCol + Col ];      { Get range }
if NewRng <> MouRng then                      { New range? }
begin                                        { YES }
    if NewRng = NO_RANGE then                { Outside of a range? }
        MouDefinePtr( StdPtr )              { YES, default cursor }
    else                                    { NO, range recognized }
        MouDefinePtr( ActRngPtr^[ NewRng ].PtrMask );
end;
MouRng := NewRng;      { Place range number in global variable }
end;

{ ***** }
{ * MouSetDefaultPtr: Defines default cursor appearance for screen * }
{ * ranges not assigned as special ranges * }
{ **-----** }
{ * Input : default = Cursor and screen mask for mouse cursor * }
{ * Info: - The parameters should be created with the help of the * }
{ * MouPtrMask function * }
{ ***** }

procedure MouSetDefaultPtr( default : PTRVIEW );

begin
    StdPtr := default;      { Reserve bitmask in global variable }

```

[illegible]

```

{*****}
*  MouInit:  Initializes mouse functions and procedures as well as  *
*            variables                                              *
**-----**
*  Input   :  Columns = Number of screen columns                    *
*            Lines  = Number of screen lines                        *
*  Output  :  TRUE if a mouse driver is installed, else FALSE      *
*  Info:    - This function must be the first called from an      *
*            application program, before other procedures and     *
*            functions can be called                               *
{*****}

```

```
function MouInit( Columns, Lines : byte ) : boolean;
```

```
var Regs : Registers;           { Processor regs for interrupt call }
```

```
begin
```

```
  TLine := Lines;               { Store number of lines and }
  TCol  := Columns;             { columns in global variables }
```

```
  ExitOld := ExitProc;          { Set address of exit procedure }
  ExitProc := @MouEnd;          { Define MouEnd as exit procedure }
```

```
{-- Allocate and fill mouse range -----}
```

```
  BLen := TLine * TCol;         { Number of characters in screen }
  GetMem( BufPtr, BLen );       { Allocate internal range buffer }
  MouIBufFill( 0, 0, TCol-1, TLine-1, NO_RANGE );
```

```
  Regs.AX := 0;                 { Initialize mouse driver }
  Intr( $33, Regs );            { Call mouse driver }
  MouInit := ( Regs.AX <> 0 );   { Mouse driver installed? }
```

```

MouSetMoveArea( 0, 0, TCol-1, TLine-1 );           { Set move area }

MouCol    := MouIGetX;                             { Load current mouse position }
MouRow    := MouIGetY;                             { into global variables }
MouRng    := NO_RANGE;                             { cursor in no set range }
MouEvent  := EV_LEFT_REL or EV_RIGHT_REL;          { No mouse button pressed }
StdPtr    := MouPtrMask( PTRSAMECHAR, PTRINVCOL ); { Std. cursor }
OldPtr    := PTRVIEW( 0 );

{-- Install assembler event handler "AssmHand" -----}
MouISetEventHandler( EV_MOU_ALL, FNCTPTR(@AssmHand) );

end;

{*****
*                               M A I N       P R O G R A M                               *
*****}

const Ranges : array[0..4] of RANGE =              { The mouse range }
(
  ( x1: 0; y1: 0; x2: 79; y2: 0 ),                 { Top line }
  ( x1: 0; y1: 1; x2: 0; y2: 23 ),                 { Left column }
  ( x1: 0; y1: 24; x2: 78; y2: 24 ),               { Bottom line }
  ( x1: 79; y1: 1; x2: 79; y2: 23 ),               { Right column }
  ( x1: 79; y1: 24; x2: 79; y2: 24 )              { Lower right corner }
);

var Dummy : integer;                               { Get result from MouEventWait }

begin
  {-- Configure mouse cursor for the different mouse ranges -----}

```

```

Ranges[ 0 ].PtrMask := MouPtrMask( PtrDifChar($18), PtrInvCol);
Ranges[ 1 ].PtrMask := MouPtrMask( PtrDifChar($1b), PtrInvCol);
Ranges[ 2 ].PtrMask := MouPtrMask( PtrDifChar($19), PtrInvCol);
Ranges[ 3 ].PtrMask := MouPtrMask( PtrDifChar($1a), PtrInvCol);
Ranges[ 4 ].PtrMask := MouPtrMask( PtrDifChar($58), PtrDifCol($40));

writeln(#13#10,'MOUSEP - (c) 1989 by MICHAEL TISCHER'#13#10);
if MouInit( 80, 25 ) then
begin
    { Initialize mouse module }
    { OK, there's an installed mouse driver }
    writeln('Move the mouse cursor around the screen. As you move ',CRLF,
        'it around the edge of the screen, you will see the mouse',CRLF,
        'cursor change its appearance. The cursor shape changes ',CRLF,
        'as you move the mouse from edge to edge. ',CRLF,CRLF,
        'To end this program, move the mouse cursor to the ',CRLF,
        'lower right corner of the screen, and press both the ',CRLF,
        'left and right mouse buttons at the same time. ');

    MouSetDefaultPtr( MouPtrMask( PtrDifChar( $DB ), PtrDifCol( 3 ) ) );
    MouDefRange( 5, @Ranges );
    MouShowMouse;
    { Range definition }
    { Display mouse cursor on the screen }

    {-- Wait until the user presses both the left and right mouse ----}
    {-- buttons simultaneously while the cursor is in range 4 ----}

    repeat
    Dummy := MouEventWait( EAND, EV_LEFT_PRESS or EV_RIGHT_PRESS );
    until EvRng = 4;
    end
    else
    { No mouse installed OR no mouse driver installed }
    writeln('Sorry, no mouse driver currently installed.');
```

end.