

```

/*****
/*          C D D E V . C          */
/**-----**/
/* Task          : Utilities for CD Device Driver programming */
/**-----**/
/* Author         : Michael Tischer / Bruno Jennrich          */
/* Developed on    : 04/08/1994                                */
/* Last update     : 04/18/1994                                */
/*                                                        */
/* Warning: Struct-Member Alignment must be set to 1 byte, since */
/*          driver doesn't operate on logical structures, but    */
/*          instead, operates on BYTE offsets!                   */
/*          CDDEV.C will not function without CDUTIL.C/OBJ!      */
/*****
/* CDDEV.C can also be #Included */
#define __CDDEV_C

#include <stdio.h>
#include <conio.h>

#include "types.h"
#include "cddev.h"
#include "cdutil.h"

/* Plaintext for Device-Status-Bits */
static char *DevStat [13][2] = {
{ "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" } };

/*****
/* _CallStrategy: Send Device-Request to device driver */
/**-----**/
/* Input   : iCD_Drive_Letter - Number of drive, to whose device */
/*                               driver a device request is to be */
/*                               sent. (0=A, 1=B, etc.) */
/*                               lpReqHdr - Address of Device-Request-Header */
/* Output   : Status of device after operation */
/*                               (see RequestHeader.wStatus) */
/**-----**/
/* Info: This function first calls the strategy, then the interrupt */
/*        function of the device driver whose header and SubUnit are */
/*        determined by the passed drive letters. */
/*        This function is called when the version number of MSCDEX */
/*        is < 2.10. */
*****/
INT _CallStrategy( INT iCD_Drive_Letter, LPRequestHeader lpReqHdr )
{ static VOID (_FP *Strategy)( VOID ); /* Function addresses */
  static VOID (_FP *Interrupt)( VOID );

```

```

static INT S, O;                                /* Segment, Offset */
static INT i;
static INT      iNumUnits = -1;
static CHAR     cCDLetters[ 26 ];               /* All CD drive letters */
static DevElement DevElements[ 26 ];

if( iNumUnits == -1 )                          /* All static variables initialized ? */
{static INT iStart, iVersion;
  MSCDEX_GetNumberOfDriveLetters( &iNumUnits, &iStart );
  MSCDEX_GetCDRomDriveDeviceList( DevElements );

  if( iVersion >= 0x200 )                      /* Does GetCDRomDriveLetters() exist? */
    MSCDEX200_GetCDRomDriveLetters( cCDLetters );
  else                                  /* otherwise sequential drive letters */
    for( i = 0; i < iNumUnits; i++ ) cCDLetters[ i ] =
                                          ( char ) ( iStart + i );
}

for( i = 0; i < iNumUnits; i++ )
if( cCDLetters[ i ] == iCD_Drive_Letter )      /* Which drive? */
{ LPDeviceHeader lpDevHdr;
  lpDevHdr = DevElements[ i ].lpDeviceHeader; /* DeviceHeader and */
  lpReqHdr->bSubUnit = DevElements[ i ].bSubUnit; /* SubUnit */
  if( FP_OFF( lpDevHdr ) ) return -1;          /* Always to Offset 0 */
  Strategy = ( VOID ( _FP * ) ( VOID ) ) MK_FP( FP_SEG( lpDevHdr ),
                                                  lpDevHdr->wStrategy );
  Interrupt = ( VOID ( _FP * ) ( VOID ) ) MK_FP( FP_SEG( lpDevHdr ),
                                                  lpDevHdr->wInterrupt );
  S = FP_SEG( lpReqHdr );                      /* Get Segment and Offset before */
  O = FP_OFF( lpReqHdr );                      /* executing asm instructions! */

  asm mov es,S                                /* Set ES:BX */

```

```

    _asm mov bx,0
    Strategy();          /* First set internal buffer addresses... */
    Interrupt();         /* ...then execute code */
    return lpReqHdr->wStatus;
}
return 0xFFFF;
}

/*****
/* 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. */
**-----*/
INT cd_IsError( WORD wStatus )
{
    return ( wStatus & DEV_ERROR ) ? TRUE : FALSE;
}

/*****
/* cd_GetReqHdrError : Get error of last device communication */
**-----*/
/* Input      : lpReqHdr - Address of Request-Header to be checked */
/*             for errors. */
/* Output     : > 0 - ErrorCode + 1 */
/*             == 0 - No error */
**-----*/
INT cd_GetReqHdrError( LPRequestHeader lpReqHdr )

```

```

{
    return cd_IsError( lpReqHdr->wStatus ) ?                /* Test Bit 15 */
        ( lpReqHdr->wStatus & 0x00FF ) + 1 : 0;
}

/*****
/* cd_IsReqHdrBusy : Is device still busy executing commands ?      */
/**-----**/
/* Input   : lpReqHdr - Address of device header                    */
/* Output  : TRUE  - Device still busy with old command             */
/*          FALSE - Device no longer busy                           */
/*****
INT cd_IsReqHdrBusy( LPRequestHeader lpReqHdr )
{
    return ( lpReqHdr->wStatus & DEV_BUSY ) ? TRUE : FALSE;    /* Bit 9 */
}

/*****
/* cd_IsReqHdrDone : Has last device command been completely      */
/*                  executed ?                                     */
/**-----**/
/* Input   : lpReqHdr - Address of Device-Header                  */
/* Output  : TRUE  - Device command has been executed             */
/*          FALSE - Device still BUSY                             */
/*****
INT cd_IsReqHdrDone( LPRequestHeader lpReqHdr )
{
    return ( lpReqHdr->wStatus & DEV_DONE ) ? TRUE : FALSE;    /* Bit 8 */
}

/*- Wrapper Functions for MSCDEX Device Commands -----*/

```

```

/*****
/* cd_GetDeviceHeaderAddress : Get address of device header */
/**-----**/
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*            lpRA                - Raddr-ControlBlock */
/* Output     : Device-Status */
/**-----**/
/* Info : After the call, lpRA->lpDeviceHeaderAddress contains */
/*        the address of the header of device driver responsible */
/*        for the specified CD drive */
/*****
INT cd_GetDeviceHeaderAddress( INT          iCD_Drive_Letter,
                              LPMSDEX_Raddr lpRA )

{
    lpRA->bControlBlockCode = IOCTLI_GET_DEVHDR_ADDRESS;          /* = 0 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                               lpRA, sizeof( LPMSDEX_Raddr ) );
}

/*****
/* cd_GetLocationOfHead : Get location of Read/Write Head */
/**-----**/
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*            lpLH                - LocHead-ControlBlock */
/* Output     : Device-Status */
/**-----**/
/* Info : Prior to the call, lpLH->bAddressingMode must contain */
/*        addressing mode (HSG(0) or REDBOOK(1)). After the call, */
/*        lpLH->lLocation contains the current location of the */
/*        Read/Write head. */
/*****
INT cd_GetLocationOfHead( INT iCD_Drive_Letter, LPMSDEX_LocHead lpLH )

```

```

{
    lpLH->bControlBlockCode = IOCTLI_GET_HEAD_LOCATION;          /* = 1 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpLH, sizeof( LPMSCDEX_Raddr ) );
}

/*****
/* cd_GetAudioChannelInfo : Get allocation of CD channels to
/*                          CD outputs
/*
/*-----**
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
/*              lpAI              - AudInfo-ControlBlock
/* Output     : Device-Status
/*-----**
/* Info : lpAI->Output0-4 and lpAI->Volume0-4 contain allocation and
/*         volume of the 4 possible channels
/*-----**
INT cd_GetAudioChannelInfo( INT          iCD_Drive_Letter,
                           LPMSCDEX_AudInfo lpAI )
{
    lpAI->bControlBlockCode = IOCTLI_GET_AUDIO_CHANNEL_INFO;      /* = 4 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpAI, sizeof( LPMSCDEX_AudInfo ) );
}

/*****
/* cd_ReadDriveBytes : Read drive bytes
/*
/*-----**
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )
/*              lpDB              - DrvBytes-ControlBlock
/* Output     : Device-Status
/*-----**

```

```

/* Info : Prior to the call, lpDB->bNumReadBytes contains the number */
/*         of bytes to be read. After the call, the read bytes are in */
/*         lpDB->bReadBuffer. If more than 128 bytes are to be read, */
/*         the previous 128 byte blocks are discarded in order to */
/*         pass the block specified by INT(NumReadBytes/128). */
/*****
INT cd_ReadDriveBytes( INT iCD_Drive_Letter, LPMSCDEX_DrvBytes lpDB )
{
    lpDB->bControlBlockCode = IOCTL_READ_DRIVE_BYTES;          /* = 5 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpDB, sizeof( LPMSCDEX_DrvBytes ) );
}

/*****
/* cd_GetDevStat : Read Device Status (drive status ) */
/**-----*/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*          lpDS                - DevStat-ControlBlock */
/* Output  : Device-Status */
/**-----*/
/* Info : After the call, lpDS->lDeviceStatus contains device status */
/*****
INT cd_GetDevStat( INT iCD_Drive_Letter, LPMSCDEX_DevStat lpDS )
{
    lpDS->bControlBlockCode = IOCTL_GET_DEV_STAT;              /* = 6 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpDS, sizeof( MSCDEX_DevStat ) );
}

/*****
/* cd_PrintDevStat : Output Device Status */
/**-----*/

```



```

/* Input      : lpDS - Device Status determined previously by GetDevStat */
/*****
VOID cd_PrintDevStat( LPMSCDEx_DevStat lpDS )
{
    INT i;
    /* Go through all the bits and output message for set/cleared */
    /* bit */
    for( i = 0; i < 13; i++ )
        printf("%s\n", DevStat[i][(lpDS->lDeviceStatus&(1<<i))?0:1]);
}

/*****
/* cd_GetSectorSize : Get size of a sector */
/**-----**
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*              lpSec             - SectSize-ControlBlock */
/*              iReadMode         - COOKED ( 0 ) or RAW ( 1 ) */
/* Output     : Device-Status */
/**-----**
/* Info : After call, lpSec->wSectorSize contains sector size (bytes) */
/*****
INT cd_GetSectorSize( INT          iCD_Drive_Letter,
                     LPMSCDEx_SectSize lpSEC,
                     INT          iReadMode )
{
    lpSEC->bControlBlockCode = IOCTLI_SECTOR_SIZE; /* = 7 */
    lpSEC->bReadMode = ( BYTE ) iReadMode;
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                               lpSEC, sizeof( MSCDEX_SectSize ) );
}

/*****
/* cd_GetVolumeSize : Get number of sectors of a CD */

```

```

/**-----**/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )      */
/*          lpVS               - VolSize-ControlBlock      */
/* Output   : Device-Status                                */
/**-----**/
/* Info : After the call, lpVS->lVolumeSize contains the number of */
/*          sectors of the inserted CD.                          */
/*****
INT cd_GetVolumeSize( INT iCD_Drive_Letter, LPMSCDEX_VolSize lpVS )
{
    lpVS->bControlBlockCode = IOCTL_VOLUME_SIZE; /* = 8 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpVS, sizeof( LPMSCDEX_VolSize ) );
}

/*****
/* cd_GetMediaChanged : CD changed?                          */
/**-----**/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )      */
/*          lpMC               - MedChng-ControlBlock      */
/* Output   : Device-Status                                */
/**-----**/
/* Info : After the call, lpMC->bMediaChanged contains the status */
/*          of the medium.                                         */
/*          1   : CD has not been changed since the last call.    */
/*          0   : CD may have been changed since the last call.  */
/*          255 : CD has been changed since the last call.       */
/*****
INT cd_GetMediaChanged( INT iCD_Drive_Letter, LPMSCDEX_MedChng lpMC )
{
    lpMC->bControlBlockCode = IOCTL_MEDIA_CHANGED; /* = 9 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,

```

```

        lpMC, sizeof( LPMSCDEX_MedChng ) );
}

/*****
/* cd_GetAudioDiskInfo : Get number of titles (songs, tracks)
/*
/* of an Audio CD.
/*-----*/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )
/*          lpDI               - DiskInfo-ControlBlock
/* Output   : Device-Status
/*-----*/
/* Info : After the call, lpDI->bLowestTrack, lpDI->bHighestTrack
/*         contain the number of the first and last song.
/*         lpDI->bLeadOutTrack contains the address of the first
/*         unused sector (after last title)
*****/
INT cd_GetAudioDiskInfo( INT iCD_Drive_Letter, LPMSCDEX_DiskInfo lpDI )
{
    lpDI->bControlBlockCode = IOCTL_AUDIO_DISK_INFO; /* = 10 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpDI, sizeof( LPMSCDEX_DiskInfo ) );
}

/*****
/* 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.
/*          lpTI              - TackInfo-ControlBlock
/* Output   : Device-Status
/*-----*/
*****/

```

```

/* Info : lpTI->lStartingPoint      - REDBOOK address of title start */
/*      lpTI->bTrackControlInfo - Information about track type      */
/*****
INT cd_GetAudioTrackInfo( INT          iCD_Drive_Letter,
                          INT          iTrackNo,
                          LPMSCDEX_TnoInfo lpTI )

{
    lpTI->bControlBlockCode = IOCTL_AUDIO_TRACK_INFO;          /* = 11 */
    lpTI->bTrackNo = ( BYTE ) iTrackNo;
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpTI, sizeof( LPMSCDEX_TnoInfo ) );
}

/*****
/* cd_IsDataTrack : Is track a pure data track, or does it
/*                  contain audio information?
/*
/*-----*/
/* Input   : lpTI - Address of a TrackInfo-ControlBlock
/*           (s. GetAudioTrackInfo)
/*
/* Output  : TRUE  - lpTI describes data track
/*           FALSE - Track contains audio data
/*
/*****
INT cd_IsDataTrack( LPMSCDEX_TnoInfo lpTI )
{
    return ( ( lpTI->bTrackControlInfo & TCI_TRACK_MASK ) ==
             TCI_DATA_TRACK ) ? TRUE : FALSE;
}

/*****
/* cd_GetTrackLen : Get size of a track (number of 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.                                     */
/*****/
LONG cd_GetTrackLen( INT iCD_Drive_Letter, INT iTno )
{
    MSCDEX_DiskInfo DI;
    MSCDEX_TnoInfo  TI;
    MSCDEX_VolSize  VS;
    LONG            lStart;
    LONG            lEnd;

    cd_GetAudioDiskInfo( iCD_Drive_Letter, &DI );
    if( ( iTno < DI.bLowestTrack ) || ( iTno > DI.bHighestTrack ) )
        return 0L; /* not a valid track */

    cd_GetAudioTrackInfo( iCD_Drive_Letter, iTno, &TI );
    lStart = REDBOOK2HSG( TI.lStartingPoint );
    if( iTno < DI.bHighestTrack ) /* Difference sequential tracks */
    {
        cd_GetAudioTrackInfo( iCD_Drive_Letter, iTno + 1, &TI );
        lEnd = REDBOOK2HSG( TI.lStartingPoint );
    }
    else /* Difference [End of CD - last track] */
    {
        cd_GetVolumeSize( iCD_Drive_Letter, &VS );
        lEnd = VS.lVolumeSize - 150L; /* Convert frames to HSG */
    }
    return lEnd - lStart;
}

```

```

}

/*****
/* cd_QueryAudioChannel : Get Online information about audio title */
/*                          currently playing */
/**-----**
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*              lpQI              - QInfo-ControlBlock */
/* Output     : Device-Status */
/**-----**
/* Info : lpQI->bTrackNo/bIndex/bTrackMin/bTrackSec/bTrackFrame */
/*        Information about song currently playing */
/*        (Number and playing time) */
/*        lpQI->bDiskMin / bDiskSec / bDiskFrame */
/*        Total playing time */
/*        This function can be called on a regular basis to get */
/*        current playing times */
*****/
INT cd_QueryAudioChannel( INT iCD_Drive_Letter, LPMSCDEX_QInfo lpQI )
{
    lpQI->bControlBlockCode = IOCTLI_AUDIO_QUERY_CHANNEL; /* = 12 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                               lpQI, sizeof( LPMSCDEX_QInfo ) );
}

/*****
/* cd_GetAudioSubChannelInfo : Get Sub-Channel information */
/**-----**
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*              lpSCI              - SubChanInfo-ControlBlock */
/* Output     : Device-Status */
/**-----**

```

[illegible]

```

/*****
/* cd_PrintUPCode : Output UPC BCD number. */
/**-----**/
/* Input      : lpUPC - UPCCode-ControlBlock */
/**-----**/
/* Info : The last nibble of the bUPC_EAN-Array is not used */
/*****
VOID cd_PrintUPCode( LPMSCEX_UPCode lpUPC )
{
    INT i;
    printf(" UPC: ");
    for( i = 0; i < 6; i++ )
        /* The first 12 digits */
        printf("%c%c", ( ( lpUPC->bUPC_EAN[ i ] >>  4 ) & 0x0F ) + '0',
                    ( ( lpUPC->bUPC_EAN[ i ] ) & 0x0F ) + '0' );
    printf("%c\n", ( ( lpUPC->bUPC_EAN[ i ] >>  4 ) & 0x0F ) + '0');
}

/*****
/* cd_GetAudioStatusInfo : Get current audio status */
/*                          (Play/Pause) */
/**-----**/
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*              lpAS              - AudStat-ControlBlock */
/* Output     : Device-Status */
/**-----**/
/* Info :      After the call, Bit 0 of the lpAS->wAudioStatus variable */
/*              gives information about the play mode of the CD drive. */
/*              (Set = Pause) */
/*              lpAS->lResumeStart contains the next position to play */
/*              for a RESUME. lpAS->lResumeEnd indicates when playback */
/*              of the song should be stopped. */
/*              (see STOP_AUDIO / PLAY_AUDIO ) */
/*****

```



```

INT cd_GetAudioStatusInfo( INT iCD_Drive_Letter, LPMSCDEX_AudStat lpAS )
{
    lpAS->bControlBlockCode = IOCTLI_AUDIO_STATUS_INFO;          /* = 15 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_READ,
                                lpAS, sizeof( LPMSCDEX_AudStat ) );
}

/*****
/* cd_Eject : Open tray                                          */
/**-----**/
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )        */
/* Output     : Device-Status                                     */
/**-----**/
/* Info : Before tray is opened, it is unlocked                */
/*****
INT cd_Eject( INT iCD_Drive_Letter )
{
    MSCDEX_Eject E;
    E.bControlBlockCode = IOCTLI_EJECT_DISK;                     /* = 0 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_WRITE,
                                ( LPVOID )&E, sizeof( MSCDEX_Eject ) );
}

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

```

```

LD.bControlBlockCode = IOCTL_LOCK_DOOR;                                /* = 1 */
LD.bLock = ( BYTE ) ( iLock ? 1 : 0 );
return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_WRITE,
                           &LD, sizeof( MSCDEX_LockDoor ) );
}

/*****
/* cd_Reset : Reset drive                                           */
/**-----**/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )               */
/* Output   : Device-Status                                         */
*****/
INT cd_ResetDrive( INT iCD_Drive_Letter )
{
    MSCDEX_Reset R;
    R.bControlBlockCode = IOCTL_RESET_DRIVE;                          /* = 2 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_WRITE,
                               ( LPVOID )&R, sizeof( MSCDEX_Reset ) );
}

/*****
/* cd_SetAudioChannelInfo : Set allocation of CD channels to       */
/*                          CD outputs                               */
/**-----**/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )               */
/*          : lpAI             - AudInfo-ControlBlock              */
/* Output   : Device-Status                                         */
*****/
/* Info : lpAI->Output0-4 and lpAI->Volume0-4 contain allocation and */
/*        volume of the 4 possible channels                         */
*****/
INT cd_SetAudioChannelInfo( INT iCD_Drive_Letter,
                           LPMSCDEX_AudInfo lpAI )

```

```

{
    lpAI->bControlBlockCode = IOCTL_SET_AUDIO_CHANNEL_INFO;          /* = 3 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_WRITE,
                                lpAI, sizeof( LPMSCDEX_AudInfo ) );
}

/*****
/* cd_WriteDriveBytes : Passes drive specific data                  */
/**-----*/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )                */
/*          lpDB               - DrvBytes-ControlBlock              */
/* Output   : Device-Status                                          */
*****/
INT cd_WriteDriveBytes( INT iCD_Drive_Letter, LPMSCDEX_DrvBytes lpDB )
{
    lpDB->bControlBlockCode = IOCTL_WRITE_DRIVE_BYTES;              /* = 4 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_WRITE,
                                lpDB, sizeof( LPMSCDEX_DrvBytes ) );
}

/*****
/* cd_CloseTray : Close tray                                        */
/**-----*/
/* Input   : iCD_Drive_Letter : Drive ID( 0=A, etc )                */
/* Output   : Device-Status                                          */
*****/
INT cd_CloseTray( INT iCD_Drive_Letter )
{
    MSCDEX_CloseTray CT;
    CT.bControlBlockCode = IOCTL_CLOSE_TRAY;                        /* = 5 */
    return MSCDEX_ReadWriteReq( iCD_Drive_Letter, DEVCMD_IOCTL_WRITE,
                                ( LPVOID )&CT, sizeof( MSCDEX_CloseTray ) );
}

```

```

/*****
/* 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. */
/*****
INT cd_PlayAudio( INT iCD_Drive_Letter,
                  INT iAdressMode,
                  LONG lStart,
                  LONG lLen )
{ MSCDEX_PlayReq PR;

    lLen = lLen > 0 ? lLen : 0L; /* 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;
    return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &PR.RegHdr );
}

```

```

}

/*****
/* 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. */
*****/
INT cd_StopAudio( INT iCD_Drive_Letter )
{
    RequestHeader ReqHdr;

    ReqHdr.bLength  = sizeof( RequestHeader );
    ReqHdr.bSubUnit = 0;
    ReqHdr.bCommand = ( BYTE )DEVCMND_STOP_AUDIO;

    return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &ReqHdr );
}

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

```

```

/*****
INT cd_ResumeAudio( INT iCD_Drive_Letter )
{ RequestHeader ReqHdr;

    ReqHdr.bLength = sizeof( RequestHeader );
    ReqHdr.bSubUnit = 0;
    ReqHdr.bCommand = ( BYTE )DEVCMND_RESUME_AUDIO;

    return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &ReqHdr );
}

/*****
/* 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). */
/*****
INT cd_Seek( INT iCD_Drive_Letter, INT iAdressingMode, LONG lSector )
{ MSCDEX_SeekReq SR;

    SR.RegHdr.bLength = sizeof( RequestHeader );
    SR.RegHdr.bSubUnit = 0;
    SR.RegHdr.bCommand = ( BYTE )DEVCMND_SEEK;

```

```

    SR.bAddressingMode    = ( BYTE ) iAddressingMode;
    SR.lpTransferAddress  = NULL;
    SR.wNumSec            = 0;
    SR.lStartingSector    = lSector;
    return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &SR.RegHdr );
}

/*****
/* 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 2352 bytes large. COOKED sectors, on      */
/*         the other hand, are only 2048 bytes.                      */
/**-----**/
INT cd_ReadLong( INT      iCD_Drive_Letter,
                 INT      iAddressMode,
                 LONG      lStart,
                 INT      iNumSec,
                 LPVOID     lpReadData,
                 INT      iDataMode )
{
    MSCDEX_ReadWriteL RWL;

    RWL.RegHdr.bLength    = sizeof( RequestHeader );
    RWL.RegHdr.bSubUnit   = 0;

```

```

RWL.RegHdr.bCommand          = ( BYTE ) DEVCMD_READLONG;

RWL.bAddressingMode          = ( BYTE ) iAdressMode;
RWL.lpTransferAddress        = lpReadData;
RWL.wNumSec                  = iNumSec;
RWL.lStartingSector          = lStart;
RWL.bDataReadWriteMode       = ( BYTE ) iDataMode;
RWL.bInterleaveSize          = 1;
RWL.bInterleaveSkipFactor    = 1;

return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &RWL.RegHdr );
}

/*****
/* cd_ReadLongPrefetch : Tells drive next sector to read.          */
/*-----*/
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )                */
/*           iAddressingMode   - HSG(0) or REDBOOK(1)              */
/*           lStart            - First sector to read                */
/* Output   : Device-Status                                          */
/*-----*/
/* Info : To compensate a little for the slow speed of the 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".                                         */
*****/
INT cd_ReadLongPrefetch( INT iCD_Drive_Letter,
                        INT iAdressMode,
                        LONG lStart )
{ MSCDEX_ReadWriteL RWL;

```



```

RWL.RegHdr.bLength      = sizeof( RequestHeader );
RWL.RegHdr.bSubUnit     = 0;
RWL.RegHdr.bCommand     = ( BYTE )DEVCMD_READLONG_PREFETCH;

RWL.bAdressingMode      = ( BYTE ) iAdressMode;
RWL.lpTransferAddress   = NULL;
RWL.wNumSec             = 1;
RWL.lStartingSector    = lStart;
RWL.bDataReadWriteMode  = 0;
RWL.bInterleaveSize    = 0;
RWL.bInterleaveSkipFactor = 0;

return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &RWL.RegHdr );
}

/*****
/* 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), Model(1), */
/*                               Mode2Form1(2), Mode2Form2(3) */
/* Output   : Device-Status */
/**-----*/
/* Info : Mode0 - Sectors are written with zeroes */
/*         Model - COOKED-Write (2048 byte sectors) */
/*         Mode2Form1 - COOKED-Write (2048 byte sectors) */
/*         Mode2Form2 - RAW-Write (2352 byte sectors) */
*****/

```

```

INT cd_WriteLong( INT      iCD_Drive_Letter,
                  INT      iAdressMode,
                  LONG      lStart,
                  INT      iNumSec,
                  LPVOID    lpWriteData,
                  INT      iDataMode )
{
    MSCDEX_ReadWriteL RWL;

    RWL.RegHdr.bLength      = sizeof( RequestHeader );
    RWL.RegHdr.bSubUnit     = 0;
    RWL.RegHdr.bCommand     = ( BYTE )DEVCMND_WRITELONG;

    RWL.bAdressingMode      = ( BYTE ) iAdressMode;
    RWL.lpTransferAddress   = lpWriteData;
    RWL.wNumSec             = iNumSec;
    RWL.lStartingSector     = lStart;
    RWL.bDataReadWriteMode  = ( BYTE ) iDataMode;
    RWL.bInterleaveSize     = 0;
    RWL.bInterleaveSkipFactor = 0;

    return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &RWL.RegHdr );
}

/*****
/* cd_WriteLongVerify : Writes and checks data on CD with data      */
/*                      in memory                                   */
/*-----**
/* Input   : iCD_Drive_Letter - Drive ID( 0=A, etc )              */
/*           iAdressingMode    - 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 : Model1:      COOKED-Verify (2048 byte sectors)      */
/*        Mode2Form1: COOKED-Verify (2048 byte sectors)      */
/*        Mode2Form2: RAW-Verify (2352 byte sectors)         */
/* **** */
/***** */
INT cd_WriteLongVerify( INT          iCD_Drive_Letter,
                        INT          iAdressMode,
                        LONG         lStart,
                        INT          iNumSec,
                        LPVOID       lpVerifyData,
                        INT          iDataMode )
{
    MSCDEX_ReadWriteL RWL;

    RWL.RegHdr.bLength      = sizeof( RequestHeader );
    RWL.RegHdr.bSubUnit     = 0;
    RWL.RegHdr.bCommand     = ( BYTE )DEVCMD_WRITELONG_VERIFY;

    RWL.bAdressingMode      = ( BYTE )iAdressMode;
    RWL.lpTransferAdress    = lpVerifyData;
    RWL.wNumSec             = iNumSec;
    RWL.lStartingSector     = lStart;
    RWL.bDataReadWriteMode  = ( BYTE )iDataMode;
    RWL.bInterleaveSize     = 0;
    RWL.bInterleaveSkipFactor = 0;

    return MSCDEX_SendDeviceRequest( iCD_Drive_Letter, &RWL.RegHdr );
}

/***** */

```

```

/* cd_PrintDiskTracks : Show all titles and their playing times      */
/**-----**/
/* Input      : iCD_Drive_Letter - Drive ID( 0=A, etc )            */
/******/
VOID cd_PrintDiskTracks( INT iCD_Drive_Letter )
{
    INT          i;
    MSCDEX_DiskInfo DI;
    MSCDEX_TnoInfo TI;

    if( !cd_IsError( cd_GetAudioDiskInfo( iCD_Drive_Letter, &DI ) ) )
        for( i = DI.bLowestTrack; i <= DI.bHighestTrack; i++ )
        {
            static INT iMin, iSec, iFrame;
            if( !cd_IsError( cd_GetAudioTrackInfo( iCD_Drive_Letter,
                                                    i, &TI ) ) )
            {
                LONG lLen;
                REDBOOK2Time( TI.lStartingPoint, &iMin, &iSec, &iFrame );
                printf( " %2d: %2d:%02d.%02d  ", i, iMin, iSec, iFrame );
                lLen = cd_GetTrackLen( iCD_Drive_Letter, i );
                Frame2Time( lLen, &iMin, &iSec, &iFrame );
                printf( " Length: %2d: %2d:%02d.%02d", i, iMin, iSec, iFrame );
                printf( " Type : " );
                switch( TI.bTrackControlInfo & TCI_TRACK_MASK )
                {
                    case TCI_DATA_TRACK: printf( "Data " ); break;
                    case TCI_4AUDIO_CHANNELS: printf( "Audio/Quadro " ); break;
                    default: printf( "Audio/Stereo " );
                }
            }
            if( TI.bTrackControlInfo & TCI_PRE_EMPHASIS )
                printf( "Pre-Emphasis " );
            if( TI.bTrackControlInfo & TCI_DIGITAL_COPY_PROHIBITED )
                printf( "Digital Copy prohibited " );
            printf( "\n" );
        }
    }
}

```

```

    }
}

/*****
/* cd_FastForward : Fast forward current audio output by 2 seconds */
/**-----**/
/* Input : iCD_Drive_Letter - Drive ID( 0=A, etc ) */
/*****
VOID cd_FastForward( INT iCD_Drive_Letter )
{
    MSCDEX_AudStat AS;
    LONG          LHSG;

    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 );
}

/*****
/* 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 */
/*          : FALSE           - CD player finished with playback */
/*****
INT cd_PrintActPlay( INT iCD_Drive_Letter )

```

```

{ INT          iStat;
  MSCDEX_QInfo  QI;

  iStat = cd_QueryAudioChannel( iCD_Drive_Letter, &QI );
  printf(" Track: %01d%01d Index: %02d  ",
        ( INT ) ( ( QI.bTrackNo & 0xF0 ) >> 4 ),
        ( INT ) ( QI.bTrackNo & 0x0F ),
        ( INT ) QI.bIndex );
  printf(" Track: %02d:%02d.%02d", ( INT ) QI.bTrackMin,
        ( INT ) QI.bTrackSec,
        ( INT ) QI.bTrackFrame );
  printf(" Disk: %02d:%02d.%02d\r", ( INT ) QI.bDiskMin,
        ( INT ) QI.bDiskSec,
        ( INT ) QI.bDiskFrame );

  return ( iStat & DEV_BUSY ) ? TRUE : FALSE;
}

/*****
/* 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                                         */
*****/
VOID cd_PrintSector( LPVOID lpSector, INT iSize, INT iCols, INT iRows )
{ LPBYTE lpS;
  INT i, j, k;

  lpS = lpSector;

```

```

k = 0;
for( i = 0; i < iSize; )
{
    for( j = 0; j < iCols; j++ )
        if( i + j < iSize )
            printf("%c", isprint( lpS[ i + j ] ) ? lpS[ i + j ] : '.' );
        else printf(" ");

    printf("  ");
    for( j = 0; j < iCols; j++ )
        if( i + j < iSize )    printf("%02X ", lpS[ i + j ] );
        else printf("    ");

    printf("\n");
    i += iCols;

    if( iRows > 0 ) k++;
    if( ( k == iRows ) && ( i < iSize ) )
    {
        printf("< Key >\r");
        _getch();
        k = 0;
    }
}

/*****
/* cd_PrintDirEntry : Display elements of a DirEntry structure          */
/*-----**/
/* Input      : lpDirEntry - Address of structure to be displayed      */
/*****
VOID cd_PrintDirEntry( LPDIR_ENTRY lpDirEntry )

```

```

{
    printf(" XAR-Length          : %d\n",
           ( INT )lpDirEntry->XAR_len );
    printf(" Start-LBN          : %ld\n",
           lpDirEntry->loc_extent );
    printf(" LBN-Size              : %d\n",
           lpDirEntry->lb_size );
    printf(" File-Length            : %ld\n",
           lpDirEntry->data_len );
    /* printf(" Date/Time                : ",
           lpDirEntry->record_time ); */
    printf(" Flags                  : %d\n",
           lpDirEntry->file_flags );
    printf(" Interleave-Size        : %d\n",
           (INT) lpDirEntry->il_size );
    printf(" Interleave-Skip        : %d\n",
           (INT) lpDirEntry->il_skip );
    printf(" Volume Set Sequence Number: %d\n",
           lpDirEntry->VSSN );
    printf(" Length of filename      : %d\n",
           lpDirEntry->len-fi );
    printf(" FileName                : %s\n",
           lpDirEntry->file_id );
    printf(" Version                 : %d.%02d\n",
           (lpDirEntry->file_version >> 8),
           lpDirEntry->file_version & 0x00FF );
    printf(" System data length      : %d\n",
           (INT) lpDirEntry->len-su );
}

#endif

```