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: 11 SCSI Device

```
The Small Computer System Interface (SCSI) hardware of the A3000 and
A2091/A590 is controlled by the SCSI device.  The SCSI device allows
an application to send Exec I/O commands and SCSI commands to
a SCSI peripheral.  Common SCSI peripherals include hard drives, streaming
tape units and CD-ROM drives.
```


1.2 11 SCSI Device / SCSI Device Commands and Functions

```
SCSI Device Command  Operation
-------------------  ---------
HD_SCSICMD           Issue a SCSI-direct command to a SCSI unit.


Trackdisk Device Commands Supported by the SCSI Device
------------------------------------------------------
TD_CHANGESTATE       Return the disk present/not-present status of a drive.
TD_FORMAT            Initialize one or more tracks with a data buffer.
TD_PROTSTATUS        Return the write-protect status of a disk.
TD_SEEK              Move the head to a specific track.


Exec Commands Supported by SCSI Device
--------------------------------------
CMD_READ             Read one or more sectors from a disk.
CMD_START            Restart a SCSI unit that was previously stopped with
                     CMD_STOP.
CMD_STOP             Stop a SCSI unit.
```

```
CMD_WRITE            Write one or more sectors to a disk.


Exec Functions as Used in This Chapter
--------------------------------------
AbortIO()            Abort an I/O request to the SCSI device.
AllocMem()           Allocate a block of memory.
AllocSignal()        Allocate a signal bit.
CheckIO()            Return the status of an I/O request.
CloseDevice()        Relinquish use of the SCSI device. All requests must
                     be complete before closing.

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

FreeMem()            Free a block of previously allocated memory.
FreeSignal()         Free a previously allocated signal.
OpenDevice()         Obtain use of the SCSI device.  You specify the type
                     of unit and its characteristics in the call to
                     OpenDevice().

WaitIO()             Wait for completion of an I/O request and remove it
                     from the reply port.


Exec Support Functions as Used in This Chapter
----------------------------------------------
CreateExtIO()        Create an extended IORequest structure for use in
                     communicating with the SCSI device.

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

DeleteExtIO()        Delete the extended IORequest structure created by
                     CreateExtIO().

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

1.3 11 SCSI Device / Device Interface

```
The SCSI device operates like other Amiga devices.  To use it, you must
first open the SCSI 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 power of the SCSI device comes from its special facility for passing
SCSI and SCSI-2 command blocks to any SCSI unit on the bus.  This facility
is commonly called SCSI-direct and it allows the Amiga to perform SCSI
functions that are "non-standard" in terms of the normal Amiga I/O model.

To send SCSI-direct or other commands to the SCSI device, an extended I/O
request structure named IOStdReq is used.

   struct IOStdReq
```

```
{
    struct  Message io_Message;
    struct  Device  *io_Device;/* device node pointer  */
    struct  Unit    *io_Unit;  /* unit (driver private)*/
    UWORD   io_Command;        /* device command */
    UBYTE   io_Flags;
    BYTE    io_Error;          /* error or warning num */
    ULONG   io_Actual;         /* actual number of bytes transferred */
    ULONG   io_Length;         /* requested number bytes transferred*/
    APTR    io_Data;           /* points to data area */
    ULONG   io_Offset;         /* offset for block structured devices */
};
```

See the include file exec/io.h for the complete structure definition.

Opening The SCSI Device
Closing The SCSI Device

1.4 11 / Device Interface / Opening The SCSI Device

Three primary steps are required to open the SCSI device:

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

* Create an I/O request structure of type IOStdReq. The IOStdReq structure is created by the CreateExtIO() function. CreateExtIO will initialize your IOStdReq to point to your reply port.

* Open the SCSI device. Call OpenDevice() passing it the I/O request and the SCSI unit encoded in the unit field.

SCSI unit encoding consists of three decimal digits which refer to the SCSI Target ID (bus address) in the 1s digit, the SCSI logical unit (LUN) in the 10s digit, and the controller board in the 100s digit. For example:

SCSI unit	Meaning
6	drive at address 6
12	LUN 1 on multiple drive controller at address 2
104	second controller board, address 4
88	not valid: both logical units and addresses range from 0-7

The Commodore 2090/2090A unit numbers are encoded differently. The SCSI logical unit (LUN) is in the 100s digit, and the SCSI Target ID is a permuted 1s digit: Target ID 0-6 maps to unit 3-9 (7 is reserved for the controller).

2090/A unit	Meaning
3	drive at address 0
109	drive at address 6, logical unit 1
1	not valid: this is not a SCSI unit.Perhaps it's an ST506 unit.

Some controller boards generate a unique name for the second controller board, instead of implementing the 100s digit (e.g., the 2090A’s iddisk.device).

```
    struct MsgPort *SCSIMP;      /* Message port pointer */
    struct IOStdReq *SCSIIO;     /* IORequest pointer */

        /* Create message port */
    if (!(SCSIMP = CreatePort(NULL,NULL)))
        cleanexit("Can't create message port\n",RETURN_FAIL);

        /* Create IORequest */
    if (!(SCSIIO = CreateExtIO(SCSIMP,sizeof(struct IOStdReq))))
        cleanexit("Can't create IORequest\n",RETURN_FAIL);

        /* Open the SCSI device */
    if (error = OpenDevice("scsi.device",6L,SCSIIO,0L))
        cleanexit("Can't open scsi.device\n",RETURN_FAIL);
```

In the code above, the SCSI unit at address 6 of logical unit 0 of board 0 is opened.

1.5 11 / Device Interface / Closing The SCSI Device

Each OpenDevice() must eventually be matched by a call to CloseDevice(). All I/O requests must be complete before calling CloseDevice(). If any requests are still pending, abort them with AbortIO().

```
    if (!(CheckIO(SCSIIO)))
        {
        AbortIO(SCSIIO);   /* Ask device to abort any pending requests */
        WaitIO(SCSIIO);    /* Wait for abort, then clean up */
        }
    CloseDevice(SCSIIO);   /* Close SCSI device */
```

1.6 11 SCSI Device / SCSI-Direct

SCSI-direct is the facility of the Amiga’s SCSI device interface that allows low-level SCSI commands to be passed directly to a SCSI unit on the bus. This makes it possible to support the special features of tape drives, hard disks and other SCSI equipment that do not fit into the Amiga’s normal I/O model. For example, with SCSI-direct, special commands can be sent to hard drives to modify various drive parameters that are normally inaccessible or which differ from drive to drive.

In order to use SCSI-direct, you must first open the SCSI device for the unit you want to use in the manner described above. You then send an HD_SCSICMD I/O request with a pointer to a SCSI command data structure.

The SCSI device uses a special data structure for SCSI-direct commands named SCSICmd.

```
struct SCSICmd
{
 UWORD *scsi_Data;      /* word aligned data for SCSI Data Phase */
                        /* (optional) data need not be byte aligned */
                        /* (optional) data need not be bus accessible */
 ULONG  scsi_Length;    /* even length of Data area */
                        /* (optional) data can have odd length */
                        /* (optional) data length can be > 2**24 */
 ULONG  scsi_Actual;    /* actual Data used */
 UBYTE *scsi_Command;   /* SCSI Command (same options as scsi_Data) */
 UWORD  scsi_CmdLength; /* length of Command */
 UWORD  scsi_CmdActual; /* actual Command used */
 UBYTE  scsi_Flags;     /* includes intended data direction */
 UBYTE  scsi_Status;    /* SCSI status of command */
 UBYTE *scsi_SenseData; /* sense data: filled if SCSIF_[OLD]AUTOSENSE */
                        /* is set and scsi_Status has CHECK CONDITION */
                        /* (bit 1) set */
 UWORD  scsi_SenseLength;/* size of scsi_SenseData, also bytes to */
                        /* request w/ SCSIF_AUTOSENSE, must be 4..255 */
 UWORD  scsi_SenseActual;/* amount actually fetched (0 = no sense) */
};
```

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

SCSICmd will contain the SCSI command and any associated data that you wish to pass to the SCSI unit. You set its fields to the values required by the unit and the command. When you have opened the SCSI device and set the SCSICmd to the proper values, you are ready for SCSI-direct.

You send a SCSI-direct command by passing an IOStdReq to the SCSI device with a pointer to the SCSICmd structure set in io_Data, the size of the SCSICmd structure set in io_Length and HD_SCSICMD set in io_Command.:

```
struct IOStdReq *SCSIreq = NULL;
struct SCSICmd Cmd;           /* where the actual SCSI command goes */

SCSIreq->io_Length  = sizeof(struct SCSICmd);
SCSIreq->io_Data    = (APTR)&Cmd;
SCSIreq->io_Command = HD_SCSICMD;
DoIO(SCSIreq);
```

The SCSICmd structure must be filled in prior to passing it to the SCSI unit. How it is filled in depends on the SCSI-direct being passed to the unit. Below is an example of setting up a SCSICmd structure for the MODE_SENSE SCSI-direct command.

```
UBYTE *buffer;           /* a data buffer used for mode sense data */
UBYTE  Sense[20];        /* buffer for request sense data */
struct SCSICmd Cmd;      /* where the actual SCSI command goes */

/* the command being sent */

static UBYTE ModeSense[]={ 0x1a,0,0xff,0,254,0 };

Cmd.scsi_Data = (UWORD *)buffer;     /* where we put mode sense data */
```

```
   Cmd.scsi_Length = 254;               /* how much we will accept      */
   Cmd.scsi_CmdLength = 6;              /* length of the command        */
   Cmd.scsi_Flags = SCSIF_AUTOSENSE |   /* do automatic REQUEST_SENSE   */
                    SCSIF_READ;         /* set expected data direction  */
   Cmd.scsi_SenseData = (UBYTE *)Sense; /* where sense data will go     */
   Cmd.scsi_SenseLength = 18;           /* how much we will accept      */
   Cmd.scsi_SenseActual = 0;            /* how much has been received   */

   Cmd.scsi_Command=(UBYTE *)ModeSense; /* issuing a MODE_SENSE command */
```

The fields of the SCSICmd are:

scsi_data
This field points to the data buffer for the SCSI data phase (if any is expected).It is generally the job of the driver software to ensure that the given buffer is DMA-accessible and to drop to programmed I/O if it isn't. The filing system provides a stop-gap fix for non-conforming drivers with the AddressMask parameter in DEVS:mountlist. For absolute safety, restrict all direct reads and writes to Chip RAM.

scsi_Length
This is the expected length of data to be transferred. If an unknown amount of data is to be transferred from target to host, set the scsi_Length to be larger than the maximum amount of data expected. Some controllers explicitly use scsi_Length as the amount of data to transfer. The A2091, A590 and A3000 drivers always do programmed I/O for data transfers under 256 bytes or when the DMA chip doesn't support the required alignment.

scsi_Actual
How much data was actually received from or sent to the SCSI unit in response to the SCSI-direct command.

scsi_Command
The SCSI-direct command.

scsi_CmdLength
The length of the SCSI-direct command in bytes.

scsi_CmdActual
The actual number of bytes of the SCSI-direct command that were transferred to the SCSI unit.

scsi_Flags
These flags contain the intended data direction for the SCSI command. It is not strictly necessary to set the data direction flag since the SCSI protocol will inform the driver which direction data transfers will be going. However, some controllers use this information to set up DMA before issuing the command . It can also be used as a sanity check in case the data phase goes the wrong way.

One flag in particular, is worth noting. SCSIF_AUTOSENSE is used to make the driver perform an automatic REQUEST SENSE if the target returns CHECK CONDITION for a SCSI command. The reason for having the driver do this is the multitasking nature of the Amiga.If two tasks were accessing the same drive and the first received a CHECK CONDITION, the second task would destroy the sense information when it sent a command.

SCSIF_AUTOSENSE prevents the caller from having to make two I/O requests and removes this window of vulnerability.

scsi_Status
The status of the SCSI-direct command. The values returned in this field can be found in the SCSI specification. For example, 2 is CHECK_CONDITION.

scsi_SenseActual
If the SCSIF_AUTOSENSE flag is set, it is important to initialize this field to zero before issuing a SCSI command because some drivers don't support AUTOSENSE and won't initialize the field.

scsi_SenseData
This field is used only for SCSIF_AUTOSENSE. If a REQUEST SENSE command is directly sent to the driver, the data will be deposited in the buffer pointed to by scsi_Data.

Keep in mind that SCSI-direct is geared toward an initiator role so it can't be expected to perform target-like operations. You can only send commands to a device, not receive them from an initiator. There is no provision for SCSI messaging, either. This is due mainly to the interactive nature of the extended messages (such as synchronous transfer requests) which have to be handled by the driver because it knows the limitations of the controller card and has to be made aware of such protocol changes.

1.7 11 SCSI Device / RigidDiskBlock - Fields and Implementation

The RigidDiskBlock (RDB) standard was borne out of the same development effort as HD_SCSICMD and as a result has a heavy bias towards SCSI. However, there is nothing in the RDB specification that makes it unusable for devices using other bus protocols. The XT style disks used in the A590 also support the RDB standard.

The RDB scheme was designed to allow the automatic mounting of all partitions on a hard drive and subsequent booting from the highest priority partition even if it has a soft loaded filing system. Disks can be removed from one controller and plugged into another (supporting the RDB scheme) and will carry with it all the necessary information for mounting and booting with them.

The preferred method of creating RigidDiskBlocks is with the HDToolBox program supplied by Commodore. Most controllers include an RDB editor or utility.

When a driver is initialized, it uses the information contained in the RDB to mount the required partitions and mark them as bootable if needed. The driver is also responsible for loading any filing systems that are required if they are not already available on the filesystem.resource list. File- systems are added to the resource according to DosType and version number.

The following is a listing of devices/hardblocks.h that describes all the fields in the RDB specification.

```
/*------------------------------------------------------------------
 *
 *      This file describes blocks of data that exist on a hard disk
 *      to describe that disk.  They are not generically accessable to
 *      the user as they do not appear on any DOS drive.  The blocks
 *      are tagged with a unique identifier, checksummed, and linked
 *      together.  The root of these blocks is the RigidDiskBlock.
 *
 *      The RigidDiskBlock must exist on the disk within the first
 *      RDB_LOCATION_LIMIT blocks.  This inhibits the use of the zero
 *      cylinder in an AmigaDOS partition: although it is strictly
 *      possible to store the RigidDiskBlock data in the reserved
 *      area of a partition, this practice is discouraged since the
 *      reserved blocks of a partition are overwritten by "Format",
 *      "Install", "DiskCopy", etc.  The recommended disk layout,
 *      then, is to use the first cylinder(s) to store all the drive
 *      data specified by these blocks: i.e. partition descriptions,
 *      file system load images, drive bad block maps, spare blocks,
 *      etc.
 *
 *      Though only 512 byte blocks are currently supported by the
 *      file system, this proposal tries to be forward-looking by
 *      making the block size explicit, and by using only the first
 *      256 bytes for all blocks but the LoadSeg data.
 *
 *------------------------------------------------------------------*/

/*
 *  NOTE
 *      optional block addresses below contain $ffffffff to indicate
 *      a NULL address, as zero is a valid address
 */
struct RigidDiskBlock {
  ULONG   rdb_ID;             /* 4 character identifier */
  ULONG   rdb_SummedLongs;    /* size of this checksummed structure */
  LONG    rdb_ChkSum;         /* block checksum (longword sum to zero) */
  ULONG   rdb_HostID;         /* SCSI Target ID of host */
  ULONG   rdb_BlockBytes;     /* size of disk blocks */
  ULONG   rdb_Flags;          /* see below for defines */
  /* block list heads */
  ULONG   rdb_BadBlockList;   /* optional bad block list */
  ULONG   rdb_PartitionList;  /* optional first partition block */
  ULONG   rdb_FileSysHeaderList; /* optional file system header block */
  ULONG   rdb_DriveInit;      /* optional drive-specific init code */
                              /* DriveInit(lun,rdb,ior): */
                                 /* "C" stk & d0/a0/a1 */
  ULONG   rdb_Reserved1[6];   /* set to $ffffffff */
  /* physical drive characteristics */
  ULONG   rdb_Cylinders;      /* number of drive cylinders */
  ULONG   rdb_Sectors;        /* sectors per track */
  ULONG   rdb_Heads;          /* number of drive heads */
  ULONG   rdb_Interleave;     /* interleave */
  ULONG   rdb_Park;           /* landing zone cylinder */
  ULONG   rdb_Reserved2[3];
  ULONG   rdb_WritePreComp;   /* starting cylinder: write precompensation */
  ULONG   rdb_ReducedWrite;   /* starting cylinder: reduced write current */
```

```
  ULONG   rdb_StepRate;      /* drive step rate */
  ULONG   rdb_Reserved3[5];
  /* logical drive characteristics */
  ULONG   rdb_RDBBlocksLo;   /* low block of range reserved for
                                /* hardblocks */
  ULONG   rdb_RDBBlocksHi;   /* high block of range for these hardblocks */
  ULONG   rdb_LoCylinder;    /* low cylinder of partitionable disk area */
  ULONG   rdb_HiCylinder;    /* high cylinder of partitionable data area */
  ULONG   rdb_CylBlocks;     /* number of blocks available per cylinder */
  ULONG   rdb_AutoParkSeconds; /* zero for no auto park */
  ULONG   rdb_HighRDSKBlock; /* highest block used by RDSK */
                             /* (not including replacement bad blocks) */
  ULONG   rdb_Reserved4;
  /* drive identification */
  char    rdb_DiskVendor[8];
  char    rdb_DiskProduct[16];
  char    rdb_DiskRevision[4];
  char    rdb_ControllerVendor[8];
  char    rdb_ControllerProduct[16];
  char    rdb_ControllerRevision[4];
  ULONG   rdb_Reserved5[10];
};

#define IDNAME_RIGIDDISK        0x5244534B      /* 'RDSK' */

#define RDB_LOCATION_LIMIT      16

#define RDBFB_LAST      0      /* no disks exist to be configured after */
#define RDBFF_LAST      0x01L  /*   this one on this controller */
#define RDBFB_LASTLUN   1      /* no LUNs exist to be configured */
#define RDBFF_LASTLUN   0x02L  /*   greater than this one at this SCSI */
                               /*   Target ID */
#define RDBFB_LASTTID   2      /* no Target IDs exist to be configured */
#define RDBFF_LASTTID   0x04L  /*   greater than this one on this */
                               /*   SCSI bus */
#define RDBFB_NORESELECT 3     /* don't bother trying to perform */
                               /*   reselection when talking */
#define RDBFF_NORESELECT 0x08L /*   to this drive */
#define RDBFB_DISKID    4      /* rdb_Disk... identification valid */
#define RDBFF_DISKID    0x10L
#define RDBFB_CTRLRID   5      /* rdb_Controller...identification valid */
#define RDBFF_CTRLRID   0x20L
                               /* added 7/20/89 by commodore: */
#define RDBFB_SYNCH     6      /* drive supports scsi synchronous mode */
#define RDBFF_SYNCH     0x40L  /* DANGEROUS TO USE IF IT DOESN'T! */

/*---------------------------------------------------------------*/
struct BadBlockEntry {
    ULONG   bbe_BadBlock;      /* block number of bad block */
    ULONG   bbe_GoodBlock;     /* block number of replacement block */
};

struct BadBlockBlock {
    ULONG   bbb_ID;            /* 4 character identifier */
    ULONG   bbb_SummedLongs;   /* size of this checksummed structure */
    LONG    bbb_ChkSum;        /* block checksum (longword sum to zero) */
    ULONG   bbb_HostID;        /* SCSI Target ID of host */
```

```
    ULONG   bbb_Next;          /* block number of the next BadBlockBlock */
    ULONG   bbb_Reserved;
    struct BadBlockEntry bbb_BlockPairs[61]; /* bad block entry pairs */
    /* note [61] assumes 512 byte blocks */
};

#define IDNAME_BADBLOCK         0x42414442      /* 'BADB' */

/*--------------------------------------------------------------------*/
struct PartitionBlock {
    ULONG   pb_ID;             /* 4 character identifier */
    ULONG   pb_SummedLongs;    /* size of this checksummed structure */
    LONG    pb_ChkSum;         /* block checksum (longword sum to zero) */
    ULONG   pb_HostID;         /* SCSI Target ID of host */
    ULONG   pb_Next;           /* block number of the next PartitionBlock */
    ULONG   pb_Flags;          /* see below for defines */
    ULONG   pb_Reserved1[2];
    ULONG   pb_DevFlags;       /* preferred flags for OpenDevice */
    UBYTE   pb_DriveName[32]; /* preferred DOS device name: BSTR form */
                               /* (not used if this name is in use) */
    ULONG   pb_Reserved2[15]; /* filler to 32 longwords */
    ULONG   pb_Environment[17]; /* environment vector for this partition */
    ULONG   pb_EReserved[15]; /* reserved for future environment vector */
};

#define IDNAME_PARTITION       0x50415254      /* 'PART' */

#define PBFB_BOOTABLE   0     /* this partition intended to be bootable */
#define PBFF_BOOTABLE   1L    /*  (expected directories and files exist) */
#define PBFB_NOMOUNT    1     /* do not mount this partition (manually */
#define PBFF_NOMOUNT    2L    /*   mounted, but space reserved here) */

/*--------------------------------------------------------------------*/
struct FileSysHeaderBlock {
    ULONG   fhb_ID;            /* 4 character identifier */
    ULONG   fhb_SummedLongs; /* size of this checksummed structure */
    LONG    fhb_ChkSum;        /* block checksum (longword sum to zero) */
    ULONG   fhb_HostID;        /* SCSI Target ID of host */
    ULONG   fhb_Next;          /* block number of next FileSysHeaderBlock */
    ULONG   fhb_Flags;         /* see below for defines */
    ULONG   fhb_Reserved1[2];
    ULONG   fhb_DosType;       /* file system description: match this with */
                               /* partition environment's DE_DOSTYPE entry */
    ULONG   fhb_Version;       /* release version of this code */
    ULONG   fhb_PatchFlags;    /* bits set for those of the following that */
                               /*  need to be substituted into a standard */
                               /*  device node for this file system: e.g. */
                               /*  0x180 to substitute SegList & GlobalVec */
    ULONG   fhb_Type;          /* device node type: zero */
    ULONG   fhb_Task;          /* standard dos "task" field: zero */
    ULONG   fhb_Lock;          /* not used for devices: zero */
    ULONG   fhb_Handler;       /* filename to loadseg: zero placeholder */
    ULONG   fhb_StackSize;     /* stacksize to use when starting task */
    LONG    fhb_Priority;      /* task priority when starting task */
    LONG    fhb_Startup;       /* startup msg: zero placeholder */
    LONG    fhb_SegListBlocks; /* first of linked list of LoadSegBlocks: */
                               /*   note that this entry requires some */
```

```
                             /*   processing before substitution */
    LONG    fhb_GlobalVec;   /* BCPL global vector when starting task */
    ULONG   fhb_Reserved2[23];/* (those reserved by PatchFlags) */
    ULONG   fhb_Reserved3[21];
};

#define IDNAME_FILESYSHEADER    0x46534844      /* 'FSHD' */

/*------------------------------------------------------------------*/
struct LoadSegBlock {
    ULONG   lsb_ID;             /* 4 character identifier */
    ULONG   lsb_SummedLongs;    /* size of this checksummed structure */
    LONG    lsb_ChkSum;         /* block checksum (longword sum to zero) */
    ULONG   lsb_HostID;         /* SCSI Target ID of host */
    ULONG   lsb_Next;           /* block number of the next LoadSegBlock */
    ULONG   lsb_LoadData[123];  /* data for "loadseg" */
    /* note [123] assumes 512 byte blocks */
};

#define IDNAME_LOADSEG
```

How A Driver Uses RDB
Alien Filing Systems

1.8 11 / RigidDiskBlock-Fields and Implementation / How A Driver Uses RDB

The information contained in the RigidDiskBlock and subsequent PartitionBlocks, et al., is used by a driver in the following manner.

After determining that the target device is a hard disk (using the SCSI-direct command INQUIRY), the driver will scan the first RDB_LOCATION_LIMIT (16) blocks looking for a block with the "RDSK" identifier and a correct sum-to-zero checksum. If no RDB is found then the driver will give up and not attempt to mount any partitions for this unit. If the RDB is found then the driver looks to see if there's a partition list for this unit (rdb_PartitionList). If none, then just the rdb_Flags will be used to determine if there are any LUNs or units after this one. This is used for early termination of the search for units on bootup.

If a partition list is present, and the partition blocks have the correct ID and checksum, then for each partition block the driver does the following.

1. Checks the PBFB_NOMOUNT flag. If set then this partition is just reserving space. Skip to the next partition without mounting the current one.

2. If PBFB_NOMOUNT is false, then the partition is to be mounted. The driver fetches the given drive name from pb_DriveName. This name will be of the form dh0, work, wb_2.x etc. A check is made to see if this name already exists on eb_MountList or DOS's device list. If it does, then the name is algorithmically altered to remove duplicates. The A590, A2091 and A3000 append .n (where n is a number) unless another name ending with .n is found. In that case the name is

changed to .n+1 and the search for duplicates is retried.

3. Next the driver constructs a parameter packet for MakeDosNode() using the (possibly altered) drive name and information about the Exec device name and unit number. MakeDosNode() is called to create a DOS device node. MakeDosNode() constructs a filesystem startup message from the given information and fills in defaults for the ROM filing system.

4. If MakeDosNode() succeeds then the driver checks to see if the entry is using a standard ("DOS\0") filing system. If not then the routine for patching in non-standard filing systems is called (see "Alien File Systems" below).

5. Now that the DOS node has been set up and the correct filing system segment has been associated with it, the driver checks PBFB_BOOTABLE to see if this partition is marked as bootable. If the partition is not bootable, or this is not autoboot time (DiagArea == 0) then the driver simply calls AddDosNode() to enqueue the DOS device node. If the partition is bootable, then the driver constructs a boot node and enqueues it on eb_MountList using the boot priority from the environment vector. If this boot priority is -128 then the partition is not considered bootable.

1.9 11 / RigidDiskBlock - Fields and Implementation / Alien Filing Systems

When a filing system other than the ROM filing system is to be used, the following steps take place.

1. First, open filesystem.resource in preparation for finding the filesystem segment we want. If filesystem.resource doesn't exist then create it and add it via AddResource(). Under 2.0 the resource is created by the system early on in the initialization sequence. Under pre-V36 Kickstart, it is the responsibility of the first RDB driver to create it.

2. Scan filesystem.resource looking for a filesystem that matches the DosType and version that we want. If it exists go to step 4.

3. Since the driver couldn't find the filesystem it needed, it will have to load it from the RDB area. The list of FileSysHeaderBlocks (pointed to by the "RDSK" block) is scanned for a filesystem of the required DosType and version. If none is found then the driver will give up and abort the mounting of the partition. If the required filesystem is found, then it is LoadSeg()'ed from the "LSEG" blocks and added as a new entry to the filesystem.resource.

4. The SegList pointer of the found or loaded filesystem is held in the FileSysEntry structure (which is basically an environment vector for this filing system). Using the patch flags, the driver now patches the newly created environment vector (pointed to by the new DosNode) using the values in the FileSysEntry being used. This ensures that the partition will have the correct filing system set up with the correct mount variables using a shared SegList.

The eb_Mountlist will now be set up with prioritized bootnodes and maybe some non-bootable, but mounted partitions. The system bootstrap will now take over.

1.10 11 SCSI Device / Amiga BootStrap

At priority -40 in the system module initialization sequence, after most other modules are initialized, appropriate expansion boards are configured. Appropriate boards will match a FindConfigDev(, -1,-1) - these are all boards on the expansion library board list. Furthermore, they will meet all of the following conditions:

1. CDB_CONFIGME set in cd_Flags,
2. ERTB_DIAGVALID set in cd_Rom->er_Type,
3. diagnostic area pointer (in cd_Rom->er_Reserved0c) is non-zero,
4. DAC_CONFIGTIME set in da_Config, and
5. at least one valid resident tag within the diagnostic area, the first of which is used by InitResident() below. This resident structure was patched to be valid during the ROM diagnostic routine run when the expansion library first initialized the board.

Boards meeting all these conditions are initialized with the standard InitResident() mechanism, with a NULL SegList. The board initialization code can find its ConfigDev structure with the expansion library's GetCurrentBinding() function. This is an appropriate time for drivers to Enqueue() a boot node on the expansion library's eb_MountList for use by the strap module below, and clear CDB_CONFIGME so a C:BindDrivers command will not try to initialize the board a second time.

This module will also enqueue nodes for 3.5" trackdisk device units. These nodes will be at the following priorities:

Priority	Drive
5	df0:
-10	df1:
-20	df2:
-30	df3:

Next, at priority -60 in the system module initialization sequence, the strap module is invoked. Nodes from the prioritized eb_MountList list is used in priority order in attempts to boot. An item on the list is given a chance to boot via one of two different mechanisms, depending on whether it it uses boot code read in off the disk (BootBlock booting), or uses boot code provided in the device ConfigDev diagnostic area (BootPoint booting). Floppies always use the BootBlock method. Other entries put on the eb_MountList (e.g. hard disk partitions) used the BootPoint mechanism for pre-V36 Kickstart, but can use either for V36/V37.

The eb_MountList is modified before each boot attempt, and then restored and re-modified for the next attempt if the boot fails:

1. The node associated with the current boot attempt is placed at the head of the eb_MountList.

2. Nodes marked as unusable under AmigaDOS are removed from the list. Nodes that are unusable are marked by the longword bn_DeviceNode->dn_Handler having the most significant bit set. This is used, for example, to keep UNIX partitions off the AmigaDOS device list when booting AmigaDOS instead of UNIX.

The selection of which of the two different boot mechanisms to use proceeds as follows:

1. The node must be valid boot node, i.e. meet both of the following conditions:

   a) ln_Type is NT_BOOTNODE,
   b) bn_DeviceNode is non-zero,

2. The type of boot is determined by looking at the DosEnvec pointed to by fssm_Environ pointed to by the dn_Startup in the bn_DeviceNode:

   a) if the de_TableSize is less than DE_BOOTBLOCKS, or the de_BootBlocks entry is zero, BootPoint booting is specified, otherwise

   b) de_BootBlocks contains the number of blocks to read in from the beginning of the partition, checksum, and try to boot from.

Bootblock Booting
Bootpoint Booting

1.11 11 / Amiga BootStrap / Bootblock Booting

In BootBlock booting the sequence of events is as follows:

1. The disk device must contain valid boot blocks:

   a) the device and unit from dn_Startup opens successfully,

   b) memory is available for the <de_BootBlocks> * <de_SizeBlock> * 4 bytes of boot block code,

   c) the device commands CMD_CLEAR, TD_CHANGENUM, and CMD_READ of the boot blocks execute without error,

   d) the boot blocks start with the three characters "DOS" and pass the longword checksum (with carry wraparound), and

   e) memory is available to construct a boot node on the eb_MountList to describe the floppy. If a device error is reported in 1.c., or if memory is not available for 1.b. or 1.e., a recoverable alert is presented before continuing.

2. The boot code in the boot blocks is invoked as follows:

```
        a) The address of the entry point for the boot code is offset
           BB_ENTRY into the boot blocks in memory.

        b) The boot code is invoked with the I/O request used to issue the
           device commands in 1.c. above in register A1, with the io_Offset
           pointing to the beginning of the partition (the origin of the
           boot blocks) and SysBase in A6.

    3. The boot code returns with results in both D0 and A0.

        a) Non-zero D0 indicates boot failure.  The recoverable alert
           AN_BootError is presented before continuing.

        b) Zero D0 indicates A0 contains a pointer to the function to
           complete the boot.  This completion function is chained to with
           SysBase in A6 after the strap module frees all its resources.
           It is usually the dos.library initialization function, from the
           dos.library resident tag.  Return from this function is
           identical to return from the strap module itself.
```

1.12 11 / Amiga BootStrap / Bootpoint Booting

```
BootPoint booting follows this sequence:

    1. The eb_MountList node must contain a valid BootPoint:

        a) ConfigDev pointer (in ln_Name) is non-zero,

        b) diagnostic area pointer (in cd_Rom er_Reserved0c) is non-zero,

        c) DAC_CONFIGTIME set in da_Config.

    2. The boot routine of a valid boot node is invoked as follows:

        a) The address of the boot routine is calculated from da_BootPoint.

        b) The resulting boot routine is invoked with the ConfigDev pointer
           on the stack in C fashion (i.e., (*boot)(configDev);). Moreover,
           register A2 will contain the address of the associated
           eb_MountList node.

    3. Return from the boot routine indicates failure to boot.

If all entries fail to boot, the user is prompted to put a bootable disk
into a floppy drive with the "strap screen".  The system floppy drives
are polled for new disks.  When one appears, the "strap screen" is removed
and the appropriate boot mechanism is applied as described above.  The
process of prompting and trying continues till a successful boot occurs.
```

1.13 11 SCSI Device / Additional Information on the SCSI Device

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

For information on the SCSI commands, see either the ANSI-X3T9 (draft SCSI-2) or ANSI X3.131 (SCSI-1) specification. The NCR SCSI BBS - phone number (316)636-8700 (2400 baud) - has electronic copies of the current SCSI specifications.

```
      SCSI Device Information
-------------------------------------
INCLUDES        devices/scsidisk.h
                devices/scsidisk.i
                devices/hardblocks.h
                devices/hardblocks.i
```