


```

end;

type MRCIREQUEST = record
    SourcePtr    : pointer;      { pointer in source buffer }
    SourceLen    : word;         { size of the source buffer in bytes }
    Reserved1    : word;
    DestPtr      : pointer;      { pointer in destination buffer }
    DestLen      : word;         { size of the destination buffer }
    ChunkLen     : word;         { block size for compressed data }
    IncDecomp    : longint;      { status for incremental decompression }
end;

type BUF = record
    adresse : pointer;          { pointer to buffer }
    lenb    : word;             { size of the buffer in bytes }
end;

{-- Constants -----}

{-- Flags and order codes -----}
const STANDARD_COMPRESS = 1;      { standard compression }
const STANDARD_DECOMPRESS = 2;    { standard decompression }
const MAX_COMPRESS = 8;          { maximum compression }
const INC_DECOMPRESS = 32;       { incremental decompression }

{-- Type of MRCI client -----}
const TYP_APPLICATION = 0;        { MRCI client is an application }
const TYP_TSRDRIVER = 1;         { MRCI client is a TSR or device driver }

{-- Error codes -----}
const MRCI_OK = 0;               { everything o.k. }
const MRCI_ERR_FUNCTION = 1;     { wrong function code }

```

```

const MRCI_ERR_BUSY      = 2;           { server is busy }
const MRCI_ERR_OVERFLOW  = 3;           { destination buffer too small }
const MRCI_ERR_NOCOMPRESS = 4;          { data can't be compressed }

{-- Constants for demo program -----}
const TYP_APP            = TYP_APPLICATION;      { this is not a TSR }
const CHUNK              = 1;                    { compress down to 1 byte if possible }

{-- Global variables -----}

var mrci : mrcinfo;                             { takes MRCI Info block after calling }
                                                { MrciQuery }

{*****}
* MrciQuery : checks for the existence of an MRCI server in hardware or *
* software and provides information about this server                  *
*-----*
* Input : MrciInfo = after successfully calling a function, contains *
*               an MRCI Info block with information about the      *
*               server                                              *
* Output : TRUE if Mrci-server is installed, otherwise FALSE       *
{*****}

function MrciQuery ( var mrci : mrcinfo ) : boolean;

var server : boolean;           { is there an MRCI server? }
    regs   : registers;

type mrcip = ^mrcinfo;         { pointer to MRCINFO }

begin
    regs.ax := $4A12;           { MUX code for MRCI server }

```

```

regs.cx := ( ord( 'M' ) shl 8 ) + ord( 'R' );           { CX = 'MR' }
regs.dx := ( ord( 'C' ) shl 8 ) + ord( 'I' );           { DX = 'CI' }
intr( $2F, regs);
if ( regs.cx = ( (ord( 'I' ) shl 8 ) + ord( 'C' ) ) ) and { CX == 'IC' }
    ( regs.dx = ( (ord( 'R' ) shl 8 ) + ord( 'M' ) ) ) ) then {DX == 'RM' }
    server := true                                     { CX and DX o.k. ---> server found }
else
    begin
        { no software server found, test hardware server }
        regs.ax := $B001;                             { call Int 1A, Function B001h }
        regs.cx := ( ord( 'M' ) shl 8 ) + ord( 'R' );   { CX = 'MR' }
        regs.dx := ( ord( 'C' ) shl 8 ) + ord( 'I' );   { DX = 'CI' }
        intr( $1A, regs);
        if ( regs.cx = ( (ord( 'I' ) shl 8 ) + ord( 'C' ) ) ) and
            ( regs.dx = ( (ord( 'R' ) shl 8 ) + ord( 'M' ) ) ) ) then
            server := true                             { CX and DX o.k. ---> server found }
        end;

    if server then { if server found, then indicates ES:DI in the Info block }
        mrci := mrcip(ptr( Regs.es, Regs.di ))^;
    MrciQuery := server;
end;

```

```

{ *****
* MrciCall : calls on of the different server features *
* ----- *
* Input : OperationCode = command code (see constants) *
*         SourceBuf      = source buffer *
*         DestBuf        = destination buffer *
* Output : error code of the MRCI server *
* Info : This function can only be called after *
*         MrciQuery has been successfully called *
* Globals : mrci *

```

```
*****}
```

```
function MRCICall(      OperationCode : word;
                    var sourcebuf    : buf;
                    var destbuf      : buf ) : integer;
```

```
var mrcir : MRCIREQUEST;      { to build an MRCI Request block }
    mrcrp : pointer;          { points to MRCIR }
    err   : integer;          { returned error code }
```

```
begin
```

```
    mrcir.SourcePtr := sourcebuf.address;      { register source buffer in }
    mrcir.SourceLen := sourcebuf.lenb;         { request block }
```

```
    mrcir.DestPtr := destbuf.address; { register destination buffer Buffer }
    mrcir.DestLen := destbuf.lenb;    { in request block }
```

```
    mrcir.ChunkLen := CHUNK;           { get Chunk size from constant }
    mrcrp := @mrcir;                  { note pointer to request block }
```

```
asm
```

```
    push    bp      { store register }
    push    ds
```

```
{-- First enter into a Windows Critical Section so that multiple VMs ----}
{-- can't call the MRCI server at the same time          ----}
{-- under Windows in 486 Enhanced Mode                    ----}
}
```

```
    push    ax      { Windows expects exactly this }
    mov     ax,8001h { series of commands and no other }
    int     2ah
```

```

pop      ax

{--- Prepare to call the MRCI server -----}
mov      ax,OperationCode
mov      cx,TYP_APP

mov      dx,seg mrci      { pointer to MRCI Info block following ES:BX }
mov      es,dx
mov      bx,offset mrci
lds      si,mrcrp          { set DS:SI to request block }
call     es:MRCInfo.EntryPoint[bx]      { call server }

{-- Leave Windows Critical Section -----}
push     ax                { the command sequence must be }
mov      ax,8101h          { followed exactly here as well }
int      2ah
pop      ax

pop      ds                { retrieve register }
pop      bp

mov      err,ax            { store returned error code }
end;
MRCICall := err;           { return error code as a function result }
destbuf.lenb := mrcir.DestLen; { return size of the destination buffer }
end;

{*****
* Compress : compresses a data block using the MRCI server *
*-----*
* Input : TypCompress = one of the two constants, STANDARD_COMPRESS *
*              or MAX_COMPRESS *

```

```

*           SourceBuf   = source buffer                               *
*           DestBuf     = destination buffer                         *
* Output : error code of the MRCI server                             *
* Info   : This function can only be called after MrciQuery         *
*           has been successfully called                             *
*****}

```

```

function Compress(      TypCompress : word;
                     var sourcebuf   : buf;
                     var destbuf     : buf ) : integer;

```

```

begin
  Compress := MRCICall( STANDARD_COMPRESS, sourcebuf, destbuf );
end;

```

```

{*****}
* Decompress : decompresses a data block using the MRCI server      *
* -----*
* Input : SourceBuf = source buffer                                  *
*         DestBuf   = destination buffer                            *
* Output : error code of the MRCI server                             *
* Info   : This function can't be called until MrciQuery           *
*           has been successfully called                             *
*         The data block must have been compressed by the MRCI server *
(*****}

```

```

function Decompress( var sourcebuf : buf; var destbuf : buf ) : integer;

```

```

begin
  Decompress := MRCICall( STANDARD_DECOMPRESS, sourcebuf, destbuf );
end;

```

```

{*****}
* YesNo : checks flags and outputs Yes/No message *
* -----*
* Input : Status = status in which the desired flag is situated *
*         Flag   = bit value of the flag *
* Output : Yes or No as a string, each 4 characters long *
{*****}

```

```
function YesNo( Status, Flag : word ) : string;
```

```
begin
  If (Status and Flag) = Flag then
    YesNo := 'Yes '
  else
    YesNo := 'No  '
end;
```

```

{*****}
* PrintMrCiInfos: provides status information about the MRCI server *
* -----*
* Input : none *
* Output : none *
* Info   : This function can't be called until after MrCiQuery *
*         has been successfully called *
* Globals : mrci *
{*****}

```

```
procedure PrintMrCiInfos;
```

```
begin
  writeln ( 'Manufacturer           : ',
            chr( ( mrci.ManufacturerID and 255) ),
```



```

        chr( ( mrci.ManufacturerID shr 8 ) and 255 ) ,
        chr( ( mrci.ManufacturerID shr 16 ) and 255 ) ,
        chr( mrci.ManufacturerID shr 24 ) );
writeln ( 'Manufacturer-Version      : ',
mrci.ManufacturerVersion shr 8, '.',
mrci.ManufacturerVersion and 255 );
writeln ( 'MRCI-Version              : ',
mrci.MrciVersion shr 8, '.',
mrci.MrciVersion and 255 );

writeln;
writeln ( 'Features                               Software   Hardware');
writeln ( '-----' );
writeln ( 'Standard compression      : ',
YesNo( mrci.Flags, STANDARD_COMPRESS ), ' ',
YesNo( mrci.HardwareFlags, STANDARD_COMPRESS ) );
writeln ( 'Standard decompression      : ',
YesNo( mrci.Flags, STANDARD_DECOMPRESS ), ' ',
YesNo( mrci.HardwareFlags, STANDARD_DECOMPRESS ) );
writeln ( 'Maximum compression        : ',
YesNo( mrci.Flags, MAX_COMPRESS ), ' ',
YesNo( mrci.HardwareFlags, MAX_COMPRESS ) );
writeln ( 'Incremental decompression    : ',
YesNo( mrci.Flags, INC_DECOMPRESS ), ' ',
YesNo( mrci.HardwareFlags, INC_DECOMPRESS ) );

writeln;
end;

{-----}
{-- Main program --}
{-----}

const SBUFLen = 12196;                { Amount of data to be compressed }

```

```

var sourcebuf : array [1..SBUFLEN] of byte; ( source and destination buffer }
destbuf      : array [1..SBUFLEN] of byte;
sb,
db           : BUF;                { describes source and destination buffer }
err,
i            : integer;            { error code }
                                   { loop counter }

begin
  writeln( 'MRCIP - (c) 1993 by Michael Tischer');
  writeln;
  if not MrCiQuery( mrci ) then
    begin
      writeln ( 'MRCI not installed');
      exit;
    end;

  {-- MRCI is installed; first display information about the MRCI server ----}

  writeln ( 'MRCI installed');
  PrintMrCiInfos;

  {-- Fill buffer with data, compress and decompress -----}
  sb.adresse := @sourcebuf;        { build descriptor for source and }
  sb.lenb    := SBUFLEN;           { destination buffers }
  db.adresse := @destbuf;
  db.lenb    := SBUFLEN;

  for i := 1 to SBUFLEN do          { fill source buffer with data }
    sourcebuf[i] := 1;

  writeln('Compress');

```

```
writeln('Size before: ', SBUFLEN, ' bytes ');
err := Compress( STANDARD_COMPRESS, sb, db );
writeln ( 'Error code = ', err );
writeln('Size after: ', db.lenb, ' bytes ');

writeln;
writeln('Decompress');
writeln('Size before: ', db.lenb, ' bytes ');
err := Decompress( db, sb );
writeln ( 'Error code = ', err );
writeln('Size after: ', sb.lenb, ' bytes ');
end.
```