Devices

COLLABORATORS			
	TITLE : Devices		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		March 28, 2025	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Devices

1.1 Amiga® RKM Devices: 14 Trackdisk Device

The Amiga trackdisk device directly drives the disk, controls the disk motors, reads raw data from the tracks, and writes raw data to the tracks. Normally, you use the AmigaDOS functions to write or read data from the disk. The trackdisk device is the lowest-level software access to the disk data and is used by AmigaDOS to access the disks. The trackdisk device supports the usual commands such as CMD_WRITE and CMD_READ. In addition, it supports an extended form of these commands to allow additional control over the trackdisk device.

```
                    NEW FEATURES FOR VERSION 2.0

                    Feature            Description
                ---------------      ---------------
                TD_GETGEOMETRY       Device Command
                TD_EJECT             Device Command
                IOTF_INDEXSYNC       Device Command Flag
                IOTF_WORDSYNC        Device Command Flag
                Fast RAM Buffers     Now Supported

   Compatibility Warning:
   ----------------------
   The new features for 2.0 are not backwards compatible.
```

1.2 14 Trackdisk Device / Trackdisk Device Commands and Functions

```
Command             Operation
--------            ----------
CMD_CLEAR           Mark track buffer as invalid. Forces the track to be
ETD_CLEAR           re-read.  ETD_CLEAR also checks for a diskchange.

CMD_READ            Read one or more sectors from a disk.  ETD_READ also
ETD_READ            reads the sector label area and checks for a diskchange.

CMD_UPDATE          Write out track buffer if it has been changed. ETD_UPDATE
ETD_UPDATE          also checks for a diskchange.

CMD_WRITE           Write one or more sectors to a disk.  ETD_WRITE also
ETD_WRITE           writes the sector label area and checks for a diskchange.

TD_ADDCHANGEINT     Add an interrupt handler to be activated on a diskchange.

TD_CHANGENUM        Return the current value of the diskchange counter used
                    by the ETD commands to determine if a diskchange has
                    occurred.

TD_CHANGESTATE      Return the disk present/not-present status of a drive.

TD_EJECT            Eject a disk from a drive.  This command will only work
                    on drives that support an eject command (V36).

TD_FORMAT           Initialize one or more tracks with a data buffer.
ETD_FORMAT          ETD_FORMAT also initializes the sector label area.

TD_GETDRIVETYPE     Return the type of disk drive in use by the unit.

TD_GETGEOMETRY      Return the disk geometry table (V36).

TD_GETNUMTRACKS     Return the number of tracks usable with the unit.

TD_MOTOR            Turn the motor on or off.  ETD_MOTOR also checks for a
ETD_MOTOR           diskchange.

TD_PROTSTATUS       Return the write-protect status of a disk.

TD_RAWREAD          Read RAW sector data from disk (unencoded MFM).
ETD_RAWREAD         ETD_RAWREAD also checks for a diskchange.

TD_RAWWRITE         Write RAW sector data to disk.  ETD_RAWWRITE also checks
ETD_RAWWRITE        for a diskchange.

TD_REMCHANGEINT     Remove a diskchange interrupt handler.

TD_SEEK             Move the head to a specific track.  ETD_SEEK also checks
                    for a

ETD_SEEK            diskchange.


Exec Functions as Used in This Chapter
--------------------------------------
AbortIO()           Abort a command to the trackdisk device.
```

```
BeginIO()        Initiate a command and return immediately (asynchronous
                 request).

CloseDevice()    Relinquish use of a disk unit.

DoIO()           Initiate a command and wait for completion (synchronous
                 request).

OpenDevice()     Obtain exclusive use of a particular disk unit.


Exec Support Functions as Used in This Chapter
----------------------------------------------
CreateExtIO()    Create an extended I/O request structure of type IOExtTD.
                 This structure will be used to communicate commands to
                 the trackdisk device.

CreatePort()     Create a signal message port for reply messages from the
                 trackdisk device.  Exec will signal a task when a message
                 arrives at the reply port.

DeleteExtIO()    Delete an I/O request structure created by CreateExtIO().

DeletePort()     Delete the message port created by CreatePort().
```

1.3 14 Trackdisk Device / Device Interface

The trackdisk device operates like other Amiga devices. To use it, you must first open the device, then send I/O requests to it, and then close it when finished. See the "Introduction to Amiga System Devices" chapter for general information on device usage.

The trackdisk device uses two different types of I/O request blocks, IOStdReq and IOExtTD and two types of commands, standard and extended. An IOExtTD is required for the extended trackdisk commands (those beginning with "ETD_"), but can be used for both types of commands. Thus, the IOExtTD is the type of I/O request that will be used in this chapter.

```
    struct IOExtTD
    {
        struct  IOStdReq iotd_Req;
        ULONG   iotd_Count;         /* Diskchange counter */
        ULONG   iotd_SecLabel;      /* Sector label data */
    };
```

See the include file devices/trackdisk.h for the complete structure definition.

The enhanced commands listed above - those beginning with "ETD_" - are similar to their standard counterparts but have additional features: they allow you to control whether a command will be executed if the disk has been changed and they allow you to read or write to the sector label portion of a sector.

Enhanced commands require a larger I/O request, IOExtTD, than the IOStdReq request used by the standard commands. IOExtTD contains extra information needed by the enhanced command; since the standard form of a command ignores the extra fields, IOExtTD requests can be used for both types. The extra information takes the form of two extra longwords at the end of the data structure. These commands are performed only if the change count is less than or equal to the value in the iotd_Count field of the command's request block.

The iotd_Count field keeps old I/O requests from being performed when the disk is changed. Any request found with an iotd_Count less than the current change counter value will be returned with a characteristic error (TDERR_DiskChange) in the io_Error field. This allows stale I/O requests to be returned to the user after a disk has been changed. The current disk-change counter value can be obtained by TD_CHANGENUM. If the user wants enhanced disk I/O but does not care about disk removal, then iotd_Count may be set to the maximum unsigned long integer value (0xFFFFFFFF).

The iotd_SecLabel field allows access to the sector identification section of the sector header. Each sector has 16 bytes of descriptive data space available to it; the trackdisk device does not interpret this data. If iotd_SecLabel is NULL, then this descriptive data is ignored. If it is not NULL, then iotd_SecLabel should point to a series of contiguous 16-byte chunks (one for each sector that is to be read or written). These chunks will be written out to the sector's label region on a write or filled with the sector's label area on a read. If a CMD_WRITE (the standard write call) is done, then the sector label area is left unchanged.

1.4 14 / Device Interface / About Amiga Floppy Disks

The standard 3.5 inch Amiga floppy disk consists of a number of tracks that are NUMSECS (11) sectors of TD_SECTOR (512) usable data bytes plus TD_LABELSIZE (16) bytes of label area. There are usually 2 tracks per cylinder (2 heads) and 80 cylinders per disk. The number of tracks can be found using the TD_GETNUMTRACKS command.

For V36 and higher systems, the NUMSECS in some drives may be variable and may change when a disk is inserted. Use TD_GETGEOMETRY to determine the current number of sectors.

```
   Think Tracks not Cylinders.
   ---------------------------
   The result is given in tracks and not cylinders. On a standard 3.5"
   drive, this gives useful space of 880K bytes plus 28K bytes of sector
   label area per floppy disk.
```

Although the disk is logically divided up into sectors, all I/O to the disk is done a track at a time. This allows access to the drive with no interleaving and increases the useful storage capacity by about 20 percent. Each disk drive on the system has its own buffer which holds the track data going to and from the drive.

Normally, a read of a sector will only have to copy the data from the track buffer. If the track buffer contains another track's data, then the buffer will first be written back to the disk (if it is "dirty") and the new track will be read in. All track boundaries are transparent to the programmer (except for FORMAT, SEEK, and RAWREAD/RAWWRITE commands) because you give the device an offset into the disk in the number of bytes from the start of the disk. The device ensures that the correct track is brought into memory.

The performance of the disk is greatly enhanced if you make effective use of the track buffer. The performance of sequential reads will be up to an order of magnitude greater than reads scattered across the disk. In addition, only full-sector writes on sector boundaries are supported.

The trackdisk device is based upon a standard device structure. It has the following restrictions:

* All reads and writes must use an io_Length that is an integer multiple of TD_SECTOR bytes (the sector size in bytes).

* The offset field must be an integer multiple of TD_SECTOR.

* The data buffer must be word-aligned.

* Under pre-V36, the data buffer must be also be in Chip RAM.

1.5 14 / Device Interface / Opening The Trackdisk Device

Three primary steps are required to open the trackdisk device:

* Create a message port by calling CreatePort(). Reply messages from the device must be directed to a message port.

* Create an extended I/O request structure of type IOExtTD. The IOExtTD structure is created by the CreateExtIO() function.

* Open the trackdisk device. Call OpenDevice(), passing it the extended I/O request.

For the trackdisk device, the flags parameter of the OpenDevice() function specifies whether you are opening a 3.5" drive (flags=0) or a 5.25" drive (flags=1). With flags set to 0 trackdisk will only open a 3.5" drive. To tell the device to open any drive it understands, set the flags parameter to TDF_ALLOW_NON_3_5. (See the include file devices/trackdisk.h for more information.)

```
   #include <devices/trackdisk.h>

   struct MsgPort *TrackMP;         /* Pointer for message port */
   struct IOExtTD *TrackIO;         /* Pointer for IORequest */

   if (TrackMP=CreatePort(0,0) )
```

```
       if (TrackIO=(struct IOExtTD *)
              CreateExtIO(TrackMP,sizeof(struct IOExtTD)) )
           if (OpenDevice(TD_NAME,0L,(struct IORequest *)TrackIO,Flags) )
               printf("%s did not open\n",TD_NAME);

   Disk Drive Unit Numbers.
   ------------------------
   The unit number - second parameter of the OpenDevice() call - can be
   any value from 0 to 3.  Unit 0 is the built-in 3.5" disk drive.
   Units 1 through 3 represent additional disk drives that may be
   connected to an Amiga system.
```

1.6 14 / Device Interface / Reading From The Trackdisk Device

You read from the trackdisk device by passing an IOExtTD to the device with CMD_READ set in io_Command, the number of bytes to be read set in io_Length, the address of the read buffer set in io_Data and the track you want to read - specified as a byte offset from the start of the disk - set in io_Offset.

The byte offset of a particular track is calculated by multiplying the number of the track you want to read by the number of bytes in a track. The number of bytes in a track is obtained by multiplying the number of sectors (NUMSECS) by the number of bytes per sector (TD_SECTOR). Thus you would multiply 11 by 512 to get 5632 bytes per track. To read track 15, you would multiply 15 by 5632 giving 84,480 bytes offset from the beginning of the disk.

```
    #define TRACK_SIZE ((LONG) (NUMSECS * TD_SECTOR))
    UBYTE *Readbuffer;
    SHORT tracknum;

    if (Readbuffer = AllocMem(TRACK_SIZE,MEMF_CLEAR|MEMF_CHIP))
        {
        DiskIO->iotd_Req.io_Length = TRACK_SIZE;
        DiskIO->iotd_Req.io_Data = (APTR)Readbuffer;
        DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track);
        DiskIO->iotd_Req.io_Command = CMD_READ;
        DoIO((struct IORequest *)DiskIO);
        }
```

For reads using the enhanced read command ETD_READ, the IOExtTD is set the same as above with the addition of setting iotd_Count to the current diskchange number. The diskchange number is returned by the TD_CHANGENUM command (see below). If you wish to also read the sector label area, you must set iotd_SecLabel to a non-NULL value.

```
    DiskIO->iotd_Req.io_Length = TRACK_SIZE;
    DiskIO->iotd_Req.io_Data = (APTR)Readbuffer;
    DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track);
    DiskIO->iotd_Count = change_count;
    DiskIO->iotd_Req.io_Command = ETD_READ;
    DoIO((struct IORequest *)DiskIO);
```

ETD_READ and CMD_READ obey all of the trackdisk device restrictions noted above. They transfer data from the track buffer to the user’s buffer. If the desired sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before a new track is read.

1.7 14 / Device Interface / Writing To The Trackdisk Device

You write to the trackdisk device by passing an IOExtTD to the device with CMD_WRITE set in io_Command, the number of bytes to be written set in io_Length, the address of the write buffer set in io_Data and the track you want to write - specified as a byte offset from the start of the disk - set in io_Offset.

```
    #define TRACK_SIZE ((LONG) (NUMSECS * TD_SECTOR))
    UBYTE *Writebuffer;

    if (Writebuffer = AllocMem(TRACK_SIZE,MEMF_CLEAR|MEMF_PUBLIC))
        {
        DiskIO->iotd_Req.io_Length = TRACK_SIZE;
        DiskIO->iotd_Req.io_Data = (APTR)Writebuffer;
        DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * tracknum);
        DiskIO->iotd_Req.io_Command = CMD_WRITE;
        DoIO((struct IORequest *)DiskIO);
        }
```

For writes using the enhanced write command ETD_WRITE, the IOExtTD is set the same as above with the addition of setting iotd_Count to the current diskchange number. The diskchange number is returned by the TD_CHANGENUM command (see below). If you wish to also write the sector label area, you must set iotd_SecLabel to a non-NULL value.

```
    DiskIO->iotd_Req.io_Length = TRACK_SIZE;
    DiskIO->iotd_Req.io_Data = (APTR)Writebuffer;
    DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * tracknum);
    DiskIO->iotd_Count = change_count;
    DiskIO->iotd_Req.io_Command = ETD_WRITE;
    DoIO((struct IORequest *)DiskIO);
```

ETD_WRITE and CMD_WRITE obey all of the trackdisk device restrictions noted above. They transfer data from the user’s buffer to the track buffer. If the track that contains this sector is already in the track buffer, no disk activity is initiated. If the desired sector is not in the buffer, the track containing that sector is automatically read in. If the data in the current track buffer has been modified, it is written out to the disk before a new track is read in for modification.

1.8 14 / Device Interface / Closing The Trackdisk Device

As with all devices, you must close the trackdisk device when you have finished using it. To release the device, a CloseDevice() call is executed

with the same IOExtTD used when the device was opened. This only closes
the device and makes it available to the rest of the system. It does not
deallocate the IOExtTD structure.

```
    CloseDevice((struct IORequest *)DiskIO);
```

1.9 14 Trackdisk Device / Advanced Commands

```
 Determining The Drive Geometry Table     Updating A Track Sector
 Clearing The Track Buffer      Formatting A Track
 Controlling The Drive Motor      Ejecting A Disk
```

1.10 14 / Advanced Commands / Determining The Drive Geometry Table

The layout geometry of a disk drive can be determined by using the
TD_GETGEOMETRY command. The layout can be defined three ways:

* TotalSectors

* Cylinders and CylSectors

* Cylinders, Heads, and TrackSectors.

Of the three, TotalSectors is the most accurate, Cylinders and CylSectors
is less so, and Cylinders, Heads and TrackSectors is the least accurate.
All are usable, though the last two may waste some portion of the
available space on some drives.

The TD_GETGEOMETRY commands returns the disk layout geometry in a
DriveGeometry structure:

```
    struct DriveGeometry
    {
        ULONG dg_SectorSize;        /* in bytes */
        ULONG dg_TotalSectors;      /* total # of sectors on drive */
        ULONG dg_Cylinders;         /* number of cylinders */
        ULONG dg_CylSectors;        /* number of sectors/cylinder */
        ULONG dg_Heads;             /* number of surfaces */
        ULONG dg_TrackSectors;      /* number of sectors/track */
        ULONG dg_BufMemType;        /* preferred buffer memory type */
                                    /* (usually MEMF_PUBLIC) */
        UBYTE dg_DeviceType;        /* codes as defined in the SCSI-2 spec*/
        UBYTE dg_Flags;             /* flags, including removable */
        UWORD dg_Reserved;
    };
```

See the include file devices/trackdisk.h for the complete structure
definition and values for the dg_DeviceType and dg_Flags fields.

You determine the drive layout geometry by passing an IOExtTD with
TD_GETGEOMETRY set in io_Command and a pointer to a DriveGeometry
structure set in io_Data.

```
   struct DriveGeometry *Euclid;

   Euclid = (struct DriveGeometry *)
             AllocMem(sizeof(struct DriveGeometry),
                      MEMF_PUBLIC | MEMF_CLEAR);

   DiskIO->iotd_Req.io_Data = Euclid;      /* put layout geometry here */
   DiskIO->iotd_Req.io_Command = TD_GETGEOMETRY;
   DoIO((struct IORequest *)DiskIO);
```

For V36 and higher versions of the operating system, TD_GETGEOMETRY is preferred over TD_GETNUMTRACKS for determining the number of tracks on a disk. This is because new drive types may have more sectors or different sector sizes, etc., than standard Amiga drives.

1.11 14 / Advanced Commands / Clearing The Track Buffer

ETD_CLEAR and CMD_CLEAR mark the track buffer as invalid, forcing a reread of the disk on the next operation. ETD_UPDATE or CMD_UPDATE would be used to force data out to the disk before turning the motor off. ETD_CLEAR or CMD_CLEAR is usually used after having locked out the trackdisk device via the use of the disk resource, when you wish to prevent the track from being updated, or when you wish to force the track to be re-read. ETD_CLEAR or CMD_CLEAR will not do an update, nor will an update command do a clear.

You clear the track buffer by passing an IOExtTD to the device with CMD_CLEAR or ETD_CLEAR set in io_Command. For ETD_CLEAR, you must also set iotd_Count to the current diskchange number.

```
   DiskIO->iotd_Req.io_Command = TD_CLEAR;
   DoIO((struct IORequest *)DiskIO);
```

1.12 14 / Advanced Commands / Controlling The Drive Motor

ETD_MOTOR and TD_MOTOR give you control of the motor. When the trackdisk device executes this command, the old state of the motor is returned in io_Actual. If io_Actual is zero, then the motor was off. Any other value implies that the motor was on. If the motor is just being turned on, the device will delay the proper amount of time to allow the drive to come up to speed. Normally, turning the drive on is not necessary - the device does this automatically if it receives a request when the motor is off.

However, turning the motor off is the programmer’s responsibility. In addition, the standard instructions to the user are that it is safe to remove a disk if, and only if, the motor is off (that is, if the disk light is off).

You control the drive motor by passing an IOExtTD to the device with CMD_MOTOR or ETD_MOTOR set in io_Command and the state you want to put the motor in set in io_Length. If io_Length is set to 1, the trackdisk device will turn on the motor; a 0 will turn it off. For ETD_MOTOR, you must also set iotd_Count to the current diskchange number.

```
   DiskIO->iotd_Req.io_Length = 1;          /* Turn on the drive motor */
   DiskIO->iotd_Req.io_Command = TD_MOTOR;
   DoIO((struct IORequest *)DiskIO);
```

1.13 14 / Advanced Commands / Updating A Track Sector

The Amiga trackdisk device does not write data sectors unless it is necessary (you request that a different track be used) or until the user requests that an update be performed. This improves system speed by caching disk operations. The update commands ensure that any buffered data is flushed out to the disk. If the track buffer has not been changed since the track was read in, the update commands do nothing.

You update a data sector by passing an IOExtTD to the device with CMD_UPDATE or ETD_UPDATE set in io_Command. For ETD_UPDATE, you must also set iotd_Count to the current diskchange number.

```
   DiskIO->iotd_Req.io_Command = TD_UPDATE;
   DoIO((struct IORequest *)DiskIO);
```

1.14 14 / Advanced Commands / Formatting A Track

ETD_FORMAT and TD_FORMAT are used to write data to a track that either has not yet been formatted or has had a hard error on a standard write command. TD_FORMAT completely ignores all data currently on a track and does not check for disk change before performing the command. The device will format the requested tracks, filling each sector with the contents of the buffer pointed to by io_Data field. You should do a read pass to verify the data.

If you have a hard write error during a normal write, you may find it possible to use the TD_FORMAT command to reformat the track as part of your error recovery process. ETD_FORMAT will write the sector label area if the iotd_SecLabel is non-NULL.

You format a track by passing an IOExtTD to the device with CMD_FORMAT or ETD_FORMAT set in io_Command, io_Data set to at least track worth of data, io_Offset field set to the byte offset of the track you want to write and the io_Length set to the length of a track. For ETD_FORMAT, you must also set iotd_Count to the current diskchange number.

```
    #define TRACK_SIZE ((LONG) (NUMSECS * TD_SECTOR))
    UBYTE *Writebuffer;

    if (WriteBuffer = AllocMem(TRACK_SIZE,MEMF_CLEAR|MEMF_CHIP))
        {
        DiskIO->iotd_Req.io_Length=TRACK_SIZE;
        DiskIO->iotd_Req.io_Data=(APTR)Writebuffer;
        DiskIO->iotd_Req.io_Offset=(ULONG)(TRACK_SIZE * track);
        DiskIO->iotd_Req.io_Command = TD_FORMAT;
        DoIO((struct IORequest *)DiskIO);
        }
```

1.15 14 / Advanced Commands / Ejecting A Disk

Certain disk drive manufacturers allow software control of disk ejection. The trackdisk device provides the TD_EJECT command to tell such drives to eject a disk.

You eject a disk by passing an IOExtTD to the device with TD_EJECT set in io_Command.

```
   DiskIO->iotd_Req.io_Command = TD_EJECT;
   DoIO((struct IORequest *)DiskIO);
```

```
   Read the Instruction Manual.
   ----------------------------
   The Commodore 3.5" drives for the Amiga and most other Amiga drive
   manufacturers do not support software disk ejects.  Attempting this
   command on those drives will result in an error condition.  Consult
   the instruction manual for your disk drive to determine whether this
   is supported.
```

1.16 14 Trackdisk Device / Disk Status Commands

Disk status commands return status on the current disk in the opened unit. These commands may be done with quick I/O and thus may be called within interrupt handlers (such as the trackdisk disk change handler). See the "Exec: Device Input/Output" chapter of the Amiga ROM Kernel Reference Manual: Libraries for more detailed information on quick I/O.

1.17 14 / Disk Status Commands / Determining The Presence Of A Disk

You determine the presence of a disk in a drive by passing an IOExtTD to the device with TD_CHANGESTATE set in io_Command. For quick I/O, you must set io_Flags to IOF_QUICK.

```
   DiskIO->iotd_Req.io_Flags = IOF_QUICK;
   DiskIO->iotd_Req.io_Command = TD_CHANGESTATE;
   BeginIO((struct IORequest *)DiskIO);
```

TD_CHANGESTATE returns the presence indicator of a disk in io_Actual. The value returned will be zero if a disk is currently in the drive and nonzero if the drive has no disk.

1.18 14 / Disk Status Commands / Determining The Write-Protect Status Of A Disk

You determine the write-protect status of a disk by passing an IOExtTD to the device with TD_PROTSTATUS set in io_Command. For quick I/O, you must set io_Flags to IOF_QUICK.

```
   DiskIO->iotd_Req.io_Flags = IOF_QUICK;
   DiskIO->iotd_Req.io_Command = TD_PROTSTATUS;
   BeginIO((struct IORequest *)DiskIO);
```

TD_PROTSTATUS returns the write-protect status in io_Actual. The value will be zero if the disk is not write-protected and nonzero if the disk is write-protected.

1.19 14 / Disk Status Commands / Determining The Drive Type

You determine the drive type of a unit by passing an IOExtTD to the device with TD_GETDRIVETYPE set in io_Command. For quick I/O, you must set io_Flags to IOF_QUICK.

```
   DiskIO->iotd_Req.io_Flags = IOF_QUICK;
   DiskIO->iotd_Req.io_Command = TD_GETDRIVETYPE;
   BeginIO((struct IORequest *)DiskIO);
```

TD_GETDRIVETYPE returns the drive type for the unit that was opened in io_Actual. The value will be DRIVE3_5 for 3.5" drives and DRIVE5_25 for 5.25" drives. The unit can be opened only if the device understands the drive type it is connected to.

1.20 14 / Disk Status Commands / Determining The Number Of Tracks Of A Drive

You determine the number of a tracks of a drive by passing an IOExtTD to the device with TD_GETNUMTRACKS set in io_Command. For quick I/O, you must set io_Flags to IOF_QUICK.

```
    DiskIO->iotd_Req.io_Flags = IOF_QUICK;
    DiskIO->iotd_Req.io_Command = TD_GETNUMTRACKS;
    BeginIO((struct IORequest *)DiskIO);
```

TD_GETNUMTRACKS returns the number of tracks on that device in io_Actual. This is the number of tracks of TD_SECTOR * NUMSECS size. It is not the number of cylinders. With two heads, the number of cylinders is half of the number of tracks. The number of cylinders is equal to the number of tracks divided by the number of heads (surfaces). The standard 3.5" Amiga drive has two heads

TD_GETGEOMETRY is the preferred over TD_GETNUMTRACKS for V36 and higher versions of the operating system especially since new drive types may have

```
more sectors or different sector sizes, etc., than standard Amiga drives.
```

1.21 14 / Disk Status Commands / Determining The Current Diskchange Number

```
You determine the current diskchange number of a disk by passing an
IOExtTD to the device with TD_CHANGENUM set in io_Command.  For quick I/O,
you must set io_Flags to IOF_QUICK.

    DiskIO->iotd_Req.io_Flags = IOF_QUICK;
    DiskIO->iotd_Req.io_Command = TD_CHANGENUM;
    BeginIO((struct IORequest *)DiskIO);

TD_CHANGENUM returns the current value of the diskchange counter (as used
by the enhanced commands) in io_Actual. The disk change counter is
incremented each time the disk is inserted or removed.

    ULONG change_count;

    DiskIO->iotd_Req.io_Flags = IOF_QUICK;
    DiskIO->iotd_Req.io_Command = TD_CHANGENUM;
    BeginIO((struct IORequest *)DiskIO);
    /* store current diskchange value */
    change_count = DiskIO->iotd_Req.io_Actual;

    DiskIO->iotd_Req.io_Length = 1;     /* Turn on the drive motor */
    DiskIO->iotd_Count = change_count;
    DiskIO->iotd_Req.io_Command = ETD_MOTOR;
    DoIO((struct IORequest *)DiskIO);
```

1.22 14 Trackdisk Device / Commands for Diagnostics and Repair

```
The trackdisk device provides commands to move the drive heads to a
specific track. These commands are provided for internal diagnostics, disk
repair, and head cleaning only.

 Moving The Drive Head To A Specific Track
```

1.23 14 / Diagnostics and Repair / Moving The Drive Head To A Specific Track

```
You move the drive head to a specific track by passing an IOExtTD to the
device with TD_SEEK or ETD_SEEK set in io_Command, and io_Offset set to
the byte offset of the track to which the seek is to occur.

    DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track);
    DiskIO->iotd_Req.io_Command = TD_SEEK;
    DoIO((struct IORequest *)DiskIO);

   Seeking is not Reading.
   -----------------------
   TD_SEEK and ETD_SEEK do not verify their position until the next
   read. That is, they only move the heads; they do not actually read
   any data.
```

1.24 14 Trackdisk Device / Notification of Disk Changes

Many programs will wish to be notified if the user has changed the disk in the active drive. While this can be done via the Intuition DISKREMOVED and DISKINSERTED messages, sometimes more tightly controlled testing is required. The trackdisk device provides commands to initiate interrupt processing when disks change.

 Adding A Diskchange Software Interrupt Handler
 Removing A Diskchange Software Interrupt Handler

1.25 14 / / Adding A Diskchange Software Interrupt Handler

The trackdisk device lets you add a software interrupt handler that will be Cause()'ed when a disk insert or remove occurs. Within the handler, you may only call the status commands that can use IOF_QUICK.

You add a software interrupt handler by passing an IOExtTD to the device with a pointer to an Interrupt structure set in io_Data, the length of the structure set in io_Length and TD_ADDCHANGEINT set in io_Command.

```
   DiskIO->iotd_Req.io_Length = sizeof(struct Interrupt)
   DiskIO->iotd_Req.io_Data   = (APTR)Disk_Interrupt;
   DiskIO->iotd_Req.io_Command = TD_ADDCHANGEINT;
   SendIO((struct IORequest *)DiskIO);

   Going, going, gone.
   -------------------
   This command does not return when executed. It holds onto the
   IORequest until the TD_REMCHANGEINT command is executed with that
   same IORequest.  Hence, you must use SendIO() with this command.
```

1.26 14 / / Removing A Diskchange Software Interrupt Handler

You remove a software interrupt handler by passing an IOExtTD to the device with a pointer to an Interrupt structure set in io_Data, the length of the structure set in io_Length and TD_REMCHANGEINT set in io_Command. You must pass it the same Interrupt structure used to add the handler.

```
    DiskIO->iotd_Req.io_Length = sizeof(struct Interrupt)
    DiskIO->iotd_Req.io_Data   = (APTR)Disk_Interrupt;
    DiskIO->iotd_Req.io_Command = TD_REMCHANGEINT;
    DoIO((struct IORequest *)DiskIO);

   Don't use with pre-V36 and earlier versions.
   --------------------------------------------
   Under pre-V36 and earlier versions of the Amiga system software,
   TD_REMCHANGEINT does not work and should not be used.  Instead, use
   the workaround listed in the "trackdisk.doc" of the Amiga ROM
   Kernel Reference Manual: Includes and Autodocs.
```

1.27 14 Trackdisk Device / Commands for Low-Level Access

The trackdisk device provides commands to read and write raw flux changes on the disk. The data returned from a low-level read or sent via a low-level write should be encoded into some form of legal flux patterns. See the Amiga Hardware Reference Manual and books on magnetic media recording and reading.

```
   Proceed at your own risk with V1.3 and earlier versions.
   --------------------------------------------------------
   In V1.3 Kickstart and earlier these functions are unreliable even
   though under certain configurations the commands may appear to work.
```

1.28 14 / Commands for Low-Level Access / Reading Raw Data From A Disk

ETD_RAWREAD and TD_RAWREAD perform a raw read from a track on the disk. They seek to the specified track and read it into the user’s buffer.

No processing of the track is done. It will appear exactly as the bits come off the disk - typically in some legal flux format (such as MFM, FM, GCR, etc; if you don’t know what these are, you shouldn’t be using this call). Caveat programmer.

This interface is intended for sophisticated programming only. You must fully understand digital magnetic recording to be able to utilize this call. It is also important that you understand that the MFM encoding scheme used by the higher level trackdisk routines may change without notice. Thus, this routine is only really useful for reading and decoding other disks such as MS-DOS formatted disks.

You read raw data from a disk by passing an IOExtTD to the device with TD_RAWREAD or ETD_RAWREAD set in io_Command, the number of bytes to be read set in io_Length (maximum 32K), a pointer to the read buffer set in io_Data, and io_Offset set to the byte offset of the track where you want to the read to begin. For ETD_RAWREAD, you must also set iotd_Count to the current diskchange number.

```
  DiskIO->iotd_Req.io_Length = 1024;          /* number of bytes to read */
  DiskIO->iotd_Req.io_Data = (APTR)Readbuffer; /* pointer to buffer */
  DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track); /* track no. */
  DiskIO->iotd_Req.io_Flags = IOTDF_INDEX      /* Set for index sync */
  DiskIO->iotd_Count = change_count;           /* diskchange number */
  DiskIO->iotd_Req.io_Command = ETD_RAWREAD;
  DoIO((struct IORequest *)DiskIO);
```

```
A raw read may be synched with the index pulse by setting the
IOTDF_INDEXSYNC flag or synched with a $4489 sync pattern by setting the
IOTDF_WORDSYNC flag.  See the "trackdisk.doc" of the Amiga ROM Kernel
Reference Manual: Includes and Autodocs for more information about these
flags.

   Forewarned is Forearmed.
   ------------------------
   Commodore-Amiga may make enhancements to the disk format in the
   future. Commodore-Amiga intends to provide compatibility within the
   trackdisk device.  Anyone who uses these raw routines is bypassing
   this upward-compatibility and does so at her own risk.
```

1.29 14 / Commands for Low-Level Access / Writing Raw Data To A Disk

```
ETD_RAWWRITE and TD_RAWWRITE perform a raw write to a track on the disk.
They seek to the specified track and write it from the user's buffer.

No processing of the track is done. It will be written exactly as the bits
come out of the buffer - typically in some legal flux format (such as
MFM, FM, GCR; if you don't know what these are, you shouldn't be using
this call). Caveat Programmer.

This interface is intended for sophisticated programming only. You must
fully understand digital magnetic recording to be able to utilize this
call. It is also important that you understand that the MFM encoding
scheme used by the higher level trackdisk routines may change without
notice. Thus, this routine is only really useful for encoding and writing
other disk formats such as MS-DOS disks.

You write raw data to a disk by passing an IOExtTD to the device with
TD_RAWRITE or ETD_RAWRITE set in io_Command, the number of bytes to be
written set in io_Length (maximum 32K), a pointer to the write buffer set
in io_Data, and io_Offset set to the byte offset of the track where you
want to the write to begin.  For ETD_RAWWRITE, you must also set
iotd_Count to the current diskchange number.

  DiskIO->iotd_Req.io_Length = 1024;         /* number of bytes to write */
  DiskIO->iotd_Req.io_Data = (APTR)Writebuffer; /* pointer to buffer */
  DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * track); /* track no. */
  DiskIO->iotd_Req.io_Flags = IOTDF_INDEX      /* Set for index sync */
  DiskIO->iotd_Count = change_count;           /* diskchange number */
  DiskIO->iotd_Req.io_Command = ETD_RAWWRITE;
  DoIO((struct IORequest *)DiskIO);

A raw read may be synched with the index pulse by setting the
IOTDF_INDEXSYNC flag or synched with a $4489 sync pattern by setting the
IOTDF_WORDSYNC flag.  See the "trackdisk.doc" of the Amiga ROM Kernel
Reference Manual: Includes and Autodocs for more information about these
flags.
```

1.30 14 / Low-Level Access Commands / Limitations For Synced Reads And Writes

```
There is a delay between the index pulse and the start of bits coming in
from the drive (e.g. dma started). It is in the range of 135-200
microseconds. This delay breaks down as follows: 55 microseconds for
software interrupt overhead (this is the time from interrupt to the write
of the DSKLEN register); 66 microsecs for one horizontal line delay
(remember that disk I/O is synchronized with Agnus' display fetches). The
last variable (0-65 microseconds) is an additional scan line since DSKLEN
is poked anywhere in the horizontal line. This leaves 15 microseconds
unaccounted for. In short, you will almost never get bits within the first
135 microseconds of the index pulse, and may not get it until 200
microseconds. At 4 microsecs/bit, this works out to be between 4 and 7
bytes of user data delay.

   Forewarned is Forearmed.
   ------------------------
   Commodore-Amiga may make enhancements to the disk format in the
   future. Commodore-Amiga intends to provide compatibility within the
   trackdisk device.  Anyone who uses these raw routines is bypassing
   this upward-compatibility and does so at her own risk.
```

1.31 14 Trackdisk Device / Trackdisk Device Errors

```
The trackdisk device returns error codes whenever an operation is
attempted.

    DiskIO->iotd_Req.io_Length = TRACK_SIZE;
    DiskIO->iotd_Req.io_Data = (APTR)Writebuffer;
    DiskIO->iotd_Req.io_Offset = (ULONG)(TRACK_SIZE * tracknum);
    DiskIO->iotd_Count = change_count;
    DiskIO->iotd_Req.io_Command = ETD_WRITE;
    if (DoIO((struct IORequest *)DiskIO))
        printf("ETD_WRITE failed.  Error: %ld\n",DiskIO-iotd.io_Error);

When an error occurs, these error numbers will be returned in the io_Error
field of your IOExtTD block.

              TRACKDISK DEVICE ERROR CODES

        Error       Value         Explanation
------------------  ----- -----------------------------
TDERR_NotSpecified   20   Error could not be determined
TDERR_NoSecHdr       21   Could not find sector header
TDERR_BadSecPreamble 22   Error in sector preamble
TDERR_BadSecID       23   Error in sector identifier
TDERR_BadHdrSum      24   Header field has bad checksum
TDERR_BadSecSum      25   Sector data field has bad checksum
TDERR_TooFewSecs     26   Incorrect number of sectors on track
TDERR_BadSecHdr      27   Unable to read sector header
TDERR_WriteProt      28   Disk is write-protected
TDERR_DiskChanged    29   Disk has been changed or is not currently present
TDERR_SeekError      30   While verifying seek position, found seek error
TDERR_NoMem          31   Not enough memory to do this operation
```

```
TDERR_BadUnitNum      32   Bad unit number (unit # not attached)
TDERR_BadDriveType    33   Bad drive type (not an Amiga 3 1/2 inch disk)
TDERR_DriveInUse      34   Drive already in use (only one task exclusive)
TDERR_PostReset       35   User hit reset; awaiting doom
```

1.32 14 Trackdisk Device / Additional Information on the Trackdisk Device

```
Additional programming information on the trackdisk device can be found in
the include files and the autodocs for the trackdisk device.  Both are
contained in the Amiga ROM Kernel Reference Manual: Includes and Autodocs.

                      Trackdisk Device Information
                  -----------------------------------
                  INCLUDES        devices/trackdisk.h
                                  devices/trackdisk.h

                  AUTODOCS        trackdisk.doc
```