

Libraries

COLLABORATORS

	TITLE : Libraries		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		July 23, 2024	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	Libraries	1
1.1	Amiga® RKM Libraries: 32 Expansion Library	1
1.2	32 Expansion Library / AUTOCONFIG(TM)	1
1.3	32 Expansion Library / The Expansion Sequence	2
1.4	32 / The Expansion Sequence / Simple Expansion Library Example	3
1.5	32 Expansion Library / Expansion Board Drivers	3
1.6	32 / Expansion Board Drivers / Disk Based Drivers	4
1.7	32 / Expansion Board Drivers / Expansion Drivers and DOS	5
1.8	32 / Expansion Board Drivers / ROM Based and Autoboot Drivers	6
1.9	32 // ROM Based and Autoboot Drivers / Events At DIAG Time	7
1.10	32 // ROM Based and Autoboot Drivers / Events At ROMTAG INIT Time	11
1.11	32 // ROM Based and Autoboot Drivers / Events At BOOT Time	12
1.12	32 / Expansion Board Drivers / RigidDiskBlock and Alternate Filesystems	12
1.13	32 // RigidDiskBlock and Alternate Filesystems / RigidDiskBlock	14
1.14	32 // RigidDiskBlock and Alternate Filesystems / BadBlockBlock	17
1.15	32 // RigidDiskBlock and Alternate Filesystems / PartitionBlock	17
1.16	32 // RigidDiskBlock and Alternate Filesystems / FileSysHeaderBlock	19
1.17	32 // RigidDiskBlock and Alternate Filesystems / LoadSegBlock	20
1.18	32 // RigidDiskBlock and Alternate Filesystems / filesysres.h and .i	20
1.19	32 Expansion Library / Function Reference	21

Chapter 1

Libraries

1.1 Amiga® RKM Libraries: 32 Expansion Library

Amiga RAM expansion boards and other expansion bus peripherals are designed to reside at dynamically assigned address spaces within the system. The configuration and initialization of these expansion peripherals is performed by the `expansion.library`.

AUTOCONFIG(TM)	Expansion Board Drivers
The Expansion Sequence	Function Reference

1.2 32 Expansion Library / AUTOCONFIG(TM)

The Amiga AUTOCONFIG protocol is designed to allow the dynamic assignment of available address space to expansion boards, eliminating the need for user configuration via jumpers. Such expansion boards include memory boards, hard disk controllers, network interfaces, and other special purpose expansion devices. Some expansion devices, such as RAM boards, require no special driver software. Other types of expansion devices may use a disk-loaded driver from the `DEVS:` or `SYS:Expansion` drawer, or an on-board ROM driver (for example, a self-booting hard disk controller).

This chapter will concentrate on the software and driver side of Zorro expansion devices, using a Zorro II device as an example. Zorro III devices have additional identifying bits and memory size options which are described in the Zorro III hardware documentation. For more information on Zorro II and Zorro III expansion hardware, see the "Zorro Expansion Bus" appendix of the Amiga Hardware Reference Manual, 3rd Edition from Addison-Wesley. For additional information specific to Zorro II boards, see the Commodore publication A500/A2000 Technical Reference Manual.

AUTOCONFIG occurs whenever the Amiga is powered on or reset. During early system initialization, `expansion.library` identifies the expansion boards that are installed in the Amiga and dynamically assigns an appropriate address range for each board to reside at. During this AUTOCONFIG process, each expansion board first appears in turn at `$E80000` (Zorro II) or `$FF000000` (Zorro III), presenting readable identification information, generally in a PAL or a ROM, at the beginning of the board. The

identification includes the size of the board, its address space preferences, type of board (memory or other), and a unique Hardware Manufacturer Number assigned by Commodore Applications and Technical Support (CATS), West Chester, Pennsylvania.

The unique Hardware Manufacturer number, in combination with a vendor-supplied product number, provides a way for boards to be identified and for disk-based drivers to be matched with expansion boards. All expansion boards for the Amiga must implement the AUTOCONFIG protocol.

Note:

A unique Hardware Manufacturer number assigned by CATS is not the same as a Developer number.

1.3 32 Expansion Library / The Expansion Sequence

During system initialization, `expansion.library` configures each expansion peripheral in turn by examining its identification information and assigning it an appropriate address space. If the board is a RAM board, it can be added to the system memory list to make the RAM available for allocation by system tasks.

Descriptions of all configured boards are kept in a private `ExpansionBase` list of `ConfigDev` structures. A board's identification information is stored in the `ExpansionRom` sub-structure contained within the `ConfigDev` structure. Applications can examine individual or all `ConfigDev` structures with the `expansion.library` function `FindConfigDev()`.

The `ConfigDev` structure is defined in `<libraries/configvars.h>` and `<.i>`:

```
struct ConfigDev {
    struct Node      cd_Node;
    UBYTE           cd_Flags;      /* (read/write) */
    UBYTE           cd_Pad;        /* reserved */
    struct ExpansionRom cd_Rom;     /* copy of board's expansion ROM */
    APTR            cd_BoardAddr; /* memory where board was placed */
    ULONG           cd_BoardSize; /* size of board in bytes */
    UWORD           cd_SlotAddr;   /* which slot number (PRIVATE) */
    UWORD           cd_SlotSize;   /* number of slots (PRIVATE) */
    APTR            cd_Driver;     /* pointer to node of driver */
    struct ConfigDev *cd_NextCD;   /* linked list of drivers to config */
    ULONG           cd_Unused[4]; /* for whatever the driver wants */
};

/* cd_Flags */
#define CDB_SHUTUP      0      /* this board has been shut up */
#define CDB_CONFIGME    1      /* this board needs a driver to claim it */

#define CDF_SHUTUP      0x01
#define CDF_CONFIGME    0x02
```

The `ExpansionRom` structure within `ConfigDev` contains the board

identification information that is read from the board's PAL or ROM at expansion time. The actual onboard identification information of a Zorro II board appears in the high nibbles of the first \$40 words at the start of the board. Except for the first nibble pair (\$00/\$02) which when combined form `er_Type`, the information is stored in inverted ("ones-complement") format where binary 1's are represented as 0's and 0's are represented as 1's. The `expansion.library` reads the nibbles of expansion information from the board, un-inverts them (except for \$00/\$02 `er_Type` which is already un-inverted), and combines them to form the elements of the `ExpansionRom` structure.

The `ExpansionRom` structure is defined in `<libraries/configregs.h>` and `<.i>`:

```
struct ExpansionRom {
    UBYTE  er_Type;          /* First 16 bytes of the expansion ROM */
    UBYTE  er_Product;       /* Board type, size and flags */
    UBYTE  er_Flags;         /* Product number, assigned by manufacturer */
    UBYTE  er_Reserved03;    /* Flags */
    UWORD  er_Manufacturer;   /* Must be zero ($ff inverted) */
    ULONG  er_SerialNumber;  /* Unique ID, ASSIGNED BY COMMODORE-AMIGA! */
    UWORD  er_InitDiagVec;   /* Available for use by manufacturer */
    UBYTE  er_Reserved0c;    /* Offset to optional "DiagArea" structure */
    UBYTE  er_Reserved0d;
    UBYTE  er_Reserved0e;
    UBYTE  er_Reserved0f;
};
```

Simple Expansion Library Example

1.4 32 / The Expansion Sequence / Simple Expansion Library Example

The following example uses `FindConfigDev()` to print out information about all of the configured expansion peripherals in the system. `FindConfigDev()` searches the system's list of `ConfigDev` structures and returns a pointer to the `ConfigDev` structure matching a specified board:

```
newconfigdev = struct ConfigDev *
                FindConfigDev( struct ConfigDev *oldconfigdev,
                              LONG manufacturer, LONG product )
```

The `oldconfigdev` argument may be set to `NULL` to begin searching at the top of the system list or, if it points to a valid `ConfigDev`, searching will begin after that entry in the system list. The manufacturer and product arguments can be set to search for a specific manufacturer and product by number, or, if these are set to -1, the function will match any board.

`findboards.c`

1.5 32 Expansion Library / Expansion Board Drivers

The Amiga operating system contains support for matching up disk-based drivers with AUTOCONFIG boards. Though such drivers are commonly Exec devices, this is not required. The driver may, for instance, be an Exec library or task. Since 1.3, the system software also supports the initialization of onboard ROM driver software.

Disk Based Drivers	ROM Based and Autoboot Drivers
Expansion Drivers and DOS	RigidDiskBlock And Alternate Filesystems

1.6 32 / Expansion Board Drivers / Disk Based Drivers

Disk-based expansion board drivers and their icons are generally placed in the SYS:Expansion drawer of the user's SYS: disk or partition. The icon Tool Type field must contain the unique Hardware Manufacturer number, and the Product number of the expansion board(s) the driver is written for. (For more about icon Tool Type fields refer to the chapter on "Workbench and Icon Library".)

The BindDrivers command issued during the disk startup-sequence attempts to match disk-based drivers with their expansion boards. To do this, BindDrivers looks in the Tool Types field of all icon files in SYS:Expansion. If the Tool Type "PRODUCT" is found in the icon, then this is an icon file for a driver. Binddrivers will then attempt to match the manufacturer and product number in this PRODUCT Tool Type with those of a board that was configured at expansion time.

For example, suppose you are manufacturer #1019. You have two products, #1 and #2 which both use the same driver. The icon for your driver for these two products would have a Tool Type set to "PRODUCT=1019/1|1019/2". This means: I am an icon for a driver that works with product number 1 or 2 from manufacturer 1019, now bind me. Spaces are not legal. Here are two other examples:

```
PRODUCT=1208/11      is the Tool Type for a driver for product
                      11 from manufacturer number 1208.
```

```
PRODUCT=1017         is the Tool Type for a driver for any
                      product from manufacturer number 1017.
```

If a matching board is found for the disk-based driver, the driver code is loaded and then initialized with the Exec InitResident() function. From within its initialization code, the driver can get information about the board it is bound to by calling the expansion.library function GetCurrentBinding(). This function will provide the driver with a copy of a CurrentBinding structure, including a pointer to a ConfigDev structure (possibly linked to additional ConfigDevs via the cd_NextCD field) of the expansion board(s) that matched the manufacturer and product IDs.

```
/* this structure is used by GetCurrentBinding() */
/* and SetCurrentBinding() */
struct CurrentBinding {
    struct ConfigDev *cb_ConfigDev; /* first configdev in chain */
    UBYTE *          cb_FileName; /* file name of driver */
    UBYTE *          cb_ProductString; /* product # string */
}
```

```

        UBYTE **          cb_ToolTypes;      /* tooltypes from disk object */
};

```

GetCurrentBinding() allows the driver to find out the base address and other information about its board(s). The driver must unset the CONFIGME bit in the cd_Flags field of the ConfigDev structure for each board it intends to drive, and record the driver's Exec node pointer in the cd_Driver structure. This node should contain the LN_NAME and LN_TYPE (i.e., NT_DEVICE, NT_TASK, etc.) of the driver.

Important Note:

The GetCurrentBinding() function, and driver binding in general, must be bracketed by an ObtainConfigBinding() and ReleaseConfigBinding() semaphore. The BindDrivers command obtains this semaphore and performs a SetCurrentBinding() before calling InitResident(), allowing the driver to simply do a GetCurrentBinding().

Full source code for a disk-based Expansion or DEVS: sample device driver may be found in the Addison-Wesley Amiga ROM Kernel Reference Manual: Devices. Autodocs for expansion.library functions may be found in the Addison-Wesley Amiga ROM Kernel Reference Manual: Includes and Autodocs.

1.7 32 / Expansion Board Drivers / Expansion Drivers and DOS

Two other expansion.library functions commonly used by expansion board drivers are MakeDosNode() and AddDosNode(). These functions allow a driver to create and add a DOS device node (for example DH0:) to the system. A new function, AddBootNode(), is also available in Release 2 (V36 and later versions of the OS) that can be used to add an autobooting DOS device node.

MakeDosNode() requires an initialized structure of environment information for creating a DOS device node. The format of the function is:

```

struct DeviceNode *deviceNode = MakeDosNode(parameterPkt);

```

The parameterPkt argument is a pointer (passed in A0 from assembler) to an initialized packet of environment parameters.

The parameter packet for MakeDosNode() consists of four longwords followed by a DosEnvec structure:

```

;-----
;
; Layout of parameter packet for MakeDosNode
;
;-----

```

* The packet for MakeDosNode starts with the following four
* longwords, directly followed by a DosEnvec structure.

```

APTR  dosName      ; Points to a DOS device name (ex. 'RAM1',0)
APTR  execName     ; Points to device driver name (ex. 'ram.device',0)

```



```

ULONG unit          ; Unit number
ULONG flags         ; OpenDevice flags

```

* The DosEnvec disk "environment" is a longword array that describes the
 * disk geometry. It is variable sized, with the length at the beginning.
 * Here are the constants for a standard geometry.
 * See libraries/filehandler.i for additional notes.

```

STRUCTURE DosEnvec,0
  ULONG de_TableSize      ; Size of Environment vector
  ULONG de_SizeBlock      ; in longwords: standard value is 128
  ULONG de_SecOrg         ; not used; must be 0
  ULONG de_Surfaces       ; # of heads (surfaces). drive specific
  ULONG de_SectorPerBlock ; not used; must be 1
  ULONG de_BlocksPerTrack ; blocks per track. drive specific
  ULONG de_Reserved       ; DOS reserved blocks at start of partition.
  ULONG de_PreAlloc       ; DOS reserved blocks at end of partition
  ULONG de_Interleave     ; usually 0
  ULONG de_LowCyl         ; starting cylinder. typically 0
  ULONG de_HighCyl        ; max cylinder. drive specific
  ULONG de_NumBuffers     ; Initial # DOS of buffers.
  ULONG de_BufMemType     ; type of mem to allocate for buffers
  ULONG de_MaxTransfer    ; Max number of bytes to transfer at a time
  ULONG de_Mask           ; Address Mask to block out certain memory
  LONG  de_BootPri        ; Boot priority for autoboot
  ULONG de_DosType        ; ASCII (HEX) string showing filesystem type;
                          ; 0X444F5300 is old filesystem,
                          ; 0X444F5301 is fast file system

  ULONG de_Baud           ; Baud rate for serial handler
  ULONG de_Control        ; Control word for handler/filesystem
                          ; (used as filesystem/handler desires)

  ULONG de_BootBlocks     ; Number of blocks containing boot code
                          ; (for non-AmigaDOS filesystems)

  LABEL DosEnvec_SIZEOF

```

After making a DOS device node, drivers (except for autoboot drivers) use
 AddDosNode(deviceNode) to add their node to the system. Autoboot drivers
 will instead use the new Release 2 expansion.library AddBootNode()
 function (if running under V36 or higher) or the Exec Enqueue() function
 (if running under pre-V36) to add a BootNode to the
 ExpansionBase.eb_MountList.

1.8 32 / Expansion Board Drivers / ROM Based and Autoboot Drivers

Since 1.3, the system software supports the initialization of ROM drivers
 residing on expansion peripherals, including the ability for drivers to
 provide a DOS node which the system can boot from. This feature is known
 as Autoboot.

Automatic boot from a ROM-equipped expansion board is accomplished before
 DOS is initialized. This facility makes it possible to automatically boot
 from a hard disk without any floppy disks inserted. Likewise, it is
 possible to automatically boot from any device which supports the ROM
 protocol, even allowing the initialization of a disk operating system

other than the Amiga's dos.library. ROM-based drivers contain several special entry points that are called at different stages of system initialization. These three stages are known as DIAG, ROMTAG INIT and BOOT.

Events At DIAG Time Events At ROMTAG INIT Time Events At BOOT Time

1.9 32 // ROM Based and Autoboot Drivers / Events At DIAG Time

When your AUTOCONFIG hardware board is configured by the expansion initialization routine, its ExpansionRom structure is copied into the ExpansionRom subfield of a ConfigDev structure. This ConfigDev structure will be linked to the expansion.library's private list of configured boards.

After the board is configured, the er_Type field of its ExpansionRom structure is checked. The DIAGVALID bit set declares that there is a valid DiagArea (a ROM/diagnostic area) on this board. If there is a valid DiagArea, expansion next tests the er_InitDiagVec vector in its copy of the ExpansionRom structure. This offset is added to the base address of the configured board; the resulting address points to the start of this board's DiagArea.

```
struct ExpansionRom
{
    UBYTE    er_Type;           /* <-- if ERTB_DIAGVALID set */
    UBYTE    er_Product;
    UBYTE    er_Flags;
    UBYTE    er_Reserved03;
    UWORD    er_Manufacturer;
    ULONG    er_SerialNumber;
    UWORD    er_InitDiagVec;    /* <-- then er_InitDiagVec      */
    UBYTE    er_Reserved0c;    /*      is added to cd_BoardAddr */
    UBYTE    er_Reserved0d;    /*      and points to DiagArea   */
    UBYTE    er_Reserved0e;
    UBYTE    er_Reserved0f;
};
```

Now expansion knows that there is a DiagArea, and knows where it is.

```
struct DiagArea
{
    UBYTE    da_Config;        /* <-- if DAC_CONFIGTIME is set */
    UBYTE    da_Flags;
    UWORD    da_Size;          /* <-- then da_Size bytes will  */
    UWORD    da_DiagPoint;     /*      be copied into RAM      */
    UWORD    da_BootPoint;
    UWORD    da_Name;
    UWORD    da_Reserved01;
    UWORD    da_Reserved02;
};

/* da_Config definitions */
#define DAC_BUSWIDTH      0xC0 /* two bits for bus width */
#define DAC_NIBBLEWIDE    0x00
#define DAC_BYTEWIDE      0x40 /* invalid for 1.3 - see note below */
```

```

#define DAC_WORDWIDE      0x80

#define DAC_BOOTTIME      0x30 /* two bits for when to boot */
#define DAC_NEVER         0x00 /* obvious */
#define DAC_CONFIGTIME    0x10 /* call da_BootPoint when first
                                configging the device */
#define DAC_BINDTIME      0x20 /* run when binding drivers to boards */

```

Next, expansion tests the first byte of the DiagArea structure to determine if the CONFIGTIME bit is set. If this bit is set, it checks the da_BootPoint offset vector to make sure that a valid bootstrap routine exists. If so, expansion copies da_Size bytes into RAM memory, starting at beginning of the DiagArea structure.

The copy will include the DiagArea structure itself, and typically will also include the da_DiagPoint ROM/diagnostic routine, a Resident structure (romtag), a device driver (or at least the device initialization tables or structures which need patching), and the da_BootPoint routine. In addition, the BootNode and parameter packet for MakeDosNode() may be included in the copy area for Diag-time patching. Strings such as DOS and Exec device names, library names, and the romtag ID string may also be included in the copy area so that both position-independent ROM code and position-independent routines in the copy area may reference them PC relative.

The copy will be made either nibblewise, or wordwise, according to the BUSWIDTH subfield of da_Config. Note that the da_BootPoint offset must be non-NULL, or else no copy will occur. (Note - under 1.3, DAC_BYTEWIDE is not supported. Byte wide ROMs must use DAC_NIBBLEWIDE and drivers must supply additional code to re-copy their DiagArea)

The following illustrates an example Diag copy area, and specifies the various fields which should be coded as relative offsets for later patching by your DiagPoint routine.

Example DiagArea Copy in RAM

```

DiagStart:      ; a struct DiagArea
                CCFF  ; da_Config, da_Flags
                SIZE  ; da_Size          - coded as EndCopy-DiagStart
                DIAG  ; da_DiagPoint     - coded as DiagEntry-DiagStart
                BOOT  ; da_BootPoint     - coded as BootEntry-DiagStart
                NAME  ; da_Name          - coded as DevName-DiagStart
                0000  ; da_Reserved01    - Above fields above are supposed
                0000  ; da_Reserved02    to be relative. No patching needed

Romtag:         rrrr  ; a struct Resident ("Romtag")
                ...   RT_MATCHTAG, RT_ENDSKIP, RT_NAME and RT_IDSTRING
                ...   addresses are coded relatively as label-DiagStart.
                ...   The RT_INIT vector is coded as a relative offset
                ...   from the start of the ROM. DiagEntry patches these.

DevName:        ssss..0 ; The name string for the exec device
IdString:       iiii..0 ; The ID string for the Romtag

BootEntry:      BBBB    ; Boot-time code

```

```

...

DiagEntry: DDDD      ; Diag-time code (position independent)
...           When called, performs patching of the relative-
              coded addresses which need to be absolute.

OtherData:
    dddd      ; Device node or structs/tables (patch names, vectors)
    bbbb      ; BootNode (patch ln_Name and bn_DeviceNode)
    pppp      ; MakeDosNode packet (patch dos and exec names)

    ssss      ; other name, ID, and library name strings
    ...

EndCopy:

```

Now the ROM "image" exists in RAM memory. Expansion stores the ULONG address of that "image" in the UBYTES `er_Reserved0c`, `0d`, `0e` and `0f`. The address is stored with the most significant byte in `er_Reserved0c`, the next to most significant byte in `er_Reserved0d`, the next to least significant byte in `er_Reserved0e`, and the least significant byte in `er_Reserved0f` - i.e., it is stored as a longword at the address `er_Reserved0c`.

Expansion finally checks the `da_DiagPoint` offset vector, and if valid executes the ROM/diagnostic routine contained as part of the ROM "image". This diagnostic routine is responsible for patching the ROM image so that required absolute addresses are relocated to reflect the actual location of code and strings, as well as performing any diagnostic functions essential to the operation of its associated AUTOCONFIG board. The structures which require patching are located within the copy area so that they can be patched at this time. Patching is required because many of the structures involved require absolute pointers to such things as name strings and code, but the absolute locations of the board and the RAM copy area are not known when code the structures.

The patching may be accomplished by coding pointers which require absolute addresses instead as relative offsets from either the start of the `DiagArea` structure, or the start of the board's ROM (depending on whether the final absolute pointer will point to a RAM or ROM location). The `Diag` routine is passed both the actual base address of the board, and the address of the `Diag` copy area in RAM. The routine can then patch the structures in the `Diag` copy area by adding the appropriate address to resolve each pointer.

Example `DiagArea` and `Diag` patching routine:

`Diag.asm`

Your `da_DiagPoint` ROM/diagnostic routine should return a non-zero value to indicate success; otherwise the ROM "image" will be unloaded from memory, and its address will be replaced with NULL bytes in locations `er_Reserved0c`, `0d`, `0e` and `0f`.

Now that the ROM "image" has been copied into RAM, validated, and linked to board's `ConfigDev` structure, the expansion module is free to configure all other boards on the `utility.library`'s private list of boards.

It may help to see just how a card's ROM AUTOCONFIG information corresponds to the ExpansionRom structure. This chart shows the contents of on-card memory for a fictional expansion card. Note that the ExpansionRom.Flags field (\$3F in addresses \$08/\$0A below) is shown interpreted in its inverted form of \$3F. Once the value is uninverted to become \$C0, you should use the #defines in <libraries/configregs.h> to interpret it.

Table 32-1: Sample Zorro II AUTOCONFIG ROM Information Viewed as a Hex Dump

FLAG AND FIELD DEFINITIONS				THIS BOARD
-----				-----
	1xxx chained			11 = Normal type
	x111 size			Don't addmem
	000=8meg, 001=64K, 010=128K, etc.			ROM Vector Valid
11xx type	/	1xxx nopref		Not chained
xx1x addmem	/	xlxx canshut		Size 256K
xxx1 ROM	/	xx11 reserved		Product#=~\$FE=1
\	/ ~Prod#	/		Flags=~\$3F=\$C0
\	/ / \	/ res. reserved		Prefer exp space
0000: D0003000 F000E000 3000F000 F000F000				Can't be shut up
-----				-----
	~Manufacturer#	~HiWord Serial#		Manu#=~\$F824=\$7DB=2011
/ / \ \	/ / \ \			HiSer=~\$FFDA=\$0025=37
0010: F0008000 20004000 F000F000 D000A000				
-----				-----
	~LoWord Serial#	~Rom Vector		LoSer=~\$FFFE=\$0001=1
/ / \ \	/ / \ \			Rom Vector=~\$FF7F=\$80
0020: F000F000 F000E000 F000F000 7000F000				from board base
-----				-----

The AUTOCONFIG information from the above card would appear as follows in an ExpansionRom structure:

Nibble Pairs	ExpansionRom Field	Value
-----	-----	-----
00/02	er_Type	\$D3
04/06	er_Product	\$01 = 1
08/0A	er_Flags	\$C0
10/12 and 14/16	er_Manufacturer	\$07DB = 2011
18/1A thru 24/26	er_SerialNumber	\$00250001
28/2A and 2C/2E	er_InitDiagVec	\$0080

If a card contains a ROM driver (Rom Vector valid), and the vector is at offset \$80 (as in this example) the DiagArea structure will appear at offset \$0080 from the base address of the board. This example card's Resident structure (romtag) directly follows its DiagArea structure.

WORDWIDE+CONFIGTIME	ROMTAG
\ flags	DiagPt
Devname	starts
\ \ DAsize / BootPt /	here
	DiagPt, BootPt,

\ \ /\ /\ /\ /\ res. res. /\	DevName relative
0080: 90000088 004A0076 00280000 00004AFC	to Diag struct

COLDSTART NT_DEVICE	backptr,endskip,
\ ver /priority	and DevName coded
backptr endskip \ \ / / DevName	relative, patched
0090: 0000000E 00000088 01250314 00000028	at Diag time

IDstring InitEntry	ID and InitEntry
00A0: 00000033 00000116	coded relative,
	patched at Diag

1.10 32 // ROM Based and Autoboot Drivers / Events At ROMTAG INIT Time

Next, most resident system modules (for example graphics) are initialized. As part of the system initialization procedure a search is made of the expansion.library's private list of boards (which contains a ConfigDev structure for each of the AUTOCONFIG hardware boards). If the cd_Flags specify CONFIGME and the er_Type specifies DIAGVALID, the system initialization will do three things:

First, it will set the current ConfigDev as the current binding (see the expansion.library SetCurrentBinding() function). Second, it will check the DiagArea's da_Config flag to make sure that the CONFIGTIME bit is set. Third, it will search the ROM "image" associated with this hardware board for a valid Resident structure (<exec/resident.h>); and, if one is located, will call InitResident() on it, passing a NULL segment list pointer as part of the call.

Next, the board's device driver is initialized. The Resident structure associated with this board's device driver (which has now been patched by the ROM/diagnostic routine) should follow standard system conventions in initializing the device driver provided in the boot ROMs. This driver should obtain the address of its associated ConfigDev structure via GetCurrentBinding().

Once the driver is initialized, it is responsible for some further steps. It must clear the CONFIGME bit in the cd_Flags of its ConfigDev structure, so that the system knows not to configure this device again if binddrivers is run after bootstrap. Also, though it is not currently mandatory, the driver should place a pointer to its Exec node in the cd_Driver field of the ConfigDev structure. This will generally be a device (NT_DEVICE) node. And for this device to be bootable, the driver must create a BootNode structure, and link this BootNode onto the expansion.library's eb_MountList.

The BootNode structure (see <libraries/expansionbase.h>) contains a Node of the new type NT_BOOTNODE (see <exec/nodes.h>). The driver must initialize the ln_Name field to point to the ConfigDev structure which it has obtained via the GetCurrentBinding() call. The bn_Flags subfield is currently unused and should be initialized to NULL. The bn_DeviceNode must be initialized to point to the DosNode for the device.

When the DOS is initialized later, it will attempt to boot from the first

BootNode on the eb_MountList. The eb_MountList is a priority sorted List, with nodes of the highest priority at the head of the List. For this reason, the device driver must enqueue a BootNode onto the list using the Exec library function Enqueue().

In the case of an autoboot of AmigaDOS, the BootNode must be linked to a DeviceNode of the AmigaDOS type (see <libraries/filehandler.h>), which the driver can create via the expansion library MakeDosNode() function call. When the DOS "wakes up", it will attempt to boot from this DeviceNode.

1.11 32 // ROM Based and Autoboot Drivers / Events At BOOT Time

If there is no boot disk in the internal floppy drive, the system strap module will call a routine to perform autoboot. It will examine the eb_MountList; find the highest priority BootNode structure at the head of the List; validate the BootNode; determine which ConfigDev is associated with this BootNode; find its DiagArea; and call its da_BootPoint function in the ROM "image" to bootstrap the appropriate DOS. Generally, the BootPoint code of a ROM driver will perform the same function as the boot code installed on a floppy disk, i.e., it will FindResident() the dos.library, and jump to its RT_INIT vector. The da_BootPoint call, if successful, should not return.

If a boot disk is in the internal floppy drive, the system strap will Enqueue() a BootNode on the eb_MountList for DF0: at the suggested priority (see the Autodoc for the expansion.library AddDosNode() function). Strap will then open AmigaDOS, overriding the autoboot. AmigaDOS will boot from the highest priority node on the eb_MountList which should, in this case, be DF0:. Thus, games and other bootable floppy disks will still be able to obtain the system for their own use.

In the event that there is no boot disk in the internal floppy drive and there are no ROM bootable devices on the autoconfiguration chain, the system does the normal thing, asking the user to insert a Workbench disk, and waiting until its request is satisfied before proceeding.

1.12 32 / Expansion Board Drivers / RigidDiskBlock and Alternate Filesystems

Through the use of RigidDiskBlock information and the FileSys.resource, it is possible for an autoboot driver to have access to enough information to mount all of its device partitions and even load alternate filesystems for use with these partitions.

The RigidDiskBlock specification (also known as "hardblocks") defines blocks of data that exist on a hard disk to describe that disk. These blocks are created or modified with an installation utility (such as the hard drive Prep utility for the A2090A ST506/SCSI controller card) provided by the disk controller manufacturer, and they are read and used by the device driver ROM (or expansion) code. They are not generally accessible to the user as they do not appear on any DOS device. The blocks are tagged with a unique identifier, checksummed, and linked together.

The five block types currently defined are RigidDiskBlock, BadBlockBlock, PartitionBlock, FileSysHeaderBlock, and LoadSegBlock.

The root of these blocks is the RigidDiskBlock. The RigidDiskBlock must exist on the disk within the first RDB_LOCATION_LIMIT (16) blocks. This inhibits the use of the first cylinder(s) 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. This allocation range is described in the RigidDiskBlock.

The RigidDiskBlock contains basic information about the configuration of the drive: number and size of blocks, tracks, and cylinders, as well as other relevant information. The RigidDiskBlock points to bad block, partition, file system and drive initialization description blocks.

The BadBlockBlock list contains a series of bad-block/good-block pairs. Each block contains as many as will fit in a physical sector on the drive. These mappings are to be handled by the driver on read and write requests.

The drive initialization description blocks are LoadSegBlocks that are loaded at boot time to perform drive-specific initialization. They are called with both C-style parameters on the stack, and assembler parameters in registers as follows:

```
d0 = DriveInit(lun, rdb, ior) (d0/a0/a1)
```

where lun is the SCSI logical unit number (needed to construct SCSI commands), rdb is a pointer to a memory copy of the RigidDiskBlock (which should not be altered), and ior is a standard IO request block that can be used to access the drive with synchronous DoIO() calls.

The result of DriveInit() is either -1, 0, or 1. A -1 signifies that an error occurred and drive initialization cannot continue. A 0 (zero) result reports success. In cases -1 and 0, the code is unloaded. A result of 1 reports success, and causes the code to be kept loaded. Furthermore, this resident code will be called whenever a reset is detected on the SCSI bus.

The FileSysHeaderBlock entries contain code for alternate file handlers to be used by partitions. There are several strategies that can be used to determine which of them to load. The most robust would scan all drives for those that are both required by partitions and have the highest fhb_Version, and load those. Whatever method is used, the loaded file handlers are added to the Exec resource FileSystem.resource, where they are used as needed to mount disk partitions.

The PartitionBlock entries contains most of the data necessary to add each partition to the system. They replace the Mount and DEVS:MountList mechanism for adding these partitions. The only items required by the expansion.library MakeDosNode() function which are not in this partition block are the Exec device name and unit, which is expected to be known by

driver reading this information. The file system to be used is specified in the `pb_Environment`. If it is not the default file system (i.e., `'DOS\0'` or `'DOS\1'`), the node created by `MakeDosNode()` is modified as specified in a `FileSystem.resource`'s `FileSysEntry` before adding it to the DOS list.

Only 512 byte blocks were supported by the pre-V36 file system, but this proposal was forward-looking by making the block size explicit, and by using only the first 256 bytes for all blocks but the `LoadSeg` and `BadBlock` data. Under the present filesystem, this allows using drives formatted with sectors 256 bytes or larger (i.e., 256, 512, 1024, etc). `LoadSeg` and `BadBlock` data use whatever space is available in a sector.

<code>RigidDiskBlock</code>	<code>PartitionBlock</code>	<code>LoadSegBlock</code>
<code>BadBlockBlock</code>	<code>FileSysHeaderBlock</code>	<code>filesysres.h</code> and <code>.i</code>

1.13 32 // RigidDiskBlock and Alternate Filesystems / RigidDiskBlock

This is the current specification for the `RigidDiskBlock`:

<code>rdb_ID</code>	<code>== 'RDSK'</code>
<code>rdb_SummedLongs</code>	<code>== 64</code>
 <code>rdb_ChkSum</code>	 block checksum (longword sum to zero)
 <code>rdb_HostID</code>	 SCSI Target ID of host This is the initiator ID of the creator of this <code>RigidDiskBlock</code> . It is intended that modification of the <code>RigidDiskBlock</code> , or of any of the blocks pointed to by it, by another initiator (other than the one specified here) be allowed only after a suitable warning. The user is then expected to perform an audio lock out ("Hey, is anyone else setting up SCSI stuff on this bus?"). The <code>rdb_HostID</code> may become something other than the initiator ID when connected to a real network: that is an area for future standardization.
 <code>rdb_BlockBytes</code>	 size of disk blocks Under pre-V36 filesystem, this must be 512 for a disk with any AmigaDOS partitions on it. Present filesystem supports 256, 512, 1024, etc.
 <code>rdb_Flags</code>	 longword of flags:
 <code>RDBF._LAST</code>	 no disks exist to be configured after this one on this controller (SCSI bus).
 <code>RDBF._LASTLUN</code>	 no LUNs exist to be configured greater than this one at this SCSI Target ID
 <code>RDBF._LASTTID</code>	 no Target IDs exist to be configured greater than this one on this SCSI bus

RDBF._NORESELECT	don't bother trying to perform reselection when talking to this drive
RDBF._DISKID	rdb_Disk... identification variables below contain valid data.
RDBF._CTRLRID	rdb_Controller... identification variables below contain valid data.
RDBF._SYNCH	drive supports scsi synchronous mode CAN BE DANGEROUS TO USE IF IT DOESN'T!

These fields point to other blocks on the disk which are not a part of any filesystem. All block pointers referred to are block numbers on the drive.

rdb_BadBlockList	optional bad block list A singly linked list of blocks of type PartitionBlock
rdb_PartitionList	optional first partition block A singly linked list of blocks of type PartitionBlock
rdb_FileSysHeaderList	optional file system header block A singly linked list of blocks of type FileSysHeaderBlock
rdb_DriveInit	optional drive-specific init code A singly linked list of blocks of type LoadSegBlock containing initialization code. Called as DriveInit(lun,rdb,ior) (d0/a0/a1).
rdb_Reserved1[6]	set to \$fffffffffs These are reserved for future block lists. Since NULL for block lists is \$ffffffff, these reserved entries must be set to \$ffffffff.

These fields describe the physical layout of the drive.

rdb_Cylinders	number of drive cylinders
rdb_Sectors	sectors per track
rdb_Heads	number of drive heads
rdb_Interleave	interleave This drive interleave is independent from, and unknown to, the DOS's understanding of interleave as set in the partition's environment vector.
rdb_Park	landing zone cylinder
rdb_Reserved2[3]	set to zeros

These fields are intended for ST506 disks. They are generally unused for SCSI devices and set to zero.

rdb_WritePreComp	starting cylinder: write precompensation
rdb_ReducedWrite	starting cylinder: reduced write current
rdb_StepRate	drive step rate
rdb_Reserved3[5]	set to zeros

These fields are used while partitions are set up to constrain the partitionable area and help describe the relationship between the drive's logical and physical layout.

rdb_RDBlocksLo	low block of the range allocated for blocks described here. Replacement blocks for bad blocks may also live in this range.
rdb_RDBlocksHi	high block of this range (inclusive)
rdb_LoCylinder	low cylinder of partitionable disk area Blocks described by this include file will generally be found in cylinders below this one.
rdb_HiCylinder	high cylinder of partitionable data area Usually rdb_Cylinders-1.
rdb_CylBlocks	number of blocks available per cylinder This may be rdb_Sectors*rdb_Heads, but a SCSI disk that, for example, reserves one block per cylinder for bad block mapping would use rdb_Sectors*rdb_Heads-1.
rdb_AutoParkSeconds	number of seconds to wait before parking drive heads automatically. If zero, this feature is not desired.
rdb_HighRDSKBlock	highest block used by these drive definitions Must be less than or equal to rdb_RDBlocksHi. All replacements for bad blocks should be between rdb_HighRDSKBlock+1 and rdb_RDBlocksHi (inclusive).
rdb_Reserved4	set to zeros

These fields are of the form available from a SCSI Identify command. Their purpose is to help the user identify the disk during setup. Entries exist for both controller and disk for non-embedded SCSI disks.

rdb_DiskVendor	vendor name of the disk
rdb_DiskProduct	product name of the disk
rdb_DiskRevision	revision code of the disk
rdb_ControllerVendor	vendor name of the disk controller

```

rdb_ControllerProduct    product name of the disk controller
rdb_ControllerRevision   revision code of the disk controller
rdb_Reserved5[10]       set to zeros

```

1.14 32 // RigidDiskBlock and Alternate Filesystems / BadBlockBlock

This is the current specification for the BadBlockBlock. The end of data occurs when bbb_Next is NULL (\$FFFFFFFF), and the summed data is exhausted.

```

bbb_ID                    == 'BADB'

bbb_SummedLongs           size of this checksummed structure
                          Note that this is not 64 like most of the other
                          structures. This is the number of valid longs
                          in this image, and can be from 6 to
                          rdb_BlockBytes/4. The latter is the best size
                          for all blocks other than the last one.

bbb_ChkSum               block checksum (longword sum to zero)

bbb_HostID               SCSI Target ID of host
                          This describes the initiator ID for the creator
                          of these blocks. (see rdb_HostID discussion)

bbb_Next                 block number of the next BadBlockBlock
bbb_Reserved             set to zeros

bbb_BlockPairs[61]       pairs of block remapping information
                          The data starts here and continues as long as
                          indicated by bbb_SummedLongs-6: e.g. if
                          bbb_SummedLongs is 128 (512 bytes), 61 pairs
                          are described here.

```

1.15 32 // RigidDiskBlock and Alternate Filesystems / PartitionBlock

This is the current specification for the PartitionBlock. Note that while reading these blocks you may encounter partitions that are not to be mounted because the pb_HostID does not match, or because the pb_DriveName is in use and no fallback strategy exists, or because PBF._NOMOUNT is set, or because the proper filesystem cannot be found. Some partitions may be mounted but not be bootable because PBF._BOOTABLE is not set.

```

pb_ID                    == 'PART'
pb_SummedLongs           == 64
pb_ChkSum               block checksum (longword sum to zero)

pb_HostID               SCSI Target ID of host
                          This describes the initiator ID for the owner
                          of this partition. (see rdb_HostID discussion)

```

pb_Next	block number of the next PartitionBlock
pb_Flags	see below for defines
PBF._BOOTABLE	this partition is intended to be bootable (e.g. expected directories and files exist)
PBF._NOMOUNT	this partition description is to reserve space on the disk without mounting it. It may be manually mounted later.
pb_Reserved1[2]	set to zeros
pb_DevFlags	preferred flags for OpenDevice
pb_DriveName	preferred DOS device name: BSTR form This name is not to be used if it is already in use.

Note that pb_Reserved2 will always be at least 4 longwords so that the RAM image of this record may be converted to the parameter packet to the expansion.library function MakeDosNode().

pb_Reserved2[15]	filler to make 32 longwords so far
------------------	------------------------------------

The specification of the location of the partition is one of the components of the environment, below. If possible, describe the partition in a manner that tells the DOS about the physical layout of the partition: specifically, where the cylinder boundaries are. This allows the filesystem's smart block allocation strategy to work.

pb_Environment[17]	environment vector for this partition containing:
de_TableSize	size of Environment vector
de_SizeBlock	== 128 (for 512 bytes/logical block)
de_SecOrg	== 0
de_Surfaces	number of heads (see layout discussion)
de_SectorPerBlock	== 1
de_BlocksPerTrack	blocks per track (see layout discussion)
de_Reserved	DOS reserved blocks at start of partition. Must be >= 1. 2 is recommended.
de_PreAlloc	DOS reserved blocks at end of partition Valid only for filesystem type DOS^A (the fast file system). Zero otherwise.
de_Interleave	DOS interleave Valid only for filesystem type DOS^@ (the old file system). Zero otherwise.
de_LowCyl	starting cylinder
de_HighCyl	max cylinder
de_NumBuffers	initial # DOS of buffers.

<code>de_BufMemType</code>	type of mem to allocate for buffers The second argument to <code>AllocMem()</code> .
<code>de_MaxTransfer</code>	max number of bytes to transfer at a time. Drivers should be written to handle requests of any length.
<code>de_Mask</code>	address mask to block out certain memory Normally <code>\$00ffffff</code> for DMA devices.
<code>de_BootPri</code>	Boot priority for autoboot Suggested value: zero. Keep less than five, so it won't override a boot floppy.
<code>de_DosType</code>	ASCII string showing filesystem type; DOS^@ (\$444F5300) is old filesystem, DOS^A (\$444F5301) is fast file system. UNI<anything> is a Unix partition.
<code>pb_EReserved[15]</code>	reserved for future environment vector

1.16 32 // RigidDiskBlock and Alternate Filesystems / FileSysHeaderBlock

The current specification for the `FileSysHeaderBlock` follows.

<code>fhb_ID</code>	== 'FSHD'
<code>fhb_SummedLongs</code>	== 64
<code>fhb_ChkSum</code>	block checksum (longword sum to zero)
<code>fhb_HostID</code>	SCSI Target ID of host This describes the initiator ID for the creator of this block. (see <code>rdb_HostID</code> discussion)
<code>fhb_Next</code>	block number of next <code>FileSysHeaderBlock</code>
<code>fhb_Flags</code>	see below for defines
<code>fhb_Reserved1[2]</code>	set to zero

The following information is used to construct a `FileSysEntry` node in the `FileSystem.resource`.

<code>fhb_DosType</code>	file system description This is matched with a partition environment's <code>de_DosType</code> entry.
<code>fhb_Version</code>	release version of this load image Usually MSW is version, LSW is revision.
<code>fhb_PatchFlags</code>	patch flags These are bits set for those of the following that need to be substituted into a standard

	device node for this file system, lsb first: e.g. 0x180 to substitute SegList & GlobalVec
fhb_Type	device node type: zero
fhb_Task	standard dos "task" field: zero
fhb_Lock	not used for devices: zero
fhb_Handler	filename to loadseg: zero placeholder
fhb_StackSize	stacksize to use when starting task
fhb_Priority	task priority when starting task
fhb_Startup	startup msg: zero placeholder
fhb_SegListBlocks	first of linked list of LoadSegBlocks: Note that if the fhb_PatchFlags bit for this entry is set (bit 7), the blocks pointed to by this entry must be LoadSeg'd and the resulting BPTR put in the FileSysEntry node.
fhb_GlobalVec	BCPL global vector when starting task Zero or -1.
fhb_Reserved2[23]	(those reserved by PatchFlags)
fhb_Reserved3[21]	set to zero

1.17 32 // RigidDiskBlock and Alternate Filesystems / LoadSegBlock

This is the current specification of the LoadSegBlock. The end of data occurs when lsb_Next is NULL (\$FFFFFFFF), and the summed data is exhausted.

lsb_ID	== 'LSEG'
lsb_SummedLongs	size of this checksummed structure Note that this is not 64 like most of the other structures. This is the number of valid longs in this image, like bbb_SummedLongs.
lsb_ChkSum	block checksum (longword sum to zero)
lsb_HostID	SCSI Target ID of host This describes the initiator ID for the creator of these blocks. (see rdb_HostID discussion)
lsb_Next	block number of the next LoadSegBlock
lsb_LoadData	data for "loadseg" The data starts here and continues as long as indicated by lsb_SummedLongs-5: e.g. if lsb_SummedLongs is 128 (ie. for 512 byte blocks), 123 longs of data are valid here.

1.18 32 // RigidDiskBlock and Alternate Filesystems / filesysres.h and .i

The FileSysResource is created by the first code that needs to use it. It is added to the resource list for others to use. (Checking and creation should be performed while Forbid() is in effect). Under Release 2 the resource is created by the system early on in the initialization sequence. Under 1.3 it is the responsibility of the first RDB driver to create it.

FileSysResource

fsr_Node	on resource list with the name FileSystem.resource
fsr_Creator	name of creator of this resource
fsr_FileSysEntries	list of FileSysEntry structs

FileSysEntry

fse_Node	on fsr_FileSysEntries list ln_Name is of creator of this entry
fse_DosType	DosType of this FileSys
fse_Version	release version of this FileSys Usually MSW is version, LSW is revision.
fse_PatchFlags	bits set for those of the following that need to be substituted into a standard device node for this file system: e.g. \$l80 for substitute SegList & GlobalVec
fse_Type	device node type: zero
fse_Task	standard dos "task" field
fse_Lock	not used for devices: zero
fse_Handler	filename to loadseg (if SegList is null)
fse_StackSize	stacksize to use when starting task
fse_Priority	task priority when starting task
fse_Startup	startup msg: FileSysStartupMsg for disks
fse_SegList	segment of code to run to start new task
fse_GlobalVec	BCPL global vector when starting task

No more entries need exist than those implied by fse_PatchFlags, so entries do not have a fixed size.

For additional information on initializing and booting a Rigid Disk Block filesystem device, see the SCSI Device chapter of the Addison-Wesley Amiga ROM Kernel Reference Manual: Devices. Writers of drivers for expansion devices that perform their own DMA (direct memory access) should consult the Exec chapters and Autodocs for information on Release 2 processor cache control functions including CachePreDMA() and CachePostDMA(). See the following include files for additional notes and related structures: <libraries/configvers.h> and <.i>, <libraries/configregs.h> and <.i>, <devices/hardblocks.h> and <.i>, <resources/filesysres.h> and <.i>, and <libraries/filehandler.h> and <.i>.

1.19 32 Expansion Library / Function Reference

The following are brief descriptions of the expansion library functions that are useful for expansion device drivers and related applications. See the Amiga ROM Kernel Reference Manual: Includes and Autodocs for the complete descriptions of all the expansion library functions.

Table 32-2: Expansion Library Functions

Function	Description
FindConfigDev()	Returns a pointer to the ConfigDev structure of a given expansion device.
MakeDosNode()	Creates the DOS device node for disk and similar expansion devices.
AddDosNode()	Adds a DOS device node to the system.
AddBootNode()	Adds an autobooting DOS device node to the system (V36).
GetCurrentBinding()	Returns a pointer to the CurrentBinding structure of a given device.
SetCurrentBinding()	Set up for reading the CurrentBinding with GetCurrentBinding().
ObtainConfigBinding()	Protect the ConfigDev structure with a semaphore.
ReleaseConfigBinding()	Release a semaphore on ConfigDev set up with ObtainCurrentBinding().