

```

{ ***** X M S P . P A S ***** }
*
*-----*
* Task      : Demonstrates access to extended memory and
*             high memory area using XMS functions, as
*             implemented by the HIMEM.SYS device driver,
*             for example.
*-----*
* Author     : MICHAEL TISCHER
* Developed on : 07/27/90
* Last update  : 07/29/90
*-----*
{ ***** }

```

```

program XMSP;

uses Crt, Dos;           { For interrupt call and keyboard access }

```

```

const ERR_NOERR           = $00;           { No error }
      ERR_NOTIMPLEMENTED = $80;           { Specified function not known }
      ERR_VDISKFOUND     = $81;           { VDISK-RAMDISK detected }
      ERR_A20            = $82;           { Error at handler A20 }
      ERR_GENERAL        = $8E;           { General driver error }
      ERR_UNRECOVERABLE  = $8F;           { Unrecoverable error }
      ERR_HMANOTEXIST    = $90;           { HMA does not exist }
      ERR_HMAINUSE       = $91;           { HMA already in use }
      ERR_HMAMINSIZE     = $92;           { Not enough space in HMA }
      ERR_HMANOTALLOCED  = $93;           { HMA not allocated }
      ERR_A20STILLON     = $94;           { Handler A20 still on }
      ERR_OUTOMEMORY     = $A0;           { Out of extended memory }
      ERR_OUTOHANDLES    = $A1;           { All XMS handles in use }
      ERR_INVALIDHANDLE  = $A2;           { Invalid handle }
      ERR_SHINVALID      = $A3;           { Source handle invalid }
      ERR_SOINVALID      = $A4;           { Source offset invalid }
      ERR_DHINVALID      = $A5;           { Destination handle invalid }
      ERR_DOINVALID      = $A6;           { Destination offset invalid }
      ERR_LENINVALID     = $A7;           { Invalid length for move function }
      ERR_OVERLAP        = $A8;           { Illegal overlapping }
      ERR_PARITY          = $A9;           { Parity error }
      ERR_EMBUNLOCKED    = $AA;           { UMB is unlocked }
      ERR_EMBLOCKED      = $AB;           { UMB is still locked }
      ERR_LOCKOVERFLOW   = $AC;           { Overflow of UMB lock counter }
      ERR_LOCKFAIL       = $AD;           { UMB cannot be locked }
      ERR_UMBSIZETOOBIG  = $B0;           { Smaller UMB available }
      ERR_NOUMBS         = $B1;           { No more UMB available }
      ERR_INVALIDUMB     = $B2;           { Invalid UMB segment address }

```

```

type XMSRegs = record
    { Information for XMS call }
    AX,           { Only registers AX, BX, DX and SI }
    BX,           { required, depending on called }
    DX,           { function along with a segment }
    SI,           { address }
    Segment : word
end;

```

```

{-- Global variables -----}

```

```

var XMSPtr : pointer;   { Pointer to the extended memory manager (XMM) }
    XMSErr : BYTE;      { Error code of the last operation }

```

```

{ ***** }
* XMSInit : Initializes the routines for calling the XMS functions
*-----*
* Input   : None
* Output  : TRUE, if an XMS driver was discovered, otherwise FALSE
* Info    : - The call of this function must precede calls of all
*             all other procedures and functions from this program.
*-----*
{ ***** }

```

```

function XMSInit : boolean;

```

```

var Regs : Registers;           { Registers for interrupt call }
    xr : XMSRegs;

```

```

begin
    Regs.AX := $4300;           { Determine availability of XMS manager }
    intr( $2F, Regs );
    if ( Regs.AL = $80 ) then    { XMS manager found? }
    begin                      { Yes }
        Regs.AX := $4310;       { Determine entry point of XMM }
        intr( $2F, Regs );
        XMSPtr := ptr( Regs.ES, Regs.BX ); { Store address in glob. var. }
        XMSErr := ERR_NOERR;     { Still no error found }
        XMSInit := true;         { Handler found, module initialized }
    end
    else                        { No XMS handler installed }
        XMSInit := false;

```

```

end;

{*****}
* XMSCall : General routine for calling an XMS function *
{*****}
* Input   : FctNr = Number of XMS function to be called *
*          : XRegs = Structure with registers for function call *
* Info    : - Before calling this procedure, only those registers *
*           : can be loaded that are actually required for calling *
*           : the specified function. *
*           : - After the XMS function call, the contents of the *
*           : various processor registers are copied to the *
*           : corresponding components of the passed structure. *
*           : - Before calling this procedure for the first time, the *
*           : XMSInit must be called successfully. *
{*****}

procedure XMSCall( FctNr : byte; var XRegs : XMSRegs );

begin
  inline ( $8C / $D9 / { mov cx,ds }
           $51 / { push cx }
           $C5 / $BE / $04 / $00 / { lds di,[bp+0004] }
           $8A / $66 / $08 / { mov ah,[bp+0008] }
           $8B / $9D / $02 / $00 / { mov bx,[di+0002] }
           $8B / $95 / $04 / $00 / { mov dx,[di+0004] }
           $8B / $B5 / $06 / $00 / { mov si,[di+0006] }
           $8E / $5D / $08 / { mov ds,[di+08] }
           $8E / $C1 / { mov es,cx }
           $26 / $FF / $1E / XMSPtr / { call es:[XMSPTr] }
           $8C / $D9 / { mov cx,ds }
           $C5 / $7E / $04 / { lds di,[bp+04] }
           $89 / $05 / { mov [di],ax }
           $89 / $5D / $02 / { mov [di+02],bx }
           $89 / $55 / $04 / { mov [di+04],dx }
           $89 / $75 / $06 / { mov [di+06],si }
           $89 / $4D / $08 / { mov [di+08],cx }
           $1F { pop ds }
  );

  {-- Test for error code -----}

  if ( XRegs.AX = 0 ) and ( XRegs.BX >= 128 ) then
    begin
      XMSErr := Lo(XRegs.BX) { Error, store error code }
      {
        .
        .
        .
        Another error handling routine could follow here
        .
        .
        .
      }
    end
  else
    XMSErr := ERR_NOERR; { No error, all ok }
  end;

{*****}
* XMSQueryVer: Returns the XMS version number and other status *
*             : information *
{*****}
* Input   : VerNr = Gets the version number after the function call *
*           : (Format: 235 = 2.35) *
*           : RevNr = Gets the revision number after the function call *
* Output  : TRUE, if HMA is available, otherwise FALSE *
{*****}

function XMSQueryVer( var VerNr, RevNr : integer ): boolean;

var XR : XMSRegs; { Registers for communication with XMS }

begin
  XmsCall( 0, XR );
  VerNr := Hi(XR.AX)*100 + ( Lo(XR.AX) shr 4 ) * 10 +
    ( Lo(XR.AX) and 15 );
  RevNr := Hi(XR.BX)*100 + ( Lo(XR.BX) shr 4 ) * 10 +
    ( Lo(XR.BX) and 15 );
  XMSQueryVer := ( XR.DX = 1 );
end;

{*****}
* XMSGetHMA : Returns right to access the HMA to the caller. *
{*****}
* Input   : LenB = Number of bytes to be allocated *

```

```

* Info      : TSR programs should only request the memory size that
*             they actually require, while applications should specify
*             the value $FFFF.
* Output    : TRUE, if the HMA could be made available,
*             otherwise FALSE;
*****}

function XMSGethHMA( LenB : word ) : boolean;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XR.DX := LenB;             { Pass length in DX register }
  XmsCall( 1, Xr );          { Call XMS function #1 }
  XMSGethHMA := ( XMSerr = ERR_NOERR );
end;

{*****}
* XMSReleaseHMA : Releases the HMA, making it possible to pass
*                to other programs.
*-----*
* Input      : None
* Info      : - Call this procedure before ending a program if the
*              HMA was allocated beforehand through a call for
*              XMSGethHMA, because otherwise the HMA cannot be passed
*              to any programs called afterwards.
*              - Calling this procedure causes the data stored in HAM
*              to be lost.
*-----*
*****}

procedure XMSReleaseHMA;

var Xr : XMSRegs;           { Call registers for communication with XMS }

begin
  XmsCall( 2, Xr );          { Call XMS function #2 }
end;

{*****}
* XMSA20OnGlobal: Switches on the A20 handler, making direct access
*                to the HMA possible.
*-----*
* None      : None
* Info      : - For many computers, switching on the A20 handler is a
*              relatively time-consuming process. Only call this
*              procedure when it is absolutely necessary.
*-----*
*****}

procedure XMSA20OnGlobal;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XmsCall( 3, Xr );          { Call XMS function #3 }
end;

{*****}
* XMSA20OffGlobal: A counterpart to the XMSA20OnGlobal procedure,
*                 this procedure switches the A20 handler back off,
*                 so that direct access to the HMA is no longer
*                 possible.
*-----*
* Input      : None
* Info      : - Always call this procedure before ending a program,
*              in case the A20 handler was switched on before via a
*              call for XMSA20OnGlobal.
*-----*
*****}

procedure XMSA20OffGlobal;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XmsCall( 4, Xr );          { Call XMS function #4 }
end;

{*****}
* XMSA20OnLocal: See XMSA20OnGlobal
*-----*
* Input      : None
* Info      : - This local procedure differs from the global procedure
*              in that it only switches on the A20 handler if it
*              hasn't already been called.
*-----*
*****}

procedure XMSA20OnLocal;

```

```

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XmsCall( 5, Xr );         { Call XMS function #5 }
end;

{*****
* XMSA200ffLocal : See XMSA290ffGlobal
*-----**
* Input      : None
* Info       : - This local procedure only differs from the global
*               procedure in that the A20 handler is only switched
*               off if hasn't already happened through a previous
*               call.
*****}

procedure XMSA200ffLocal;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XmsCall( 6, Xr );         { Call XMS function #6 }
end;

{*****
* XMSIsA20On : Returns the status of the A20 handler
*-----**
* Input      : None
* Output     : TRUE, if A20 handler is on, otherwise FALSE.
*             FALSE.
*****}

function XMSIsA20On : boolean;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XmsCall( 7, Xr );         { Call XMS function #7 }
  XMSIsA20On := ( Xr.AX = 1 ); { AX = 1 ---> Handler is free }
end;

{*****
* XMSQueryFree : Returns the size of free extended memory and the
*               largest free block
*-----**
* Input      : TotFree: Gets the total size of free extended memory.
*             MaxBl  : Gets the size of the largest free block.
* Info       : - Both specifications in kilobytes.
*             - The size of the HMA is not included in the count,
*             even if it hasn't yet been assigned to a program.
*****}

procedure XMSQueryFree( var TotFree, MaxBl : integer );

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  XmsCall( 8, Xr );         { Call XMS function #8 }
  TotFree := Xr.AX;         { Total size in AX }
  MaxBl   := Xr.DX;         { Free memory in DX }
end;

{*****
* XMSGetMem : Allocates an extended memory block (EMB)
*-----**
* Input      : LenKb : Size of requested block in kilobytes
* Output     : Handle for further access to block or 0, if no block
*               can be allocated. The appropriate error code would
*               also be in the global variable, XMSErr.
*****}

function XMSGetMem( LenKb : integer ) : integer;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  Xr.DX := LenKb;           { Length passed in DX register }
  XmsCall( 9, Xr );         { Call XMS function #9 }
  XMSGetMem := Xr.DX;       { Return handle }
end;

{*****
* XMSFreeMem : Releases previously allocated extended memory block
*             (EMB).
*-----**

```

```

Input      : Handle : Handle for access to the block returned when
              XMSGetMem was called.
* Info      : - The contents of the EMB are irretrievably lost and
              the handle becomes invalid when you call this procedure.
*             - Before ending a program, use this procedure to release
              all allocated memory areas, so that they can be
              allocated for the next program to be called.
*****}

procedure XMSFreeMem( Handle : integer );

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  Xr.DX := Handle;           { Handle passed in DX register }
  XmsCall( 10, Xr );         { Call XMS function #10 }
end;

{*****}
* XMSCopy : Copies memory areas between extended memory and
*           conventional memory or within the two memory groups.
*-----**
* Input   : FrmHandle   : Handle of memory area to be copied.
*           FrmOffset   : Offset in block being copied.
*           ToHandle    : Handle of memory area to which memory is
*                       being copied.
*           ToOffset    : Offset in the target block.
*           LenW        : Number of words to be copied.
* Info     : - To include normal memory in the operation, 0 must be
*             specified as the handle and the segment and offset
*             address must be specified as the offset in the usual
*             form (offset before segment).
*-----**}

procedure XMSCopy( FrmHandle   : integer; FrmOffset   : longint;
                  ToHandle    : integer; ToOffset    : longint;
                  LenW        : longint );

type EMMS = record           { An extended memory move structure }
  LenB      : longint;       { Number of bytes to be moved }
  SHandle   : integer;       { Source handle }
  SOffset   : longint;       { Source offset }
  DHandle   : integer;       { Destination handle }
  DOffset   : longint;       { Destination offset }
end;

var Xr : XMSRegs;           { Registers for communication with XMS }
    Mi : EMMS;              { Gets EMMS }

begin
  with Mi do
    begin
      LenB := 2 * LenW;
      SHandle := FrmHandle;
      SOffset := FrmOffset;
      DHandle := ToHandle;
      DOffset := ToOffset;
    end;

    Xr.Si      := ofs( Mi );   { Offset address of EMMS }
    Xr.Segment := Seg(Mi);     { Segment address of EMMS }
    XmsCall( 11, Xr );        { Call XMS function #11 }
  end;

  {*****}
  * XMSLock : Locks an extended memory block from being moved by the
  *           XMM, returning its absolute address at the same time.
  *-----**
  * Input   : Handle : Handle of memory area returned during a prev-
  *           ious call by XMSGetMem.
  * Output  : The linear address of the block of memory.
  *-----**}

function XMSLock( Handle : integer ) : longint;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
  Xr.DX := Handle;           { Handle of EMB }
  XmsCall( 12, Xr );         { Call XMS function #12 }
  XMSLock := longint(Xr.DX) shl 16 + Xr.BX; { Compute 32 bit address }
end;

{*****}
* XMSUnlock : Releases a locked extended memory block again.
*-----**}

```

```

Input      : Handle of memory area returned during a previous call by XMSGetMem.
*****}

procedure XMSUnlock( Handle : integer );

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
    Xr.DX := Handle;           { Handle of EMB }
    XmsCall( 13, Xr );         { Call XMS function #13 }
end;

{*****}
* XMSQueryInfo : Gets various information about an extended memory block that has been allocated.
*
*-----*
* Input      : Handle : Handle of memory area
*              Lock   : Variable, in which the lock counter is entered
*              LenKB  : Variable, in which the length of the block is entered in kilobytes
*              FreeH  : Number of free handles
* Info       : You cannot use this procedure to find out the start address of a memory block, use the XMSLock function instead.
*
*-----*
procedure XMSQueryInfo( Handle      : integer; var Lock, LenKB : integer;
                       var FreeH : integer );

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
    Xr.DX := Handle;           { Handle of EMB }
    XmsCall( 14, Xr );         { Call XMS function #14 }
    Lock  := Hi( Xr.BX );      { Evaluate register }
    FreeH := Lo( Xr.BX );
    LenKB := Xr.DX;
end;

{*****}
* XMSRealloc : Enlarges or shrinks an extended memory block previously allocated by XMSGetMem
*
*-----*
* Input      : Handle : Handle of memory area
*              NewLenKB : New length of memory area in kilobytes
* Output     : TRUE, if the block was resized, otherwise FALSE
* Info       : The specified block cannot be locked!
*
*-----*
function XMSRealloc( Handle, NewLenKB : integer ) : boolean;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
    Xr.DX := Handle;           { Handle of EMB }
    Xr.BX := NewLenKB;         { New length in the BX register }
    XmsCall( 15, Xr );         { Call XMS function #15 }
    XMSRealloc := ( XMSerr = ERR_NOERR );
end;

{*****}
* XMSGetUMB : Allocates an upper memory block (UMB).
*
*-----*
* Input      : LenPara : Size of area to be allocated in paragraphs of 16 bytes each
*              Seg      : Variable that gets the segment address of the allocated UMB in successful cases
*              MaxPara  : Variable that specifies the length of the largest available UMB in unsuccessful cases
* Output     : TRUE, if a UMB could be allocated, otherwise FALSE
* Info       : Warning! This function is not supported by all XMS drivers and is extremely hardware-dependent.
*
*-----*
function XMSGetUMB( LenPara      : integer;
                   var Seg, MaxPara : word ) : boolean;

var Xr : XMSRegs;           { Registers for communication with XMS }

begin
    Xr.DX := LenPara;           { Desired length to }
    XmsCall( 16, Xr );         { Call XMS function #16 }
    Seg := Xr.BX;              { Return segment address }
    MaxPara := Xr.DX;          { Length of largest UMB }
    XMSGetUMB := ( XMSerr = ERR_NOERR );

```

```

end;

{*****
* XMSFreeUMB : Releases UMB previously allocated by XMSGetUMB. *
*****}
-----
* Input      : Seg : Segment address of UMB being released *
* Info       : Warning! This function is not supported by all XMS *
*              drivers and is extremely hardware-dependent. *
*****}

procedure XMSFreeUMB( var Seg : word );

var Xr : XMSRegs;           { Registers for communication wit XMS }

begin
  Xr.DX := Seg;              { Segment address of UMB to DX }
  XmsCall( 17, Xr );         { Call XMS function #17 }
end;

{-----}
{-- Test and Demo procedures --}
{-----}

{*****
* HMA Test : Tests the availability of HMA and demonstrates its use. *
*****}
-----
* Input      : None *
*****}

procedure HMA Test;

type HMA R = array [1..65520] of BYTE;           { HMA array }
      HMA R PTR = ^HMA R;                       { Pointer to HMA array }

var ch      : char;                               { For reading keys }
    A20     : boolean;                           { Current status of A20 handler }
    hmap    : HMA R PTR;                         { Pointer to HMA }
    i       : integer;                           { Loop counter }
    err     : word;                              { Number of errors in HMA access }

begin
  write( 'HMA Test - Please press a key to start the test...' );
  ch := ReadKey;
  writeln( #10 );

  {-- Allocate HMA and test each memory location -----}

  if ( XMSGetHMA( $FFFF ) ) then                  { HMA acquired? }
  begin                                           { Yes }
    A20 := XMSIsA20On;                          { Determine handler status }
    if ( A20 = FALSE ) then                     { Is A20 handler on? }
      XMSA20OnGlobal;                          { No, switch it on now }

    hmap := HMA R PTR( Ptr( $FFFF, $0010 ) );   { Pointer to HMA }

    err := 0;                                   { No errors up until now }
    for i := 1 to 65520 do                      { Test each single memory location }
    begin
      write( #13, 'Memory location: ', i );
      hmap^ [i] := i mod 256;                   { Write memory location }
      if ( hmap^ [i] <> i mod 256 ) then        { And read out again }
      begin                                     { Error! }
        writeln( ' ERROR!' );
        inc( err );
      end;
    end;

    XMSReleaseHMA;                             { Release HMA }
    if ( A20 = FALSE ) then                    { Was A20 handler on? }
      XMSA20OffGlobal;                        { No, switch it off }

    writeln( #13 );
    if ( err = 0 ) then                        { Evaluate results of test }
      writeln( 'HMA O.K., no defective memory location.' )
    else
      writeln( 'ATTENTION: ', err, ' defective memory locations ' +
        'detected in HMA! ' );
    end
  else
    writeln( 'ATTENTION: No access to HMA possible.' );
  end;
end;

{*****
* EMB Test : Tests extended memory and demonstrates the calls of *
*            different XMS functions *
*****}
-----

```



```

Input      : None
*****}

procedure EMBTest;

type BAR = array [1..1024] of BYTE;
      BARPTR = ^BAR;
      { Byte array with 1K }
      { Pointer to byte array }

var ch      : char;
    ADR     : longint;
    barp    : BARPTR;
    i, j,   : integer;
    err,    : integer;
    Handle, : integer;
    TotFree, { Size of total free extended memory }
    MaxBl   : integer; { Largest free block }

begin
  write( 'EMB Test - Please press a key to start the test...' );
  ch := ReadKey;
  writeln( #10 );

  XMSQueryFree( TotFree, MaxBl ); { Determine size of extended memory }
  writeln( 'Total size of free extended memory (incl. HMA): ',
    TotFree, ' KB' );
  writeln( 'Largest free block: ',
    MaxBl, ' KB' );

  TotFree := TotFree - 64; { Calculate actual size without HMA }
  if ( MaxBl >= TotFree ) then { Can the value be right? }
    MaxBl := MaxBl - 64; { No }

  if ( MaxBl <> 0 ) then { Still enough memory free? }
    begin { Yes }
      Handle := XMSGetMem( MaxBl );
      writeln( MaxBl, ' KB allocated.' );
      writeln( 'Handle = ', Handle );
      ADR := XMSLock( Handle ); { Determine address }
      XMSUnlock( Handle ); { Unlock again }
      writeln( 'Start address = ', ADR, ' (', ADR div 1024, 'K)' );

      GetMem( barp, 1024 ); { Buffer to Turbo heap allocated }
      err := 0; { No errors up to now }

      {-- Execute allocated EMB KB for KB and test -----}

      for i := 0 to MaxBl-1 do
        begin
          write( #13, 'KB test: ', i+1 );
          FillChar( barp^, 1024, i mod 255 );
          XMSCopy( 0, longint(barp), Handle, longint(i)*1024, 512 );
          FillChar( barp^, 1024, 255 );
          XMSCopy( Handle, longint(i)*1024, 0, longint(barp), 512 );

          {-- Compare copied buffer with expected result -----}

          j := 1;
          while ( j <= 1024 ) do
            if ( barp^[j] <> i mod 255 ) then
              begin { Error! }
                writeln( ' ERROR!' );
                inc( err );
                j := 1025;
              end
            else { No error, next memory location }
              inc( j );
            end;

          writeln( #13 );
          if ( err = 0 ) then { Evaluate results of test }
            writeln( 'EMB ok, none of the tested 1K blocks ' +
              'were defective.' )
          else
            writeln( 'ATTENTION! ', err, ' defective 1K blocks detected' +
              ' in EMB' );

          FreeMem( barp, 1024 ); { Release buffer again }
          XMSFreeMem( Handle ); { Release EMB again }
        end;
      end;

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

var VerNr,

```



```

RevNr   : integer;

begin
  ClrScr;
  writeln( 'XMSP   -   XMS-Demo program by MICHAEL TISCHER' );
  writeln;
  if XMSInit then
    begin
      if XMSQueryVer( VerNr, RevNr ) then
        writeln( 'Access to HMA possible.' )
      else
        writeln( 'No access to HMA.' );
        writeln( 'XMS version number: ', VerNr div 100,
                  '.', VerNr mod 100 );
        writeln( 'Revision number   : ', RevNr div 100,
                  '.', RevNr mod 100 );

        writeln;
        HMAtest;                                { Test HMA }
        writeln;
        EMBtest;                                { Test extended memory }
      end
    else
      writeln( 'No XMS driver installed!' );
    end.
end.

```