

```

{*****}
{                                C D D E V . P A S                                }
{*-----*}
{Task          : Utilities for CD device driver programming}
{*-----*}
{Author        : Michael Tischer / Bruno Jennrich}
{Developed on  : 04/08/1994}
{Last update   : 10/08/1994}
{*****}

```

```

{$A-}                                { no word alignment of structures }
{$X+}                                { extended syntax, evaluation of Fct results optional }

```

Unit CDDEV;

Interface

Uses CDUTIL, CRT;

Const

```

DEV_ERROR = ( 1 shl 15 );           { If set, then error in bits 0-7 }
DEV_BUSY  = ( 1 shl 9 );            { Device is busy }
DEV_DONE  = ( 1 shl 8 );            { Device completed last operation }

```

{- With set error bit, following error messages in Low-Byte -----}

```

DEV_WRITE_PROTECT_VIOLATION = 0;
DEV_UNKNOWN_UNIT            = 1;
DEV_DRIVE_NOT_READY         = 2;

```

```

DEV_UNKNOWN_COMMAND           = 3;
DEV_CRC_ERROR                 = 4;
DEV_BAD_DRIVE_REQUEST_LEN     = 5;
DEV_SEEK_ERROR                = 6;
DEV_UNKNOWN_MEDIA             = 7;
DEV_SECTOR_NOT_FOUND          = 8;
DEV_PRINTER_OUT_OF_PAPER      = 9;
DEV_WRITE_FAULT               = 10;
DEV_READ_FAULT                = 11;
DEV_GENERAL_FAILURE           = 12;
DEV_RESERVED1                 = 13;
DEV_RESERVED2                 = 14;
DEV_INVALID_DISK_CHANGE       = 15;

{- Device Commands -----}
DEVCMD_INIT = 0;      { INIT command is only called once during      }
                      { booting, after that no more.                }
DEVCMD_IOCTL_READ = 3;

{-- Returns address of device header in RAM ----}
Const IOCTLI_GET_DEVHDR_ADRESS = 0;
Type MSCDEX_Raddr = Record
    bControlBlockCode      : Byte;                                { = 0 }
    lpDeviceHeaderAddress : ^DeviceHeader; { DeviceHeader address }
End;

{-- Gets read/write head location -----}

```

```

Const IOCTLI_GET_HEAD_LOCATION = 1;
Type MSCDEX_LocHead = Record
    bControlBlockCode,                { = 1 }
    bAddressingMode      : Byte;      { = HSG (0) or REDBOOK (1) }
    lLocation             : Longint;   { Location of read/write }
End;                                  { head (Sector) }

{-- Reserved Command -----}
Const IOCTLI_RESERVED1 = 2;

{-- Gets error statistics (not yet specified). -----}
Const IOCTLI_ERROR_STATISTICS = 3;
Type MSCDEX_ErrStat = Record
    bControlBlockCode : Byte;                { = 3 }
    bErrStat           : Array[0..0] of Byte; { Size of array }
End;                  { not yet defined }

{-- Determines which CD channels are on which CD-Player output -----}
{-- and how loud the channels are. -----}
Const IOCTLI_GET_AUDIO_CHANNEL_INFO = 4;
Type MSCDEX_AudInfo = Record
    bControlBlockCode,                { = 4 }
    bOutput0, bVolume0,               { Which audio tracks are re- }
    bOutput1, bVolume1,               { directed to which CD output, }
    bOutput2, bVolume2,               { and how loud are the CD outputs? }
    bOutput3, bVolume3 : Byte
End;

```

```

{-- Read data directly from drive (Data is hardware dependent) -----}
Const IOCTL_READ_DRIVE_BYTES = 5;
Type MSCDEX_DrvBytes = Record
    bControlBlockCode,                                { = 5 }
    bNumReadBytes      : Byte;                        { = Number of bytes to read }
    bReadBuffer        : Array[0..127] of Byte;
End;           { In 128 byte blocks (previous blocks are discarded) }

{-- Get drive status -----}
Const IOCTL_GET_DEV_STAT = 6;
Type MSCDEX_DevStat = Record
    bControlBlockCode : Byte;                                { = 6 }
    lDeviceStatus      : Longint;                            { after call contains the }
End;                                   { drive status }

Const
    {- Door open -----}
    DS_DOOR_OPEN                = $00000001;

    {- Door unlocked -----}
    DS_DOOR_UNLOCKED            = $00000002;

    {- Data can be read cooked and raw, else only cooked -----}
    DS_READS_COOKED_AND_RAW     = $00000004;

    {- Drive can also write (e.g., WORM), else only read -----}

```

```

DS_READ_WRITE                = $00000008;

{- Drive supports audio and video CDs, else only data -----}
DS_AUDIO_VIDEO                = $00000010;

DS_ISO_9600_INTERLEAVE       = $00000020;           { Bit 6 reserved }

{- Prefetch possible -----}
DS_PREFETCH                   = $00000080;

{- Audio channels can be redirected to CD outputs -----}
DS_MANIPULATE_AUDIO_CHANNEL = $00000100;

{- HSG and REDBOOK addressing modes, else only HSG -----}
DS_HSG_AND_REDBOOK           = $00000200;

{- No disc in drive -----}
DS_NO_DISC_IN_DRIVE          = $00000800;

{- Sub channel info can be read -----}
DS_SUPPORT_RW_SUB_CHANNELS   = $00001000;

{-- Gets sector size in bytes -----}
Const IOCTLI_SECTOR_SIZE = 7;
Type MSCDEX_SectSize = Record
    bControlBlockCode,
    bReadMode          : Byte;
                        { = 7 }
                        { COOKED or RAW }

```

```

        wSectorSize      : Word;    { Contains sector size after call }
    End;                  { COOKED: 2048 or RAW: 2352 bytes }

{-- Location of lead-out tracks (after end of CD) -----}
Const IOCTLI_VOLUME_SIZE = 8;
Type MSCDEX_VolSize = Record
    bControlBlockCode : Byte;                { = 8 }
    lVolumeSize       : Longint;    { After call, location of last }
    End;              { track + 1 }

{-- CD change occurred? -----}
Const IOCTLI_MEDIA_CHANGED = 9;
Type MSCDEX_MedChng = Record
    bControlBlockCode,                { = 9 }
    bMedChng       : Byte;            { After call: NOT_CHANGED,
    End;              { DONT_KNOW, CHANGED }

Const
    NOT_CHANGED = 1;
    DONT_KNOW   = 0;
    CHANGED     = $FF;

{-- Gets CD information -----}
Const IOCTLI_AUDIO_DISK_INFO = 10;
Type MSCDEX_DiskInfo = Record
    bControlBlockCode,                { = 10 }
    bLowestTrack,                    { First track }

```

```

        bHighestTrack      : Byte;                                { Last track }
        bLeadOutTrack      : Longint;                             { Size of CD (REDBOOK) }
    End;

    {-- Gets information about a specific track -----}
    Const IOCTL_AUDIO_TRACK_INFO = 11;
    Type MSCDEX_TnoInfo=record
        bControlBlockCode,                                       { = 11 }
        bTrackNo          : Byte;      { Number of track to be searched }
        lStartingPoint    : Longint;    { Address in REDBOOK format }
        bTrackControlInfo: Byte;        { Info about track }
    End;

    {-- Definitions for TrackControlInfo -----}
    Const
        TCI_4AUDIO_CHANNELS      = $80;
        TCI_DATA_TRACK           = $40;
        TCI_TRACK_MASK           = $C0;
        TCI_PRE_EMPHASIS         = $10;
        TCI_DIGITAL_COPY_PROHIBITED = $20;

    {-- Get current track and disc playing time -----}
    Const IOCTL_AUDIO_QUERY_CHANNEL = 12;
    Type MSCDEX_QInfo = Record
        bControlBlockCode,                                       { = 12 }
        bCONTROLandADR,      { TrackControl, Addressing (s.TnoInfo) }
        bTrackNo,            { Number of current track (song) }

```

```

        bIndex,                { Current index within current track }
        bTrackMin,             { Current track playing time }
        bTrackSec,
        bTrackFrame,
        bZero,                 { always 0 }
        bDiskMin,              { Total CD playing time }
        bDiskSec,
        bDiskFrame            : Byte;
    End;

    {-- Gets Sub Channel Info during playback -----}
    Const IOCTL_I_AUDIO_SUB_CHANNEL_INFO = 13;
    Type MSCDEX_SubChanInfo = Record
        bControlBlockCode      : Byte;                { = 13 }
        lStartingSector        : Longint;              { = REDBOOK address }
        lpTransferAddress       : Pointer;             { = Address of buffer }
        lNumberOfSectorsToRead : Longint;              { = Amount of Sub Channel }
    End;                                                { information to be read. }

    {-- Get UPC/EAN number of inserted CD. -----}
    Const IOCTL_I_UPCODE = 14;
    type MSCDEX_UPCode = Record
        bControlBlockCode,                { = 14 }
        bCONTROLandADR : Byte;              { s. TnoInfo }
        bUPC_EAN       : array[0..6] of Byte; { 13 figure BCD number }
        bZero,         { (last nibble unused) }
        bAFrame        : Byte;

```



```

End;

{-- Gets Audio Status -----}
Const IOCTL_AUDIO_STATUS_INFO = 15;
Type MSCDEX_AudStat = Record
    bControlBlockCode : Byte; { = 15 }
    wAudioStatus       : Word; { Bit 0 = Audio Pause, 1-15 reserv. }
    lResumeStart,      { Start position for DEVCMD_RESUME (REDBOOK) }
    lResumeEnd         : Longint; { End position for }
End; { DEVCMD_RESUME (REDBOOK) }

Const
    {-- INPUTFLUSH uses Standard-RequestHeader -----}
    DEVCMD_INPUTFLUSH = 7;

    {-- OUTPUTFLUSH uses Standard-RequestHeader -----}
    DEVCMD_OUTPUTFLUSH = 11;

    {-- Some drive parameters are changed using the WRITE command. ----}
    {-- The MSCDEX_IOCTLIO block is used for this purpose (see above) }
    DEVCMD_IOCTL_WRITE = 12;

    {-- Opens CD door -----}
    Const IOCTL_EJECT_DISK = 0;
    Type MSCDEX_Eject = Record
        bControlBlockCode : Byte; { = 0 }
    End;

```

```

{-- Locks door -----}
Const IOCTL_LOCK_DOOR = 1;
Type MSCDEX_LockDoor = Record
    bControlBlockCode,
    bLock          : Byte; { = 1: Lock, 0: Open the door }
End;

{-- Resetting of drive -----}
Const IOCTL_RESET_DRIVE = 2;
Type MSCDEX_Reset = Record
    bControlBlockCode : Byte;
End;

{-- Equivalent of IOCTL_GET_AUDIO_CHANNEL_INFO -----}
Const IOCTL_SET_AUDIO_CHANNEL_INFO = 3;

{-- Equivalent of IOCTL_READ_DRIVE_BYTES -----}
Const IOCTL_WRITE_DRIVE_BYTES = 4;

{-- Close drive tray -----}
Const IOCTL_CLOSE_TRAY = 5;
Type MSCDEX_CloseTray = Record
    bControlBlockCode : Byte;
End;

Const

```

```

{-- Open device -----}
DEVCMDC_DEVICE_OPEN  = 13;

{-- Close device -----}
DEVCMDC_DEVICE_CLOSE = 14;

{-- Reads data from a CD -----}
DEVCMDC_READLONG     = 128;

{- The following structure is used for reading and writing data -----}
Type MSCDEX_ReadWriteL = Record
    ReqHdr      : RequestHeader;
    bAdressingMode : Byte;           { = HSG(0) or REDBOOK(1) }
    lpTransferAddress : Pointer;     { = Address of buffer }
    wNumSec      : Word; { =Num. of sectors to be transferred }
    lStartingSector : Longint;       { = Read/write position }
    bDataReadWriteMode,           { = Transfer mode see bel. }
    bInterleaveSize,              { = Number of sequential sectors }
    bInterleaveSkipFactor : Byte;    { = Size of gap }
End;

{-- bDataReadWriteMode Definitions: -----}
{ Only RAW and COOKED are used for reading: }

Const
    COOKED      = 0;
    RAW         = 1;

{-- The following constants are effective during writing: -----}

```

```

MODE0      = 0;           { Deleting of sectors (overwrite with 0) }
MODE1      = 1;           { Inscription of sectors with specified data }
MODE2FORM1 = 2;           { COOKED Data-Write }
MODE2FORM2 = 3;           { RAW Data-Write }

{- Sends next sector to be read to driver, so that the driver can    }
{- "cache" the sector end.                                           }
Const DEVCMD_READLONG_PREFETCH = 130;

{- Sets position of read/write head on entire CD -----}
Const DEVCMD_SEEK = 131;
Type MSCDEX_SeekReq = Record
    ReqHdr      : RequestHeader;
    bAddressingMode : Byte;           { = HSG(0) or REDBOOK(1) }
    lpTransferAdress : Pointer;      { is ignored, therefore = set 0 }
    wNumSec       : Word;            { is ignored, therefore = set 0 }
    lStartingSector : Longint;       { = starting location }
End;

{ Play audio track -----}
Const DEVCMD_PLAY_AUDIO = 132;
Type MSCDEX_PlayReq = Record
    ReqHdr      : RequestHeader;
    bAddressingMode : Byte;           { = HSG(0) or REDBOOK(1) }
    lStartSector,  : Longint;         { = first sector to play }
    lSectorCount   : Longint;         { = Number of tracks to play }
End;

```

```

{- Stop audio output (uses Standard RequestHeader) -----}
Const DEVCMD_STOP_AUDIO = 133;

{- Write data (see READ_LONG) -----}
Const DEVCMD_WRITE_LONG = 134;

{- Compare written data with data in memory -----}
Const DEVCMD_WRITE_LONG_VERIFY = 135;

{- Resume playing music interrupted by nonrecurring STOP_AUDIO -----}
const DEVCMD_RESUME_AUDIO = 136;
Type HSG_DIR_ENTRY = Record
    len_dr,                                { Length of this structure }
    XAR_len      : Byte; {XAR-Length in LBN (logical block numbers) }
    loc_extentI,                                { Start-LBNs (Intel Format) }
    loc_extentM,                                { Start-LBNs (Motorola Format) }
    data_lenI,                                { File length (Intel Format) }
    data_lenM    : Longint;                    { File length (Motorola Format) }
    record_time  : Array[1..5] of Byte;        { Date and time }
    file_flags_hsg,                                { Flags }
    reserved,                                { reserved }
    il_size,                                { Interleave: Number sequential sectors }
    il_skip      : Byte;                    { Interleave: Size of gap }
    VSSNI,                                { Volume Set Sequence Number (Intel Format) }
    VSSNM,                                { Volume Set Sequence Number (Motorola Format) }
    len_fi       : Word;                    { Length of name }

```

```

{  BYTE  file_id[ ... ];Filename, variable length (max. 32 char.)  }
{  BYTE  padding; Pad-Byte (So that system data is word aligned)  }
{  BYTE  sys_data[ ... ];          System data (variable length)  }
End;

```

Type ISO\_DIR\_ENTRY = Record

```

    len_dr,                                { Length of this structure }
    XAR_len      : Byte;  { XAR-Length in LBN (logical block numbers) }
    loc_extentI,                                { Start-LBN (Intel Format) }
    loc_extentM,                                { Start-LBN (Motorola Format) }
    data_lenI,                                { File length (Intel Format) }
    data_lenM    : Longint;                    { File length (Motorola Format) }
    record_time  : array[0..6] of Byte;        { Date and time }
    file_flags_iso,                                { Flags }
    il_size,                                { Interleave: Number sequential sectors }
    il_skip      : Byte;                      { Interleave: Size of gap }
    VSSNI,                                { Volume Set Sequence Number (Intel Format) }
    VSSNM        : Word;{Volume Set Sequence Number (Motorola Format) }
    len-fi       : Byte;                      { Length of name }
{  BYTE  file_id[ ... ];Filename, variable length (max. 32 char)  }
{  BYTE  padding; Pad-Byte (So that system data is word aligned)  }
{  BYTE  sys_data[ ... ];          System data (variable length)  }
End;

```

Type DIR\_ENTRY = Record

```

    XAR_len      : Byte;  { XAR-Length in LBN (logical block numbers) }
    loc_extent    : Longint;                    { Start-LBN }

```

```

lb_size      : Word ;    { Size of an (LBN). If LBN size = 2048,      }
                        { then one LBN corresponds to a physical    }
                        { Sector/Frame.                             }
data_len     : Longint;   { File length (BYTES)                    }
record_time  : Array[0..6] of Byte;   { Date and Time              }
file_flags,  :                               { Flags                 }
il_size,     :                               { Interleave: Number sequential sectors }
il_skip      : Byte;      { Interleave: Size of gap                 }
VSSN         : Word;      { Volume Set Sequence Number             }
len_fi       : Byte;      { Length of file name                    }
file_id      : Array[0..37] of Byte;   { File name ended with #0    }
file_version : Word;      { File version                           }
len_su       : Byte;      { Length of system data                  }
su_data      : Array[0..219] of byte;   { System data              }
End;

                                { Plaintext for Device-Status-Bits }

Const
  DevStat : Array[0..12,0..1] of String =
    ( ( 'Door is open', 'Door is closed' ),
      ( 'Door is unlocked', 'Door is locked' ),
      ( 'Drive reads COOKED and RAW', 'Drive reads only COOKED' ),
      ( 'Drive reads and writes', 'Drive only reads' ),
      ( 'Drive reads DATA and plays AUDIO/VIDEO tracks',
        'Drive reads only DATA tracks' ),
      ( 'Drive supports ISO 9600 interleave',
        'Drive supports no interleave'),
      ( 'reserved', 'reserved' ),

```

```

        ( 'Drive supports prefetching requests',
          'Drive doesn't support prefetching requests'),
        ( 'Drive supports audio-channel manipulation',
          'Drive doesn't support audio-channel manipulation' ),
        ( 'Drive supports HSG and REDBOOK addressing',
          'Drive supports HSG Addressing only'),
        ( 'reserved', 'reserved' ),
        ( 'No disk in drive', 'Disk in drive'),
        ( 'Drive supports R/W sub-channel-info',
          'Drive doesn't support R/W sub-channel-info' ) );

Function cd_IsError( wStatus : Word ) : Boolean;

Function cd_GetReqHdrError( var ReqHdr : RequestHeader ) : Integer;

Function cd_IsReqHdrBusy( var ReqHdr : RequestHeader ) : Boolean;

Function cd_IsReqHdrDone( var ReqHdr : RequestHeader ) : Boolean;

Function cd_GetDeviceHeaderAddress( iCD_Drive_Letter : Integer;
                                   var RA              : MSCDEX_Raddr
                                   ) : Integer;

Function cd_GetLocationOfHead( iCD_Drive_Letter : Integer;
                              var LH            : MSCDEX_LocHead
                              ) : Integer;

```



```

Function cd_GetAudioChannelInfo( iCD_Drive_Letter : Integer;
                                var AI           : MSCDEX_AudInfo
                                Integer;

Function cd_ReadDriveBytes( iCD_Drive_Letter : Integer;
                            var DB           : MSCDEX_DrvBytes ) : Integer;

Function cd_GetDevStat( iCD_Drive_Letter : Integer;
                       var DS           : MSCDEX_DevStat ) : Integer;

Procedure cd_PrintDevStat( var DS : MSCDEX_DevStat );

Function cd_GetSectorSize( iCD_Drive_Letter : Integer;
                           var SEC          : MSCDEX_SectSize;
                           iReadMode       : Integer ) : Integer;

Function cd_GetVolumeSize( iCD_Drive_Letter : Integer;
                           var VS          : MSCDEX_VolSize ) : Integer;

Function cd_GetMediaChanged( iCD_Drive_Letter : Integer;
                             var MC          : MSCDEX_MedChng ) : Integer;

Function cd_GetAudioDiskInfo( iCD_Drive_Letter : Integer;
                              var DI          : MSCDEX_DiskInfo
                              Integer;

Function cd_GetAudioTrackInfo( iCD_Drive_Letter : integer;

```

```

                                iTrackNo      : Integer;
                                var TI         : MSCDEX_TnoInfo
Integer;

Function cd_IsDataTrack( var TI : MSCDEX_TnoInfo ) : Boolean;

Function cd_GetTrackLen( iCD_Drive_Letter, iTno : Integer ) : Longint;

Function cd_QueryAudioChannel( iCD_Drive_Letter : Integer;
                                var QI         : MSCDEX_QInfo ) : Integer;

Function cd_GetAudioSubChannel( iCD_Drive_Letter : Integer;
                                var SCI         : MSCDEX_SubChanl
):Integer;

Function cd_GetUPCode( iCD_Drive_Letter : Integer;
                        var UPC         : MSCDEX_UPCode ) : Integer;

Procedure cd_PrintUPCode( var UPC : MSCDEX_UPCode );

Function cd_GetAudioStatusInfo( iCD_Drive_Letter : Integer;
                                var AS         : MSCDEX_AudStat
Integer;

Function cd_Eject( iCD_Drive_Letter : Integer ) : Integer;

```

```

Function cd_LockDoor( iCD_Drive_Letter : Integer; iLock : Boolean ) :
Integer;

Function cd_ResetDrive( iCD_Drive_Letter : Integer ) : Integer;

Function cd_SetAudioChannelInfo( iCD_Drive_Letter : Integer;
                                var AI           : MSCDEX_AudInfo
                                Integer;

Function cd_WriteDriveBytes( iCD_Drive_Letter : Integer;
                             var DB           : MSCDEX_DrvBytes ) : Integer;

Function cd_CloseTray( iCD_Drive_Letter : Integer ) : Integer;

Function cd_PlayAudio( iCD_Drive_Letter : integer;
                      iAdressMode       : Integer;
                      lStart             : longint;
                      lLen               : Longint ):Integer;

Function cd_StopAudio( iCD_Drive_Letter : Integer ) : Integer;

Function cd_ResumeAudio( iCD_Drive_Letter : Integer ) : Integer;

Function cd_Seek( iCD_Drive_Letter : integer;
                  iAdressingMode     : Integer;
                  lSector            : Longint ) : Integer;

```

```

Function cd_ReadLong( iCD_Drive_Letter : integer;
                     iAdressMode       : Integer;
                     lStart             : Longint;
                     iNumSec           : Integer;
                     lpReadData        : Pointer;
                     iDataMode         : Integer ) : Integer;

Function cd_ReadLongPrefetch( iCD_Drive_Letter : integer;
                              iAdressMode       : Integer;
                              lStart             : Longint ) : Integer;

Function cd_WriteLong( iCD_Drive_Letter : integer;
                      iAdressMode       : Integer;
                      lStart             : Longint;
                      iNumSec           : Integer;
                      lpWriteData        : Pointer;
                      iDataMode         : Integer ) : Integer;

Function cd_WriteLongVerify( iCD_Drive_Letter : integer;
                             iAdressMode       : Integer;
                             lStart             : Longint;
                             iNumSec           : Integer;
                             lpVerifyData      : Pointer;
                             iDataMode         : Integer ) : Integer;

Procedure cd_PrintDiskTracks( iCD_Drive_Letter : Integer );

```

```

Procedure cd_FastForward( iCD_Drive_Letter : Integer );

Function cd_PrintActPlay( iCD_Drive_Letter : Integer ) : Boolean;

Procedure cd_PrintSector( lpSector : Pointer;
                          iSize     : integer;
                          iCols     : integer;
                          iRows     : Integer );

Procedure cd_PrintDirEntry( var DirEntry : DIR_ENTRY );
Function BCDOut( z:Integer ) : String;
Function IntOut( z:Integer; l:Integer ) : String;

```

#### Implementation

```

{ ***** }
{ cd_IsError : Does status word describe an error? }
{ *-----* }
{ Input      : wStatus - Status word to be tested }
{ Output     : TRUE    - Status word describes error }
{              FALSE   - Status word does not describe an error }
{ *-----* }
{ Info       : This function determines whether the error bit (Bit15) }
{              is set. }
{ ***** }
Function cd_IsError( wStatus : Word ) : Boolean;

```

```

Begin
  if ( wStatus and DEV_ERROR ) <> 0 then cd_IsError := TRUE
                                     else cd_IsError := FALSE;
End;

{*****}
{ cd_GetReqHdrError : Get error from last device }
{               communication }
{*****}
{ *-----* }
{ Input      : ReqHdr - Request-Header, to be checked for errors. }
{ Output     : > 0 - Error code + 1 }
{               == 0 - No error }
{*****}
Function cd_GetReqHdrError( var ReqHdr : RequestHeader ) : Integer;

Begin
  if cd_IsError( ReqHdr.wStatus ) then { Test Bit 15 }
    cd_GetReqHdrError := ( ReqHdr.wStatus and $00FF ) + 1
  else
    cd_GetReqHdrError := 0;
End;

{*****}
{ cd_IsReqHdrBusy : Is device still busy executing commands ? }
{*****}
{ *-----* }
{ Input      : ReqHdr - Device-Header }
{ Output     : TRUE - Device still busy with old commands }
{*****}

```

```

{
    FALSE - Device no longer busy
}
{*****}
Function cd_IsReqHdrBusy( var ReqHdr : RequestHeader ) : Boolean;

Begin
    if ( ReqHdr.wStatus and DEV_BUSY ) <> 0 then
        cd_IsReqHdrBusy := TRUE
    else
        cd_IsReqHdrBusy := FALSE;    { Bit 9 }
End;

{*****}
{ cd_IsReqHdrDone : Has last device command been completely
  executed ?
  *-----*
  Input   : ReqHdr - Device-Header
  Output  : TRUE  - Device command has been executed
            FALSE - Device still BUSY
}
{*****}
Function cd_IsReqHdrDone( var ReqHdr : RequestHeader ) : Boolean;

Begin
    if( ReqHdr.wStatus and DEV_DONE ) <> 0 then
        cd_IsReqHdrDone := TRUE
    else
        cd_IsReqHdrDone := FALSE;    { Bit 8 }
End;

```

```

{- Wrapper Functions for MSCDEX-Device commands -----}

{*****}
{ cd_GetDeviceHeaderAddress : Get address of device header }
{*-----*}
{ Input   : iCD_Drive_Letter - Drive ID (0=A, etc ) }
{         RA           - Raddr-ControlBlock }
{ Output  : Device-Status }
{*-----*}
{ Info : After the call, lpRA->lpDeviceHeaderAddress contains the }
{        address of the header of the device driver for the }
{        specified CD-ROM drive }
{*****}
Function cd_GetDeviceHeaderAddress( iCD_Drive_Letter : Integer;
                                   var RA           : MSCDEX_Raddr
Integer;
Begin
  RA.bControlBlockCode := IOCTLI_GET_DEVHDR_ADRESS;           { = 0 }
  cd_GetDeviceHeaderAddress :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                        DEVCMD_IOCTL_READ,
                        @RA,
                        sizeof( MSCDEX_Raddr ) );
End;

{*****}

```



```

{ cd_GetLocationOfHead : Get location of read/write head
*-----*
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )
          LH               - LocHead-ControlBlock
{ Output  : Device-Status
*-----*
{ Info : Prior to the call, LH.bAddressingMode must contain address-
        ing mode (HSG(0) or REDBOOK(1)). After the call, the
        current location of the read/write head will be in
        LH.lLocation.
{*****}
Function cd_GetLocationOfHead( iCD_Drive_Letter : Integer;
                             var LH           : MSCDEX_LocHead
Integer;
Begin
    LH.bControlBlockCode := IOCTLI_GET_HEAD_LOCATION;           { = 1 }
    cd_GetLocationOfHead :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                             DEVCMD_IOCTL_READ,
                             @LH,
                             sizeof( MSCDEX_Raddr ) );
End;

{*****}
{ cd_GetAudioChannelInfo : Get allocation of CD channels to
                        CD outputs
*-----*

```

```

{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
  AI          - AudInfo-ControlBlock
}
{ Output     : Device-Status
}
{ *-----*
}
{ Info : AI.Output0-4 and AI.Volume0-4 contain allocation and
  volume of 4 possible channels
}
{ *****
}
Function cd_GetAudioChannelInfo( iCD_Drive_Letter : Integer;
                                var AI           : MSCDEX_AudInfo
                                : MSCDEX_AudInfo
Integer;
Begin
  AI.bControlBlockCode := IOCTLI_GET_AUDIO_CHANNEL_INFO;      { = 4 }
  cd_GetAudioChannelInfo :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                        DEVCMD_IOCTL_READ,
                        @AI,
                        sizeof( MSCDEX_AudInfo ) );
End;

{ *****
}
{ cd_ReadDriveBytes : Read Drive Bytes
}
{ *-----*
}
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
  DB          - DrvBytes-ControlBlock
}
{ Output     : Device-Status
}
{ *-----*
}
{ Info : Prior to the call, DB.bNumReadBytes contains the number of
}

```

```

{      bytes to be read. After the call, DB.bReadBuffer contains      }
{      the read bytes. If more than 128 bytes are to be read,        }
{      the previous 128 byte blocks are discarded, to pass the      }
{      block specified by INT(NumReadBytes/128)                      }
{*****}
Function cd_ReadDriveBytes( iCD_Drive_Letter : Integer;
                           var DB           : MSCDEX_DrvBytes ) : Integer;

Begin
  DB.bControlBlockCode := IOCTL_READ_DRIVE_BYTES;      { = 5 }
  cd_ReadDriveBytes :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                        DEVCMD_IOCTL_READ,
                        @DB,
                        sizeof( MSCDEX_DrvBytes ) );

End;

{*****}
{ cd_GetDevStat : Read Device-Status                                }
{-----*}
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )                }
{         DS                  - DevStat-ControlBlock              }
{ Output  : Device-Status                                           }
{-----*}
{ Info : After the call, DS.lDeviceStatus contains the Device-Status }
{*****}
Function cd_GetDevStat( iCD_Drive_Letter : Integer;
                       var DS           : MSCDEX_DevStat ) : Integer;

```

```

Begin
  DS.bControlBlockCode := IOCTLI_GET_DEV_STAT;           { = 6 }
  cd_GetDevStat :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                        DEVCMD_IOCTL_READ,
                        @DS,
                        sizeof( MSCDEX_DevStat ) );
End;

{ ***** }
{ cd_PrintDevStat : Output Device-Status }
{ *-----* }
{ Input      : DS - Device-Status obtained via GetDevStat }
{ ***** }
Procedure cd_PrintDevStat( var DS : MSCDEX_DevStat );
var i : Integer;
Begin
  { Go through all bits and output message for set/cleared }
  { bit }
  for i := 0 to 12 do
    if (DS.lDeviceStatus and ( 1 shl i ) ) <> 0 then
      writeln( DevStat[i,0] ) else writeln( DevStat[i,1] );
End;

{ ***** }
{ cd_GetSectorSize : Get size of a sector }
{ *-----* }

```

```

{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
  Sec        : SectSize-ControlBlock
  iReadMode  : COOKED ( 0 ) or RAW ( 1 )
}
{ Output     : Device-Status
}
*-----*
{ Info : After the call, lpSec->wSectorSize contains sector size
}
*****
Function cd_GetSectorSize( iCD_Drive_Letter : Integer;
                          var SEC          : MSCDEX_SectSize;
                          iReadMode       : Integer ) : Integer;

Begin
  SEC.bControlBlockCode := IOCTLI_SECTOR_SIZE;           { = 7 }
  SEC.bReadMode := Byte (iReadMode);
  cd_GetSectorSize :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                        DEVCMD_IOCTL_READ,
                        @SEC,
                        sizeof( MSCDEX_SectSize ) );
End;

*****
{ cd_GetVolumeSize : Get number of sectors of a CD
}
*-----*
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
  VS        : VolSize-ControlBlock
}
{ Output     : Device-Status
}
*-----*

```

```

{ Info : After the call, VS.lVolumeSize contains the number of      }
  sectors of inserted CD.                                           }
{ ***** }
function cd_GetVolumeSize( iCD_Drive_Letter : Integer;
                           var VS           : MSCDEX_VolSize ) : Integer;
Begin
  VS.bControlBlockCode := IOCTL_VOLUME_SIZE;                        { = 8 }
  cd_GetVolumeSize :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                          DEVCMD_IOCTL_READ,
                          @VS,
                          sizeof( MSCDEX_VolSize ) );
End;

{ ***** }
{ cd_GetMediaChanged : CD changed? }
{ *-----* }
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{          MC                - MedChng-ControlBlock }
{ Output  : Device-Status }
{ *-----* }
{ Info : After the call, MC.bMediaChanged contains the status }
{        of the medium }
{          1 : CD has not been changed since last call. }
{          0 : May have been changed }
{          255 : CD has been changed since the last call. }
{ ***** }

```

```

Function cd_GetMediaChanged( iCD_Drive_Letter : Integer;
                             var MC           : MSCDEX_MedChng ) : Integer;
Begin
    MC.bControlBlockCode := IOCTL_MEDIA_CHANGED;           { = 9 }
    cd_GetMediaChanged :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                              DEVCMD_IOCTL_READ,
                              @MC,
                              sizeof( MSCDEX_MedChng ) );
End;

{ ***** }
{ cd_GetAudioDiskInfo : Get number of titles (songs, tracks) }
{                   of an Audio CD. }
{ *-----* }
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{         DI      - DiskInfo-ControlBlock }
{ Output  : Device-Status }
{ *-----* }
{ Info : After the call, DI.bLowestTrack, DI.bHighestTrack contain }
{        the number of the first and last song. }
{        DI.bLeadOutTrack contains the address of the first unused }
{        sector (after last title) }
{ ***** }
function cd_GetAudioDiskInfo( iCD_Drive_Letter : Integer;
                             var DI           : MSCDEX_DiskInfo
Integer;

```

```

Begin
    DI.bControlBlockCode := IOCTL_AUDIO_DISK_INFO;           { = 10 }
    cd_GetAudioDiskInfo :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                             DEVCMD_IOCTL_READ,
                             @DI,
                             sizeof( MSCDEX_DiskInfo ) );
End;

{ ***** }
{ cd_GetAudioTrackInfo : Get data on a title }
{ *-----* }
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{              iTrackNo         - Number of track for which data }
{                               is to be obtained. }
{              TI                - TrackInfo-ControlBlock }
{ Output     : Device-Status }
{ *-----* }
{ Info : TI.StartingPoINT        - REDBOOK address of title start }
{       TI.bTrackControlInfo     - Information about track type }
{ ***** }
function cd_GetAudioTrackInfo( iCD_Drive_Letter : integer;
                               iTrackNo        : Integer;
                               var TI           : MSCDEX_TnoInfo
Integer;
Begin
    TI.bControlBlockCode := IOCTL_AUDIO_TRACK_INFO;           { = 11 }

```



```

TI.bTrackNo := Byte (iTrackNo);
cd_GetAudioTrackInfo :=
  MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                       DEVCMD_IOCTL_READ,
                       @TI,
                       sizeof( MSCDEX_TnoInfo ) );

End;

{ ***** }
{ cd_IsDataTrack : Is track a pure data track, or does it }
{                   contain audio information? }
{ *-----* }
{ Input      : TI - TrackInfo-ControlBlock }
{ Output     : TRUE  - TI describes data track }
{             FALSE - Track contains audio data }
{ ***** }
function cd_IsDataTrack( var TI : MSCDEX_TnoInfo ) : Boolean;
Begin
  if ( ( TI.bTrackControlInfo and TCI_TRACK_MASK ) = TCI_DATA_TRACK )
  then cd_IsDataTrack := TRUE
  else cd_IsDataTrack := FALSE;
End;

{ ***** }
{ cd_GetTrackLen : Get size of a track (number sectors or }
{                   frames) }
{ *-----* }

```

```

{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
  iTrackNo    - Number of track for which data
                  is to be obtained.
}
{ Output      : Number of frames of specified track
}
{ *-----*
  Info : With the last track, the Lead-Out track must be used as
          the terminator.
}
{ *****
Function cd_GetTrackLen( iCD_Drive_Letter, iTno : Integer ) : Longint;
var DI      : MSCDEX_DiskInfo;
    TI      : MSCDEX_TnoInfo;
    VS      : MSCDEX_VolSize;
    lStart,
    lEnd    : Longint;

Begin
  cd_GetAudioDiskInfo( iCD_Drive_Letter, DI );
  if( ( iTno < DI.bLowestTrack ) or ( iTno > DI.bHighestTrack ) ) then
    cd_GetTrackLen:=0
    { not a valid track }
  else
    Begin
      cd_GetAudioTrackInfo( iCD_Drive_Letter, iTno, TI );
      lStart := REDBOOK2HSG( TI.lStartingPoint );
      if( iTno < DI.bHighestTrack ) then
        Begin
          { Difference sequential tracks }
          cd_GetAudioTrackInfo( iCD_Drive_Letter, iTno + 1, TI );
          lEnd := REDBOOK2HSG( TI.lStartingPoint );

```

```

        End
    else          { Difference [End of CD - last track] }
        Begin
            cd_GetVolumeSize( iCD_Drive_Letter, VS );
            lEnd := VS.lVolumeSize - 150;           { Convert Frames to HSG }
        End;
        cd_GetTrackLen := lEnd - lStart;
    End;
End;

{ ***** }
{ cd_QueryAudioChannel : Get Online information about audio titl }
{ currently playing }
{ *-----* }
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{             QI                - QInfo-ControlBlock }
{ Output     : Device-Status }
{ *-----* }
{ Info : QI.bTrackNo/bIndex/bTrackMin/bTrackSec/bTrackFrame: }
{         Information about song currently playing (number and }
{         playing time) }
{         QI.bDiskMin/bDiskSec/bDiskFrame: Total playing time }
{         }
{         This function can be called on a regular basis to get }
{         current playing times. }
{ ***** }
Function cd_QueryAudioChannel( iCD_Drive_Letter : Integer;

```

```

var QI                                : MSCDEX_QInfo ) : Integer;

Begin
  QI.bControlBlockCode := IOCTL_AUDIO_QUERY_CHANNEL;      { = 12 }
  cd_QueryAudioChannel :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                        DEVCMD_IOCTL_READ,
                        @QI,
                        sizeof( MSCDEX_QInfo ) );
End;

{ ***** }
{ cd_GetAudioSubChannelInfo : Get Sub-Channel information }
{ *-----* }
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )      }
{          SCI               - SubChanInfo-ControlBlock }
{ Output  : Device-Status                                }
{ *-----* }
{ Info :   After this function has been called, the address }
{           specified in SCI.lpTransferAddress contains the }
{           Sub-Channel-Information of the specified sectors }
{           (SCI.lStartingSector, SCI.lNumberOfSectorsToRead) }
{           A 96 byte buffer must be provided for each sector. }
{ ***** }
function cd_GetAudioSubChannel( iCD_Drive_Letter : Integer;
                                var SCI           : MSCDEX_SubChanInfo
                                Integer;
Begin

```

```

    SCI.bControlBlockCode := IOCTL_AUDIO_SUB_CHANNEL_INFO;    { = 13 }
    cd_GetAudioSubChannel :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                              DEVCMD_IOCTL_READ,
                              @SCI,
                              sizeof( MSCDEX_SubChanInfo ) );

End;

{ ***** }
{ cd_GetUPCode : Get Universal-Product-Code of a CD }
{ *-----* }
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{          UPC               - UPCode-ControlBlock }
{ Output  : Device-Status }
{ *-----* }
{ Info   : After the call, UPC.bUPC_EAN contains a 13 figure }
{          BCD number, which states the UP-Code of the current }
{          CD. }
{ ***** }
Function cd_GetUPCode( iCD_Drive_Letter : Integer;
                      var UPC           : MSCDEX_UPCode ) : Integer;

Begin
    UPC.bControlBlockCode := IOCTL_UPCODE;    { = 14 }
    cd_GetUPCode :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                              DEVCMD_IOCTL_READ,
                              @UPC,

```

```

                                sizeof( MSCDEX_UPCode ) );

End;

{*****}
{ cd_PrintUPCode : Output UPC BCD number. }
{*****}
{-----*}
{ Input      : UPC - UPCode-ControlBlock }
{-----*}
{ Info : The last nibble of bUPC_EAN array is not used }
{*****}
procedure cd_PrintUPCode( var UPC : MSCDEX_UPCode );
var i : Integer;
Begin
    Write(' UPC: ');
    for i := 0 to 5 do
        Write( BCDOut( UPC.bUPC_EAN[ i ] ) );           { the first 12 digits }
    Inc(i);
    WriteLn( char ( ( ( UPC.bUPC_EAN[ i ] shr 4 ) and $0F ) + { output last digit "by hand" }
                ord( '0' ) ) );
End;

{*****}
{ cd_GetAudioStatusInfo : Get current audio status }
{                          (Play/Pause) }
{-----*}
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{              AS                - AudStat-ControlBlock }
{-----*}

```

```

{ Output : Device-Status
{ *-----*
{ Info : After the call, Bit 0 of the AS.wAudioStatus variable
{ gives information about the play mode of the CD drive.
{ (Set = Pause)
{ AS.lResumeStart contains the next position to play
{ for a RESUME. AS.lResumeEnd indicates when playback of
{ the song should be stopped.
{ *****
Function cd_GetAudioStatusInfo( iCD_Drive_Letter : Integer;
                                var AS : MSCDEX_AudStat
Integer;
Begin
    AS.bControlBlockCode := IOCTL_AUDIO_STATUS_INFO; { = 15 }
    cd_GetAudioStatusInfo :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                              DEVCMD_IOCTL_READ,
                              @AS,
                              sizeof( MSCDEX_AudStat ) );
End;

{ *****
{ cd_Eject : Open tray
{ *-----*
{ Input : iCD_Drive_Letter - Drive ID( 0=A, etc )
{ Output : Device-Status
{ *-----*

```

```

{ Info : Before tray is opened, it is unlocked
{ *****
Function cd_Eject( iCD_Drive_Letter : Integer ) : Integer;

var E : MSCDEX_Eject ;

Begin
  E.bControlBlockCode := IOCTL_EJECT_DISK;                                { = 0 }
  cd_Eject := MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                                   DEVCMD_IOCTL_WRITE,
                                   @E,
                                   sizeof( MSCDEX_Eject ) );
End;

{ *****
{ cd_LockDoor : Lock/unlock door
{ -----*
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
{              iLock           - == 0 : Unlock door
{                               <> 0 : Lock door
{ Output     : Device-Status
{ *****
function cd_LockDoor( iCD_Drive_Letter : Integer; iLock : Boolean
Integer;

var LD : MSCDEX_LockDoor;

```



[illegible]

```

{*****}
{ cd_SetAudioChannelInfo : Set allocation of CD channels to }
{ CD outputs }
{ *-----* }
{ Input : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{ AI - AudInfo-ControlBlock }
{ Output : Device-Status }
{ *-----* }
{ Info : AI.Output0-4 and AI.Volume0-4 contain allocation and }
{ volume of the 4 possible channels }
{*****}
Function cd_SetAudioChannelInfo( iCD_Drive_Letter : Integer;
                                var AI : MSCDEX_AudInfo
                                Integer;

Begin
    AI.bControlBlockCode := IOCTL_SET_AUDIO_CHANNEL_INFO; { = 3 }
    cd_SetAudioChannelInfo :=
        MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                             DEVCMD_IOCTL_WRITE,
                             @AI,
                             sizeof( MSCDEX_AudInfo ) );

End;

{*****}
{ cd_WriteDriveBytes : Passes drive-specific data }
{ *-----* }

```

```

{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
  DB          - DrvBytes-ControlBlock
{ Output      : Device-Status
  *****
Function cd_WriteDriveBytes( iCD_Drive_Letter : Integer;
                             var DB          : MSCDEX_DrvBytes ) : Integer;

Begin
  DB.bControlBlockCode := IOCTL_WRITE_DRIVE_BYTES;           { = 4 }
  cd_WriteDriveBytes :=
    MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                         DEVCMD_IOCTL_WRITE,
                         @DB,
                         sizeof( MSCDEX_DrvBytes ) );

End;

{ *****
{ cd_CloseTray : Close tray
{ -----*
{ Input      : iCD_Drive_Letter : Drive ID( 0=A, etc )
{ Output      : Device-Status
{ *****
function cd_CloseTray( iCD_Drive_Letter : Integer ) : Integer;

var CT : MSCDEX_CloseTray;

Begin

```

```

CT.bControlBlockCode := IOCTL_CLOSE_TRAY;                                { = 5 }
cd_CloseTray := MSCDEX_ReadWriteReq( iCD_Drive_Letter,
                                     DEVCMD_IOCTL_WRITE,
                                     @CT,
                                     sizeof( MSCDEX_CloseTray ) );

End;

{ ***** }
{ cd_PlayAudio : Plays music }
{ *-----* }
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{              iAdressMode      - HSG(0) or REDBOOK(1) addresses? }
{              lStart           - Address at which playback is }
{                               to begin }
{              lLen             - Number of sectors to play back }
{ Output      : Device-Status }
{ *-----* }
{ Info        : This command updates the BUSY and PAUSE bits }
{               (Audio-Stat). If the CD player is already busy playing }
{               music, a STOP_AUDIO-Request must be sent prior to a new }
{               PLAY_AUDIO-Request. If the drive doesn't support audio, }
{               this command will be ignored. }
{ ***** }
Function cd_PlayAudio( iCD_Drive_Letter : Integer;
                      iAdressMode      : Integer;
                      lStart           : longint;
                      lLen             : Longint ) : Integer;

```

```

var PR : MSCDEX_PlayReq;

Begin
  if lLen < 0 then lLen := 0;                                { Always positive }
  PR.RegHdr.bLength := sizeof( RequestHeader );
  PR.RegHdr.bSubUnit := 0;
  PR.RegHdr.bCommand := Byte ( DEVCMD_PLAY_AUDIO );
  PR.bAddressingMode := Byte ( iAdressMode );
  PR.lStartSector := lStart;
  PR.lSectorCount := lLen;
  cd_PlayAudio := MSCDEX_SendDeviceRequest( iCD_Drive_Letter, PR.RegHdr )
End;

{ ***** }
{ cd_StopAudio : Stops music output }
{ -----* }
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{ Output  : Device-Status }
{ -----* }
{ Info    : Musical data played by the Play command will continue }
{           playing until all sectors to be played have been output. }
{           To interrupt musical output prematurely, the Stop }
{           command must be sent. If a Play-Request is interrupted }
{           by a nonrecurring Stop call, it can be started up again }
{           by a Resume. Playback resumes at the last position. }
{           After two sequential Stop-Requests, audio output can }

```

```

{          only be started again by a new Play-Request.          }
{ ***** }
Function cd_StopAudio( iCD_Drive_Letter : Integer ) : Integer;

var ReqHdr : RequestHeader;

Begin
    ReqHdr.bLength := sizeof( RequestHeader );
    ReqHdr.bSubUnit := 0;
    ReqHdr.bCommand := Byte ( DEVCMD_STOP_AUDIO );

    cd_StopAudio := MSCDEX_SendDeviceRequest( iCD_Drive_Letter, ReqHdr );
End;

{ ***** }
{ cd_ResumeAudio : Starts audio output that has been interrupted }
{ *-----* }
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{ Output     : Device-Status }
{ ***** }
Function cd_ResumeAudio( iCD_Drive_Letter : Integer ) : Integer;

var ReqHdr : RequestHeader ;

Begin
    ReqHdr.bLength := sizeof( RequestHeader );
    ReqHdr.bSubUnit := 0;

```

```

ReqHdr.bCommand := Byte ( DEVCMD_RESUME_AUDIO );

cd_ResumeAudio := MSCDEX_SendDeviceRequest( iCD_Drive_Letter, ReqHdr );
End;

{*****}
{ cd_Seek : Position Read/Write Head }
{*****}
{
  Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )
            iAdressingMode   - HSG(0) or REDBOOK(1)
            lSector          - Starting sector
  Output  : Device-Status
}
{*****}
{
  Info : The positioning of the Read/Write Head differs from the
         positioning of the file pointer within a file. With a
         CD drive, Seek can be used "cross-filewise" (even
         with audio output). However, with audio output, a Seek
         can be used for positioning above the end of the output
         (Number of sectors to be played back).
}
{*****}
Function cd_Seek( iCD_Drive_Letter : integer;
                  iAdressingMode   : Integer;
                  lSector          : Longint ) : Integer;

var SR : MSCDEX_SeekReq;

Begin

```

```

SR.RegHdr.bLength      := sizeof( RequestHeader );
SR.RegHdr.bSubUnit     := 0;
SR.RegHdr.bCommand     := Byte ( DEVCMD_SEEK );
SR.bAddressingMode     := Byte ( iAddressingMode );
SR.lpTransferAddress   := NIL;
SR.wNumSec             := 0;
SR.lStartingSector     := lSector;
cd_Seek := MSCDEX_SendDeviceRequest( iCD_Drive_Letter, SR.RegHdr );
End;

{*****}
{ cd_ReadLong : Reads number of sectors (see AbsoluteDiskRead) }
{*-----*}
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{          iAddressingMode  - HSG(0) or REDBOOK(1) }
{          lStart           - First sector to read }
{          iNumSec          - Number of sectors to be read }
{          lpReadData       - Address of read buffer }
{          iDataMode        - RAW(1) or COOKED(0) }
{ Output  : Device-Status }
{*-----*}
{ Info : A RAW sector is 2353 bytes large. COOKED sectors, on }
{        the other hand, are only 2048 bytes. }
{*****}
Function cd_ReadLong( iCD_Drive_Letter : integer;
                     iAddressMode     : Integer;
                     lStart           : Longint;

```



```

                                iNumSec          : Integer;
                                lpReadData       : Pointer;
                                iDataMode        : Integer ) : Integer;

var RWL : MSCDEX_ReadWriteL;

Begin
    RWL.RegHdr.bLength          := sizeof( RequestHeader );
    RWL.RegHdr.bSubUnit         := 0;
    RWL.RegHdr.bCommand         := Byte ( DEVCMD_READLONG );
    RWL.bAdressingMode          := Byte ( iAdressMode );
    RWL.lpTransferAddress       := lpReadData;
    RWL.wNumSec                 := iNumSec;
    RWL.lStartingSector         := lStart;
    RWL.bDataReadWriteMode      := Byte ( iDataMode );
    RWL.bInterleaveSize        := 1;
    RWL.bInterleaveSkipFactor   := 1;

    cd_ReadLong := MSCDEX_SendDeviceRequest( iCD_Drive_Letter, RWL.RegHdr )
End;

{ ***** }
{ cd_ReadLongPrefetch : Tells drive next sector to read. }
{ *-----* }
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{             iAdressingMode    - HSG(0) or REDBOOK(1) }
{             lStart            - First sector to read }

```

```

{ Output   : Device-Status
{ *-----*
{ Info : To compensate a little for the slow speed of a CD, you
{       can build a simplified cache system (one sector) with the
{       help of prefetching. This command communicates the next
{       sector to be read to the device driver. This sector
{       is read in 'passing'.
{ *****
Function cd_ReadLongPrefetch( iCD_Drive_Letter : integer;
                             iAdressMode       : Integer;
                             lStart            : Longint ) : Integer;

var RWL : MSCDEX_ReadWriteL;
Begin
    RWL.RegHdr.bLength      := sizeof( RequestHeader );
    RWL.RegHdr.bSubUnit     := 0;
    RWL.RegHdr.bCommand     := Byte ( DEVCMD_READLONG_PREFETCH );
    RWL.bAdressingMode      := Byte ( iAdressMode );
    RWL.lpTransferAddress   := NIL;
    RWL.wNumSec             := 1;
    RWL.lStartingSector     := lStart;
    RWL.bDataReadWriteMode  := 0;
    RWL.bInterleaveSize     := 0;
    RWL.bInterleaveSkipFactor := 0;

    cd_ReadLongPrefetch := MSCDEX_SendDeviceRequest( iCD_Drive_Lett
RWL.RegHdr );
End;

```

```

{*****}
{ cd_WriteLong : Writes number of sectors (see AbsoluteDiskWrite) }
{*-----*}
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{              iAdressingMode   - HSG(0) or REDBOOK(1) }
{              lStart           - First sector to be written }
{              iNumSec          - Number of sectors to be written }
{              lpWriteData      - Address of write buffer }
{              iDataMode        - Mode0(0), Mode1(1), }
{                               Mode2Form1(2), Mode2Form2(3) }
{ Output     : Device-Status }
{*-----*}
{ Info : Mode0 -      Sectors are written with zeroes }
{        Mode1 -      COOKED-Write (2048 byte sectors) }
{        Mode2Form1 - COOKED-Write (2048 byte sectors) }
{        Mode2Form2 - RAW-Write (2352 byte sectors) }
{*****}
Function cd_WriteLong( iCD_Drive_Letter : integer;
                      iAdressMode      : Integer;
                      lStart            : Longint;
                      iNumSec           : Integer;
                      lpWriteData       : Pointer;
                      iDataMode         : Integer ) : Integer;

var RWL : MSCDEX_ReadWriteL;

```

```

Begin
    RWL.RegHdr.bLength      := sizeof( RequestHeader );
    RWL.RegHdr.bSubUnit     := 0;
    RWL.RegHdr.bCommand     := Byte ( DEVCMD_WRITELONG );
    RWL.bAddressingMode     := Byte ( iAddressMode );
    RWL.lpTransferAddress   := lpWriteData;
    RWL.wNumSec             := iNumSec;
    RWL.lStartingSector     := lStart;
    RWL.bDataReadWriteMode  := Byte ( iDataMode );
    RWL.bInterleaveSize     := 0;
    RWL.bInterleaveSkipFactor := 0;

    cd_WriteLong := MSCDEX_SendDeviceRequest( iCD_Drive_Letter, RWL.RegHdr
End;

{ ***** }
{ cd_WriteLongVerify : Writes and checks data on CD with data in }
{                      memory                                     }
{ *-----* }
{ Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )           }
{          iAddressingMode   - HSG(0) or REDBOOK(1)          }
{          lStart            - First sector to be verified     }
{          iNumSec           - Number of sectors to be verified }
{          lpVerifyData      - Address of check buffer         }
{          iDataMode         - Model(1),                       }
{                               Mode2Form1(2), Mode2Form2(3)    }
{ Output  : Device-Status }

```

```

{ *-----*
  Info : Model:          COOKED-Verify (2048 byte sectors)
        Mode2Form1: COOKED-Verify (2048 byte sectors)
        Mode2Form2: RAW-Verify (2352 byte sectors)
{ *****
Function cd_WriteLongVerify( iCD_Drive_Letter : integer;
                             iAddressMode     : Integer;
                             lStart           : Longint;
                             iNumSec         : Integer;
                             lpVerifyData    : Pointer;
                             iDataMode       : Integer ) : Integer;

var RWL : MSCDEX_ReadWriteL;

Begin
  RWL.RegHdr.bLength      := sizeof( RequestHeader );
  RWL.RegHdr.bSubUnit     := 0;
  RWL.RegHdr.bCommand     := Byte ( DEVCMD_WRITELONG_VERIFY );
  RWL.bAddressingMode     := Byte ( iAddressMode );
  RWL.lpTransferAddress   := lpVerifyData;
  RWL.wNumSec             := iNumSec;
  RWL.lStartingSector     := lStart;
  RWL.bDataReadWriteMode  := Byte ( iDataMode );
  RWL.bInterleaveSize     := 0;
  RWL.bInterleaveSkipFactor := 0;

  cd_WriteLongVerify := MSCDEX_SendDeviceRequest( iCD_Drive_Letter,

```

RWL.ReqHdr );

End;

```
{ *****
  IntOut : Output Integer including leading zeroes
  *-----*
  Input   : z : Number to be output
            l : Length of output string inc. zeroes
  Output  : Formatted number string
  ***** }
```

function IntOut( z:Integer; l:Integer ):String;

var dummy:String;

begin

Str( z, dummy );

while( length( dummy ) < l ) do dummy := '0' + dummy;

IntOut := dummy;

end;

```
{ *****
  BCDOut : Convert BCD number to string
  *-----*
  Input   : z : Number to be output
            l : Length of output string
  Output  : Formatted number string
  ***** }
```

function BCDOut( z:Integer ):String;

var dummy:String;

begin

```

dummy := char ( ( ( z shr 4 ) and $0F ) + ord( '0' ) ) +
             char ( ( ( z and $0F ) + ord( '0' ) ) );
BCDOut := dummy;
end;

{ ***** }
{ cd_PrintDiskTracks : Show all titles and their playing times }
{ -----* }
{ Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{ ***** }
Procedure cd_PrintDiskTracks( iCD_Drive_Letter : Integer );

var DI      : MSCDEX_DiskInfo;
    TI      : MSCDEX_TnoInfo;
    i,
    iMin,
    iSec,
    iFrame : Integer;
    lLen    : Longint;

Begin
    if ( not cd_IsError( cd_GetAudioDiskInfo( iCD_Drive_Letter, DI ) ) ) then
        for i := DI.bLowestTrack to DI.bHighestTrack do
            Begin
                if ( not cd_IsError(cd_GetAudioTrackInfo(
                    iCD_Drive_Letter,i, TI ) ) ) then

```

```

Begin
  REDBOOK2Time( TI.lStartingPoint, iMin, iSec, iFrame );
  Write( ' ', IntOut( i, 2 ), ': ',
          IntOut( iMin, 2 ), ': ',
          IntOut( iSec, 2 ), '.',
          IntOut( iFrame, 2 ), ' ' );
  lLen := cd_GetTrackLen( iCD_Drive_Letter, i );
  Frame2Time( lLen, iMin, iSec, iFrame );
  Write( ' Lnge: ', IntOut( i, 2 ), ': ',
          IntOut( iMin, 2 ), ': ',
          IntOut( iSec, 2 ), '.',
          IntOut( iFrame, 2 ) );

  Write( ' Typ :' );
  case ( TI.bTrackControlInfo and TCI_TRACK_MASK ) of
    TCI_DATA_TRACK: Write( 'Data ' );
    TCI_4AUDIO_CHANNELS: Write( 'Audio/Quadro ' );
    else Write( 'Audio/Stereo ' );
  End;
  if ( TI.bTrackControlInfo and TCI_PRE_EMPHASIS ) <> 0 then
    Write( 'Pre-Emphasis ' );
  if ( TI.bTrackControlInfo and TCI_DIGITAL_COPY_PROHIBITED ) <
then
    Write( 'Digital Copy prohibited ' );
  Writeln;
End;
End;
End;

```



```

{*****}
{ cd_FastForward : Fast forward current audio output by 2 seconds }
{ *-----* }
{   Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{*****}
Procedure cd_FastForward( iCD_Drive_Letter : Integer );

```

```

var AS      : MSCDEX_AudStat;
    lHSG : Longint;

```

```

Begin
    cd_GetAudioStatusInfo( iCD_Drive_Letter, AS );
    cd_StopAudio( iCD_Drive_Letter );
    lHSG := AS.lResumeStart + Time2Frame( 0, 2, 0 );
    cd_PlayAudio( iCD_Drive_Letter,
                  REDBOOK,
                  HSG2REDBOOK( lHSG ),
                  REDBOOK2HSG( AS.lResumeEnd ) - lHSG );
End;

```

```

{*****}
{ cd_PrintActPlay : Display playing times and title of title }
{                  currently playing. }
{ *-----* }
{   Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) }
{   Output  : TRUE              - CD player still busy with audio }

```

```

{                                     output                                     }
{                                     - CD player finished with playback         }
{ ***** }
Function cd_PrintActPlay( iCD_Drive_Letter : Integer ) : Boolean;

var iStat : Integer;
    QI    : MSCDEX_QInfo;

Begin
    iStat := cd_QueryAudioChannel( iCD_Drive_Letter, QI );
    Write( ' Track: ', BCDOut( QI.bTrackNo ), ' Index: ',
           IntOut( QI.bIndex, 2 ), ' ' );
    Write( ' Track: ', IntOut( QI.bTrackMin, 2 ), ':',
           IntOut( QI.bTrackSec, 2 ), '.',
           IntOut( QI.bTrackFrame, 2 ) );
    Write( ' Disk: ', IntOut( QI.bDiskMin, 2 ), ':',
           IntOut( QI.bDiskSec, 2 ), '.',
           IntOut( QI.bDiskFrame, 2 ), '#13 );
    if ( iStat and DEV_BUSY ) <> 0 then
        cd_PrintActPlay := TRUE else cd_PrintActPlay := FALSE;
End;

{ ***** }
{ Hex : Converts a hex value to a hex string }
{ *-----* }
{ Input   : hz   : Word with hex number }
{ Output  : The hex string }

```

```

{*****}
Function Hex( hz : Word ) : String;

var i   : Integer;
    hs  : String;

Begin
  hs:='';
  Repeat
    i := hz mod 16;
    hz := hz div 16;
    case i of
      0..9: hs := chr( i + ord( '0' ) ) + hs;
      10..15: hs := chr( i - 10 + ord( 'A' ) ) + hs;
    end;
  Until hz = 0;
  if length( hs ) mod 2 <> 0 then hs := '0' + hs;
  hex := hs;
End;

{*****}
{ cd_PrintSector : Display contents of a sector in character / hex. }
{ -----* }
{ Input      : lpSector - Address of sector data }
{             iSize     - Size of sector }
{             iCols     - Number of columns to be displayed }
{             iRows     - Number of rows to be displayed, after that, }

```

```

{
    wait for a key to be pressed.
}
{
    0 - Do not wait
}
{*****}
Procedure cd_PrintSector( lpSector : Pointer;
    iSize      : integer;
    iCols      : integer;
    iRows      : Integer );

var lpS      : ^byte;
    i, j, k : Integer;

Begin
    lpS := lpSector;
    k := 0;
    i := 0;
    while i < iSize do
        Begin
            for j := 0 to iCols-1 do
                if ( i + j < iSize ) then
                    Begin
                        lps := Pointer ( Longint ( lpSector ) + i + j );
                        if ( ( lpS^ >= $20 ) and ( lpS^ <= $7f ) ) then
                            Write( chr( lpS^ ) ) else Write( '.' )
                        End
                    else
                        Write( ' ' );
                End
            Write( ' ' );
        End
    End

```

```

for j := 0 to iCols-1 do
  if( i + j < iSize ) then
    begin
      lps := Pointer ( Longint ( lpSector ) + i + j );
      Write( Hex( lpS^ ) );
    end
  else
    Write( '   ' );

Writeln;
inc( i, iCols);
if ( iRows > 0 ) then inc( k );
if ( ( k = iRows ) and ( i < iSize ) ) then
  Begin
    Write( '< Key >'#13 );
    ReadKey;
    k := 0;
  End;
End;
End;

```

```

{*****}
{ cd_PrintDirEntry : Display elements of a DirEntry structure }
{ *-----* }
{ Input      : DirEntry - Structure to be displayed }
{*****}
Procedure cd_PrintDirEntry( var DirEntry : DIR_ENTRY );

```

```

var i : Integer;

Begin
  Writeln(' XAR-Length           : ', Integer ( DirEntry.XAR_len ) )
  Writeln(' Start-LBN           : ', DirEntry.loc_extent );
  Writeln(' LBN-Size             : ', DirEntry.lb_size );
  Writeln(' File-Length             : ', DirEntry.data_len );
  Writeln(' Flags                   : ', DirEntry.file_flags );
  Writeln(' Interleave-Size         : ', Integer( DirEntry.il_size ) );
  Writeln(' Interleave-Skip         : ', Integer(DirEntry.il_skip ) );
  Writeln(' Volume Set Sequence Number: ', DirEntry.VSSN );
  Writeln(' Length of filename      : ', DirEntry.len-fi );
  Write(' FileName                : ' );
  i:=0;
  While ( ( DirEntry.file_id[i] <> 0 ) and ( i < 38 ) ) do
    Begin
      Write( char ( DirEntry.file_id[i] ) );
      Inc(i);
    End;
  Writeln;
  Writeln(' Version                   : ',
    DirEntry.file_version shr 8, '.',
    DirEntry.file_version and $00FF );
  Writeln(' System data length       : ', Integer(DirEntry.len_su ) );
End;

```

End.