

```

/*****
/*
/*----- D F C . C -----*/
/*
/* Tasks : Formats 3.5" and 5.25" diskettes */
/*-----*/
/*
/* Author : Michael Tischer */
/*
/* Developed on : 08/28/91 */
/*
/* Last update : 04/07/95 */
/*-----*/
/*
/* Memory model : SMALL */
/*-----*/
/*
/* Attention : Use the following when compiling in */
/*
/* Microsoft C: CL /AS /Gs DFC.C */
/*****

/*== Link include files =====*/

#include <dos.h>
#include <stdio.h>
#include <string.h>

/*== Macros =====*/

#ifdef MK_FP /* Macro MK_FP already defined? */
#undef MK_FP /* Yes, then delete macro */
#endif

#define MK_FP(seg,ofs) ((void far *) ((unsigned long) (seg)<<16|(( ofs)))
#define LO( aval ) ( ( BYTE ) ( aval & 0xFF ) )
#define HI( aval ) ( ( BYTE ) ( aval >> 8 ) )
#define SEG( p ) ( ( unsigned int ) ( ( ( long ) p ) >> 16 ) )
#define OFS( p ) ( ( unsigned int ) ( p ) )

/*== Constants =====*/

#define NOPE 0x4E /* N for No */
#define NO_DRIVE 0 /* Drive does not exist */
#define DD_525 1 /* Drive: 5.25" DD */
#define HD_525 2 /* Drive: 5.25" HD */
#define DD_35 3 /* Drive: 3.5" DD */
#define HD_35 4 /* Drive: 3.5" HD */

#define MAXNUMTRIES 5 /* Maximum number of tries */

#define TRUE ( 0 == 0 ) /* Constants, making it easy */
#define FALSE ( 1 == 0 ) /* to read the program text */

/*== Typedefs =====*/

typedef unsigned char BYTE; /* Data type byte */

typedef BYTE DDPTType[ 11 ]; /* Field for a DDPT */
typedef DDPTType *DDPTPTR; /* Pointer to a DDPT */

typedef struct { /* Physical format parameters */
    BYTE DSides, /* Number of sides */
        STrax, /* Tracks per side */
        TSEctors; /* Sectors per track */
    DDPTPTR DDPT; /* Pointer to DDPT */
} PhysDataType;

typedef struct { /* Logical format parameters */
    BYTE Media; /* Media byte */
    BYTE Cluster; /* Number of sectors per cluster */
    BYTE FAT; /* Number of sectors for the FAT */
    BYTE RootSize; /* Entries in the root directory */
} LogDataType;

typedef BYTE TrackBfType[ 18 ][ 512 ]; /* Memory for a track */

/*== Global variables =====*/

/*-- Non variable part of the BOOT sector -----*/

BYTE BootMask[ 102 ] =

```



```

void SetIntVec( int Number, void far *Pointer )
{
    *( ( void far * far * ) MK_FP( 0, Number * 4 ) ) = Pointer;
}

/*****
*/
/* GetDriveType : Determines the disk drive type. */
/* Input      : DRIVE = Drive number (0, 1 etc.) */
/* Output     : Drive code as constants (DD_525, HD_525 etc.) */
/*****
*/
BYTE GetDriveType( BYTE Drive )
{
    union REGS regs;          /* Processor registers for interrupt call */

    regs.h.ah = 0x08;          /* Function: Determine drive type */
    regs.h.dl = Drive;         /* Drive number */
    int86( 0x13, &regs, &regs ); /* Call BIOS interrupt */
    if ( regs.x.cflag )        /* Error in call? */
        return( DD_525 );     /* Fct. 0x08 does not exist => 360K XT */
    else
        return( regs.h.bl );   /* Drive type */
}

/*****
*/
/* ResetDisk : Disk reset on all drives. */
/* Input      : None */
/* Output     : None */
/* Info       : Reset executed on all drives, regardless of drive */
/*              number loaded in DD */
/*****
*/

void DiskReset( void )
{
    union REGS regs;          /* Processor registers for interrupt call */

    regs.h.ah = 0x00;          /* Function number for interrupt call */
    regs.h.dl = 0;             /* Drive a: (see Info) */
    int86( 0x13, &regs, &regs ); /* Interrupt call */
}

/*****
*/
/* GetFormatParamter: Determines the logical and physical parameters */
/*                    necessary for formatting. */
/* Input      : FORMSTRING = Pointer to string with format */
/*              "360", "720", "1200", "1440" */
/*              DRIVETYPE = Drive code, as supplied by GetDriveType() */
/*              PDATAP    = Pointer to structure that gets physical */
/*                          format parameters */
/*              LDATAP    = Pointer to structure that gets logical */
/*                          format parameters */
/* Output     : TRUE, if format is possible, otherwise FALSE */
/* Info       : New formats can be added by expanding this procedure */
/*****
*/

BYTE GetFormatParameter( char      *FormString,
                        BYTE        DriveType,
                        PhysDataType *PDataP,
                        LogDataType *LDataP )
{
    {
        static DDPTType DDPT_360 = { 0xDF, 0x02, 0x25, 0x02, 0x09, 0x2A,
                                       0xFF, 0x50, 0xF6, 0x0F, 0x08 };
        static DDPTType DDPT_1200 = { 0xDF, 0x02, 0x25, 0x02, 0x0F, 0x1B,
                                       0xFF, 0x54, 0xF6, 0x0F, 0x08 };
        static DDPTType DDPT_1440 = { 0xDF, 0x02, 0x25, 0x02, 0x12, 0x1B,
                                       0xFF, 0x6C, 0xF6, 0x0F, 0x08 };
        static DDPTType DDPT_720  = { 0xDF, 0x02, 0x25, 0x02, 0x09, 0x2A,
                                       0xFF, 0x50, 0xF6, 0x0F, 0x08 };

        static LogDataType LOG_360 = { 0xFD, 2, 2, 0x70 };
        static LogDataType LOG_1200 = { 0xF9, 1, 7, 0xE0 };
        static LogDataType LOG_720  = { 0xF9, 2, 3, 0x70 };
        static LogDataType LOG_1440 = { 0xF0, 1, 9, 0xE0 };

        static PhysDataType PHYS_360 = { 2, 40, 9, &DDPT_360 };
    }
}

```

```

static PhysDataType PHYS_1200 = { 2, 80, 15, &DDPT_1200 };
static PhysDataType PHYS_720  = { 2, 80,  9, &DDPT_720 };
static PhysDataType PHYS_1440 = { 2, 80, 18, &DDPT_1440 };

/*-- Take format from string and fill passed structures with -----*/
/*-- data -----*/

if ( strcmp( FormString, "1200" ) == 0 )          /* 1.2 Meg on 5.25"? */
    if ( DriveType == HD_525 )                    /* Format compatible with drive? */
    {
        memcpy( PDataP, &PHYS_1200, sizeof( PhysDataType ) );
        memcpy( LDataP, &LOG_1200, sizeof( LogDataType ) );
        return TRUE;                               /* End without error */
    }
    else
        return( FALSE );                          /* Drive and format incompatible */
else if ( strcmp( FormString, "360" ) == 0 )      /* 360K? */
    if ( ( DriveType == HD_525 ) || ( DriveType == DD_525 ) )
    {
        /* Format and drive compatible, set parameters */
        memcpy ( PDataP, &PHYS_360, sizeof( PhysDataType ) );
        memcpy ( LDataP, &LOG_360, sizeof ( LogDataType ) );
        return TRUE;                               /* End without error */
    }
    else
        return( FALSE );                          /* Drive and format incompatible */
else if ( strcmp( FormString, "1440" ) == 0 )     /* 1.44 Meg on 3.5"? */
    if ( DriveType == HD_35 )                     /* Format compatible with drive? */
    {
        /* Format and drive compatible, set parameters */
        memcpy ( PDataP, &PHYS_1440, sizeof( PhysDataType ) );
        memcpy ( LDataP, &LOG_1440, sizeof ( LogDataType ) );
        return TRUE;                               /* End without error */
    }
    else
        return( FALSE );                          /* Drive and format incompatible */
else if ( strcmp( FormString, "720" ) == 0 )     /* 720K on 3.5"? */
    if ( ( DriveType == HD_35 ) || ( DriveType == DD_35 ) )
    {
        /* Format and drive compatible, set parameters */
        memcpy ( PDataP, &PHYS_720, sizeof( PhysDataType ) );
        memcpy ( LDataP, &LOG_720, sizeof ( LogDataType ) );
        return TRUE;                               /* End without error */
    }
    else
        return FALSE;                             /* Drive and format incompatible */
else
    return FALSE;                                  /* Invalid format specified */
}

/*****
/* DiskPrepare: Prepare drive, set data transfer rate.
/* Input      : DRIVE = Drive number
/*            PDATA = Table with physical parameters
/* Output     : None
*****/

void DiskPrepare( BYTE Drive, PhysDataType PData )
{
    union REGS regs;          /* Processor registers for interrupt call */

    /*-- Set media type for formatting call -----*/

    regs.h.ah = 0x18;         /* Function number for interrupt call */
    regs.h.ch = PData.STrax - 1; /* Number of tracks per side */
    regs.h.cl = PData.TSectors; /* Number of sectors per track */
    regs.h.dl = Drive;         /* Drive number */
    int86( 0x13, &regs, &regs ); /* Interrupt call */
}

/*****
/* FormatTrack: Formats a track.
/* Input      : See below
/* Output     : Error status
*****/

BYTE FormatTrack( BYTE DriveNum,          /* The drive number */
                 BYTE SideNum,           /* The side number */
                 BYTE TrackF,             /* The track */

```

```

        BYTE SecPtr )          /* Number of sectors for this track */

{
    struct FormatTyp {          /* Sector information for the BIOS */
        BYTE DTrackF, DSideNum, DCounter, DLength;
    };

    BYTE          attempts;      /* Number of tries for interrupt call */
    BYTE          Counter;        /* Loop counter */
    struct FormatTyp DataField[ 18 ]; /* Maximum 18 sectors */
    void far *    dfp = DataField; /* Pointer to data field */
    union REGS    regs;          /* Processor registers for interrupt call */
    struct SREGS   sregs;        /* Segment registers */

    for ( Counter = 0; Counter < SecPtr; Counter++ )
    {
        DataField[ Counter ].DTrackF = TrackF;
        DataField[ Counter ].DSideNum = SideNum;
        DataField[ Counter ].DCounter = Counter + 1;
        DataField[ Counter ].DLength = 2;          /* 512 bytes per sector */
    }

    attempts = MAXNUMTRIES;      /* Set maximum number of tries */
    do
    {
        regs.h.ah = 5;            /* Function number for interrupt call */
        regs.h.al = SecPtr;        /* Number of sectors for a track */
        regs.x.bx = OFS( dfp );    /* Offset addr. of buffer */
        sregs.es = SEG( dfp );     /* Segment addr. */
        regs.h.dh = SideNum;        /* Side number */
        regs.h.dl = DriveNum;       /* Drive number */
        regs.h.ch = TrackF;         /* Track number */
        int86x( 0x13, &regs, &regs, &sregs ); /* Call BIOS interrupt */
        if ( regs.x.cflag )         /* Error? */
            DiskReset();
    }
    while ( ( --attempts != 0 ) && ( regs.x.cflag ) );
    return( regs.h.ah );           /* Read error status */
}

/*****
/* VerifyTrack: Verify track.
/* Input      : See below
/* Output     : Error status
*****/

BYTE VerifyTrack( BYTE DriveNum,          /* Drive number */
                  BYTE SideNum,           /* Side number */
                  BYTE TrackF,            /* Track number */
                  BYTE TSectors )         /* Number of sectors per track */

{
    BYTE          attempts;      /* Number of tries for interrupt call */
    union REGS    regs;          /* Processor registers for interrupt call */
    struct SREGS   sregs;        /* Proc. registers for extended interrupt call */
    TrackBfType    sbuf;         /* Track buffer */
    void far *    sbpPtr = sbuf;    /* FAR pointer to track buffer */

    attempts = MAXNUMTRIES;      /* Set maximum number of tries */
    do
    {
        regs.h.ah = 0x04;        /* Function number for interrupt call */
        regs.h.al = TSectors;     /* Number of sectors per track */
        regs.h.ch = TrackF;       /* Track number */
        regs.h.cl = 1;           /* Start at sector 1 */
        regs.h.dl = DriveNum;     /* Drive number */
        regs.h.dh = SideNum;      /* Side number */
        regs.x.bx = OFS( sbpPtr ); /* Offset addr. of buffer */
        sregs.es = SEG( sbpPtr ); /* Segment addr. */
        int86x( 0x13, &regs, &regs, &sregs ); /* Call BIOS interrupt */
        if ( regs.x.cflag )         /* Error? */
            DiskReset();
    }
    while ( ( --attempts != 0 ) && ( regs.x.cflag ) );
    return( regs.h.ah );           /* Read out error status */
}

```

```

/*****
/* WriteTrack: Write track.
/* Input : See below
/* Output : Error code (0=OK)
*****/

BYTE WriteTrack( BYTE DriveNum, /* Drive number */
                BYTE SideNum, /* Side number */
                BYTE TrackF, /* Track number */
                BYTE Start, /* Start at sector */
                BYTE TSectors, /* Number of sectors per track */
                void far *DaPtr ) /* Pointer to data field */

{
    BYTE attempts; /* Number of tries for interrupt call */
    union REGS regs; /* Processor registers for interrupt call */
    struct SREGS sregs; /* Proc. registers for extended interrupt call */

    attempts = MAXNUMTRIES ; /* Set maximum number of tries */
    do
    {
        regs.h.ah = 0x03; /* Function number for interrupt call */
        regs.h.al = TSectors; /* Number of sectors per track */
        regs.h.ch = TrackF; /* Track number */
        regs.h.cl = Start; /* Start at sector */
        regs.h.dl = DriveNum; /* Drive number */
        regs.h.dh = SideNum; /* Side number */
        regs.x.bx = OFS( DaPtr ); /* Offset addr. of buffer */
        sregs.es = SEG( DaPtr ); /* Segment addr. */
        int86x( 0x13, &regs, &regs, &sregs ); /* Call BIOS interrupt */
        if ( regs.x.cflag ) /* Error? */
            DiskReset();
    }
    while ( ( --attempts != 0 ) && ( regs.x.cflag ) );
    return( regs.h.ah ); /* Read out error status */
}

/*****
/* PhysicalFormat: Physical formatting of diskette (division into
/* tracks, sectors).
/* Input : See below
/* Output : Formatting ended without errors
*****/

BYTE PhysicalFormat( BYTE Drive, /* Drive number */
                   PhysDataType PData, /* Physical parameters */
                   BYTE Verify ) /* Flag for Verify */

{
    union REGS regs; /* Processor registers for interrupt call */
    BYTE attempts, /* Number of tries for interrupt call */
        TrackF, /* Loop counter: current track */
        SideNum, /* Loop counter: current side */
        Status; /* Return value of the called functions */

    /*-- Format diskette track by track -----*/

    for ( TrackF = 0; TrackF < PData.STrax; TrackF++ )
        for ( SideNum = 0; SideNum < PData.DSides; SideNum++ )
        {
            printf( "\rTrack: %d Side: %d", TrackF, SideNum );
            /*-- Maximum of 5 tries to format a track -----*/

            attempts = MAXNUMTRIES; /* Set maximum number of tries */
            do
            {
                Status = FormatTrack( Drive, SideNum, TrackF, PData.TSectors );
                if ( Status == 3 ) /* Diskette write/protected? */
                {
                    printf( "\rDiskette is write/protected \n" );
                    return FALSE; /* End procedure with error */
                }
                if ( Status == 0 && Verify )
                    Status = VerifyTrack( Drive, SideNum, TrackF, PData.TSectors );
                if ( Status > 0 ) /* Formatting unsuccessful */

```

```

        DiskReset();
    }
    while ( ( --attempts != 0 ) && ( Status != 0 ) );
    if ( Status > 0 )
    {
        printf( "\rTrack defective          \n" );
        return FALSE;
    }
    return TRUE;
}
/* Procedure ends without error */

/*****
/* LogicalFormat : Logical formatting of diskette (writing boot
/*                  sectors, FAT and root directory).
/* Input          : See below
/* Output         : TRUE, if no error occurs
*****/
BYTE LogicalFormat( BYTE Drive,          /* Drive number */
                   PhysDataType PData,   /* Physical parameters */
                   LogDataType LData )    /* Physical parameters */
{
    BYTE          i,                      /* Loop counter */
               CurSector,
               CurSide,
               CurTrack,
               Status;
    int          TotalSectors,             /* Total number of sectors */
               SecPtr;                    /* Number of sectors to be written */
    TrackBfType TrakBuffer;                /* Gets a complete track */

    memset( TrakBuffer, 0, (int) PData.TSectors * 512 ); /* Empty track */

    /*-- Boot sector: fixed part -----*/

    memcpy( TrakBuffer, BootMask, 102 ); /* Copy boot sector mask */
    memcpy( &TrakBuffer[ 0 ][ 102 ], BootMes, sizeof( BootMes) );
    TrakBuffer[ 0 ][ 510 ] = 0x55; /* End marker of boot sector */
    TrakBuffer[ 0 ][ 511 ] = 0xAA;

    /*-- Boot sector: variable part -----*/

    TotalSectors = (int) PData.STrax * (int) PData.TSectors *
                   (int) PData.DSides; /* Total number of sectors */
    TrakBuffer[ 0 ][ 13 ] = LData.Cluster; /* Cluster size */
    TrakBuffer[ 0 ][ 17 ] = LData.RootSize; /* Number entries in root dr */
    TrakBuffer[ 0 ][ 19 ] = LO( TotalSectors );
    TrakBuffer[ 0 ][ 20 ] = HI( TotalSectors );
    TrakBuffer[ 0 ][ 21 ] = LData.Media; /* Media descriptor */
    TrakBuffer[ 0 ][ 22 ] = LData.FAT; /* Size of FAT */
    TrakBuffer[ 0 ][ 24 ] = PData.TSectors; /* Sectors per track */
    TrakBuffer[ 0 ][ 26 ] = PData.DSides; /* Number of sides */

    /*-- Create FAT and FAT copy -----*/

    TrakBuffer[ 1 ][ 0 ] = LData.Media; /* Create 1st FAT */
    TrakBuffer[ 1 ][ 1 ] = 0xFF;
    TrakBuffer[ 1 ][ 2 ] = 0xFF;
    TrakBuffer[ LData.FAT + 1 ][ 0 ] = LData.Media; /* Create 2nd FAT */
    TrakBuffer[ LData.FAT + 1 ][ 1 ] = 0xFF;
    TrakBuffer[ LData.FAT + 1 ][ 2 ] = 0xFF;

    /*-- Write boot sector and FAT -----*/

    Status = WriteTrack( Drive, 0, 0, 1, PData.TSectors, TrakBuffer );
    if ( Status ) /* Error writing? */
        return FALSE; /* Yes, return error */

    /*-- Write root directory -----*/

    memset( TrakBuffer, 0, 512 ); /* Empty sector */
    CurSector = PData.TSectors; /* First track completely written */
    CurTrack = 0; /* Current track */
    CurSide = 0; /* Current side */

    /*-- Determine remaining number of sectors and write -----*/

```

```

SecPtr = LData.FAT * 2 + (LData.RootSize*32/512) + 1-PData.TSectors;

for ( i = 1; i <= SecPtr; i++ )
{
    if ( ++CurSector > PData.TSectors )          /* End of track? */
    {
        CurSector = 1;                          /* Continue with sector 1 */
        if ( ++CurSide == PData.DSides )        /* 2nd side already? */
        {
            CurSide = 0;                        /* Back to side 0 */
            CurTrack++;
        }
    }
    Status = WriteTrack( Drive, CurSide, CurTrack,
                        CurSector, 1, TrakBuffer );
    if ( Status )                                /* Error? */
        break;                                  /* Yes, leave FOR loop prematurely */
}
return ( Status == 0 );
}

/*****
/*          MAIN PROGRAM
*****/

int main( argc, argv )

int argc;          /* Number of arguments in the command line */
char *argv[];      /* Field with parameters */

{
    BYTE      CurDrive;          /* Number of drive to be formatted */
    BYTE      CurDriveType;      /* Current disk drive type */
    PhysDataType PData;          /* Physical format parameters */
    LogDataType LData;           /* Logical format parameters */
    void far *PoldDDPT;          /* Pointer to old DDPT */
    char *Param;                /* For evaluating the command line */
    BYTE      ok;                /* Flag for program flow */
    int       ExitCode;

    printf( "DFC - (c) 1992 by Michael Tischer\n\n" );

    /*-- Evaluate command line -----*/

    if ( argc > 1 )              /* Parameters specified? */
    {
        Param = argv[ 1 ];       /* Determine drive ( 0 = a:, 1 = b: ) */
        CurDrive = upcase( Param[ 0 ] ) - 65;
        CurDriveType = GetDriveType( CurDrive );    /* Current drive type */
        if ( CurDriveType > 0 )   /* Drive exist? */
            if ( GetFormatParameter( argv[ 2 ], CurDriveType, &PData, &LData ) )
            {
                DiskPrepare( CurDrive, PData );
                PoldDDPT = GetIntVec( 0x1E );        /* Store old DDPT */
                SetIntVec( 0x1E, PData.DDPT );      /* Set new DDPT */

                Param = argv[ 3 ];
                if ( ok = PhysicalFormat( CurDrive, PData,
                    (BYTE) ( upcase( Param[ 0 ] ) != 'N' ) ) )
                {
                    printf( "\rWrite boot sector and FAT \n" );
                    ok = LogicalFormat( CurDrive, PData, LData );
                }

                /*-- Evaluation of formatting process -----*/
                if ( ok )
                {
                    printf( "\rFormatting OK \n" );
                    ExitCode = 0;                    /* Program ended successfully */
                }
                else
                {
                    printf( "\rError - format cancelled \n" );
                    ExitCode = 1;                    /* Formatting cancelled with error */
                }
            }
        }
    }
}

```



```

        SetIntVec( 0x1E, POldDDPT );          /* Restore old DDPT */
    }
    else
    {
        printf( "This drive does not support that format\n" );
        ExitCode = 2;                          /* Drive and format not compatible */
    }
    else
    {
        printf( "The specified disk drive does not exist\n" );
        ExitCode = 3;                          /* Drive does not exist */
    }
}
else
{
    printf( "Call:   DFP Drive Format [ NV ]\n" );
    printf( "                \n" );
    printf( "                \n" );
    printf( "          A: or   B:        \n" );
    printf( "          \n" );
    printf( "    360, 720, 1200, 1440  \n" );
    printf( "          \n" );
    printf( "          NV = no Verify\n" );
    ExitCode = 4;                              /* Wrong call */
}
return( ExitCode );                          /* End program with return value */
}

```