Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Tel 206 882 8080
Telex 160520
Fax 206 883 8101

Microsoft®

March 11, 1991

Dear OS/2 Developer,

The enclosed package contains Version 1.0 of the OS/2 **LA**yered **D**evice **DR**iver (LADDR) Development Kit (LDK). LADDR provides an alternative approach to developing OS/2 device support. This architecture isolates hardware-specific modules and provides an easier method for developing device drivers for OS/2 than previously available.

The development kit contains the current LADDR Specification and numerous examples of sample driver code. Sample source for Type Specific Drivers (TSD) is provided supporting Disk, Tape and CDROM. Sample source for Vendor specific drivers (VSD) is provided for standard disk and CDROM data transfer, Denon CDROM Audio support, TEAC and Wangtek Tape drives, as well as additional sample VSDs that support Fault Tolerance and caching of FAT based storage devices. Bus Adapter Drivers (BID) support includes binaries for Adaptec, Future Domain, NCR and Western Digital SCSI adaptors and generic ESDI disk adaptors. Source code for the ESDI, Future Domain and Western Digital BIDs is also included. Refer to the README file on LADDR Development Kit Disk 1 for installation instructions and tips on using the LDK.

With the broad range of samples provided and the information in the specification, it is now possible to easily develop device drivers supporting a broad spectrum of hardware for OS/2. Microsoft Systems Software Support is ready to assist you with any problem you have relating to installation or program development with this kit. Feel free to contact them through Microsoft On-Line. If you would like more information about Microsoft On-Line or to arrange for the purchase of an On-Line support package, please contact Microsoft Product Support Services.

You may request Microsoft to consider your driver for inclusion in future products by filling out and signing two copies of the enclosed Microsoft LADDR Device Driver Distribution Agreement and sending it and your driver to Microsoft (Attn: OS/2 Program Management) at the address above. Microsoft will consider your submission for inclusion into future products but can not guarantee that all drivers submitted will be incorporated.

We are confident that you will find the LADDR architecture and this kit a simpler way to successfully develop device support for Microsoft OS/2.

Gary Ferguson
OS/2 LADDR Product Manager

Part Number 20784

Addendum to Microsoft License Agreement

Insert the following after section 5:

6. Layered Device Driver (LADDR). Microsoft grants to you the royalty-free right to (a) create derivative works based upon the Sample Code and to reproduce and license such derivative works whether in source code, object or binary form, (b) to reproduce, distribute and license the Binaries, and (c) sublicense those rights granted in (a) and (b) to third parties PROVIDED that you: (i) distribute the derivative works and Binaries only in conjunction with and as a part of your software product; (ii) do not use Microsoft's name, logo, or trademarks to market your software product; (iii) include Microsoft's copyright notice for LADDR on your product label and do not remove any copyright notices that are embedded in the Sample Code or Binaries; and (iv) agree to indemnify, hold harmless, and defend Microsoft from and against any claims or lawsuits, including attorneys' fees, that arise or result from the use or distribution of your software product. The "Sample Code" is limited to those files that have the following names or extensions: makefile, *.c, *.h, *.asm, *.inc, *.lnk *.lrf and *.def. The "Binaries" are limited to those files that have the following extensions: *.sys, *.sym, *.tsd, *.vsd and *.bid.

This README file contains information to help you install and use the LADDR Development Kit (LDK).

This file contains the following sections:

CAUTION: before you can use LADDR drivers on your PC, that PC must be running release 1.21, or later, of OS/2 and must contain a LADDR compatible version of BASEDD. See section 5 for more information.

1. LDK Contents

This kit contains the following materials:

1. Four 5.25" 1.2MB source diskettes containing various source files, device driver binaries, and tools.

   Note that, except as stated in section 5 below, the source files are intended only for use by the owner of the LDK in developing LADDR compliant device drivers for OS/2.

   Also note that the tools are intended only for use by the owner of the LDK for device driver development under release 1.21, or 1.3 of OS/2.

2. One 5.25" 1.2MB sample boot diskette for AT class PC's for release 1.21 of OS/2, and one 3.5" 1.44MB sample boot disk for ABIOS machines.

3. One 5.25" 1.2MB sample boot diskette for AT class PC's for release 1.3 of OS/2, and one 3.5" 1.44MB sample boot disk for ABIOS machines.

   Note that these sample boot diskettes are intended for early device driver testing only and are not a complete OS/2 system. In particular, they do not include OS/2 Presentation Manager and have very few of the OS/2 utility programs. However, they do include the kernel debugger.

   Also note that if you wish to boot the 3.5" diskettes on an IBM PS/2, you must copy the ABIOS patch files ("*.BIO") from IBM OS/2 diskettes, or a hard disk containing IBM OS/2, onto the boot disk.

4. A printed copy of the current LADDR specification.

2. Installation Procedure

The sample keyins shown below assume that the "B" drive is to be used to read the LDK diskettes and that the LDK is to be installed on the "D" fixed disk.

Installing the source

This procedure requires approximatley 5 megabytes of fixed disk space.

1. Create an appropriately named subdirectory on a fixed disk and make that subdirectory current.

   Sample keyins:

   ```
   d:
   md \laddr
   cd \laddr
   ```

2. Insert the first LDK source diskette into a drive and copy it to the hard disk using XCOPY or some equivalent program that is capable of copying entire tree structures.

   Sample keyin:

   ```
   xcopy b: /s /e
   ```

3. Repeat step two for all the remaining LDK source diskettes.

4. Change the PATH environment variable such that it includes the LDK tools.

   Sample keyin:

   ```
   path=d:\laddr\laddrutl;%path%
   ```

   CAUTION: Note that it is your reponsibility to determine if there are any naming conflicts between files on the LDK diskettes and other programs on your PC, and to resolve any such conflicts.

   Also note that the LDK tools are only intended to be used in connection with the development of LADDR compliant device drivers for OS/2.

Making the include files

This kit contains global header files in C format and also the corresponding include files in assembler format.

The include files are made from the header files.

The include files are made automatically if you assemble all of the source code. However, if you do not assemble all the source code, you can manually make them by switching to the appropriate sub-directory and invoking NMAKE or some similar program. Sample kevin:

```
cd \laddr\laddrh
nmake
```

Assembling/Compiling the source

You do not need to assemble any of the source code in this as the binaries which are made from it are included in the ...\binaries\debug and ...\binaries\retail subdirectories.

If you decide to assemble all the source code, go to the directory created in step 1 of "Installing the source" and invoke NMAKE or some similar program. Sample keyin:

```
cd \laddr
nmake
```

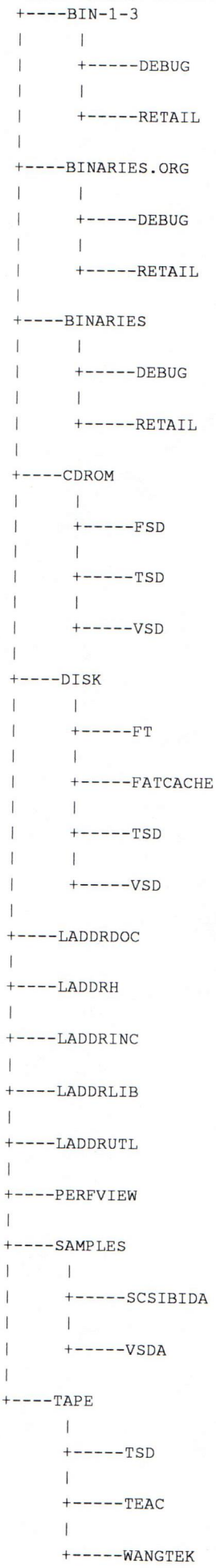
If you decide to assemble some, but not all, of the source, go to the sub-directory that you want to assemble and invoke NMAKE or some similar program. Note that NMAKE will invoke itself in lower level sub-directories if necessary.

CAUTION: All of the sample source files in this kit were compiled using the Microsoft C 6.0a compiler, assembled using the Microsoft Macro Assembler v. 5.10 and linked using LINK 5.10 provided with Microsoft C 6.0a and the small model, protect mode, Microsoft C 6.0a runtime libraries and the DOSCALLS library provided in the LDK. Using other versions of these tools could result in compile, assembly or runtime errors.

3. Directory Tree Structure

The following chart illustrates the standard LDK directory structure.

```
LADDR
  |
  +----BID
  |     |
  |     +-----ADAPTEC
  |     |
  |     +-----ESDI
  |     |
  |     +-----FD16-700
  |     |
  |     +-----FD8xx
  |     |
  |     +-----FDINC
  |     |
  |     +-----NCR
  |     |
  |     +-----WD7000AX
  |     |
  |     +-----WD7000EX
  |
  +----BIN-1-21
  |     |
  |     +-----DEBUG
  |     |
  |     +-----RETAIL
  |
```



The following explains the content of the various subdirectories.

LADDR

- This is the highest level directory in the LDK tree. It normally only contains the "master" MAKEFILE

LADDR\BID

- This directory contains only a makefile which creates the standard ESDI, WD7000AX, FD16-700 and FD8XX BIDs.

LADDR\BID\ADAPTEC

- This directory contains the Adaptec BIDs and a MAKEFILE to copy the files to the appropriate BINARIES directory.

LADDR\BID\ESDI

- This directory contains the source code for the standard ESDI BID, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\BID\FD16-700

- This directory contains the source code for the Future Domain FD16-700 BID, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\BID\FD8XX

- This directory contains the source code for the Future Domain FD8XX BID, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\BID\FDINC

- This directory contains common header files for the Future Domain FD16-700 and FD8XX BIDS.

LADDR\BID\NCR

- This directory contains the NCR BIDs and a MAKEFILE to copy the files to the appropriate BINARIES directory.

LADDR\BID\WD7000AX

- This directory contains the source code for the Western Digital WD7000AX BID, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\BID\WD7000EX

- This directory contains the source code for the Western Digital WD7000EX BID, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\BIN-1-21

- This directory contains a makefile which may be used to delete files in preparation or re-making them, and two subdirectories: DEBUG and RETAIL.

LADDR\BIN-1-21\DEBUG

- This directory contains binaries and the associated symbol files for the debug versions of OS/2 v. 1.21 drivers, and a makefile to delete all the files in preparation for re-making them.

  These drivers may contain DPRINTF's and INT 3's.

LADDR\BIN-1-21\RETAIL

- This directory contains binaries and the associated symbol files for the retail versions of OS/2 v. 1.21 drivers. and a makefile

to delete all the files in preparation for re-making them.

LADDR\BIN-1-3

- This directory contains a makefile which may be used to delete files in preparation or re-making them, and two subdirectories: DEBUG and RETAIL.

LADDR\BIN-1-3\DEBUG

- This directory contains binaries and the associated symbol files for the debug versions of OS/2 v. 1.3 drivers, and a makefile to delete all the files in preparation for re-making them.

  These drivers may contain DPRINTF's and INT 3's.

LADDR\BIN-1-3\RETAIL

- This directory contains binaries and the associated symbol files for the retail versions of OS/2 v. 1.3 drivers, and a makefile to delete all the files in preparation for re-making them.

LADDR\BINARIES.ORG

- This directory contains the subdirectories containing the original binaries provided which were built from the LDK sample code.

LADDR\BINARIES\DEBUG

- This directory contains binaries and the associated symbol files for the debug versions of selected LADDR drivers.

  These drivers may contain DPRINTF's and INT 3's.

LADDR\BINARIES\RETAIL

- This directory contains binaries and the associated symbol files for the retail versions of selected LADDR drivers.

LADDR\BINARIES

- This directory contains a makefile which may be used to delete files in preparation or re-making them, and two subdirectories: DEBUG and RETAIL.

LADDR\BINARIES\DEBUG

- This directory is the repositiory for the binaries and the associated symbol files for the debug versions of selected LADDR drivers when they are built using the makefiles provided in the kit.

LADDR\BINARIES\RETAIL

- This directory is the repositiory for the binaries and the associated symbol files for the retail versions of selected LADDR drivers when they are built using the makefiles provided in the kit.

LADDR\CDROM

- This directory contains only a MAKEFILE which creates the standard CD-ROM TSD and the default CD-ROM scsi'izer VSD.

LADDR\CDROM\FSD

- This directory contains the binary for the CD-ROM file system, the DLL for the CD-ROM utility programs (CHKDSK and FORMAT), and a README file that contains instructions for installing the CD-ROM file system.

LADDR\CDROM\TSD

- This directory contains the source code for the standard CD-ROM TSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\CDROM\VSD

- This directory contains the source code for the default CD-ROM

scsi'izer VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\DISK

- This directory contains only a MAKEFILE which creates the standard disk TSD and the default disk scsi'izer VSD.

LADDR\DISK\FATCACHE

- This directory contains the source code for the default fat cacheing VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\DISK\FT

- This directory contains the source code for the default fault tolerant VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\DISK\TSD

- This directory contains the source code for the standard disk TSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\DISK\VSD

- This directory contains the source code for the default disk scsi' izer VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\LADDRDOC

- This directory contains a test and HP LaserJet version of the LADDR Programming Documentation

LADDR\LADDRH

- This directory contains all the LADDR standard header files in "C" format, and a MAKEFILE to create assembler compatible include files.

LADDR\LADDRINC

- This directory contains the assembler compatible include files which are created from the header files in the LADDRH subdirectory, and also a MAKEFILE to create them.

LADDR\LADDRLIB

- This directory contains standard OS/2 libraries which may be needed to create certain device drivers

LADDR\LADDRUTL

- This directory contains a several utility routines which simplify routine tasks such as changing subdirectories.

LADDR\PERFVIEW

- This directory contains the debug/trace tool VIEW.EXE

LADDR\SAMPLES

- This directory contains only a MAKEFILE which creates the sample assembler language scsi BID and the sample assembler language scsi' izer VSD.

LADDR\SAMPLES\SCSIBIDA

- This directory contains the source code for the sample assembler language scsi BID, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\SAMPLES\VSDA

- This directory contains the source code for the sample assembler language scsi'izer VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\TAPE

- This directory contains only a makefile which creates the sample TAPE TSD and VSDs

LADDR\TAPE\TEAC

- This directory contains the source code for the TEAC Tape VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\TAPE\TSD

- This directory contains the source code for the Tape TSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

LADDR\TAPE\WANGTEK

- This directory contains the source code for the TEAC Tape VSD, and a MAKEFILE to create its object, listing, and binary files. It may also contain those object, listing, and binary files.

4. NMAKE/MAKEFILE Considerations

Every directory which contains source code, or has source code in any of its subdirectories, contains a MAKEFILE that will assemble, compile, and/or link the in the directory and all its subdirectories.

Thus, one can pick a level in the directory structure and make everything at that level and below it.

Normally NMAKE is invoked without parameters, which results in only changed files being assembled/compiled/linked. But NMAKE can also be invoked with the parameter CLEAN, which causes all source files to be made, or DEPEND, which causes the dependency information in the MAKEFILE to be refreshed.

CAUTION: use of the DEPEND option will cause the programs SED and INCLUDES to be invoked. Because of licensing restrictions, these programs are not shipped with the LDK and you must purchase them seperately.

Note that use of the DEPEND option is not normally required.

5. BASEDD special considerations

BASEDDxx.SYS is the component of OS/2 which contains the list of drivers to be loaded at boot time. In most OEM adaptations of OS/2, this component also combines the base disk, clock, screen, keyboard, and printer interface drivers (this is done to save space in the DOS box). Because BASEDD contains these hardware dependent drivers, it is a part of OS/2 that OEM's normally change for their hardware. It is not generally possible to create a BASEDD that will run on all PC's. For these reasons. Microsoft releases the source code for BASEDD only to

OEM's.

The BASEDDxx that Microsoft shipped to OEM's for release 1.21 did not contain the appropriate load list entries for LADDR devices. The sample boot floppy included with this LDK does contain a correct load list. If your PC is 100% IBM compatible, that basedd may be used for testing purposes in place of the one shipped with release 1.21 of OS/2. In order to run your device driver on top of an OEM adaptation of OS/2, however, you will need to wait until the OEM incorporates the new LADDR load list into their adaptation.

The BASEDDxx shipped with OS/2 release 1.3 contains the appropriate module names to fully support LADDR.

6. LADDR documentation

This kit contains the following documentation files:

LADDR.LAS
- this file contains formatting commands for HP laser jet printers using the 'Z' cartridge.

LADDR.TXT
- this file does not contain any printer control codes and is intended to be viewed with an editor rather than being printed.

CAUTION: These documents are a specification for the LADDR architecture, not a how to build a LADDR driver document. Please refer to the samples provided as examples for the various types of LADDR modules. Where the specification differs from the sample code and header files the code should be considered correct.

7. Device driver naming

The following device driver names are present in the base load list of the LADDR version of the BASEDD for ISA class PC's:

clock01.sys
screen01.sys
kbd01.sys
ios12.sys
ioconfig.sys
bootbid.bid
esdi-506.bid
aha152x.bid
aha154x.bid
aha174x.bid
wd7000ax.bid
wd7000ex.bid
wd7000sx.bid
spare001.sys
spare002.sys
stddisk.vsd
stdcdrom.vsd
stdtape.vsd
stdprint.vsd
sg.vsd
ft.vsd

fatcache.vsd
spare003.sys
print01.sys
floppy01.sys
disk.tsd
printer.tsd
cdrom.tsd
tape.tsd
iorun.sys

To test your LADDR modules you should rename the file to one of the "spare???.sys" names above and place the file in the root directory of the boot drive. Loading modules through CONFIG.SYS "device=" is not a recommend approach for LADDR driver installation.

8. Making a boot floppy

Perform the following steps to make a boot floppy for testing your device driver.

1. Use DISKCOPY or some equivalent program to make a copy of the sample boot floppy included in this kit.

2. Select an appropriate name for your device driver binary from the list of names in the base load list.

3. Copy your device driver binary into the root directory of the duplicate sample boot floppy, renaming your device driver binary to the name selected in step 2.

9. Testing your device driver

This LDK includes support for the OS/2 kernel dubugger.

The kernel debugger interacts with you via a SYMDEB like interface and a dumb terminal or terminal emulator program.

To use the kernel debugger, set up your test PC as follows:

1. Ensure that the PC does not contain a COM2 serial port.

2. Ensure that it does contain a COM1 serial port and connect that port, via an appropriate cable, to a dumb terminal or a terminal emulator.

3. Configure the terminal to operate at 9600 baud, with 8 data bits, 1 stop bit, and no parity bit, and enable XON/XOFF support.

When you boot OS/2 from the sample boot floppy, the first message displayed on the terminal is:

System Debugger 09/01/88 [80386]

The precise content of the next few messages depends on exactly how your system is configured.

Note that the drivers included on the sample boot disks in this kit are "retail" drivers and do not generate messages to the debug terminal. These drivers may, of course, be replaced with "debug" drivers.

If you are using the debug version of IOS, you will see something similar to the following:

```
Symbols linked (IOS)
...
... three lines of register content
...
xxxx:xxxx CC          INT   3
```

At this point the system is stopped and the debugger is waiting for a keyin.

If you enter "g<return>" the system will continue executing and will display a message similar to the following:

```
Symbols linked (IOCONFIG)
```

If your system includes support for ESDI/ST506/IDE fixed disk drives, messages similar to the following will be displayed:

```
Symbols linked (ESDIBID)
IOSBID:   start bid registration
IOSBID:   start "find_ice"
ESDIINIT: starting registration
IOSBID:   start esdi bid registration
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
IOSBID:   esdi device found - unit within ctlr = 0000 inquiry
                                                      data follows
...
... three lines of inquiry data in hex and ascii
...
(ESDI)BIDAER: start configure device
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
(ESDI)BIDAER: start device inquiry
(ESDI)BIDAER: end device inquiry
IOSBID:   start "find_ice"
IOSBID:   end bid registration
```

If your system includes other BID's or VSD's, messages may be displayed as they register.

Eventually, the floppy driver and the disk TSD will load. This will cause messsages similar to the following to be dispalyed:

```
Symbols linked (FLPPY01)
Symbols linked (TSD)
IOSTSD:   start tsd registration
TSDAER:   about to set rom config in dcb xxxx disk drive 0000
          before: cyls = 0000 heads = 0000 spt = 0000 phys dcb
                                                      = xxxx
          after:  cyls = xxxx heads = xxxx spt = xxxx phys dcb
                                                      = xxxx
TSDAER:   good mbr found - dcb xxxx disk drive 0080
```

```
...
... several lines of hex and ascii data
...
IOSTSD:   end tsd registration
TSDPART:  good pbr - logical dcb xxxx - physical dcb xxx
                                                - disk drive 0000
...
... several lines of hex and ascii data
...
```

No further messages are normally dislpayed.

OS/2 LADDR Device Driver Development Kit

Microsoft OS/2 LADDR Compliant Device Driver Specification

Version 1.0

for the MS OS/2 Operating System

Microsoft Corporation

Part Number 20782

Contents

Contents

List of Illustrations

PREFACE:

This document contains the specification for LADDR compliant device drivers for the following Microsoft operating system platforms:

- OS/2 release 1.21
- OS/2 release 1.3x.

This specification defines a device driver architecture in which device drivers are split into physically separate but logically related pieces.

This architecture provides the following:

- A software organization that more closely matches current hardware organization
- The uncoupling of device type specific, device vendor specific and bus interface specific code development, integration, maintenance, and support activities
- Simplification of end user product selection and installation process for add on peripherals
- The ability of a vendor to easily provide vendor specific software without having to support an entire device driver
- A mechanism to allow software vendors to create products that enhance the operation and performance of device drivers

1.0 ARCHITECTURAL OVERVIEW:

LADDR is the horizontally LAyered Device Driver aRchitecure used by Microsoft in selected versions of OS/2.

It provides device/adapter "plug and play" by dividing device drivers into functionally discrete binary files.

The following diagrams illustrate the relationship of LADDR device drivers to other major pieces of software, the layering within typical minimal LADDR device drivers, and an example of one way to enhance the driver through expanded layering.

The relationship between a LADDR disk device driver and the system as a whole:

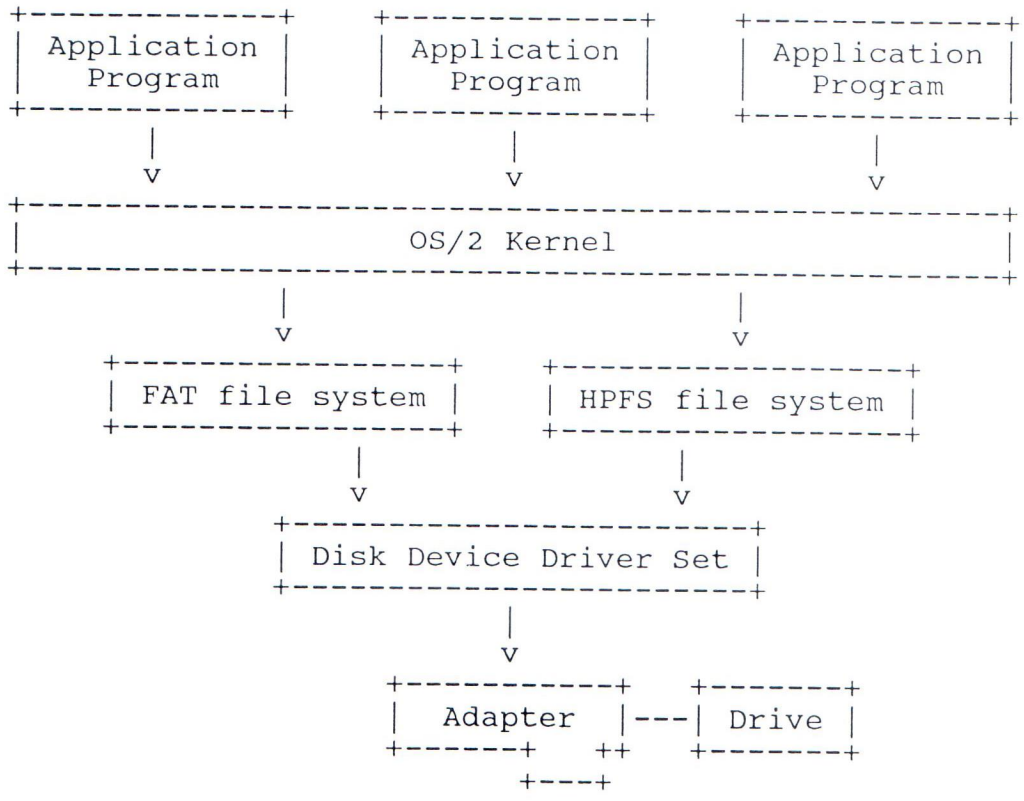


Figure 1. Relationship of LADDR Disk Device Driver to the System

Illustration of layering of a minimal SCSI LADDR disk driver:

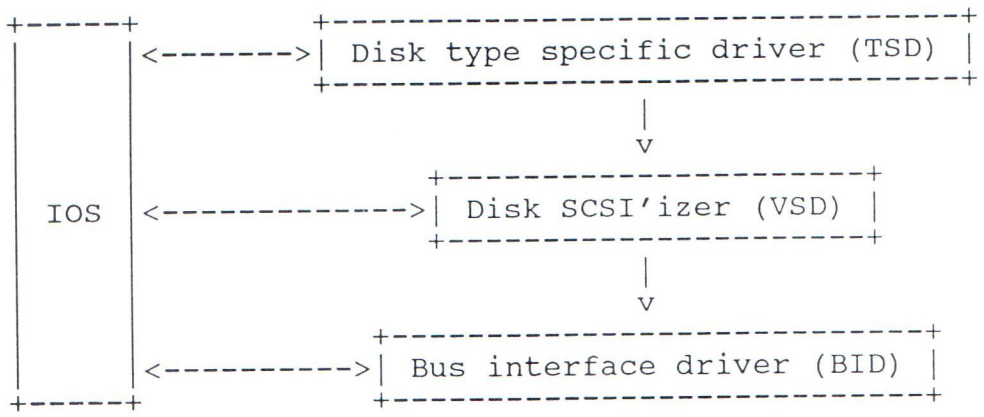
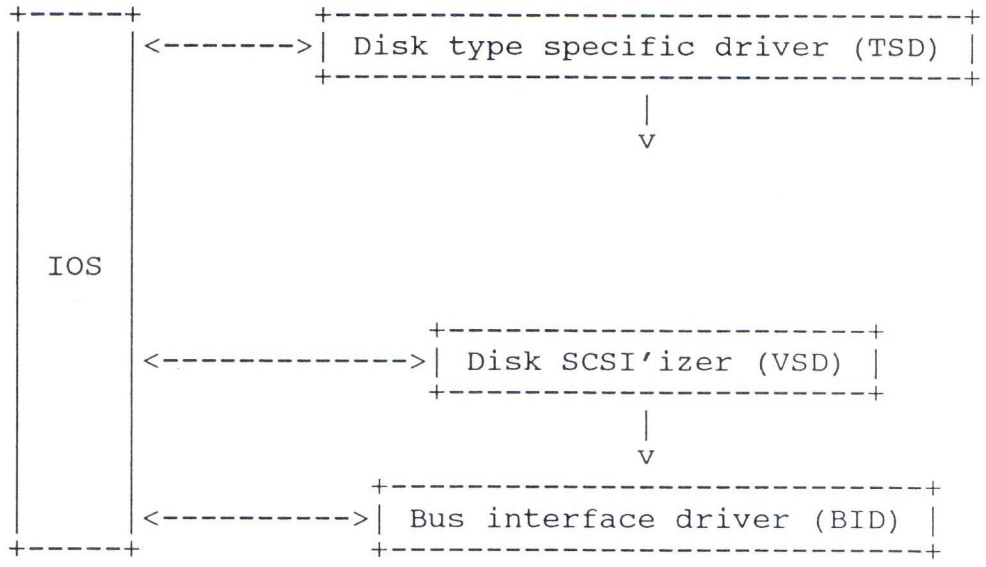


Figure 2. Minimal LADDR SCSI Device Support

To increase functionality, to add caching for example, the SCSI'izer VSD and the BID can be moved down:



and a caching VSD inserted:

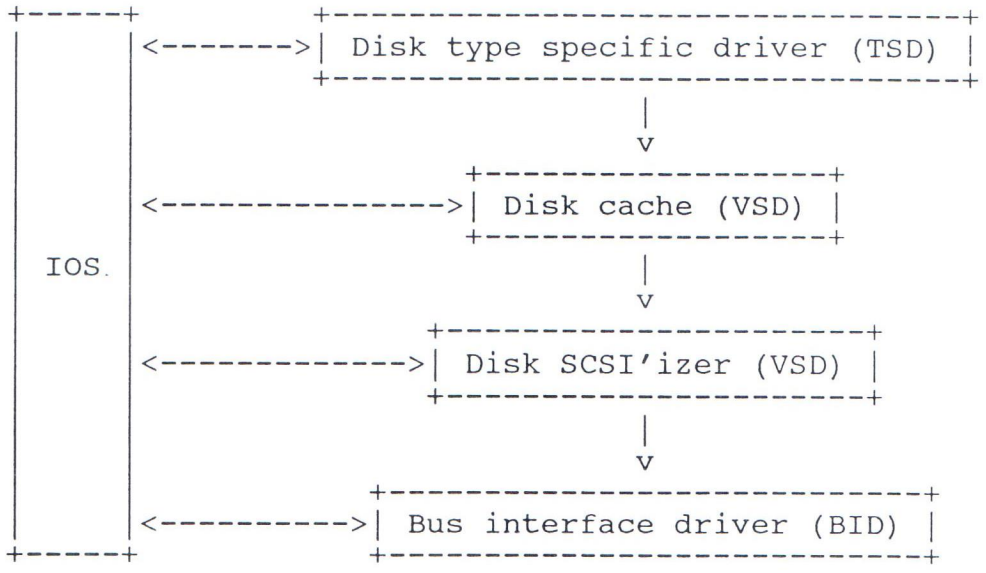


Figure 3. Adding a Disk Cache VSD

The relationship between a LADDR tape device driver and the system as a whole:

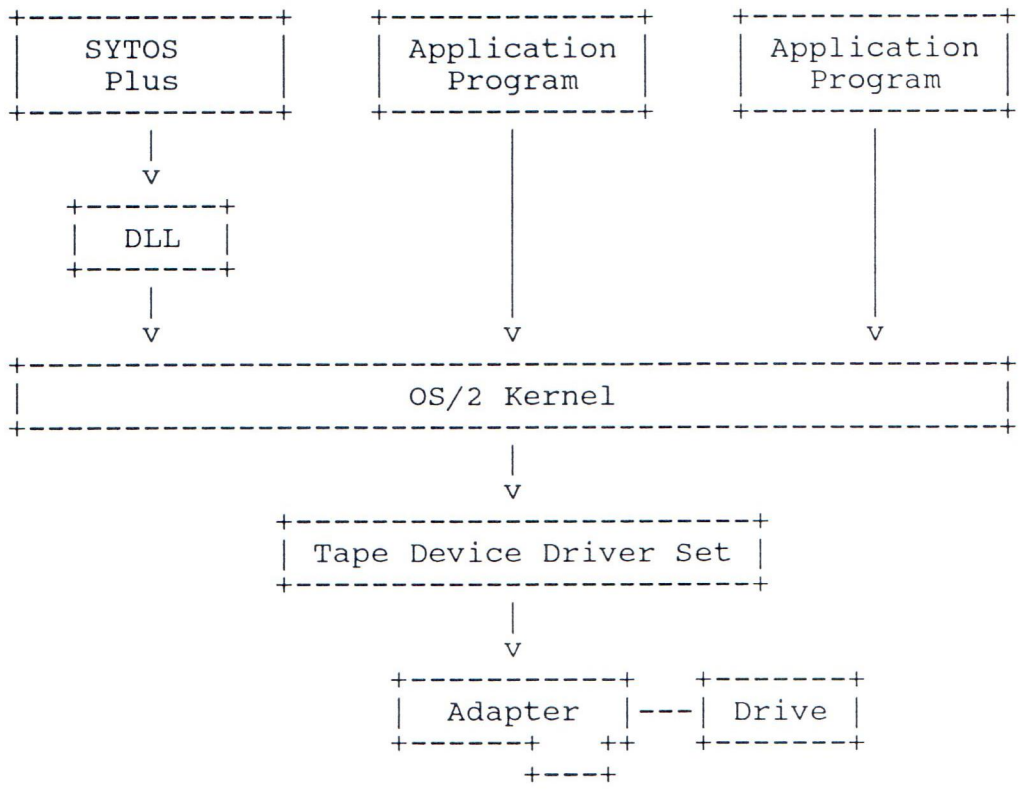


Figure 4. Relationship of LADDR Tape Device Driver to the System

Illustration of layering of a minimal SCSI LADDR tape driver for one specific vendor's drive:

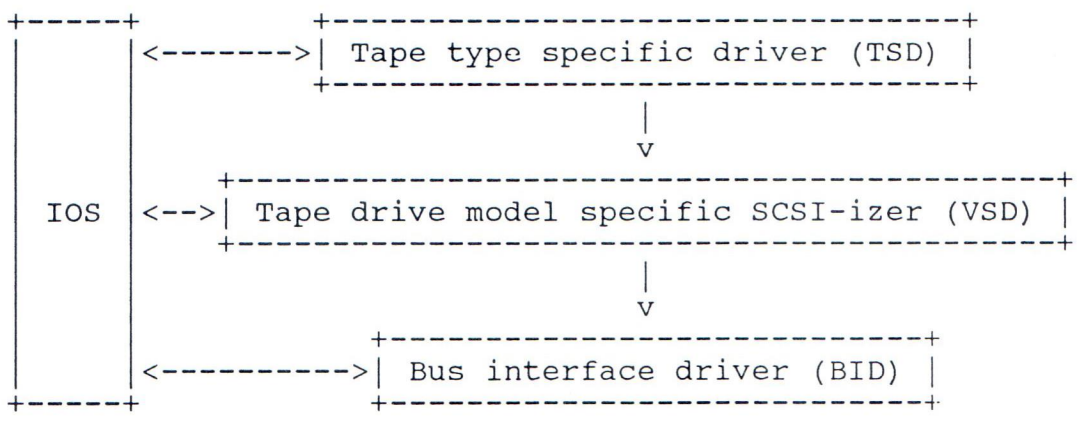
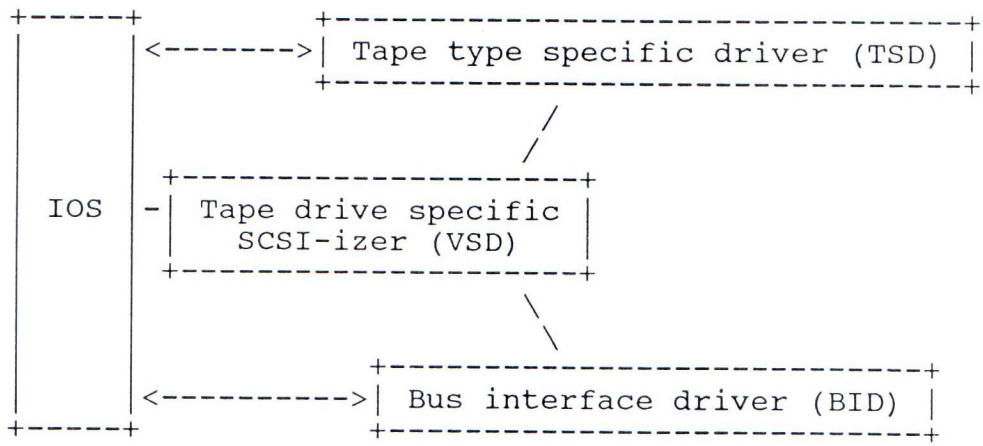


Figure 5. Minimal LADDR Tape Device Support

To support a second kind of drive, the SCSI'izer VSD can be moved aside:



and another SCSI'izer VSD inserted:

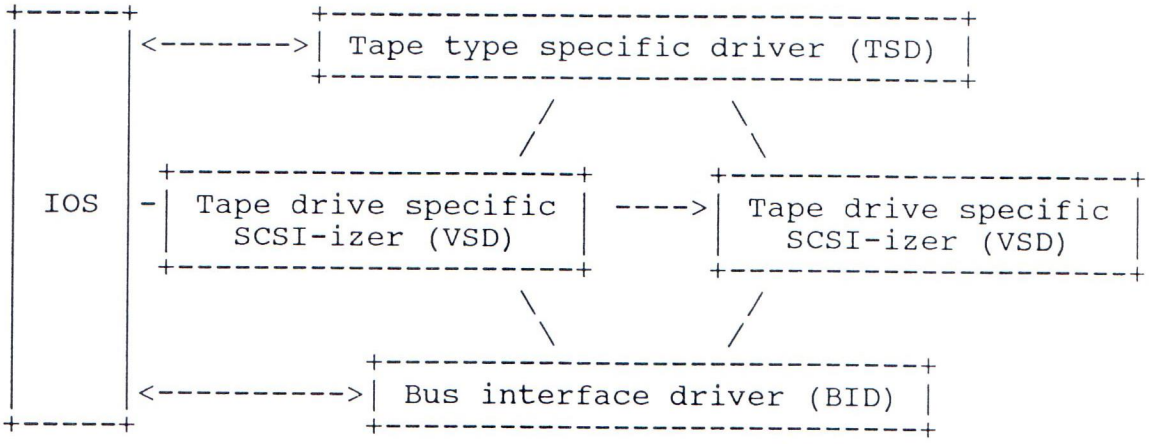


Figure 6. Adding Additional Tape Device and SCSI Support

The relationship between a LADDR CD-ROM device driver and the system as a whole:

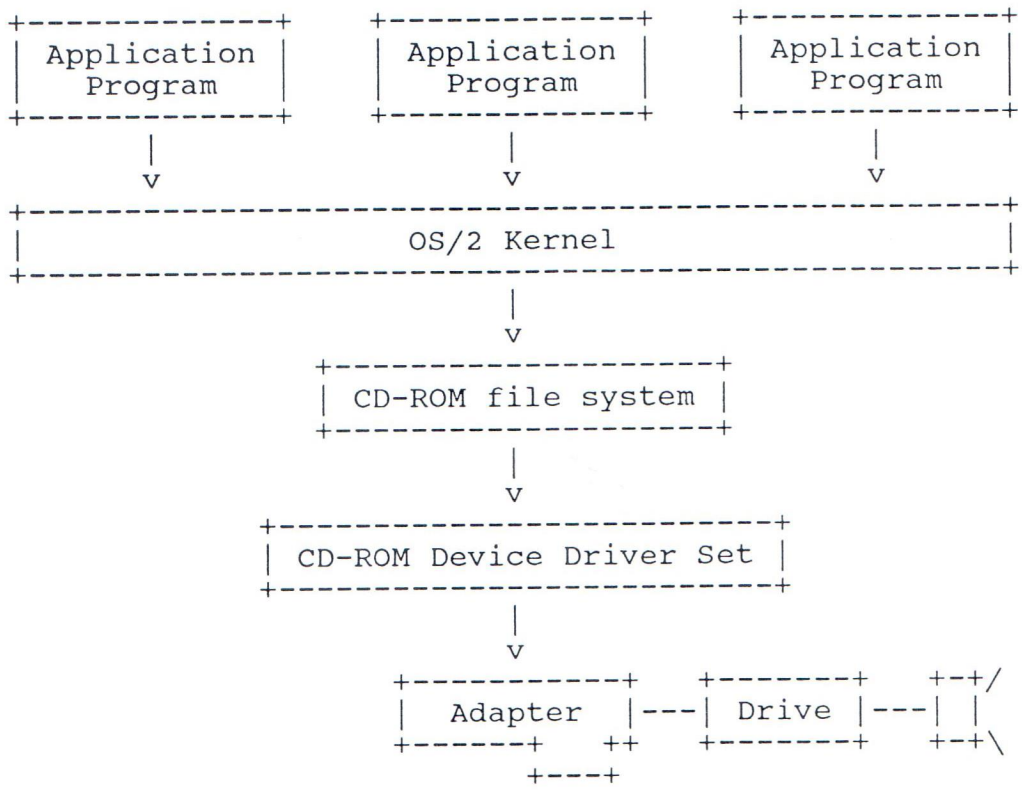


Figure 7. Relationship of LADDR CD-ROM Driver to the System

Illustration of layering of a minimal SCSI LADDR CD-ROM driver for SCSI-2 command set drives:

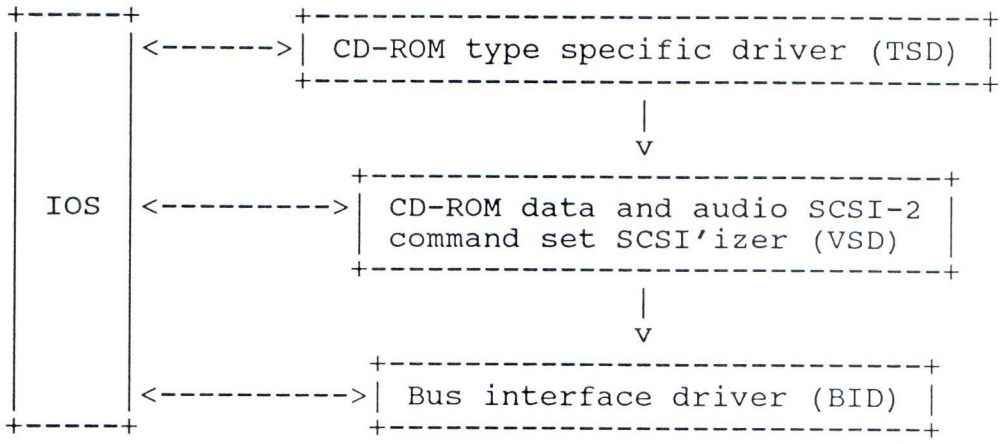
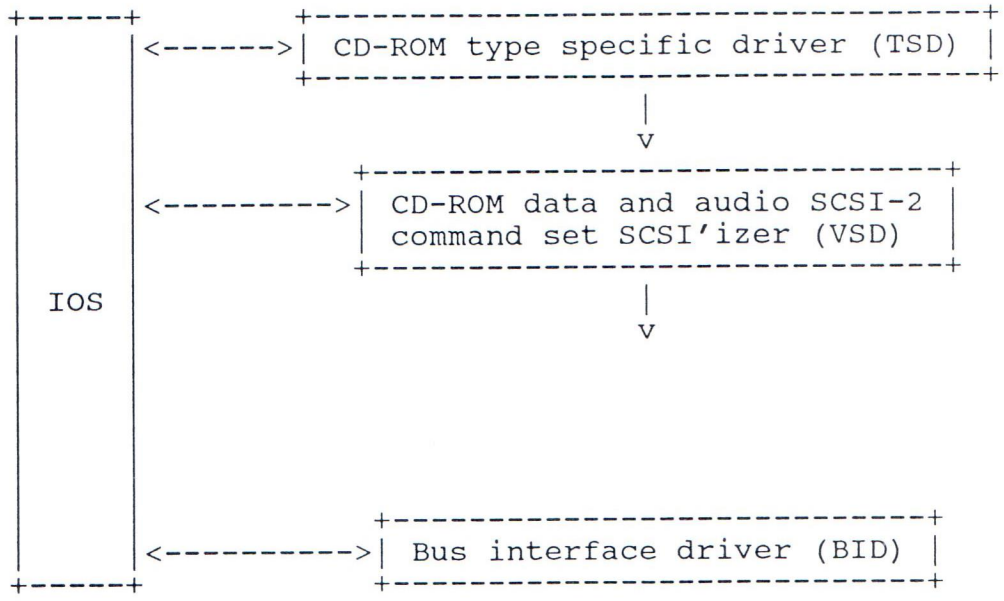


Figure 8. Minimal LADDR SCSI-2 CD-ROM Support

To support a drive with non-SCSI-2 audio, the BID can be moved down:



and a vendor specific audio SCSI'izer VSD inserted:

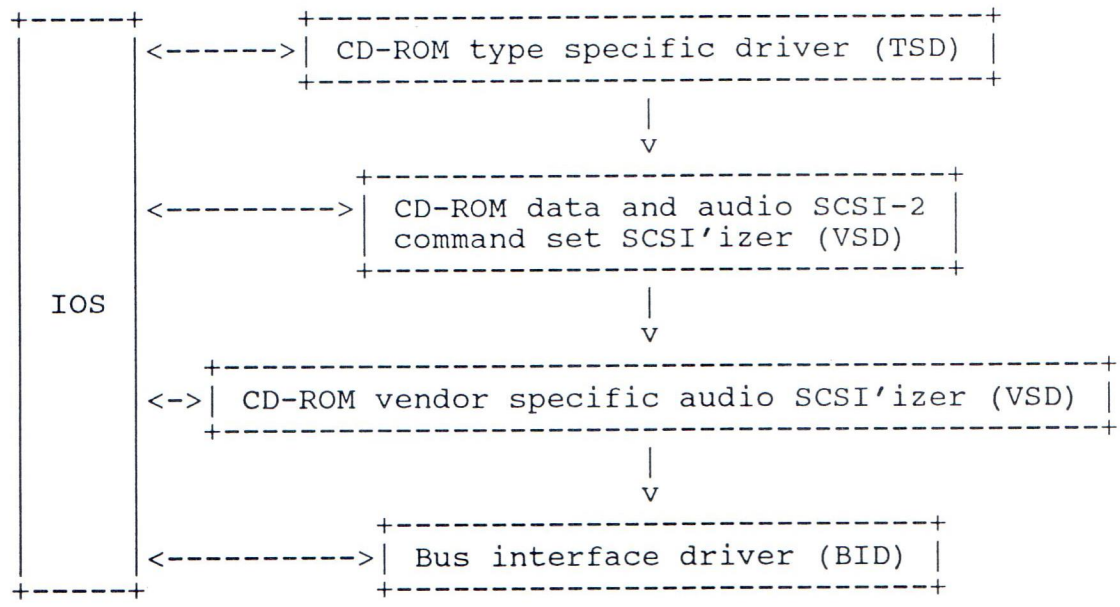


Figure 9. Adding Support for CD-ROM Audio

The relationship between a LADDR SCSI printer device driver and the system as a whole:

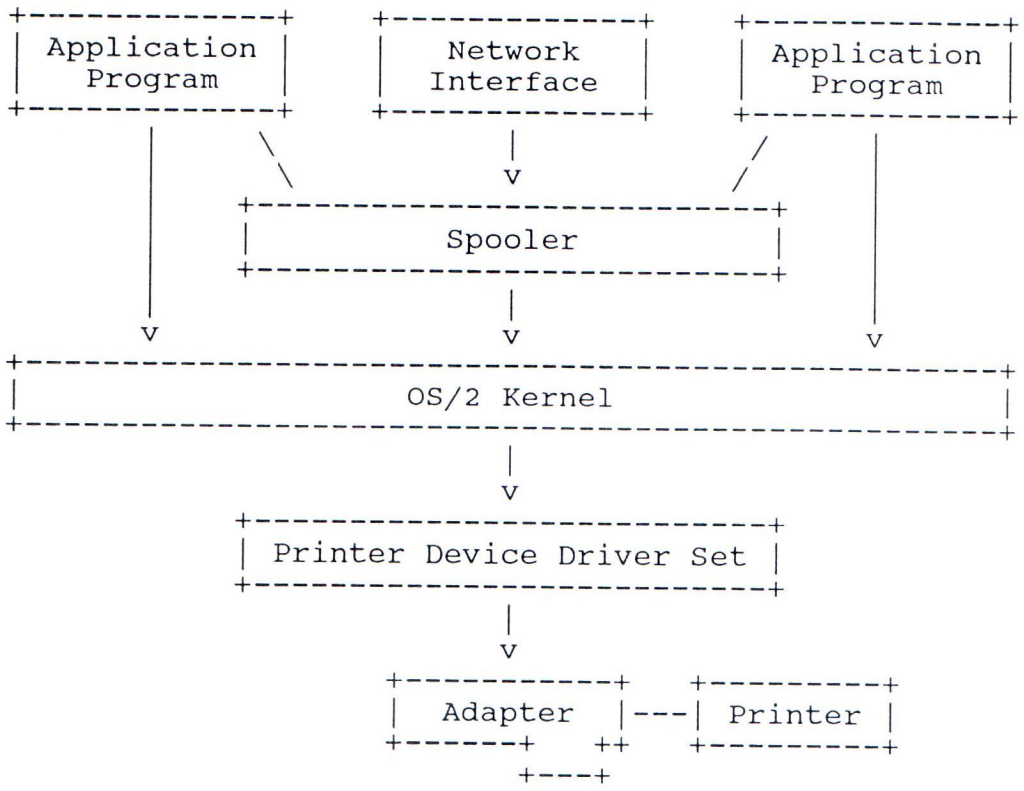


Figure 10. Relationship of LADDR Printer Driver to the System

Illustration of layering of a minimal SCSI LADDR printer driver:

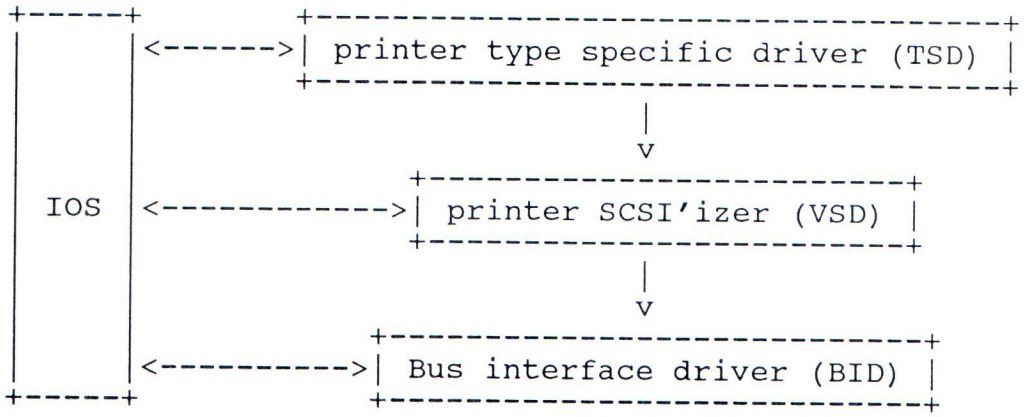
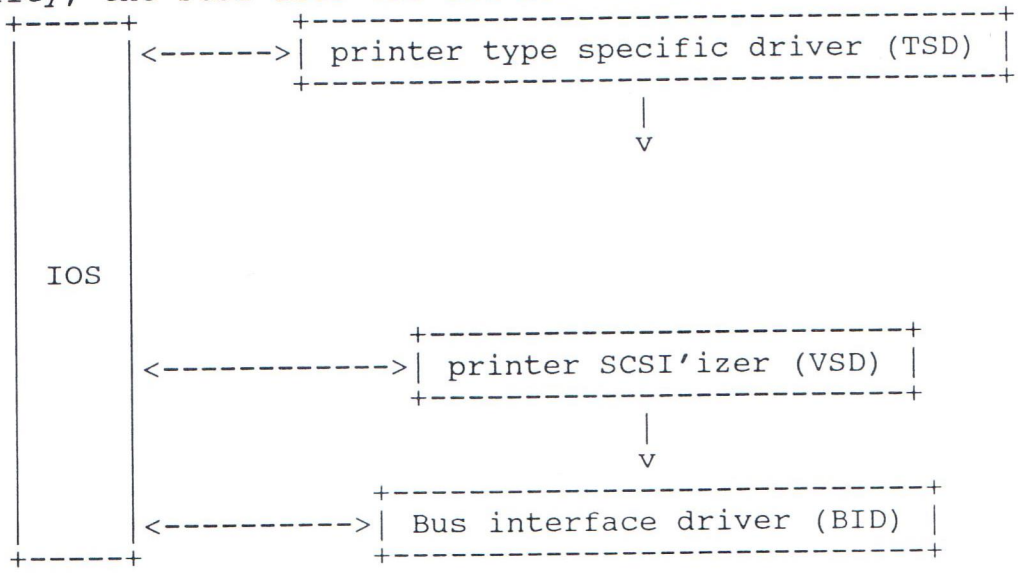


Figure 11. Minimal LADDR Printer Device Support

To support a printer with a vendor unique data de-compression capability, the SCSI'izer VSD and BID can be moved down:



and a vendor specific data compression VSD inserted:

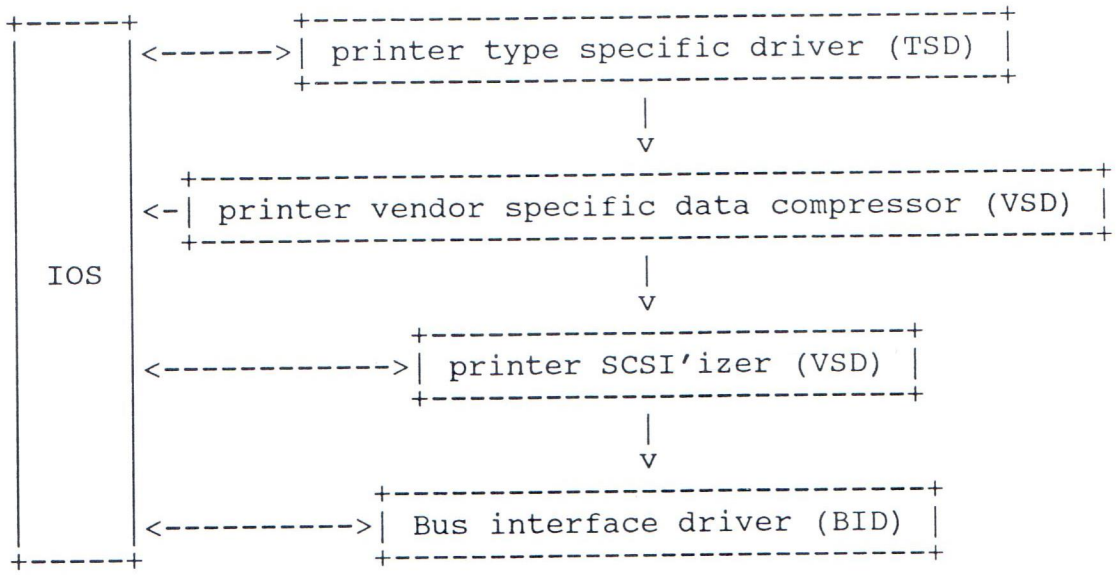


Figure 12. Enhancing LADDR Printer Support

I/O Complex Elements

Within the LADDR specification, the term "I/O complex" is used to refer to the collection of software that includes device drivers and related operating system services, initialization routines, configuration processing, interrupt routing, and timer services.

The major pieces of a LADDR I/O complex are:

I/O Supervisor (IOS) provides system services and initialization, configuration, interrupt routing, and timer services.

Type Specific Layer contains drivers (called "type specific drivers" or "TSD's") that perform device type specific processing such as validation of incoming I/O requests and conversion of logical requests to physical requests.

For any given device, this layer must contain exactly one TSD.

Vendor Enhancement Layer contains drivers (called "vendor specific drivers" or "VSD's") that provide vendors with a means to add value to their products and to compensate for aberrant operation of their peripherals, and provides a common point to build standard interface specific data structures such as SCSI's CDB.

For any given device, this layer may contain zero, one, more VSD's.

Request Routing Layer contains drivers (called "path selection drivers" or "PSD's") that pick the optimal I/O path for the request.

For any given device, this layer may contain zero or one PSD.

Bus Interface Layer contains drivers (called "bus interface drivers" or "BID's") that perform final conversion of request data structures and execute appropriate instructions to pass request to the bus interface hardware and to control that hardware's operation.

For any given device, this layer must contain exactly one BID.

For a given device, the LADDR I/O complex contains at least IOS, a TSD, and a BID, and may contain one or more VSD's and/or a PSD.

Driver Initialization

Initialization of a LADDR compliant driver occurs at one of two points in time:

- At boot time. It is at boot time that IOS; the TSD, the VSD (if there is one), and the bid for the boot device must load

and initialize.

Many other drivers are loaded at boot time as a matter of convenience for the user.

The file BASEDD0x.asm, which is part of BASEDD0x.SYS, contains a list - called "the load list" - of all drivers which are to be loaded and initialized at boot time.

During boot time initialization, drivers execute entirely at ring zero.

- During steady state either as a result of CONFIG.SYS processing or as the result of dynamic configuration processing by a ring 3 application program such as an INSTALL program.

  During steady state initialization, drivers initially gain control at ring three but soon execute a "registration" IOCTL which results in the bulk of initialization being performed at ring zero. The tail end of the initialization processing is also performed at ring three.

Request Processing

During initialization processing, for each device, the drivers construct a table of driver linkage and control information which is referred to as the "driver calldown table", or sometimes just the "calldown table".

As a request is processed, a table of completion time callout addresses, which is known as the "callback table", is built by the drivers.

Conceptually, as a request is processed, control flows "down" the driver calldown table from a TSD to a BID (leaving return addresses on the system stack), possibly via one or more VSD's and a PSD.

When the request reaches the lowest point, it is either executed, initiated, or queued and control is then returned "up" through the system stack.

When a driver gets its "turn" at processing the request, it has the following options:

- it may completely execute the request and initiate callout processing
- it may perform appropriate processing and then pass the request on to the next driver.
- it may queue the request locally and generate a new request

of its own, which then may be processed either by passing it down to the next driver, or by passing it to IOS for normal STRAT2 request processing.

Interrupt Processing

During its initialization, an interrupt driven driver registers its interrupt service routine (or routines) with IOS.

Later, when an interrupt occurs, IOS uses the interrupting IRQ number as an index to locate the drivers which are servicing interrupts from that IRQ level. These drivers are then serially given control and an opportunity to process the interrupt.

Completion Processing/Callback Processing

When a completion event - such as a timeout or I/O completion interrupt - occurs, the BID that recognizes that event initiates "callback" processing.

For BID's that operate on SRB's, the BID performs a "request element complete" callout through field SRB_callback. Additionally, the BID decrements field RCB_countdown and, upon that field becoming zero or all non-completed requests being aborted, the BID unstacks the lowest entry in the RCB callback stack and performs a "request complete" callout through that unstacked entry.

For other BID's, the BID updates fields RH_status and RLH_1st_status, and, subject to the conditions specified by field RH_request_control, performs a "request element complete" callout through field RH_notify_addr. Additionally, when the BID determines that the sum of the requests completed or aborted becomes equal to the value in the field RH_count_lo, the BID unstacks the lowest entry in the RCB callback stack and performs a "request complete" callout through that unstacked entry.

When a request or request element completion callout passes control to a VSD, the VSD must determine what if any consequential callouts are required, and perform them. Typically, these callouts are RCB and/or RH related.

When a TSD is the target of a request completion callout, the TSD must perform any required RLH related callouts. It must also perform any required RH related callouts, if, and only if, the RH was NOT passed to a VSD or BID - if the RH was passed to a VSD or BID, RH related callouts would already have been performed at that lower level.

Internal Interface

While the interfaces between the various layers are defined by a single common standard, though the use of "demand bits" a driver in one layer can indicate to a higher layer that the higher layer select one of several optional processing modes. These optional processing modes are typically used to deal with hardware constraints in either the driver or its associated device.

If a given driver does not support a particular demand bit, it passes that demand bit to the layer above.

Thus, a TSD does not need to know if it passes control to a VSD, PSD or a BID, and a VSD does not need to know if it passes control to another VSD or to a PSD or BID. Similarly, VSD's, PSD's and BID's do not need to know if they are receiving control from a TSD or a VSD.

Data Structures

The following is a brief explanation of the purpose of the various data structures used by the I/O complex.

Request Control Block (RCB) represents a request being processed by the I/O complex. It contains pointers to the other data structures involved in this request. It also contains a few fields that hold status or control information.

STRAT2 Request Packet (RLH) contains the specifics of a request being processed by the I/O complex. The RLH is the preferred I/O request packet for file systems. When file systems use the alternate request packet - the STRAT1 request packet, IOS converts it to a RLH.

STRAT1 Request Packet (RP) is the traditional I/O request packet. It is being phased out in favor of the RLH. IOS converts RP's to RLH's before passing to the drivers.

Device Control Block (DCB) contains device specific information such as the device address and pointers to the relevant device drivers.

SCSI Request Block (SRB) contains the SCSI CDB and other related information. Either a TSD or VSD builds a SRB from a RLH when a BID indicates that an SRB is required.

2.0 THE STRAT2 INTERFACE - AKA "SCATTER/GATHER":

The STRAT2 interface only supports asynchronous I/O activity. Since polling for I/O completion results in severely degraded system performance in OS/2, the STRAT2 interface includes a request completion notification facility.

This overview consists of six parts:

- Request submission
- STRAT2 Interface Calling Conventions
- The STRAT2 Packet
- Request completion notification
- I/O Request Synchronization
- Callout Calling Conventions

Overview of Request Submission

To submit a request to the STRAT2 interface, the requestor builds an object, called a "STRAT2 packet", which contains a Request List Header (more usually known as an "RLH"), one or more Request Headers (or "RH"), and one or more Scatter/Gather Descriptors (or "SGD's"). Having built this packet, the requestor then calls the STRAT2 interface with a pointer to the RLH.

Overview of STRAT2 Interface Calling Conventions

Having previously acquired the address of the STRAT2 entry point via a "Get DCSVCS" call to STRAT1, the requestor points ES:BX to a STRAT2 packet (more specifically, to the RLH) and then calls the STRAT2 entry point. Note that all of the data buffers pointed to by the STRAT2 packet must be locked down prior to the call to STRAT2.

STRAT2 performs some processing on the request and then returns control to the requestor. At this time, the request may or may not have been completely processed.

Overview of the STRAT2 Packet

The RLH contains global information that pertains to all the RH's contained in the RLH, such as the unit number. It also contains one or more RH's.

Each RH describes a piece of the request that involves one continuous area on disk. It also contains one or more SGD's.

The SGD provides the address and length of a physically continuous area of real memory which constitutes all or part of the data buffer.

Overview of Request Completion Notification

In order to be notified of the completion of a STRAT2 request, the requestor provides one or two pointers to its notification routines. The disk driver calls the requestor provided routines when appropriate. The requestor has several options as to when the notification occurs.

Overview of I/O Request Synchronization

Since the STRAT2 interface does not provide synchronous I/O request processing, higher level software must achieve request completion synchronization when necessary.

Such higher level software typically sets a semaphore prior to submitting the request and then clears the semaphore in an RLH or RH notification routine. Note that the notification may occur before control is returned from the request submission code.

Overview of Callout Calling Conventions

As the processing of the request progresses, the disk driver makes the appropriate callouts to the notify routines as specified in the RLH and RH.

RLH-level notify routines gain control with ES:BX pointing to the relevant RLH. RH-level notify routines gain control with ES:BX pointing to the relevant RH.

3.0 THE I/O SUPERVISOR (IOS):

The I/O Supervisor controls initialization and configuration of cooperating LADDR compliant drivers; and provides various services to those drivers.

The following IOS documentation is included to provide a better understanding of the operation of LADDR compliant drivers. It is neither intended as complete IOS documentation, nor as an IOS programming specification.

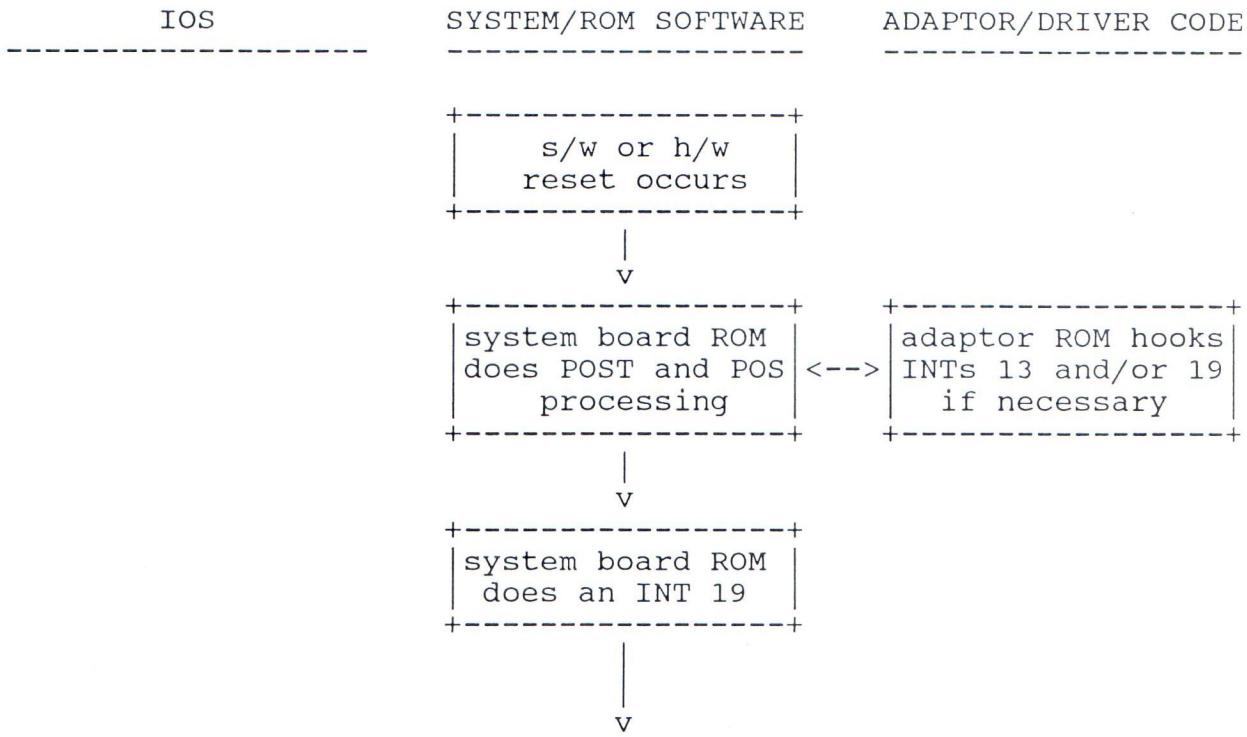
The following topics are considered:

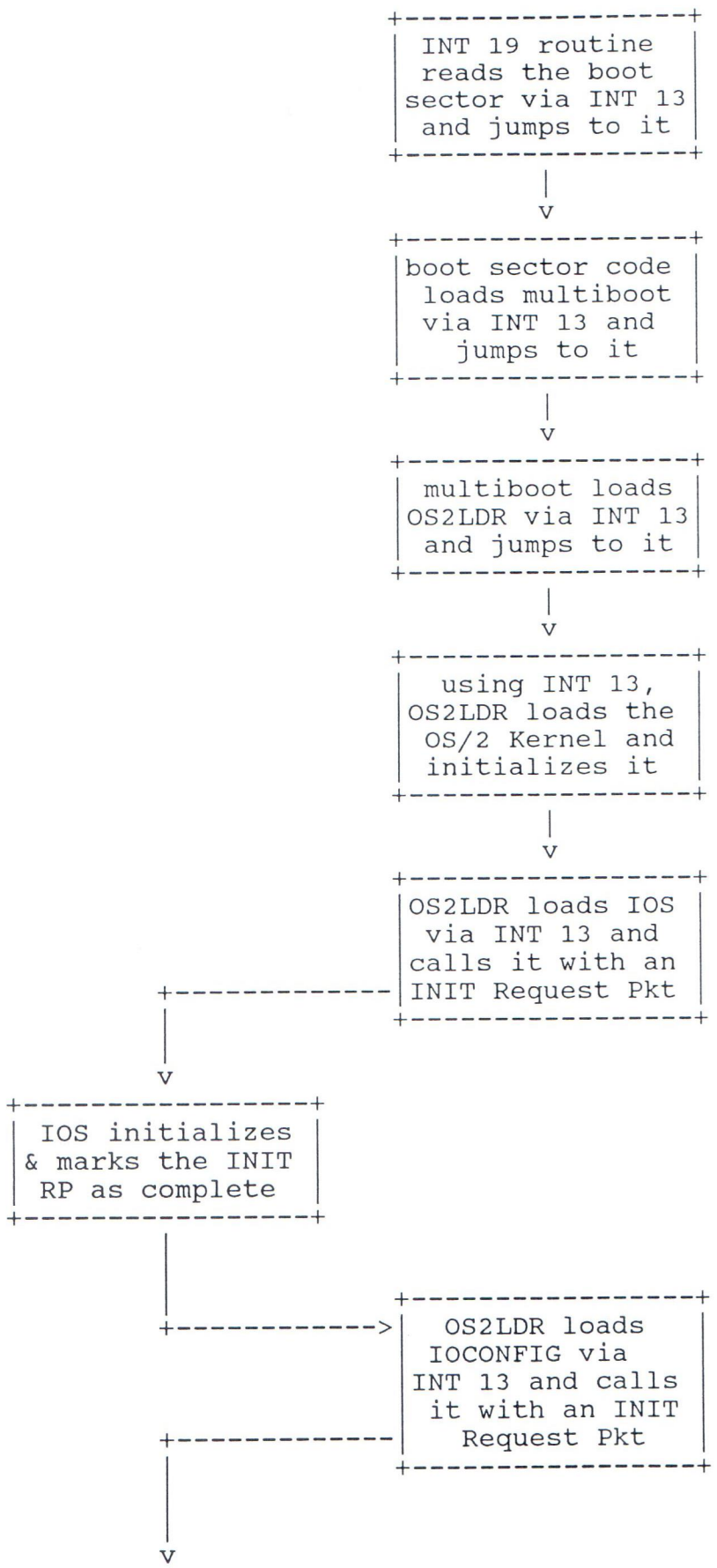
- Boot Time Processing Overview
- IOS Initialization
- IOS Configuration Processing for Drivers
- IOS Services

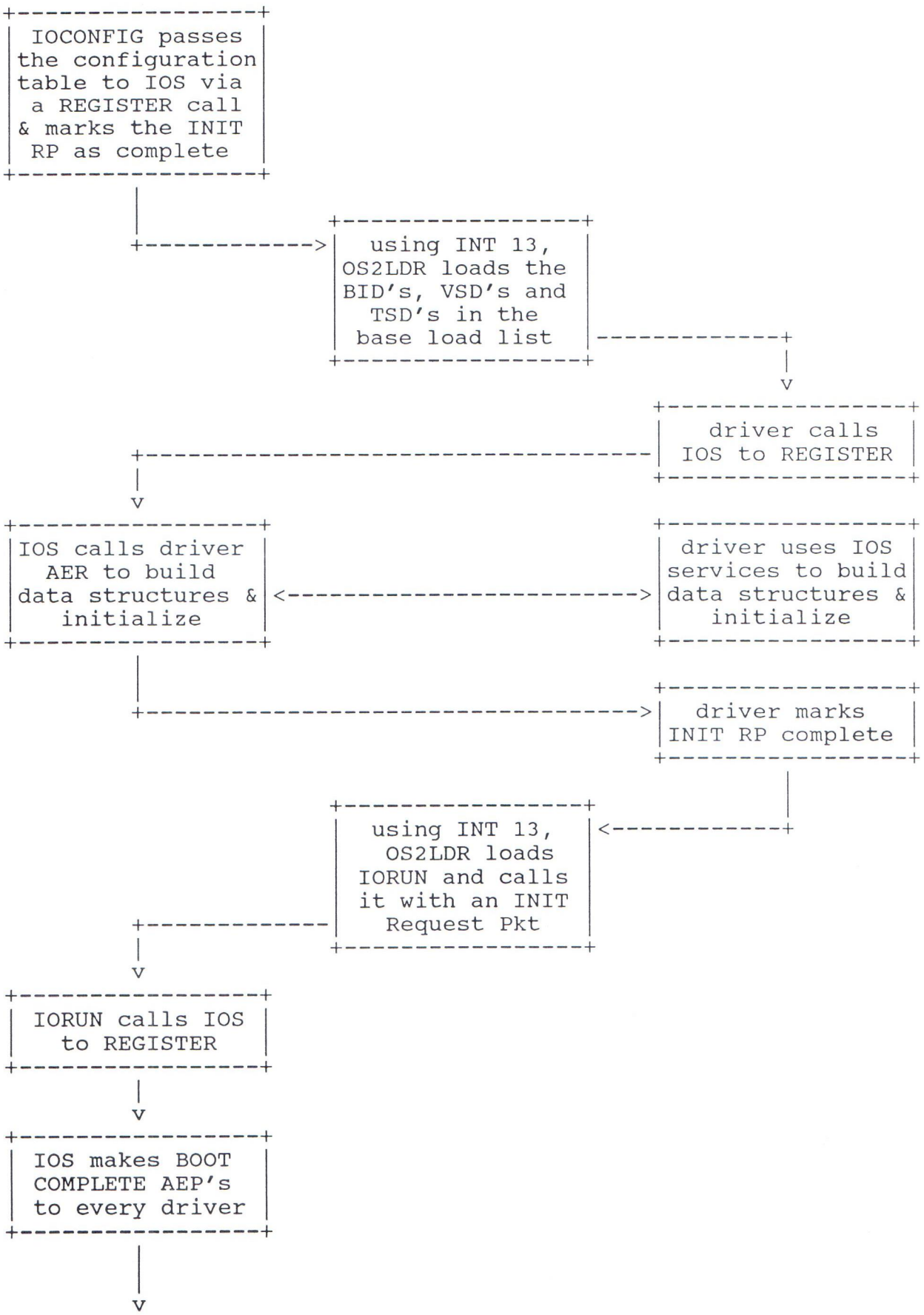
Boot Time Processing Overview

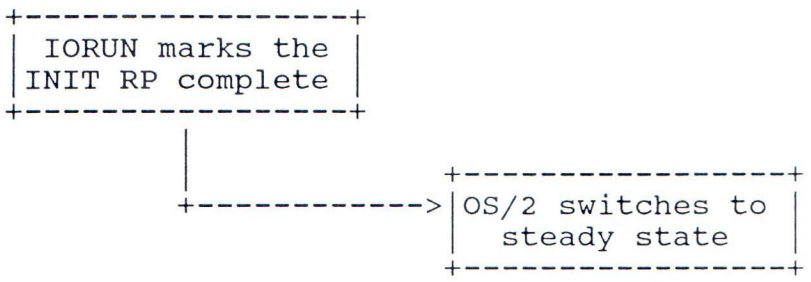
It is during OS/2 boot time processing that IOS and the drivers in the base load list are loaded and initialized.

The following zigzag diagram illustrates the boot time processing as it relates to the loading and initialization of the LADDR I/O complex.




INT 19 routine reads the boot sector via INT 13 and jumps to it
boot sector code loads multiboot via INT 13 and jumps to it
multiboot loads OS2LDR via INT 13 and jumps to it
using INT 13, OS2LDR loads the OS/2 Kernel and initializes it
OS2LDR loads IOS via INT 13 and calls it with an INIT Request Pkt
IOS initializes & marks the INIT RP as complete
OS2LDR loads IOCONFIG via INT 13 and calls it with an INIT Request Pkt


IOCONFIG passes the configuration table to IOS via a REGISTER call & marks the INIT RP as complete
using INT 13, OS2LDR loads the BID's, VSD's and TSD's in the base load list
driver calls IOS to REGISTER
IOS calls driver AER to build data structures & initialize
driver uses IOS services to build data structures & initialize
driver marks INIT RP complete
using INT 13, OS2LDR loads IORUN and calls it with an INIT Request Pkt
IORUN calls IOS to REGISTER
IOS makes BOOT COMPLETE AEP's to every driver



IOS Initialization

Loading and initialization of IOS occurs only at boot time - steady state initialization and re-initialization is not supported.

Once loaded, IOS initially receives control from the Kernel at its STRAT1 entry point with ES:BX pointing to an INIT request packet (RP).

IOS's initialization code then performs the following processing:

1. Allocates a block of memory for use as a memory pool, and initializes it

2. Initializes the IVT

3. Makes a DEVHLP call to establish an interval timer such that IOS's timeout detection logic gains control periodically. The duration of this period is normally set to two seconds but may be set to a much shorter value to facilitate the debugging of driver code.

Having completed its initialization processing, IOS returns to its caller, the Kernel, showing that initialization succeeded and indicating the amount of initialization code that is to be discarded.

Note that attempting to replace the boot time version of IOS by specifying an IOS in the CONFIG.SYS file is not supported and will have unpredictable results.

IOS Configuration Processing for Drivers

IOS provides very similar configuration processing for both drivers which initialize at boot time and drivers which initialize at steady state time.

The following describes that configuration processing.

Configuration processing for a driver is initiated by IOS's

receipt of a registration IDC from that driver.

In response to that registration IDC, IOS initializes various internal data structures and the drivers ILB.

IOS then generates a series of asynchronous event calls to the driver.

The first of these calls is a driver initialization call - in response to this call, the driver should initialize itself and, if it is a BID, it should initialize the bus interface hardware. IOS includes parametric data from IOCONFIG, if any, in the packet associated with the call.

For drivers which are BID's, IOS may generate one or more DEVICE INQUIRY asynchronous event calls to the BID. On the basis of information returned by these calls in conjunction with information provided by IOCONFIG, IOS may make one or more CONFIGURE DEVICE asynchronous event calls to the BID.

For drivers other than BID's, IOS will (after the INITIALIZE asynchronous event call has completed) make one CONFIGURE DEVICE asynchronous event call for each currently existent physical DCB. During this call, the driver should examine the DCB and, if necessary, the device, and determine if it wants to service the device. If it does, the driver should update the DCB appropriately and make calls to IOS to create logical DCB's if appropriate.

IOS Services

IOS provides two type of services to drivers:

Registration service

Support services

The IOS Registration Service

The registration service provided by IOS allow a driver to notify IOS of its existence and its characteristics, and to acquire IOS's characteristics.

When a driver calls IOS's "registration" entry point, which is made available to a driver via the DevHelp_AttachDD service, the driver passes pointers to the INIT request packet that it received from the kernel and to its own DRP, which points to its ILP. Prior to making the registration call, the driver must have properly initialized its DRP.

During its processing of the registration call, IOS fills in the

ILP and makes various calls to the drivers asynchronous events routine.

While the DRP and ILP may reside in any segments that are part of the primary code group or primary data group, it is recommended that the DRP reside in the drivers initialization data segment, and that the ILP reside in the drivers permanent code segment.

IOS Support Services

IOS provides many support services through the following interfaces:

Stack based service routine

Register based service routine

Kernel DEVHLP interface routine

Note that use of the DEVHLP interface (aside from AttachDD) should be avoided to maximize the number of Microsoft operating system platforms supported by the driver. Where the DEVHLP interface must be used because the needed function is not provided by any other interface, conditional assembly/compile/execute techniques will be required to provide cross platform capability.

4.0 THE BUS INTERFACE DRIVER (BID):

BID Structure and Sample Code

A BID can be made to meet LADDR re-entrancy, functional, and interface requirements by complying with the specifications in this document. The ESDI/ST-506 BID, ESDI-506.SYS, is an example of a LADDR compliant bid.

Physical structure The BID must be packaged as a single binary, linked as a device driver and have a name of the form "xxxxxxxx.BID". That name must either be present in the base load list, or may be specified in a "DEVICE=xxxxxxxx.BID" statement in CONFIG.SYS if the BID does not support the boot device.

Note that when a BID is loaded via CONFIG.SYS, it may be necessary to also load the associate TSD and VSD's, if any, via CONFIG.SYS.

Group structure A bid may contain either two or three groups. OS/2 release 1.21 only supports BID's with two groups. Releases 1.3 and 2.x allows a third group, which is assumed to contain swapable code.

When the BID is being loaded, the Kernel assumes that the first group encountered in the binary is a data group and expects to find the BID's device driver header at offset zero within the group. The kernel builds a writeable GDT selector for this group and nails the group down.

The Kernel assumes that the second group encountered contains code and builds a non-writeable (read/exectute) GDT selector for this group. It nails this group down.

Whenever the Kernel makes a STRAT1 call to the BID, it places the selector number for the first (data) group in the DS selector register and places the selector number for the second (code) group in the CS selector register.

If, under releases 1.3 or 2.x of OS/2, a third group is encountered, the Kernel assumes it to be swapable code and builds a non-writeable (read/execute) GDT selector for it. This group is NOT nailed down. The BID obtains the selector for this third, swapable, group through the use of selector fixup. The following assembler code fragment illustrates this:

```
EXTRN  A_Public_Label_In_The_Third_Group:FAR
```

```
MOV     AX,SEL A_Public_Label_In_The_Third_Group
MOV     ES,AX
```

The BID is responsible for properly controlling the subsequent swapping of this third group. The sample BID contains only two groups, "DGroup" for data and "CGroup" for code.

Segment structure A group is composed of one or more segments.

It is recommended that the first two groups each contain two segments. The first segment in each group is considered permanent while the second segment is discarded after the BID has initialized. The permanent data segment should only contain the BID's device driver header.

The following diagram, which corresponds to code in file BIDSEGS.INC of the sample BID, illustrates the recommended structure of a BID.

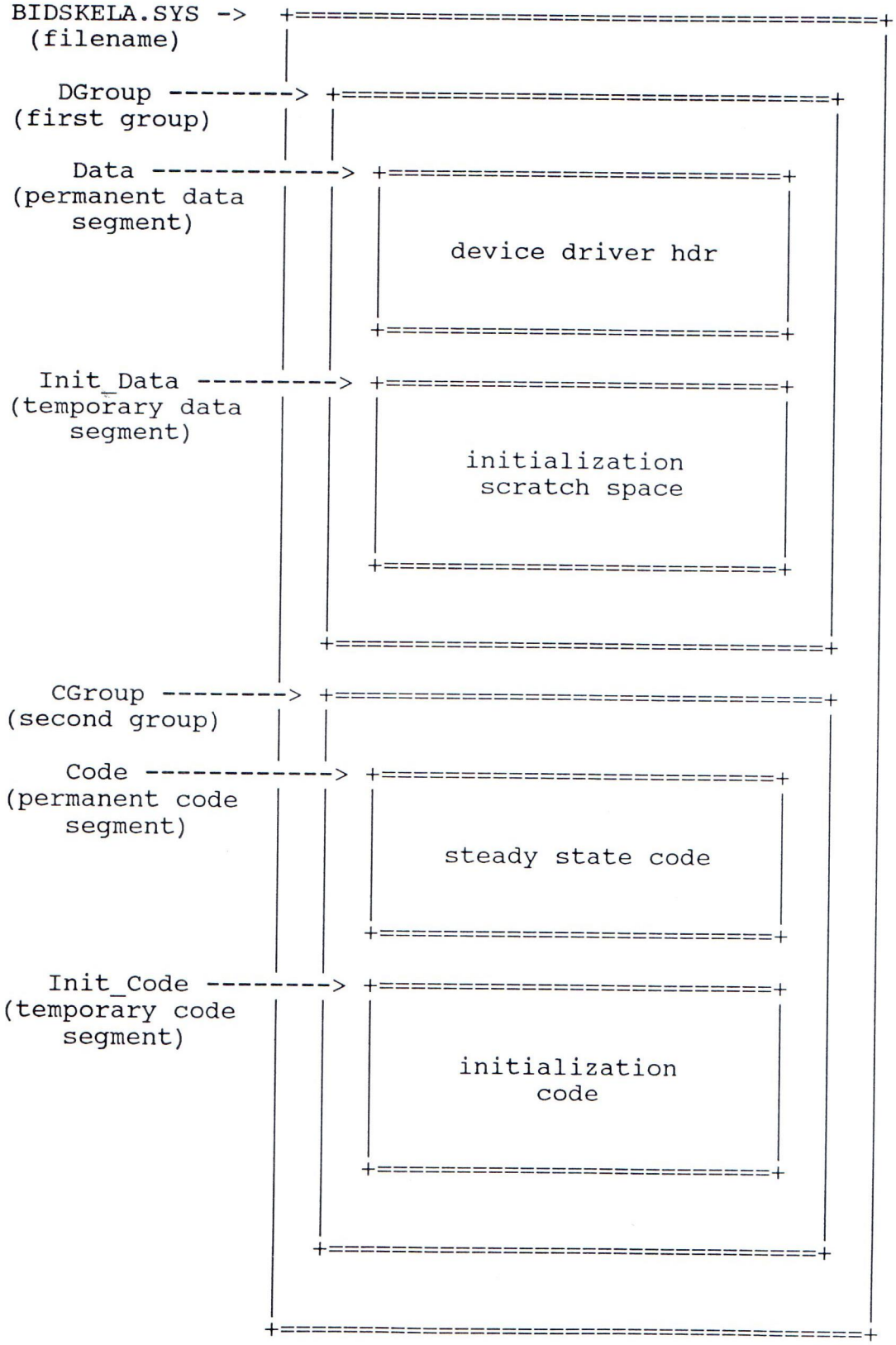


Figure 13. Recommended Structure of a BID

BID Initialization

A BID, like any other driver, is loaded by the OS/2 Kernel. If the BID is being loaded at boot time, the Kernel uses INT 13 to read the BID from disk into memory. If a DEVICE= statement in CONFIG.SYS causes a BID to be loaded, the Kernel performs normal protect mode I/O to read the BID from disk into memory.

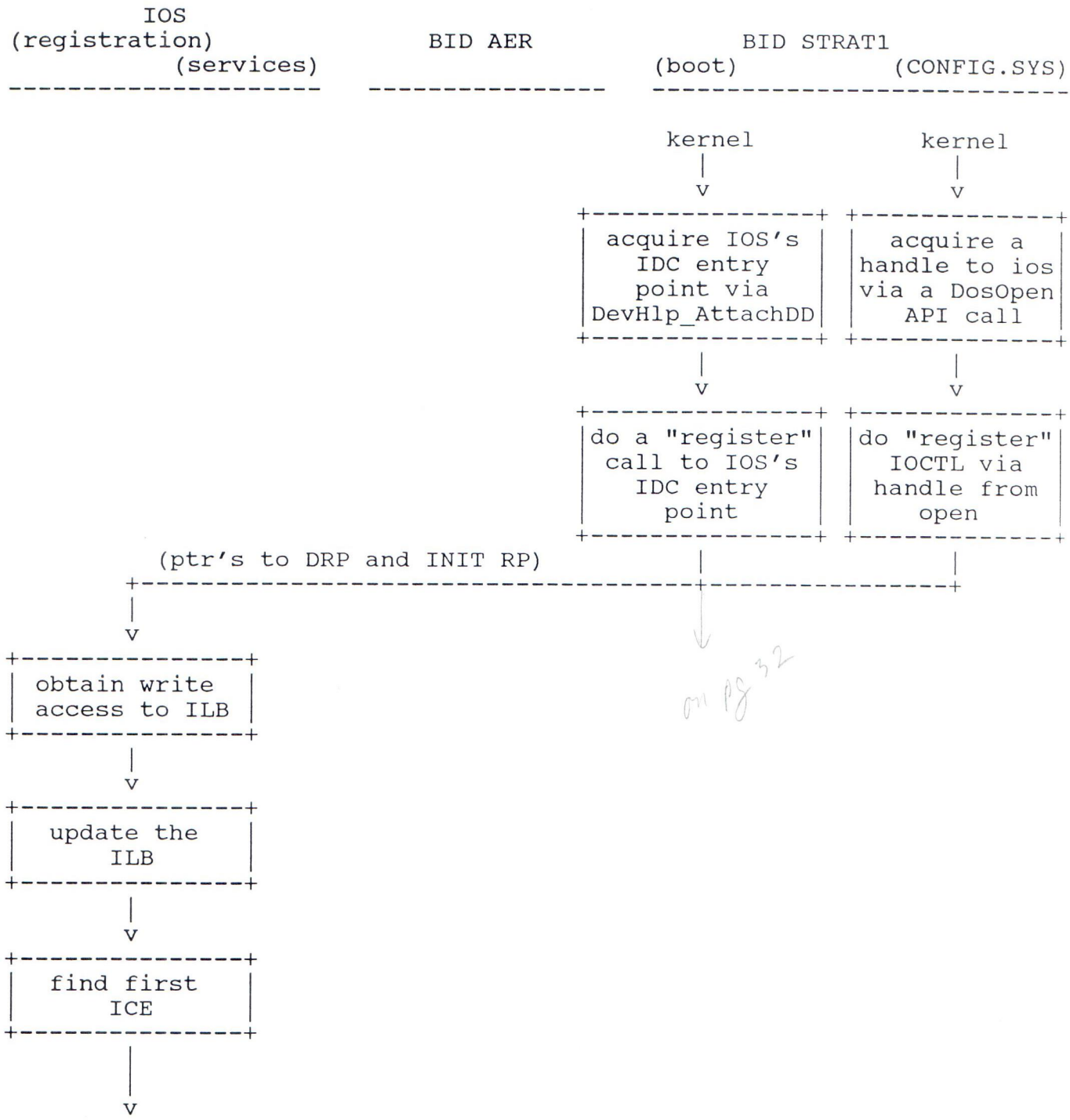
Once a BID is loaded, it initially gains control from the Kernel via a STRAT1 INIT call.

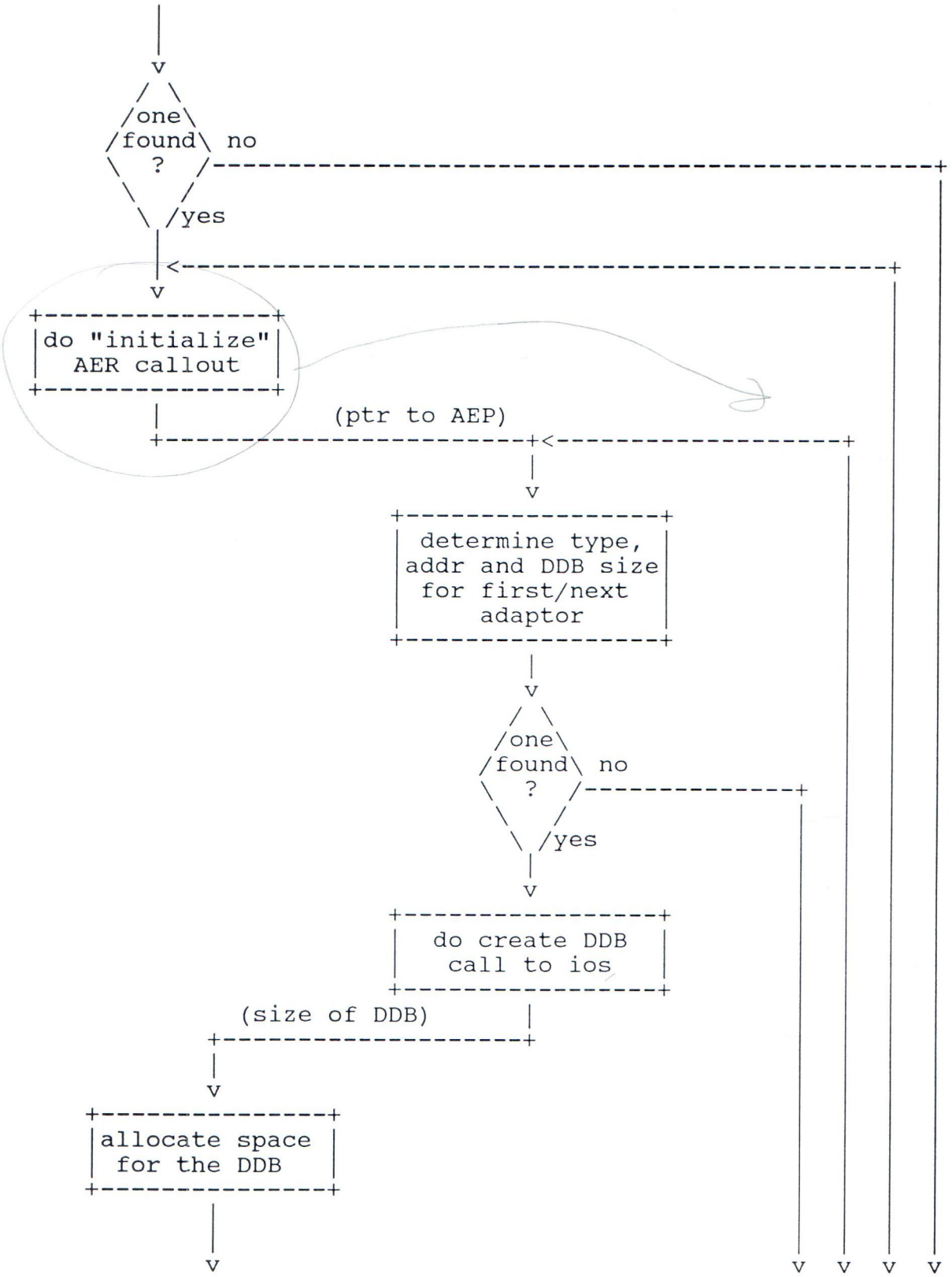
In response to that INIT call, either via an inter-driver call (for BID's loading at boot time) or via an IOCTL (for BID's loading via CONFIG.SYS), the BID makes a REGISTER call to IOS.

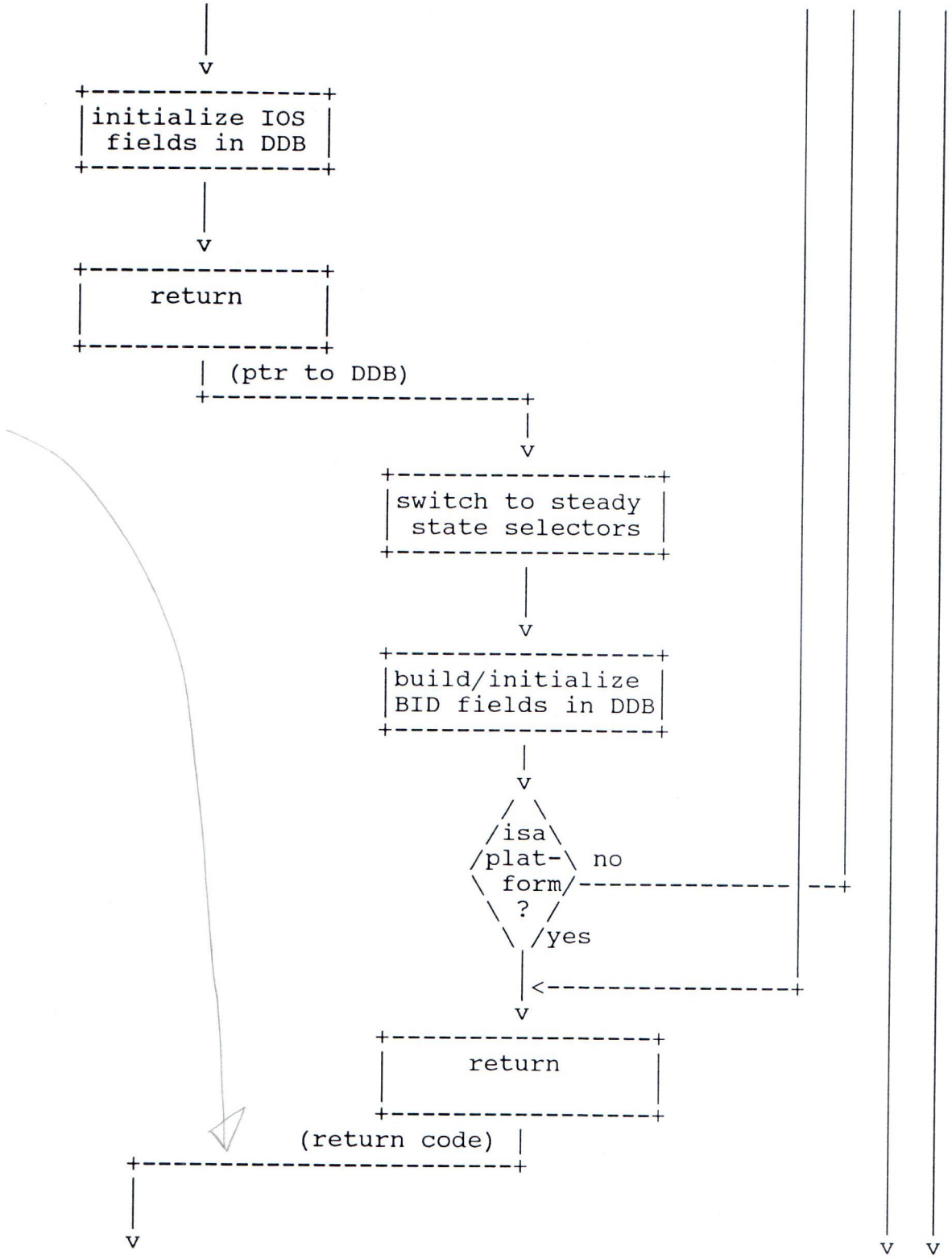
In response to the REGISTER call, IOS's BID REGISTER routine performs a series of callouts to the BID's asynchronous event routine (AER). It is these AER callouts that cause the various BID initialization routines to execute. In addition to initializing the BID itself and one or more adaptors, this process typically causes one or more device control blocks (DCB's) to be built.

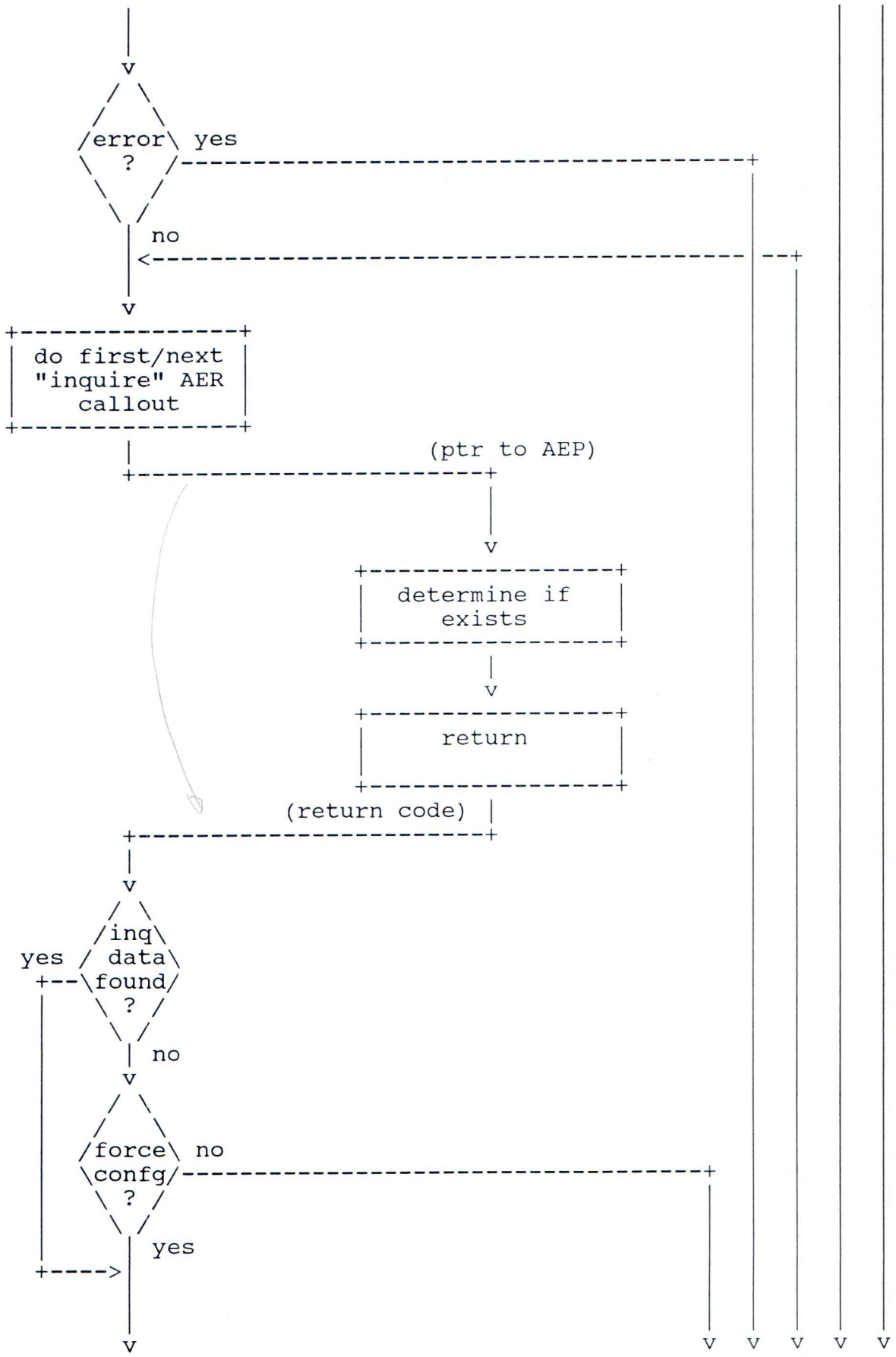
After IOS's BID REGISTER routine completes its processing, it returns control to the BID which then marks the STRAT1 INIT call complete.

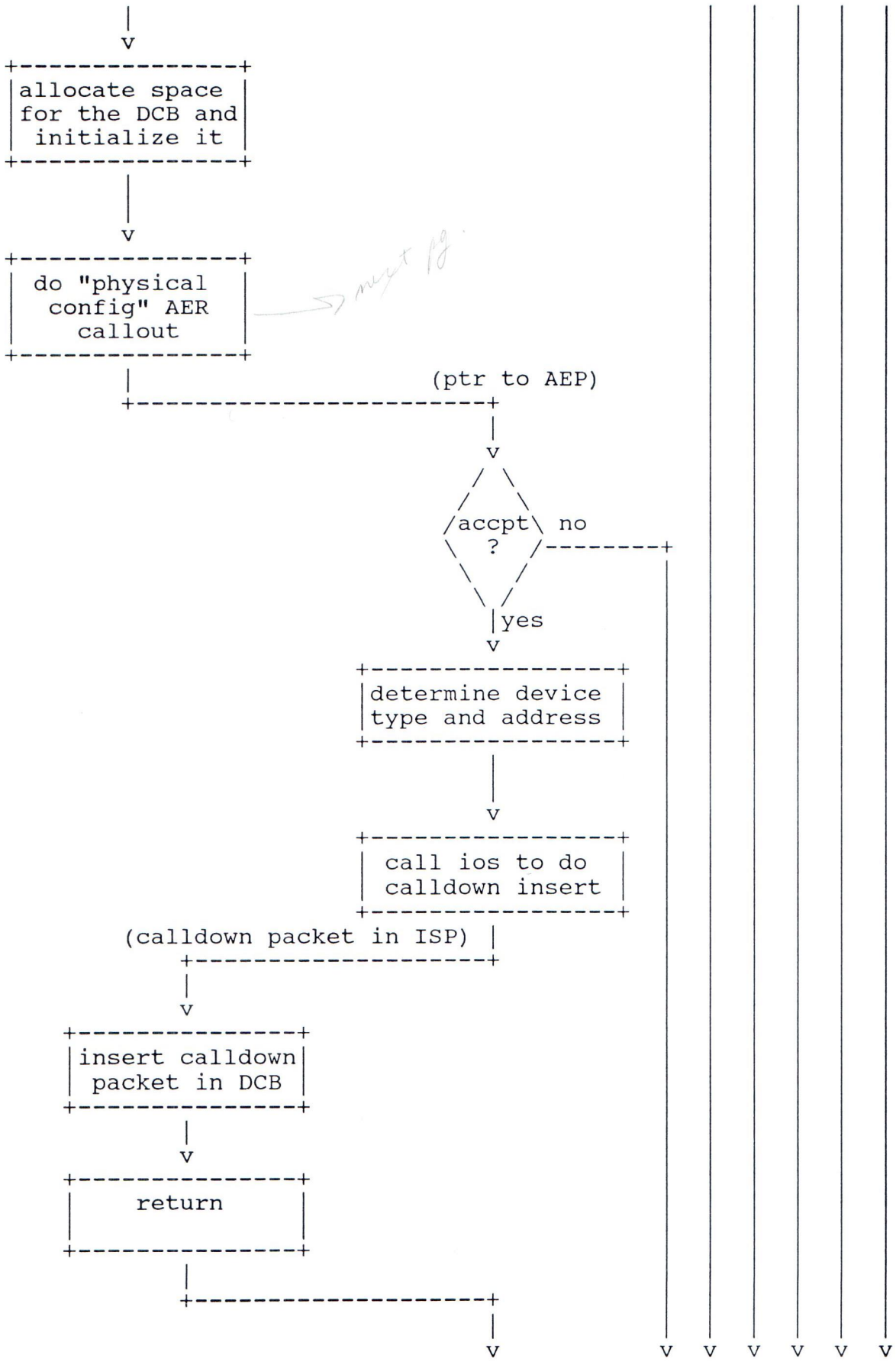
The following zigzag diagram illustrates the flow of control during the initialization of a BID.

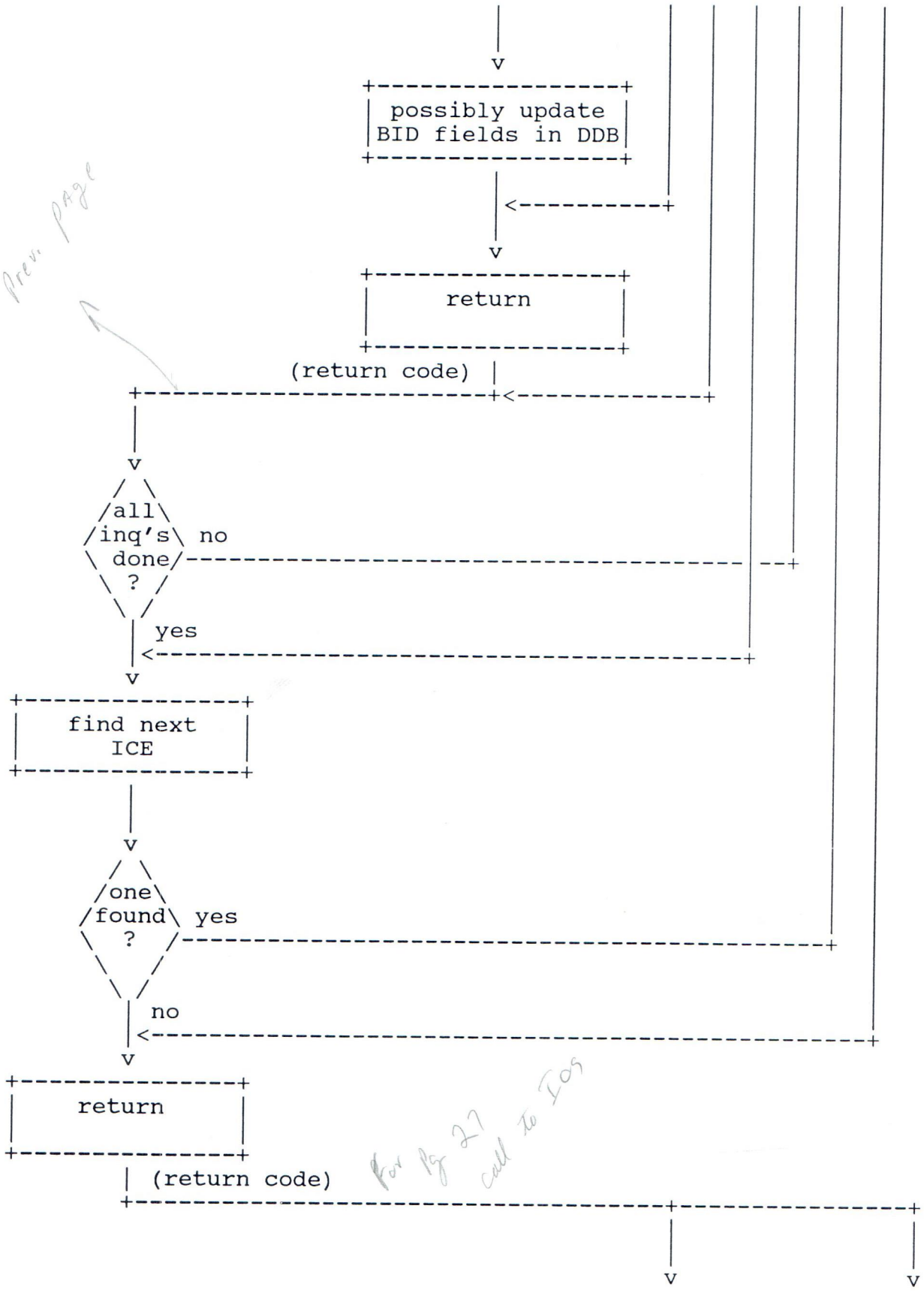



one found ?
no
yes
do "initialize" AER callout
(ptr to AEP)
determine type, addr and DDB size for first/next adaptor
one found ?
no
yes
do create DDB call to ios
(size of DDB)
allocate space for the DDB


initialize IOS fields in DDB
return
(ptr to DDB)
switch to steady state selectors
build/initialize BID fields in DDB
isa
plat-form ?
no
yes
return
(return code)


/error\ yes
\ ? /
no
do first/next "inquire" AER callout
(ptr to AEP)
determine if exists
return
(return code)
/inq\
yes / data\
\found/
\ ? /
no
/force\ no
\confg/
\ ? /
yes


allocate space for the DCB and initialize it
do "physical config" AER callout
next pg
(ptr to AEP)
accpt ?
no
yes
determine device type and address
call ios to do calldown insert
(calldown packet in ISP)
insert calldown packet in DCB
return


possibly update BID fields in DDB
return
(return code)
all inq's done ?
no
yes
find next ICE
one found ?
yes
no
return
(return code)

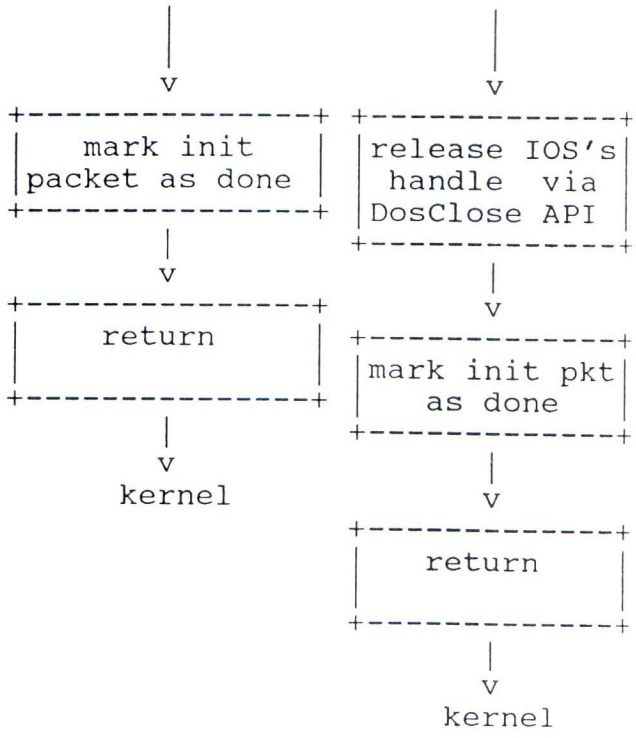


Figure 14. Example of BID Initialization

5.0 VSD INITIALIZATION:

A VSD, like any other driver, is loaded by the OS/2 Kernel. If the VSD is being loaded at boot time, the Kernel uses INT 13 to read the VSD from disk into memory. If a DEVICE= statement in CONFIG.SYS causes a VSD to be loaded, the Kernel performs normal protect mode I/O to read the VSD from disk into memory.

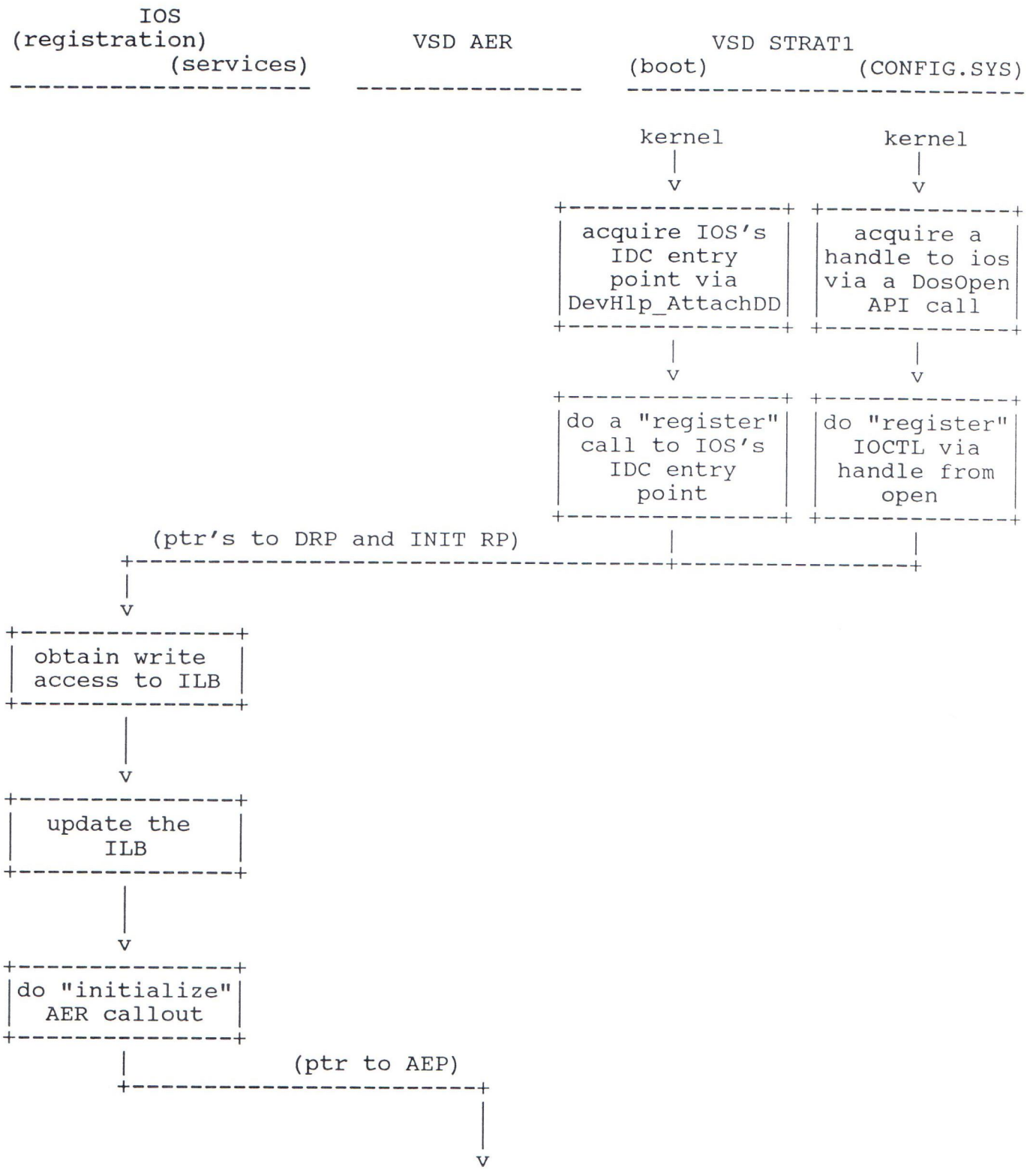
Once a VSD is loaded, it initially gains control from the Kernel via a STRAT1 INIT call.

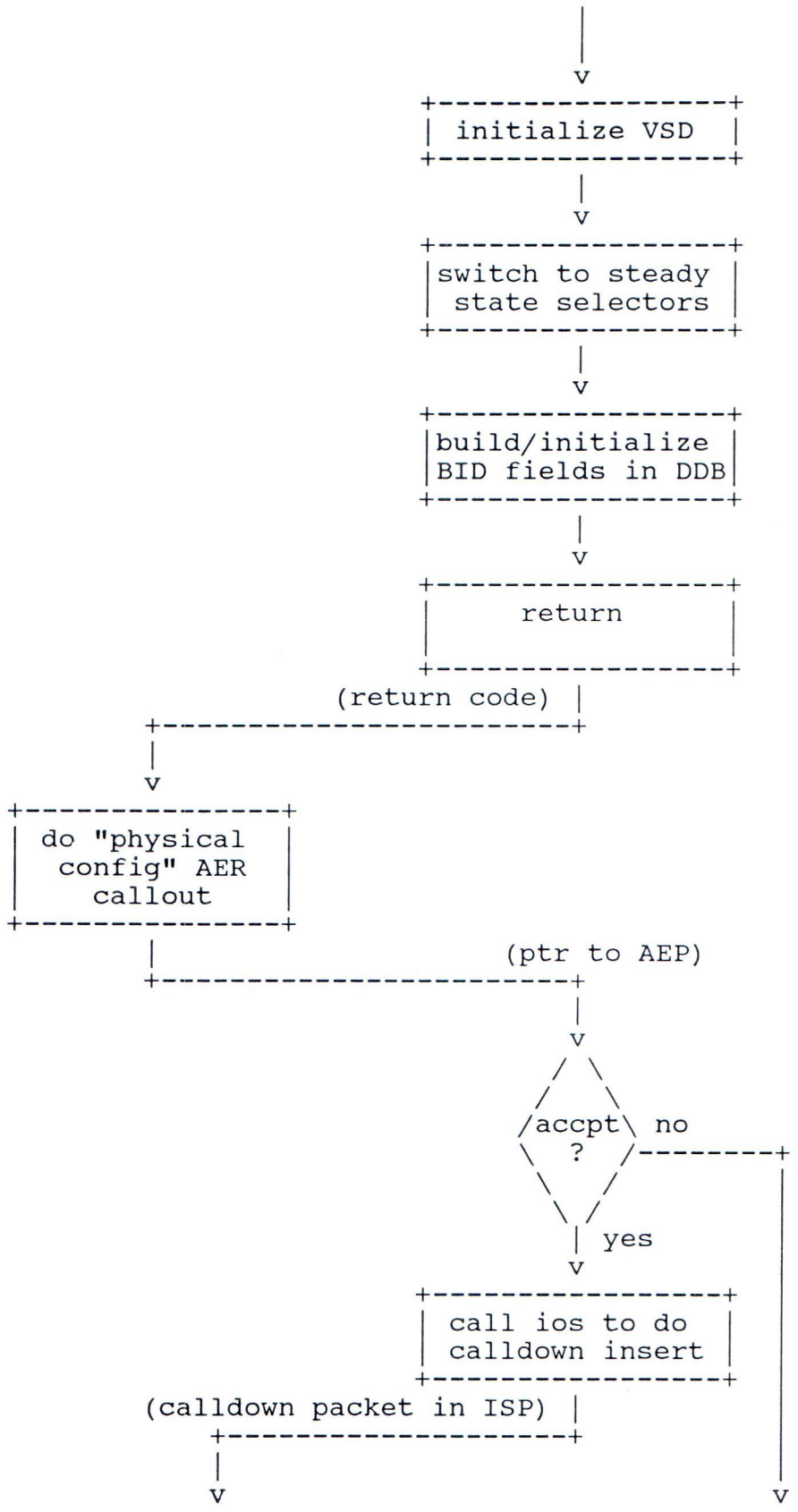
In response to that INIT call, either via an inter-driver call (for VSD's loading at boot time) or via an IOCTL (for VSD's loading via CONFIG.SYS), the VSD makes a REGISTER call to IOS.

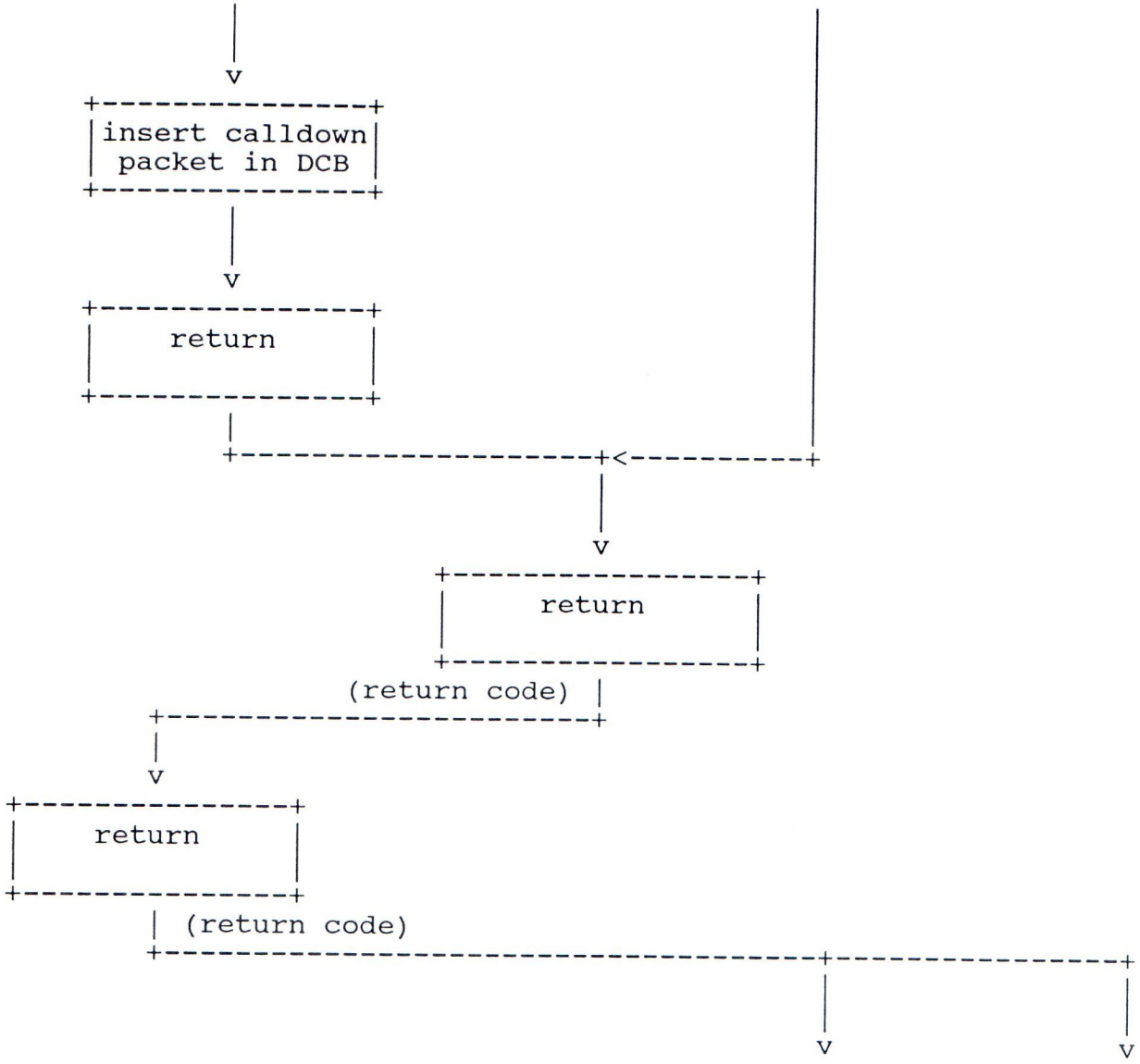
In response to the REGISTER call, IOS's VSD REGISTER routine performs a series of callouts to the VSD's asynchronous event routine (AER). It is these AER callouts that cause the various VSD initialization routines to execute.

After IOS's VSD REGISTER routine completes its processing, it returns control to the VSD which then marks the STRAT1 INIT call complete.

The following zigzag diagram illustrates the flow of control through the initialization of a VSD.




initialize VSD
switch to steady state selectors
build/initialize BID fields in DDB
return
(return code)
do "physical config" AER callout
(ptr to AEP)
accpt ?
no
yes
call ios to do calldown insert
(calldown packet in ISP)


insert calldown packet in DCB
return
return
(return code)
return
(return code)

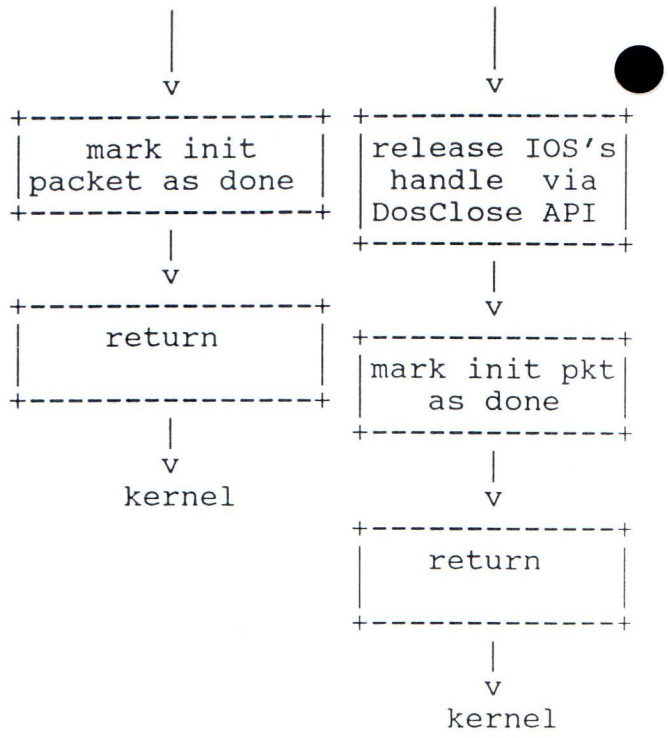


Figure 15. Example of VSD initialization

6.0 TSD INITIALIZATION:

A TSD, like any other driver, is loaded by the OS/2 Kernel. If the TSD is being loaded at boot time, the Kernel uses INT 13 to read the TSD from disk into memory. If a DEVICE= statement in CONFIG.SYS causes a TSD to be loaded, the Kernel performs normal protect mode I/O to read the TSD from disk into memory.

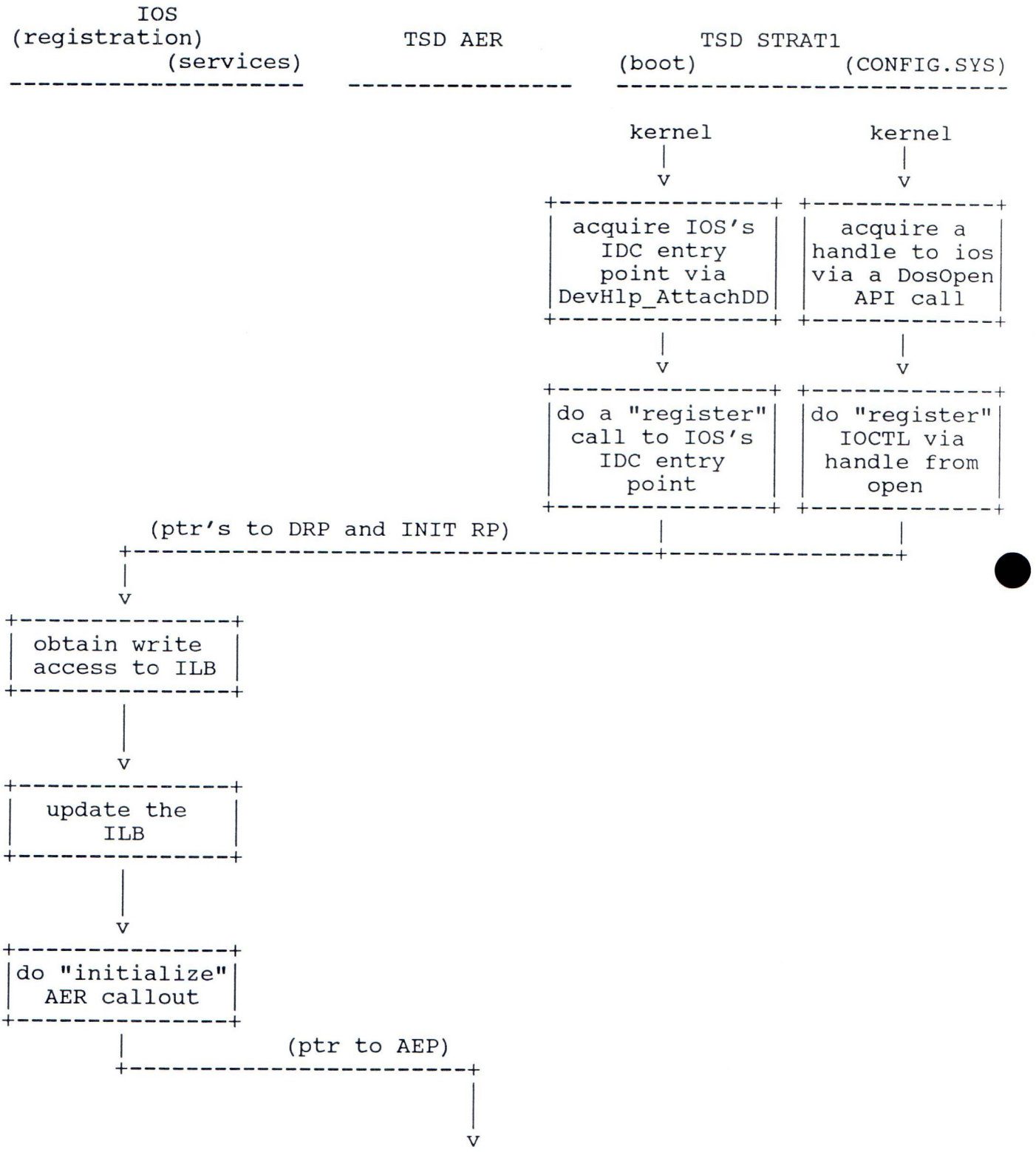
Once a TSD is loaded, it initially gains control from the Kernel via a STRAT1 INIT call.

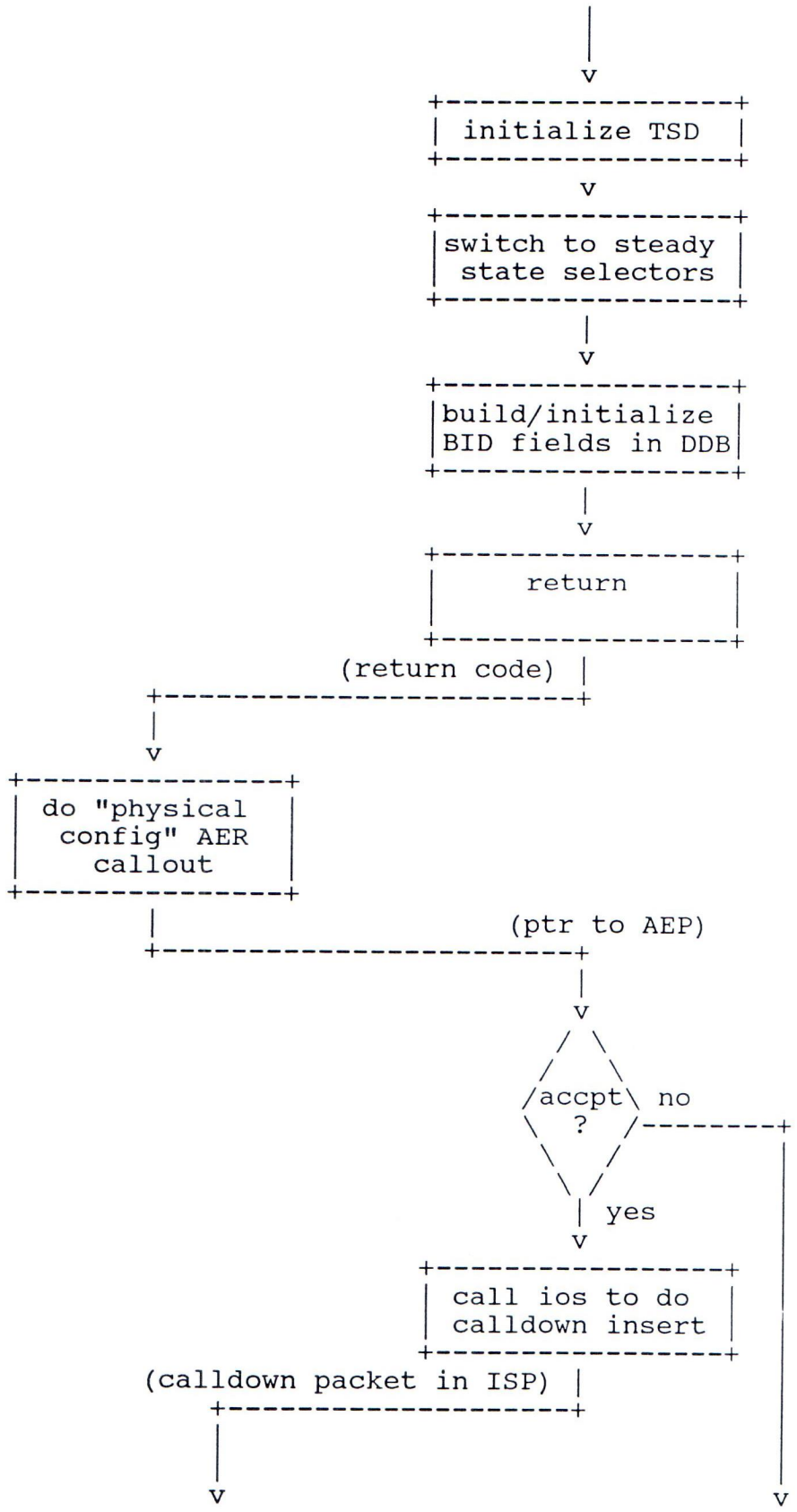
In response to that INIT call, either via an inter-driver call (for TSD's loading at boot time) or via an IOCTL (for TSD's loading via CONFIG.SYS), the TSD makes a REGISTER call to IOS.

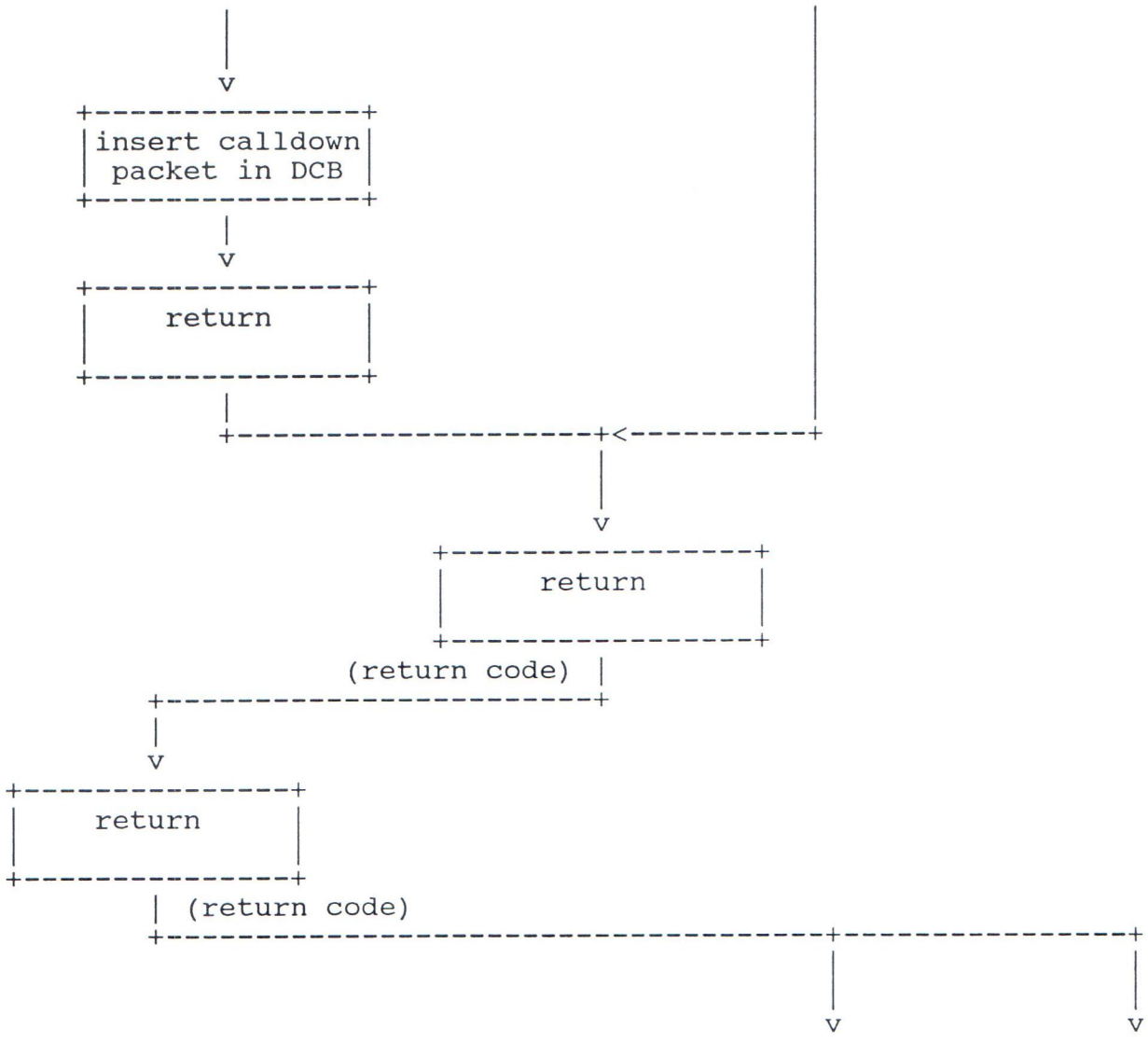
In response to the REGISTER call, IOS's TSD REGISTER routine performs a series of callouts to the TSD's asynchronous event routine (AER). It is these AER callouts that cause the various TSD initialization routines to execute.

After IOS's TSD REGISTER routine completes its processing, it returns control to the TSD which then marks the STRAT1 INIT call complete.

The following zigzag diagram illustrates the flow of control through the initialization of a TSD.




initialize TSD
switch to steady state selectors
build/initialize BID fields in DDB
return
(return code)
do "physical config" AER callout
(ptr to AEP)
accpt ?
no
yes
call ios to do calldown insert
(calldown packet in ISP)


insert calldown packet in DCB
return
return
(return code)
return
(return code)

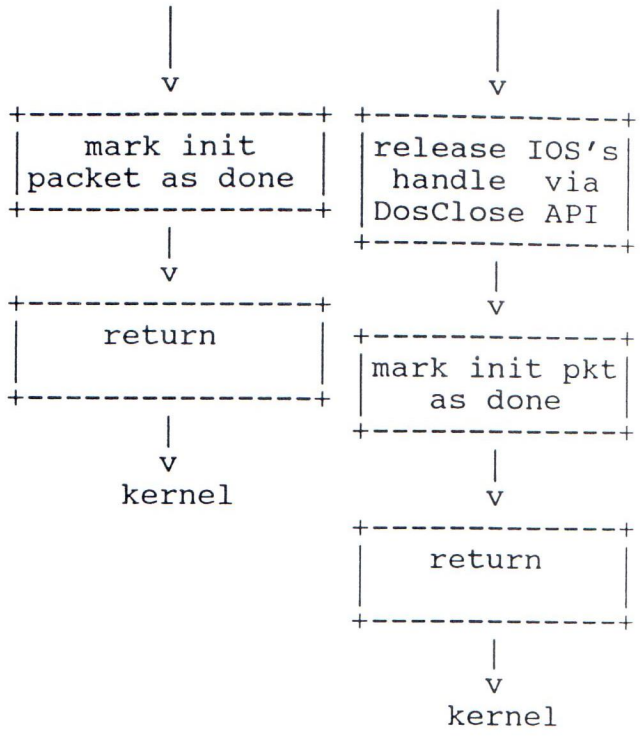


Figure 16. Example of TSD initialization

7.0 IOS INTERFACE SPECIFICATION:

IOS to Driver Interfaces

The following sections define these interfaces from IOS to drivers:

- Block Device STRAT1 Request Processing
- Block Device STRAT2 Request Processing
- Character Device STRAT1 Request Processing
- Character Device STRAT2 Request Processing
- DevHlp

Block Device STRAT1 Request Processing

This callable entry point provides TSD's with a way to transfer an RP, which has just been delivered to that TSD by the Kernel, to IOS for synchronous processing. Control is not returned to the calling TSD until the request has completed.

Having received an RP via this entry point, IOS then locates the appropriate DCB, re-packages the request in an RLH, and initiates processing of the request by passing it to the appropriate TSD.

IOS provides this entry point to every registered block device TSD in field ILB_strat1_block of the TSD's ILB. The TSD should call it whenever its own STRAT1 entry point gains control with an RP other than INIT or SHUTDOWN.

The TSD should put the 16:16 address of the RP on the stack before calling the IOS entry point.

The following code fragment illustrates use of this entry point by a block device TSD.

```
push    es                      ; put the 16:16 pointer to
push    bx                      ; rlh on the stack.

call    cs:[ILB_strat1_block]   ; call ios's STRAT1 entry point.
```

Block Device STRAT2 Request Processing

This callable entry point provides TSD's with a way to transfer an RLH, which has just been delivered to that TSD by the Kernel or an FSD, to IOS for processing. Control is returned to the calling TSD as soon as the request has been queued or initiated. Completion of the request is reported via appropriate callout processing. Note that the completion callout processing may occur before control is returned to the calling TSD.

Having received an RLH via this entry point, IOS then locates the appropriate DCB and initiates processing of the request by passing it to the appropriate TSD.

IOS provides this entry point to every registered block device TSD in field ILB_strat2_block of the TSD's ILB. The TSD should call it whenever its own STRAT2 entry point gains control.

The TSD should put the 16:32 address of the RLH on the stack before calling the IOS entry point.

The following code fragment illustrates use of this entry point by a block device TSD on a 386 system.

```
push    es                          ; put the 16:32 pointer to
push    ebx                         ; rlh on the stack.

call    cs:[ILB_strat2_block]       ; call ios's STRAT2 entry point.
```

Character Device STRAT1 Request Processing

This callable entry point provides TSD's with a way to transfer an RP, which has just been delivered to that TSD by the Kernel, to IOS for synchronous processing. Control is not returned to the calling TSD until the request has completed.

Having received an RP via this entry point, IOS then locates the appropriate DCB, re-packages the request in an RLH, and initiates processing of the request by passing it to the appropriate TSD.

IOS provides this entry point to every registered character device TSD in field ILB_strat1_char of the TSD's ILB. The TSD should call it whenever its own STRAT1 entry point gains control with an RP other than INIT or SHUTDOWN.

The TSD should put the 16:16 address of the RP on the stack before calling the IOS entry point.

The following code fragment illustrates use of this entry point by a character device TSD.

```
push    es                          ; put the 16:16 pointer to
push    bx                          ; rlh on the stack.
```

```
call    cs:[ILB_strat1_char]    ; call ios's STRAT1 entry point.
```

Character Device STRAT2 Request Processing

This callable entry point provides TSD's with a way to transfer an RLH, which has just been delivered to that TSD by the Kernel or an FSD, to IOS for processing. Control is returned to the calling TSD as soon as the request has been queued or initiated. Completion of the request is reported via appropriate callout processing. Note that the completion callout processing may occur before control is returned to the callin TSD.

Having received an RLH via this entry point, IOS then locates the appropriate DCB and initiates processing of the request by passing it to the appropriate TSD.

IOS provides this entry point to every registered character device TSD in field ILB_strat2_char of the TSD's ILB. The TSD should call it whenever its own STRAT2 entry point gains control.

The TSD should put the 16:32 address of the RLH on the stack before calling IOS.

The following code fragment illustrates use of this entry point by a character device TSD on a 386 system.

```
push    es                      ; put the 16:32 pointer to
push    ebx                     ; rlh on the stack.

call    cs:[ILB_strat2_char]    ; call ios's STRAT2 entry point.
```

DevHlp

IOS provides this entry point to every registered device driver in field ILB_devhlp of the drivers ILB. Device drivers should call this entry point (rather than the DevHlp entry point provided in an INIT RP) whenever a DevHlp service is needed.

The following code fragment illustrates use of this entry point:

```
include dmacs.inc                   ; define the DEVHLP macro.

xor     ax,ax                       ; set up a base
xor     bx,bx                       ; address of zero.
mov     cx,-1                       ; specify biggest segment possible.
mov     si,selector_to_use          ; specify gdt selector to map against.
DEVHLP  DevHlp_PhysToGDTSelector    ; call ios's devhlp entry point to
                                    ; requesting physical address to
                                    ; gdt mapping.
jc      error                       ; any error? yes, go report it.
```

IOS services

IOS provides this entry point to every registered device driver in field ILB_service_rtn of the drivers ILB. Device drivers should call this entry point whenever an IOS service is needed. Associated with a call to IOS services is an IOS services packet (ISP), the format of which varies according to the requested service, and which is defined within the IOS services packet data structure definition.

The following code fragment illustrates creation and use of the ISP by a driver on a 286 system.

```
sub     sp,size ISP_ddb_create          ; allocate isp from the stack.

mov     di,sp                           ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_create_ddb ; construct
mov     ss:[di].ISP_owner.segmt,cs      ; isp for
mov     ss:[di].ISP_owner.offst,offset $; the build
mov     ss:[di].ISP_ddb_size,ax         ; ddb service.

push    ss                              ; put the 32-bit address
push    0                               ; of the isp on
push    di                              ; the stack.

call    cs:[ILB_service_rtn]            ; create ddb.

mov     ax,ss:[di.ISP_ddb_ptr.loword]   ; pick up the ddb's address.

add     sp,size ISP_ddb_create+6        ; cleanup the stack and
                                        ; discard the isp.
```

The following IOS services are provided:

- Registration via IDC
- Registration via IOCTL
- Trace Routine
- Allocate Memory
- Create Physical DCB
- Create Logical DCB
- Create DDB
- Create SRB's
- Create RCB

- Set IRQ

Registration via IDC

This entry point to IOS is used by a driver which has received a ring zero init request packet (RP) to link itself to IOS, which, in turn, triggers driver initialization.

The code fragment below illustrates a registration call via IDC.

```
devhlpfunc  dd  ?                       ; devhelp function pointer.

ios_name    db  'IOS$    ',0            ; 8 byte asciiz name of ios.

ios_filler1 dd  ?                       ; not used.
ios_filler2 dw  ?                       ; not used.
ios_ep_prot dd  ?                       ; ios registration entry point.
ios_ds_prot dw  ?                       ; ds corresponding to above ep.

drv_reg_pkt DRP <>

mov     ax,es:[bx].INI_pDevHlp.offst    ; move the kernel's devhlp
mov     devhlpFunc.offst,ax             ; entry point address from
mov     ax,es:[bx].INI_pDevHlp.segmt    ; the init request packet
mov     devhlpFunc.segmt,ax             ; to our init data segment.

mov     bx,offset IOS_name              ; point to name to attach to.
mov     di,offset IOS_filler1           ; point to the result area.
mov     dl,DevHlp_AttachDD              ; call the kernel's attach
call    [devhlpfunc]                    ; device driver devhlp routine.

call    setup_srp                       ; setup the driver registration
                                        ; packet.
push    es                              ; set up a pointer to the
push    bx                              ; init request packet.
push    seg drv_req_pkt                 ; set up a pointer to the
push    offset drv_reg_pkt              ; drivers drp.
call    [ios_ep_prot]                   ; call ios's registration routi
add     sp,08                           ; clean up the stack.
```

The return code, which is returned in the ax register, has the following meaning:

0000 Registration accepted - driver is to remain resident after initialization is complete

0001 Registration accepted - driver is to discard itself after initialization is complete

8xxx Registration rejected - "xxx" is a standard IOS error code as defined by IEC.INC

Registration via IOCTL

This entry point to IOS is used by a driver which has received a ring three init request packet (RP) to link itself to IOS. This in turn triggers driver initialization.

Trace Routine

IOS provides this entry point to every registered device driver in field ILB_trace_rtn of the drivers ILB. It is used by a driver to cause event related information to be stored in the appropriate trace table.

Allocate Memory

This entry point to IOS services is used by a driver to obtain a piece of memory from IOS's memory pool for non-architected use. The space for all architected data structures should be obtained via the appropriate IOS services call.

In addition to allocating the requested space, IOS places the physical address of the allocated piece in the first two words of the piece in standard dword format.

The following code fragment illustrates a generic memory allocation request by a driver on a 286 system.

```
sub     sp,size ISP_mem_alloc           ; allocate isp from the stack.

mov     di,sp                           ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_alloc_mem  ; construct
mov     ss:[di].ISP_owner.segmt,cs      ; isp for
mov     ss:[di].ISP_owner.offst,offset $; the alloc
mov     ss:[di].ISP_mem_type,MED_FREE_TYPE; mem service.
mov     ss:[di].ISP_mem_size,100        ; alloc 100 bytes

push    ss                              ; set up the pointer
push    di                              ; to the isp.

call    cs:[ILB_service_rtn]            ; allocate the memory piece.

mov     ax,ss:[di.ISP_mem_ptr.loword]   ; get address of gotten piece.

add     sp,size ISP_srb_alloc+4         ; cleanup the stack and
                                        ; discard the isp.
```

Create Physical Device Control Block (DCB)

This entry point to IOS Services is used by a driver to obtain space for a physical DCB and to initialize the following fields in it:

list of fields to be added

The following code fragment illustrates a create physical dbc request by a driver on a 286 system.

```
sub     sp,size ISP_dcb_create          ; allocate isp from the stack.

mov     di,sp                           ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_create_dcb ; construct
mov     ss:[di].ISP_owner.segmt,cs      ; isp for
mov     ss:[di].ISP_owner.offst,offset $; the build
mov     ss:[di].ISP_dcb_size,size _dcb  ; dcb service.

push    ss                              ; set up the pointer
push    di                              ; to the isp.

call    cs:[ILB_service_rtn]            ; create the physical dcb.

mov     ax,ss:[di.ISP_dcb_ptr.loword]   ; get the new dcb's address.

add     sp,size ISP_dcb_create+4        ; cleanup the stack and
                                        ; discard the isp.
```

Create Logical Device Control Block (DCB)

This entry point to IOS Service is used by a driver to cause IOS to create a partially initialized logical DCB and call all drivers associated with the corresponding physical DCB to complete initialization of the logical DCB.

The following code fragment illustrates a create logical dbc request by a driver on a 286 system.

```
sub     sp,size ISP_dcb_create          ; allocate isp from the stack.

mov     di,sp                           ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_create_dcb ; construct
mov     ss:[di].ISP_owner.segmt,cs      ; isp for
mov     ss:[di].ISP_owner.offst,offset $; the build
mov     ss:[di].ISP_dcb_size,size _dcb  ; dcb service.

push    ss                              ; set up the pointer
push    di                              ; to the isp.

call    cs:[ILB_service_rtn]            ; create the logical dcb.
```

```
mov     ax,ss:[di.ISP_dcb_ptr.loword]   ; get the new dcb's address.

add     sp,size ISP_dcb_create+4        ; cleanup the stack and
                                        ; discard the isp.
```

Create Driver Data Block (DDB)

This entry point to IOS Services is used by a driver to cause IOS to create a DDB of the size specified by the driver, and to clear the driver owned portion of it to zeros.

The following code fragment illustrates a create DDB request by a driver on a 286 system.

```
sub     sp,size ISP_ddb_create          ; allocate isp from the stack.

mov     di,sp                           ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_create_ddb ; construct
mov     ss:[di].ISP_owner.segmt,cs      ; isp for
mov     ss:[di].ISP_owner.offst,offset $; the build
mov     ss:[di].ISP_ddb_size,ax         ; ddb service.

push    ss                              ; set up the pointer
push    di                              ; to the isp.

call    cs:[ILB_service_rtn]            ; create the ddb.

mov     ax,ss:[di.ISP_ddb_ptr.loword]   ; get the new ddb's address.

add     sp,size ISP_ddb_create+4        ; cleanup the stack and
                                        ; discard the isp.
```

Create Request Control Block (RCB)

This entry point to IOS Services is used by a driver to cause IOS to create an RCB and initialize the following fields in it:

list of fields to be added

The following code fragment illustrates a create RCB request by a driver on a 286 system.

```
sub     sp,size ISP_rcb_alloc           ; allocate isp from the stack.

mov     di,sp                           ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_create_rcb ; construct
mov     ss:[di].ISP_owner.segmt,cs      ; isp for
mov     ss:[di].ISP_owner.offst,offset $; the build
mov     ss:[di].ISP_rcb_size,size RCB   ; rcb service.
```

```
push    ss                                  ; set up the pointer
push    di                                  ; to the isp.

call    cs:[ILB_service_rtn]                ; create the rcb.

mov     bx,ss:[di.ISP_rcb_ptr.loword]       ; get the new rcb's address.

add     sp,size ISP_rcb_alloc+4             ; cleanup the stack and
                                            ; discard the isp.
```

Create SRB's

This entry point to IOS Services is used by a driver to cause IOS to create a chain of SRB's. The SRB size and number of SRB's is specified by the driver. IOS returns the address of an SRB from which the others are chained through link field SRB_Next_S_SRB_Logical. A link field of zero indicates the end of the chain.

The following code fragment illustrates a create RSB chain request by a driver on a 286 system.

```
sub     sp,size ISP_srb_alloc               ; allocate isp from the stack.

mov     di,sp                               ; point to the gotten isp.

mov     ss:[di].ISP_func,ISP_create_srbs; construct
mov     ss:[di].ISP_owner.segmt,cs          ; the isp
mov     ss:[di].ISP_owner.offst,offset $; for the
mov     ss:[di].ISP_srb_number, 3           ; build srb's
mov     ss:[di].ISP_srb_size,size SRB       ; services.

push    ss                                  ; set up the pointer
push    di                                  ; to the isp.

call    cs:[ILB_service_rtn]                ; create the srb's.

mov     ax,ss:[di.ISP_srb_ptr.loword]       ; get the address of the first
                                            ; srb in the chain.

add     sp,size ISP_create_srbs+4           ; cleanup the stack and
                                            ; discard the isp.
```

Set IRQ

This entry point to IOS Services is used by a driver to register an interrupt service routine for a particular IRQ.

Driver to IOS Interfaces

The following sections define these interfaces from drivers to IOS:

- Asynchronous Event Routine
- Callback Entry point
- IRQ Interrupt Routine
- Process Request Entry Point
- STRAT1 Entry Point
- STRAT2 Entry Point

Asynchronous Event Routine

Drivers provide this entry point in their DRP. IOS captures the address during registration processing and subsequently uses it to notify the driver that an asynchronous event has occurred which that driver may need to process.

Associated with a call to this entry point is a AEP, the format of which varies according to the initiating event. The various formats are defined in the asynchronous event packet data structure definition.

The following code fragment illustrates creation and use of the AEP by IOS on a 286 system.

```
sub     sp,size aep             ; allocate aep from the stack.

mov     si,sp                   ; point to the gotten aep.

mov     ss:[si.AEP_func],AEP_config_boot_device ; set up add
                                ; boot device function.
mov     ss:[si.AEP_dcb.lo],di   ; put the dcb pointer
mov     ss:[si.AEP_dcb.hi],0    ; in the aep.

push    ss                      ; put the aep address
push    0                       ; on the stack in
push    si                      ; 16:32 format and
call    es:[bx].DRP_aer         ; call the driver's AER.

add     sp,size aep+6           ; clean up the stack and
                                ; discard the aep.
```

Callback Entry Point

Drivers provide callback entry points by storing an appropriate address in a field associated with a completion event. The entry point later receives control when that completion event occurs.

Note that this entry point differs from the asynchronous event routine entry point in that this entry point is specific to a particular event and is highly performance sensitive.

IRQ Interrupt Routine

A driver provides this entry point during a REGISTER IRQ call to IOS services. IOS calls this entry point whenever an interrupt associated with the specified IRQ occurs.

Process Request Entry Point

A driver provides this entry point by storing it in the DCB's calldown table during DCB configuration processing. The process request entry point subsequently gains control whenever a request is available for it to process.

STRAT1 Entry Point

The driver provides this entry point, which is used by the kernel to initiate I/O with a STRAT1 packet, by assembling it into it's device driver header.

This entry point is called with ES:BX pointing to an RP.

STRAT2 Entry Point

The driver provides this entry point, which is used by the kernel and FSD's to initiate I/O with a STRAT2 packet, by inserting it in a DCS.

On 386 system, this entry point is called with ES:EBX pointing to an RLH.

This entry point is not supported on 286 systems.

8.0 IOS DATA STRUCTURE DEFINITIONS:

The following sections define these data structures which are used by IOS or shared between IOS and drivers.

- Asynchronous Request Packet - AEP
- Device Control Block - DCB
- Driver Data Block - DDB
- Driver Registration Packet - DRP
- Driver Vector Table - DVT
- IDC - Inter Driver Communications
- IOS Linkage Block - ILB
- IOS Service Packet - ISP
- IOS Vector Table - IVT
- Memory Element Descriptor - MED
- Request Control Block - RCB
- STRAT1 Packet - RP
- STRAT2 Packet - RLH
- SCSI Request Block - SRB

Asynchronous Event Packet - AEP

Asynchronous event packets (AEP) are generated by IOS and passed to drivers to initiate the driver's processing of an asynchronous event.

An AEP may reside anywhere in memory and is not necessarily contiguous. It is referenced via a 16:32 address.

The AEP has a variable format which is dependant on the asynchronous event being handled.

It is declared in AEP.H and its various function codes are also defined.

Device Control Block - DCB

A device control block (DCB) represents a physical or logical device. It contains information and pointers which exist on a per device basis. In general, one driver creates and partially initializes a DCB, and then one or more other drivers complete initialization. IOS and any of the drivers associated with the device may subsequently inspect or update a DCB.

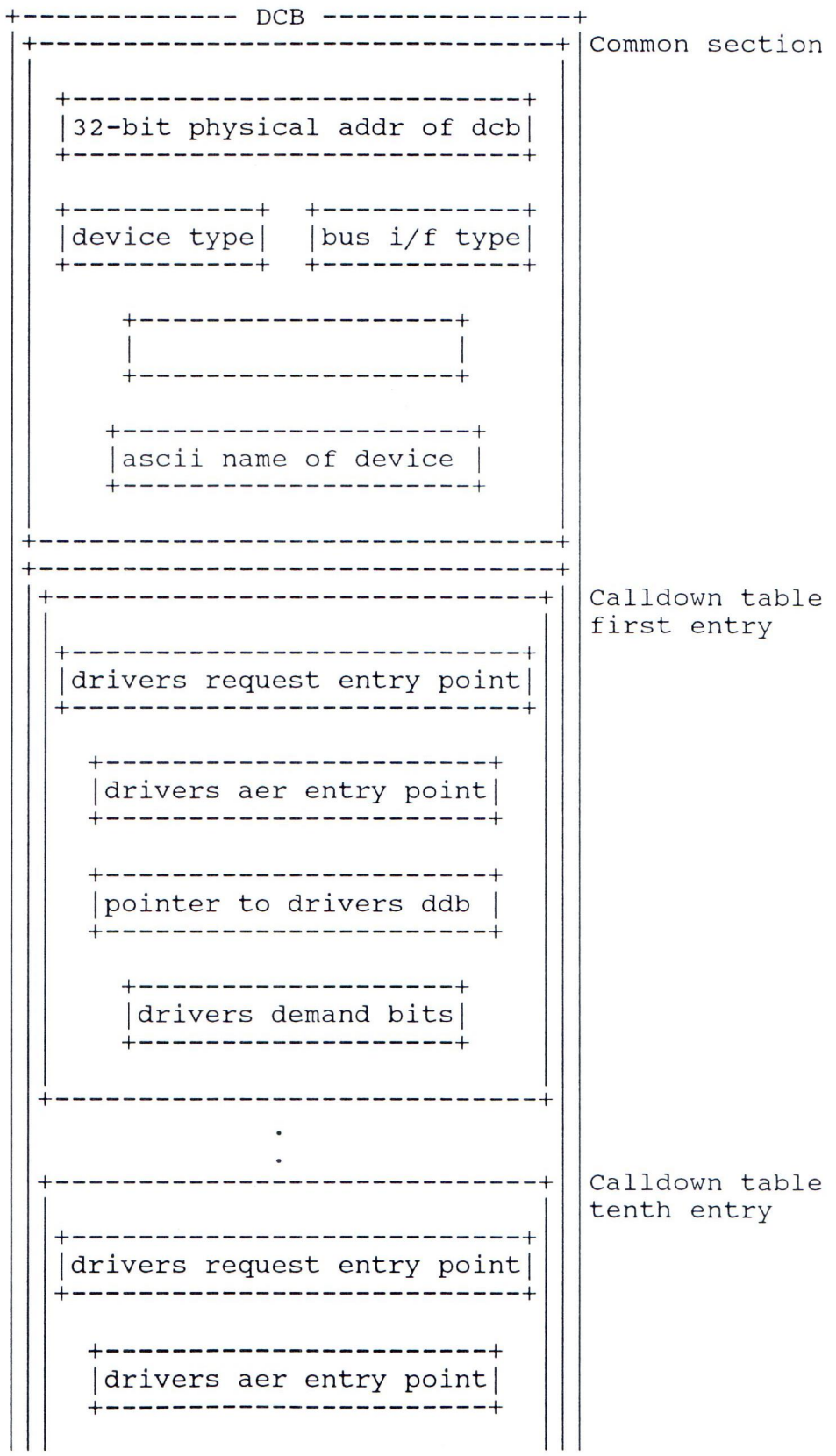
DCB's always reside in IOS's memory pool, are contiguous, and are referenced via an IOS:32 address.

The DCB has a fixed format common section and a device type specific extension.

It is declared in DCB.H.

Packed within the common section of the DCB is the call down table which has ten identically formatted entries

The following illustrates the relationship of the various elements of the DCB, with selected fields within each element shown.



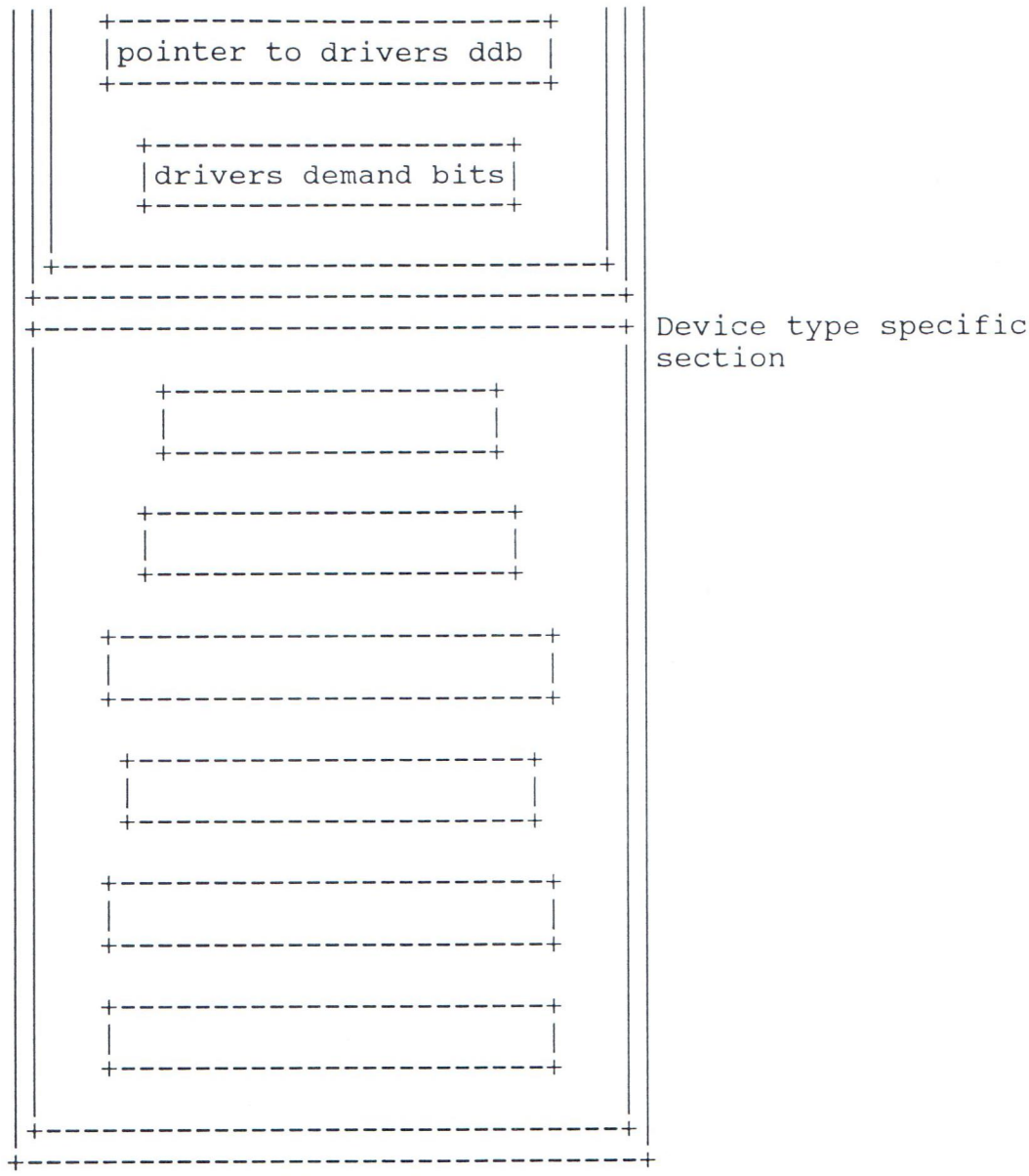
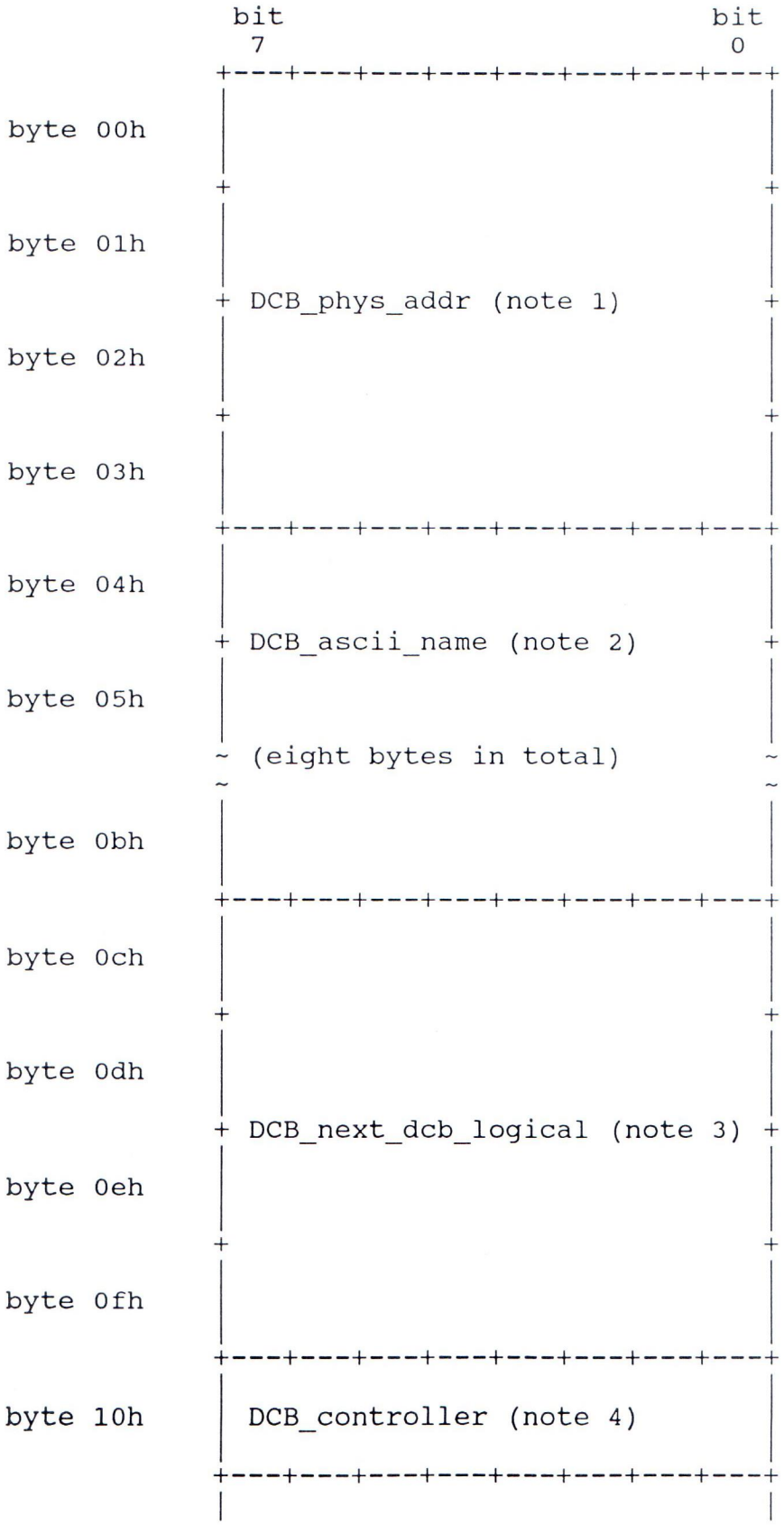


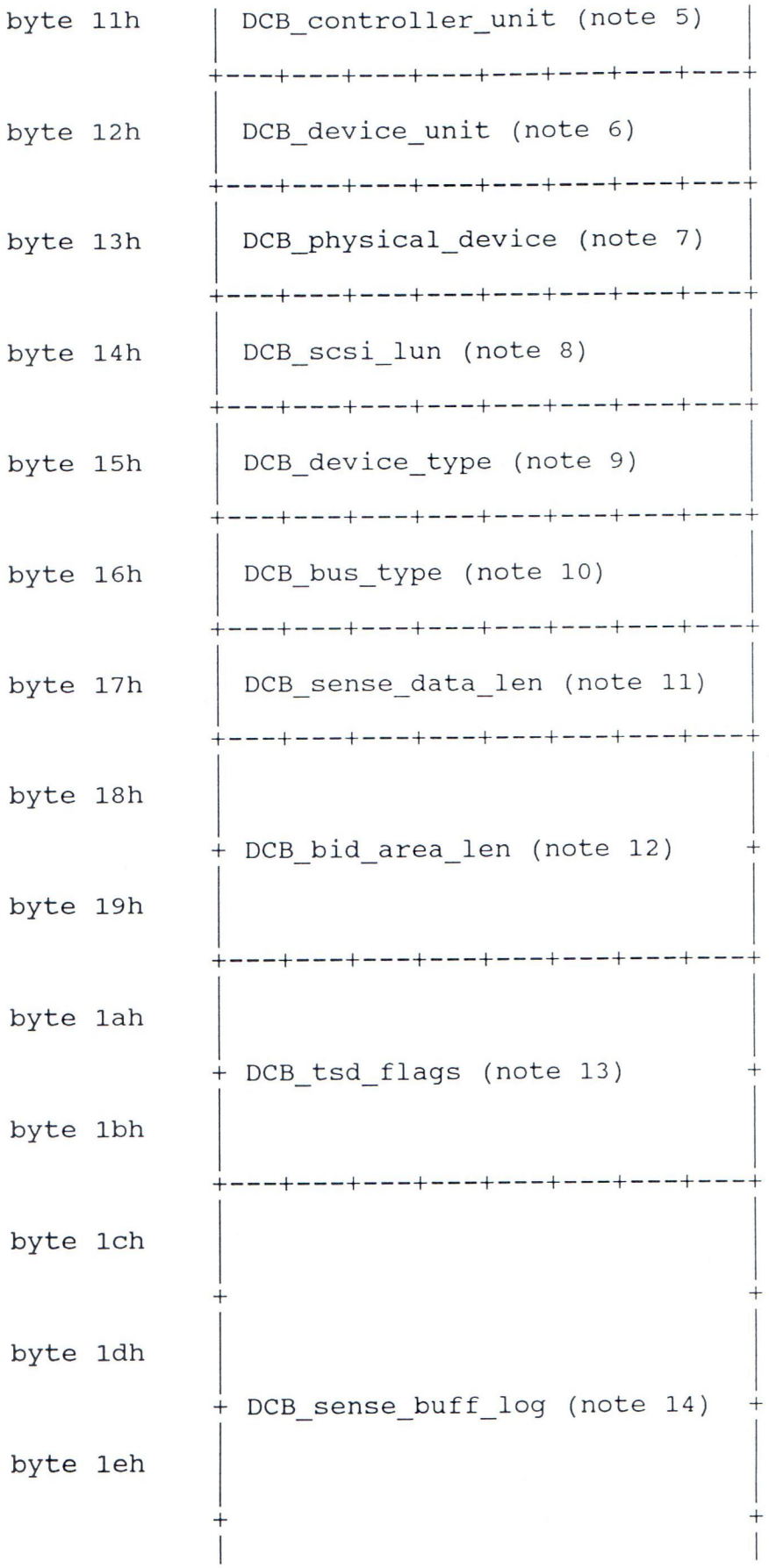
Figure 17. General Structure of the DCB

The following notes provide some basic guidelines for accessing the various parts of the DCB.

1. Drivers should not make any direct reference to a DCB's call down table. Instead they reference it via an index provided to them for that purpose.

The format of the DCB's common section is as follows:




byte 11h
DCB_controller_unit (note 5)
byte 12h
DCB_device_unit (note 6)
byte 13h
DCB_physical_device (note 7)
byte 14h
DCB_scsi_lun (note 8)
byte 15h
DCB_device_type (note 9)
byte 16h
DCB_bus_type (note 10)
byte 17h
DCB_sense_data_len (note 11)
byte 18h
DCB_bid_area_len (note 12)
byte 19h
byte 1ah
DCB_tsd_flags (note 13)
byte 1bh
byte 1ch
byte 1dh
DCB_sense_buff_log (note 14)
byte 1eh

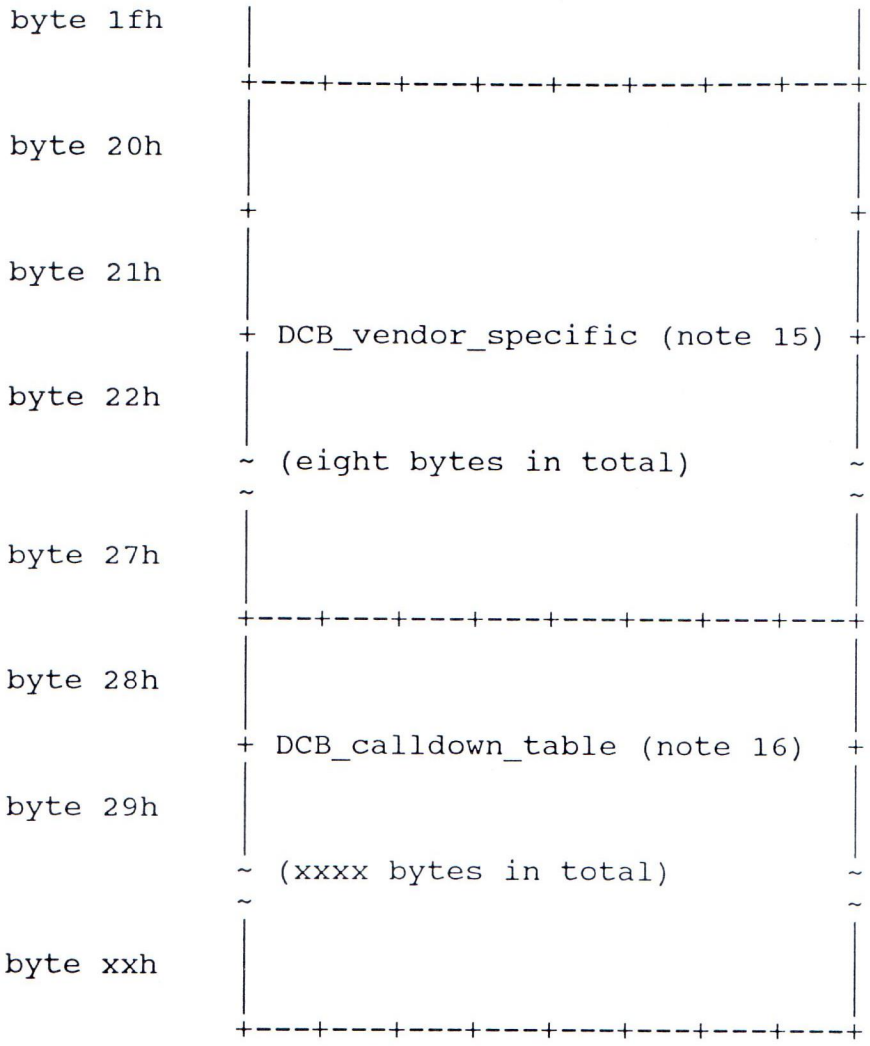


Figure 18. Device Control Block (DCB) Common Section Format

Notes:

1. **DCB_phys_addr** This field contains the 32-bit physical address of this DCB.

2. **DCB_ascii_name** This field contains the eight bytes ascii name of the device.

3. **DCB_next_dcb_logical** This field contains the IOS:32 pointer to the next DCB in IOS's DCB chain.

   This field is owned by IOS and should be neither inspected nor changed by drivers.

4. **DCB_controller** This field contains the 8-bit controller number.

5. **DCB_controller_unit** This field contains the 8-bit unit number within the controller.

6. **DCB_device_unit** This field contains the 8-bit logical unit number of the volume.

7. **DCB_physical_device** This field contains the 8-bit physical drive address in the case of an ESDI attached device, and the 8-bit target id in the case of a SCSI attached device.

8. **DCB_scsi_lun** This field contains the 8-bit logical unit number of a SCSI attached device.

9. **DCB_device_type** This field contains an 8-bit device type code, defined as follows:

   DCB_TYPE_CHS_FIXED_DISK (01 hex) The device is a fixed disk within which data is located via a cylinder, head, and sector address

   DCB_TYPE_RBA_FIXED_DISK (02 hex) The device is a fixed disk within which data is located via a relative block address

   DCB_TYPE_SEQ_TAPE (02 hex) The device is a sequentially access tape drive

   DCB_TYPE_PRINTER (03 hex) The device is a printer

   DCB_TYPE_525_FLOPPY (04 hex) The device is a floppy disk drive which supports 5.25" removable media

   DCB_TYPE_350_FLOPPY (05 hex) The device is a floppy disk drive which supports 3.5" removable media

   DCB_TYPE_525_CDROM (06 hex) The device is a cd-rom disk drive which supports 3.5" removable media

   DCB_TYPE_ASYNC (07 hex) The device is a asynchronous communications port

   DCB_TYPE_SDLC (08 hex) The device is an SDLC communications port

   DCB_TYPE_BISYNC (09 hex) The device is a BISYNC communications port

10. **DCB_bus_type** This field contains an 8-bit bus interface type code, defined as follows:

    DCB_BUS_ESDI (01 hex) The device is connected via an ESDI controller

DCB_BUS_SCSI (02 hex) The device is connected via a SCSI controller

DCB_BUS_SS (03 hex) The device is connected via a super-smart controller

11. **DCB_sense_data_len** This 8-bit field contains the number of sense bytes provided by this device

12. **DCB_bid_area_len** This 16-bit field contains the size, in bytes, of the BID specific area in SRB's built for this device

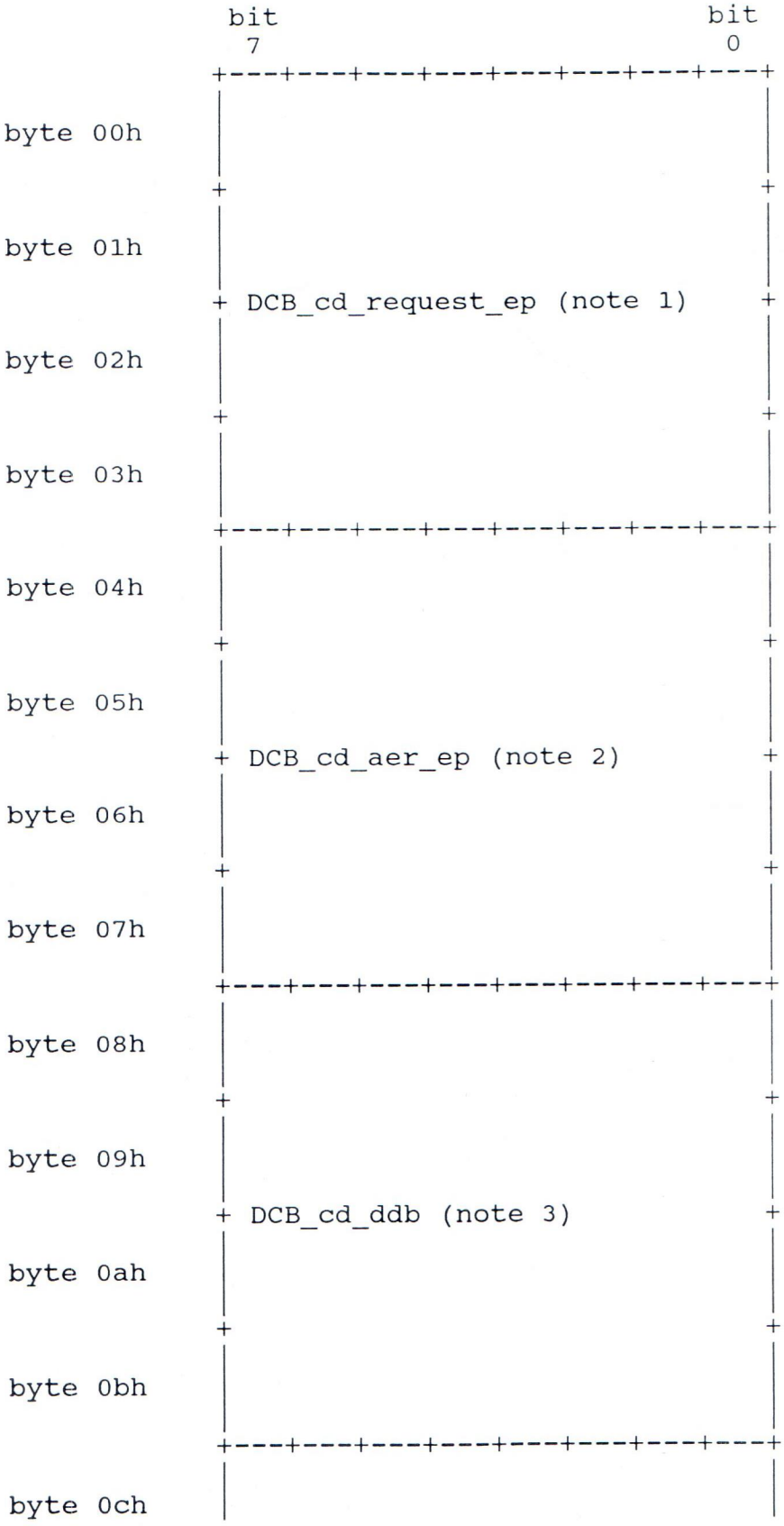
13. **DCB_tsd_flags** This 16-bit field contains flags which are private to the TSD controlling this device

14. **DCB_sense_buff_log** This field contains the IOS:32 pointer to the sense data for this device.

15. **DCB_bid_specific** This eight byte field contains sub-fields which are private to the BID currently handling any active I/O request for this device

16. **DCB_calldown_table** This field contains the device's calldown table, which is fully defined in the next section

The format of an entry in the DCB's calldown table is as follows:



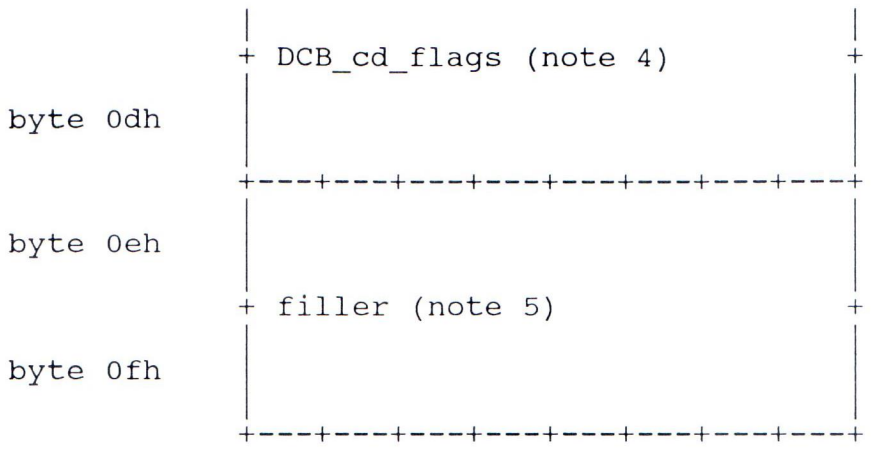


Figure 19. Device Control Block Calldown Table Format

Notes:

1. **DCB_cd_request_ep** This field contains the 16:16 address of the driver's request entry point

2. **DCB_cd_aer_ep** This field contains the 16:16 address of the driver's asynchronous event routine entry point

3. **DCB_cd_ddb** This field contains the IOS:32 address of the driver's DDB

4. **DCB_cd_flags** This 16-bit field contains the driver's demand bits. These bits represent demands being made of the next higher driver by this driver.

   As each driver initializes, before it inserts its own entry in the calldown table, it should examine the demand bits of the next lower driver. Except for TSD's those demand bits which are not supported by the initializing driver should be copied into its own demand bits. TSD's should generate an appropriate error report if any unsupported demand bits are on.

   All drivers should check all other demand bits for consistancy and an appropriate error report should be generated if they are inconsistent.

   The following demand bits are defined:

   DCB_demand_SRB_CDB (0001 hex) The driver requires that it be provided with an SRB and CDB for each RH.

   BID's which support SCSI bus interfaces normally turn this bit on.

   DCB_demand_logical (0002 hex) The driver

requires that it be provided with logical media addresses and that a logical DCB be associated with the RCB

This bit is typically turned on by certain special kinds of VSD (such as FT enhancers, and some cachers) and is then propagated to the TSD by intermediate drivers which do not support it.

DCB_demand_physical (0003 hex) The driver requires that it be provided with physical media addresses and that a physical DCB be associated with the RCB

This bit is normally turned on by a BID and then propagated to the TSD by intermidiate drivers which do not support it.

Certain special kinds of VSD (such as FT enhancers, and some cachers) do support this bit. They typically set DCB_demand_logical in their own demand bits.

DCB_demand_reserved (7ff8 hex) These bits are reserved for future use.

DCB_demand_contig_sns (8000 hex) The driver requires that the sense data area immediately follow the BID's private area in SRB's.

Failure to support this bit will render SCSI'izer VSD's incompatible with certain SCSI controller cards.

5. filler This 16-bit field exists to achieve dword alignment of calldown table entries

Driver Data Block - DDB

Driver data blocks (DDB) exist to satisfy driver re-entrancy constraints: they provide private work space.

A particular DDB exists only because its associated driver needs private memory.

In general, BID's create one DDB for each bus interface and other drivers do not need DDB's. A DDB is private to the driver which created it, although IOS may inspect or update the DDB's header.

DDB's always reside in IOS's memory pool, are contiguous, and are referenced via an IOS:32 address.

The DDB has a small fixed format header and a driver defined main section.

It is declared in DDB.H and typically is redefined by drivers.

The following code fragment is an example of driver redefinition of the DDB.

```
driver_ddb      struc

                db      size ddb dup (?)

d_ddb_field_1   dw      ?               ; appropriate comment.
d_ddb_field_2   dd      ?               ; appropriate comment.
d_ddb_field_3   db      ?               ; appropriate comment.
d_ddb_field_4   dw      ?               ; appropriate comment.

driver_ddb      ends
```

Driver Registration Packet - DRP

A driver registration packet (DRP) is generated by a driver and passed to IOS during driver registration. It contains information which identifies the version and capabilities of the driver and provides the address of the drivers asynchronous event routine. DRP's are typically assembled into a driver's initialization data segment.

A DRP may reside anywhere in memory and is not necessarily contiguous. It is referenced via a 16:32 address.

It is declared in DRP.H.

Driver Vector Table - DVT

A driver vector table (DVT) is created by IOS during driver registration. It contains various addresses within the driver and information about the driver which IOS may need at a later time. DVT's are private to IOS.

DVT's always reside in IOS's memory pool, are contiguous, and are referenced via an IOS:32 address.

It is declared in DVT.H.

Inter-driver Communication Packet - IDC

An inter-driver communication packet (IDC) is created during the processing of an AttachDD DevHlp.

An IDC may reside anywhere in memory and is not necessarily contiguous. It is referenced via a 16:16 address.

It is declared in IDC.H.

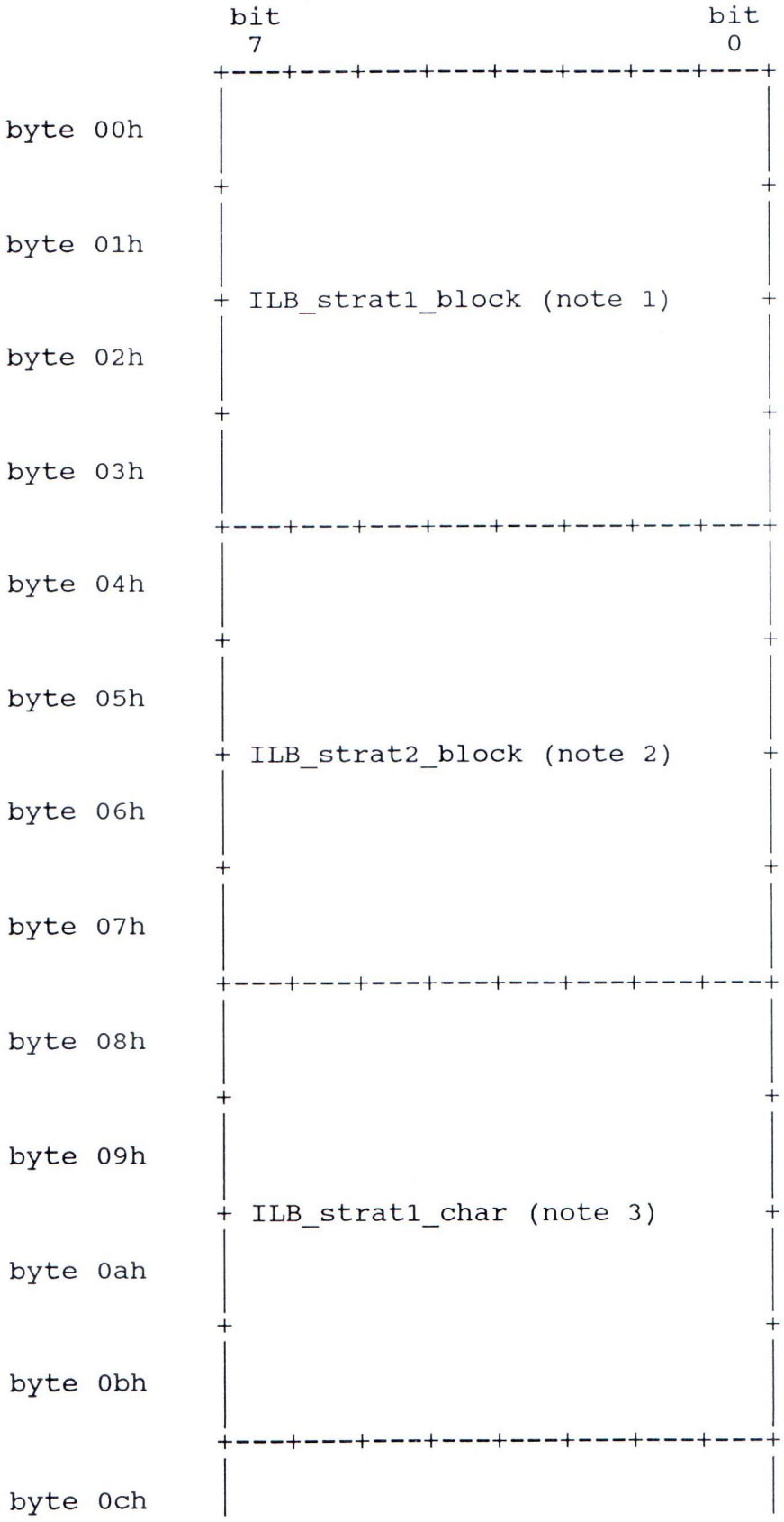
IOS Linkage Block - ILB

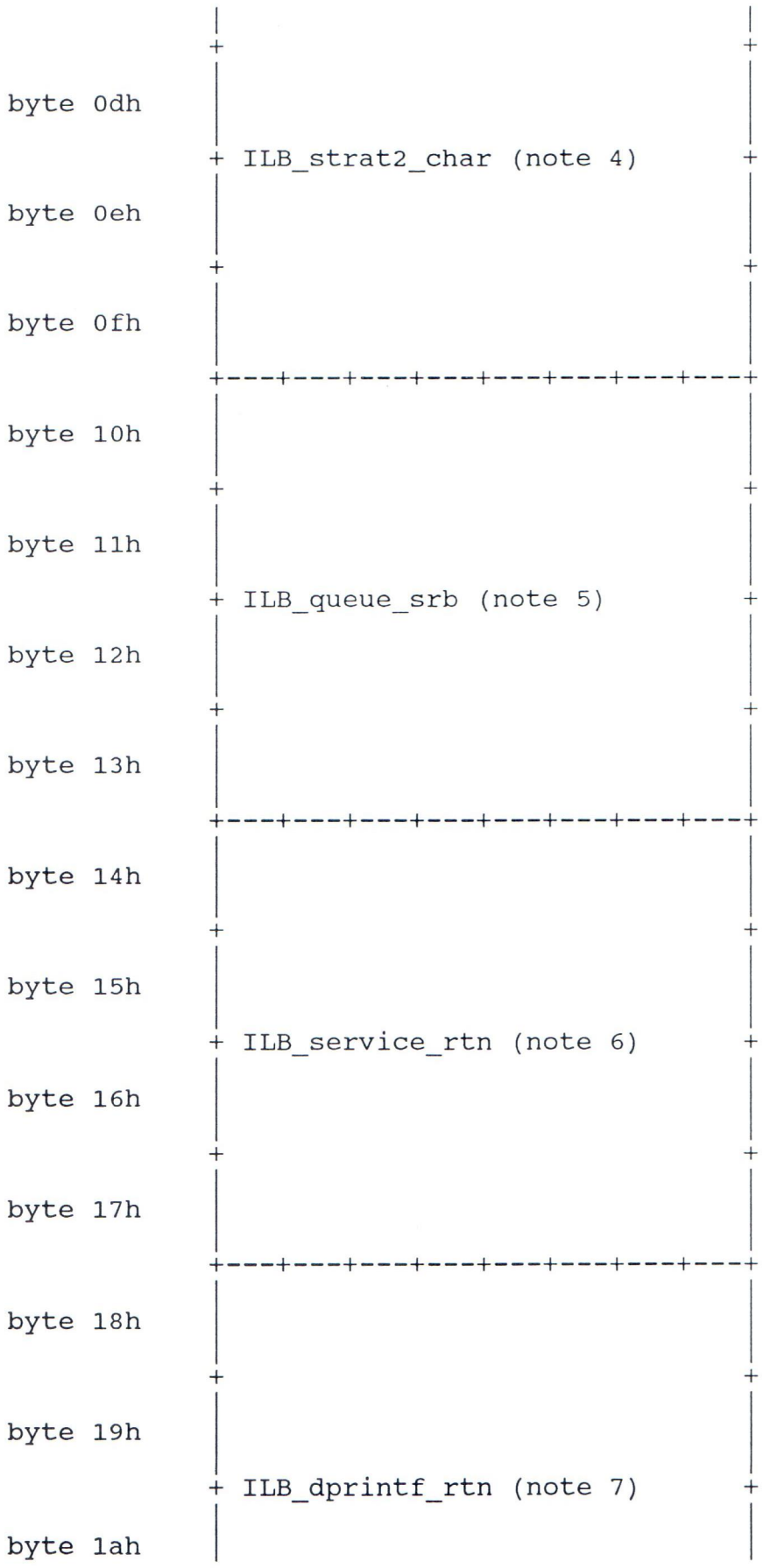
An empty IOS linkage block (ILB) is generated by a driver and passed to IOS during driver registration. IOS fills the ILB in with IOS version and configuration information and various IOS entry points for later use by the driver. DRP's are typically assembled into a driver's code segment.

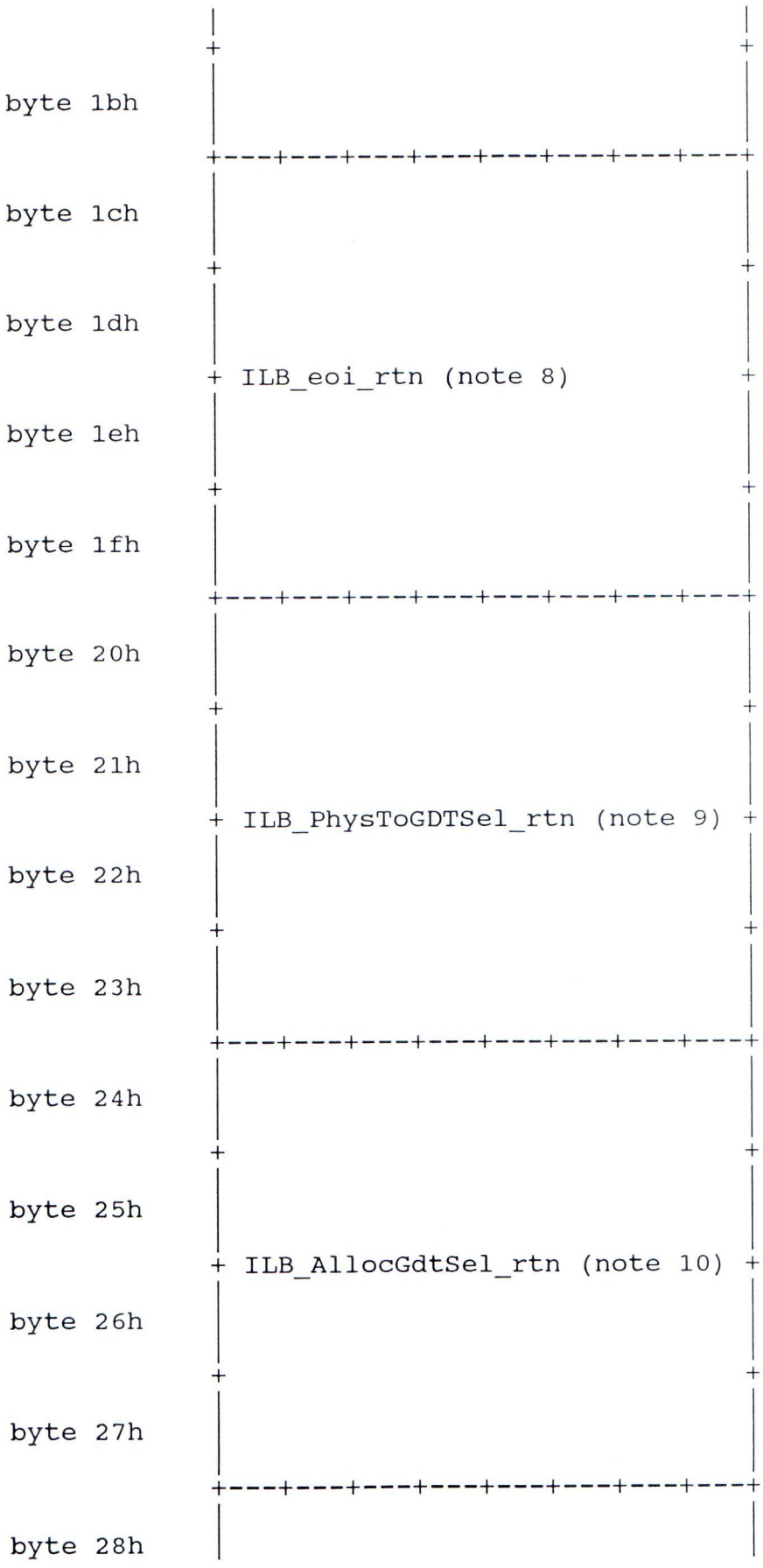
An ILB may reside anywhere in memory and is not necessarily contiguous. It is referenced via a 16:32 address.

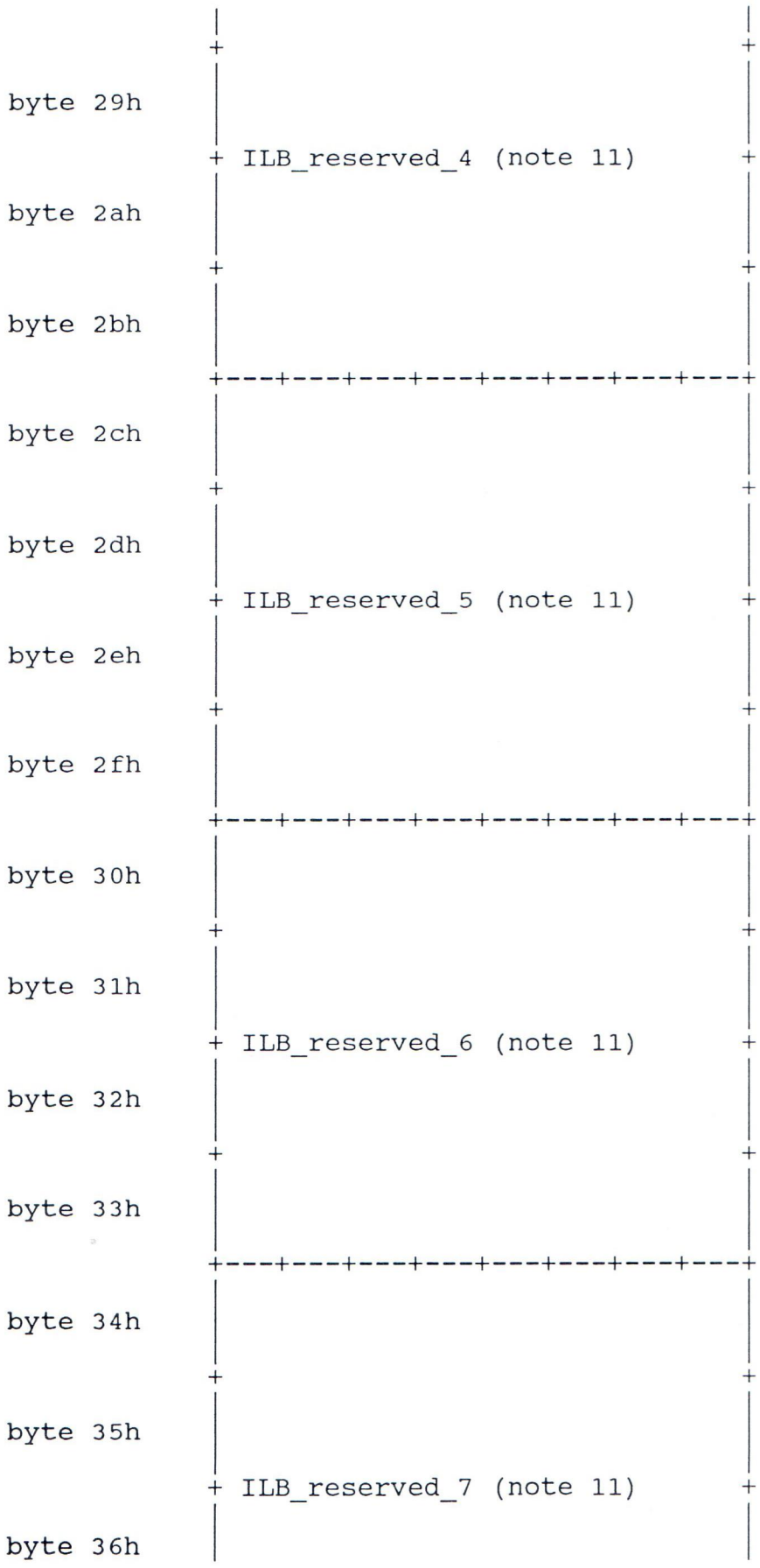
It is declared in ILB.H.

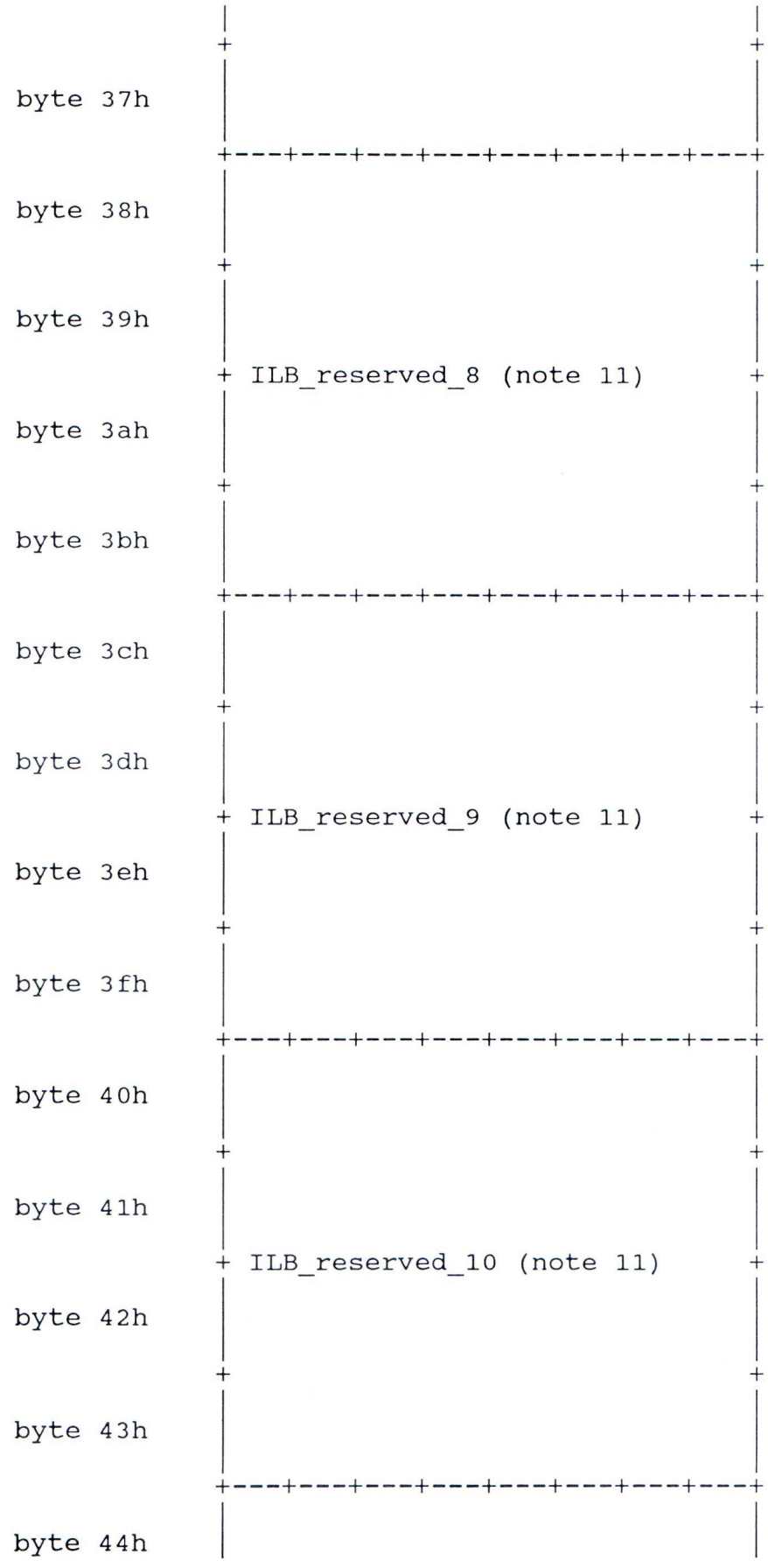
The following figure illustrates the format and content of a ILB.

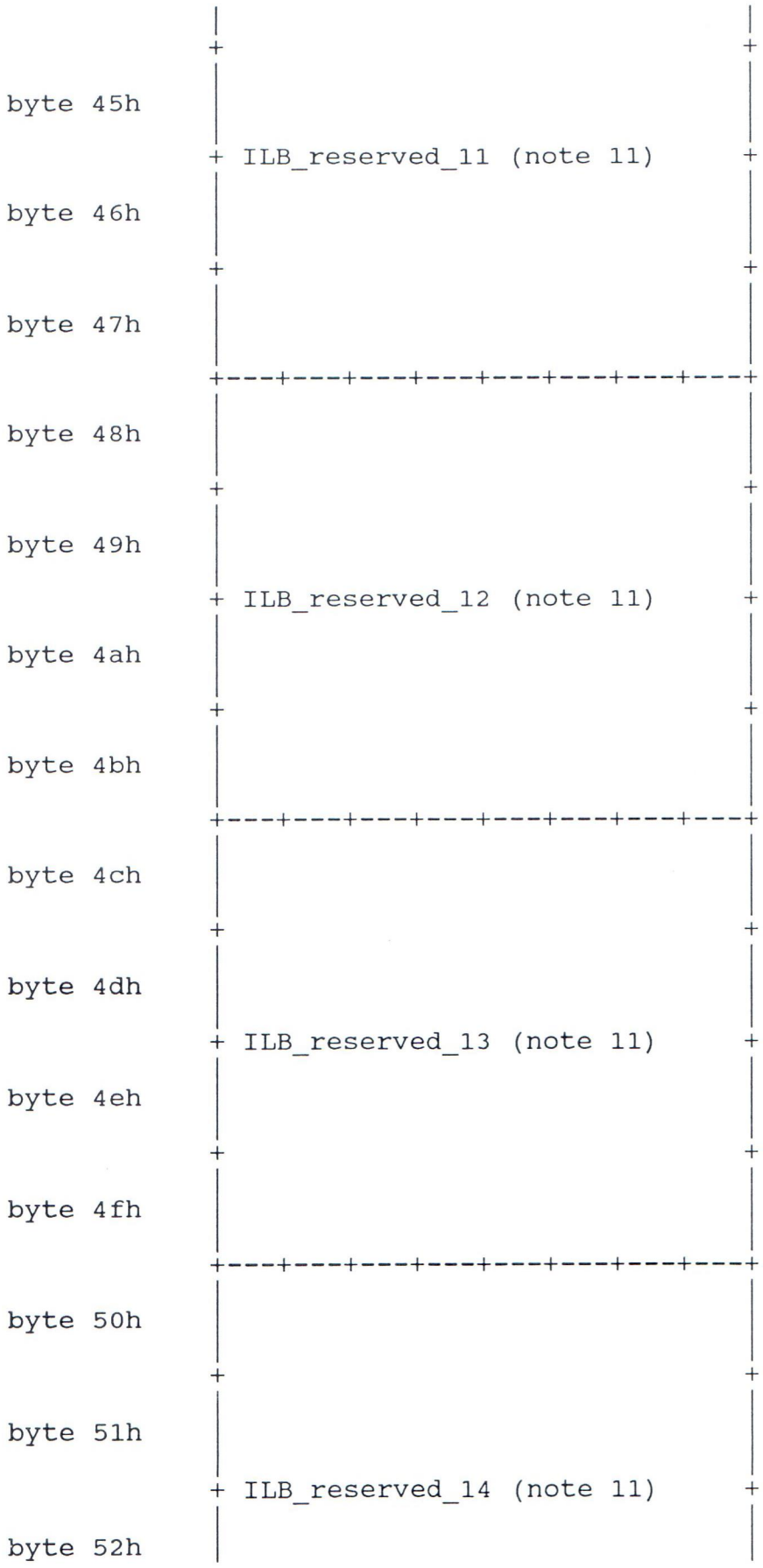


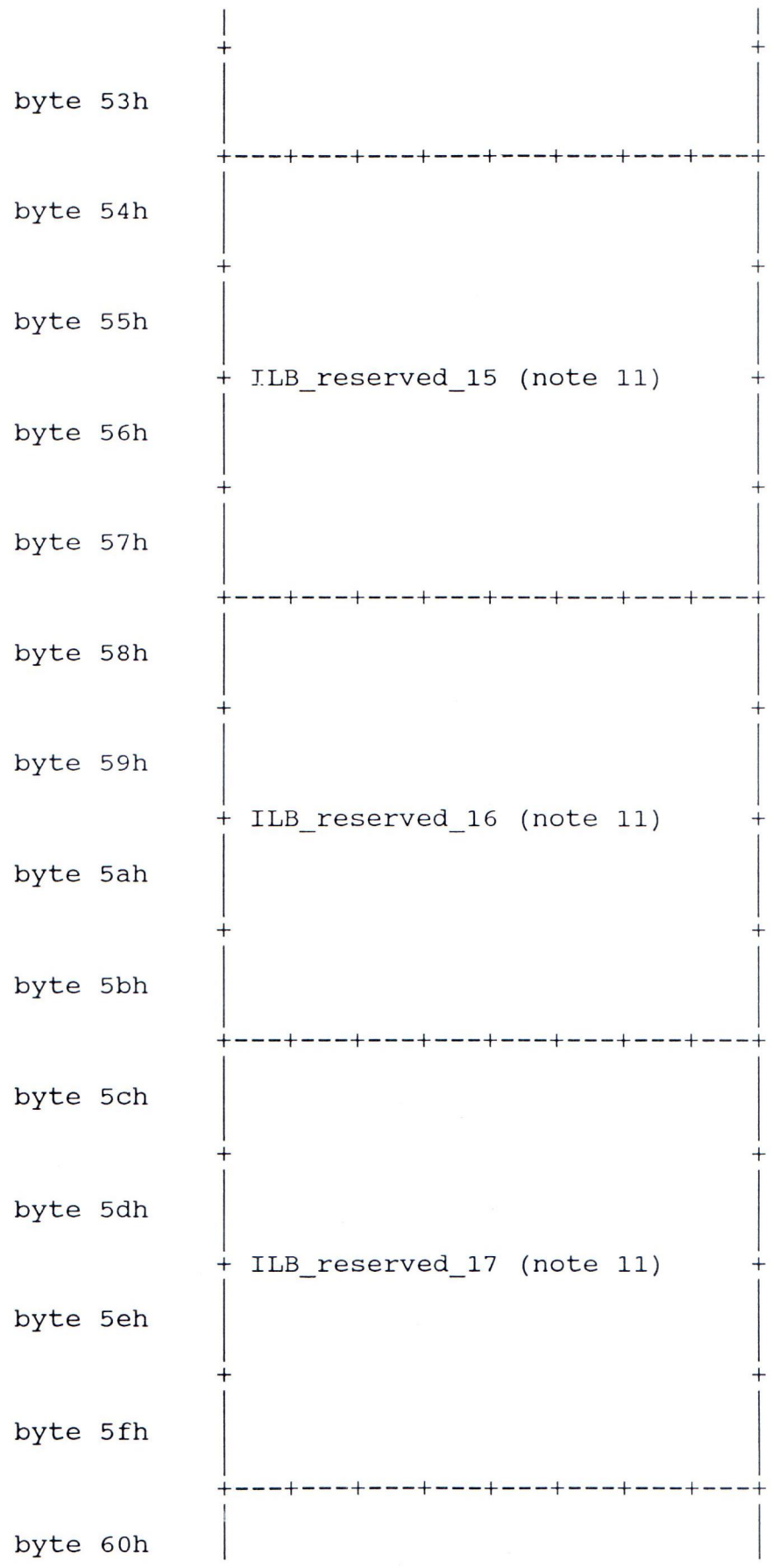

byte 0dh
ILB_strat2_char (note 4)
byte 0eh
byte 0fh
byte 10h
byte 11h
ILB_queue_srb (note 5)
byte 12h
byte 13h
byte 14h
byte 15h
ILB_service_rtn (note 6)
byte 16h
byte 17h
byte 18h
byte 19h
ILB_dprintf_rtn (note 7)
byte 1ah

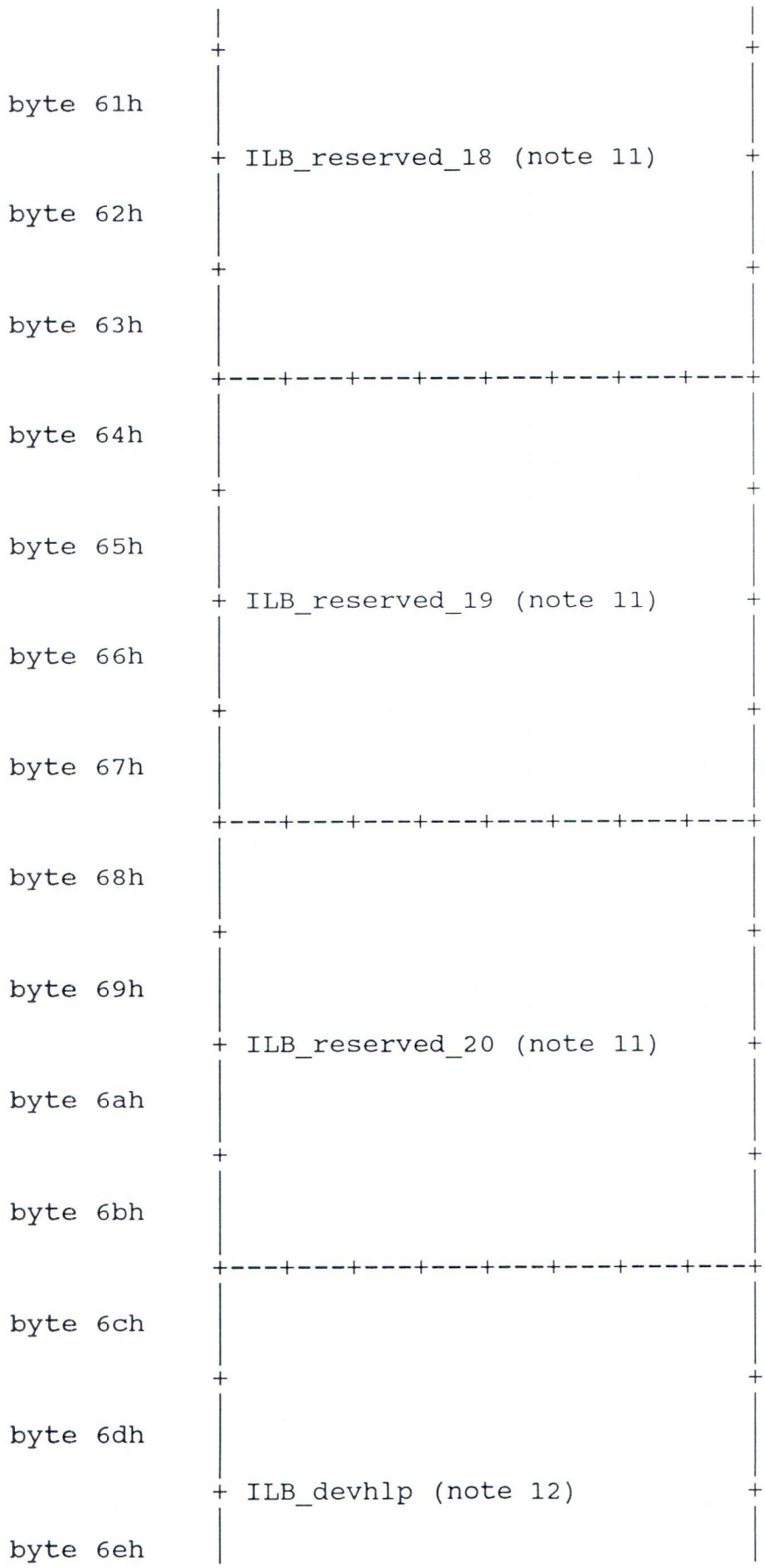

byte 1bh
byte 1ch
byte 1dh
ILB_eoi_rtn (note 8)
byte 1eh
byte 1fh
byte 20h
byte 21h
ILB_PhysToGDTSel_rtn (note 9)
byte 22h
byte 23h
byte 24h
byte 25h
ILB_AllocGdtSel_rtn (note 10)
byte 26h
byte 27h
byte 28h

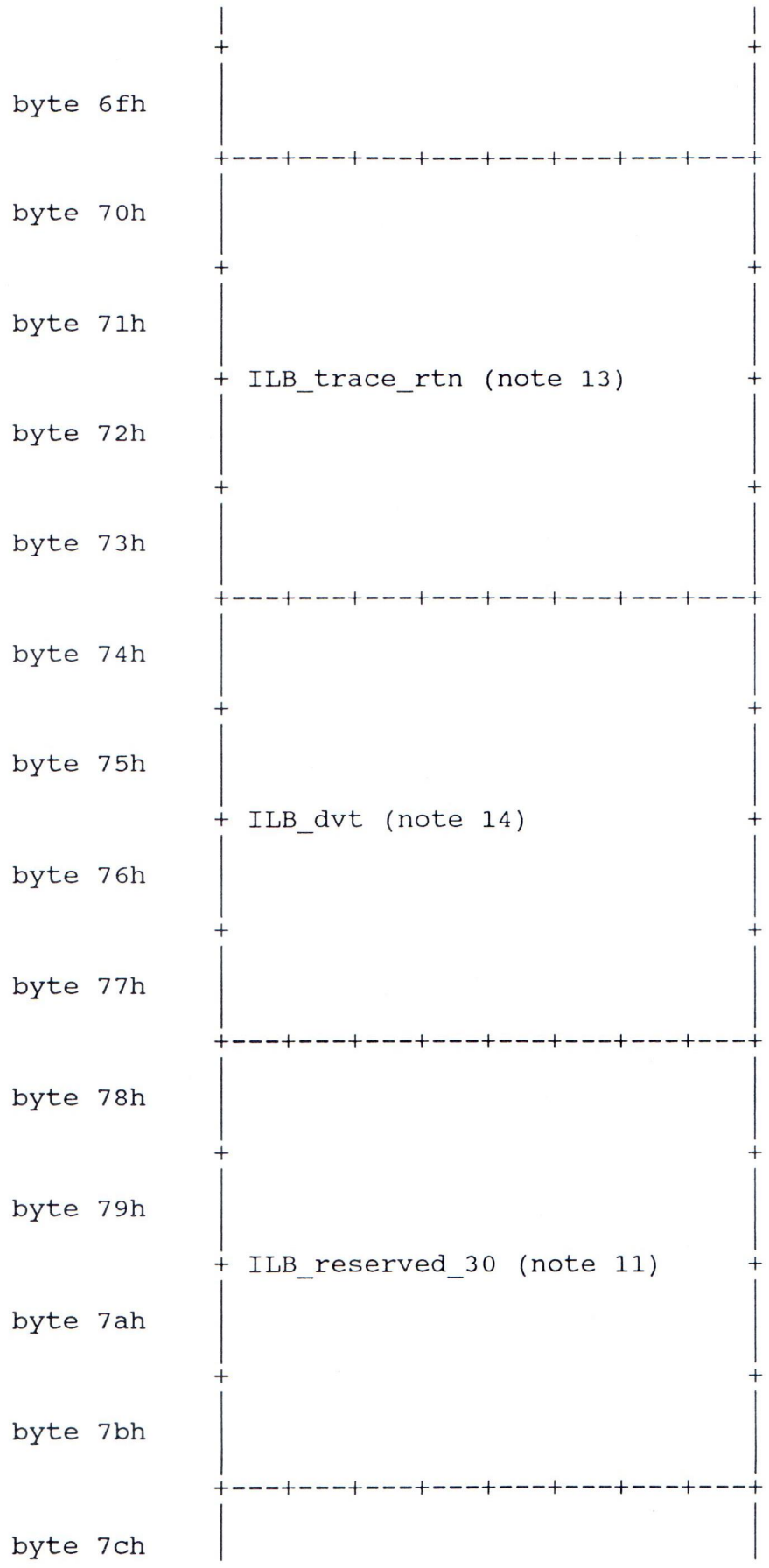

byte 29h
ILB_reserved_4 (note 11)
byte 2ah
byte 2bh
byte 2ch
byte 2dh
ILB_reserved_5 (note 11)
byte 2eh
byte 2fh
byte 30h
byte 31h
ILB_reserved_6 (note 11)
byte 32h
byte 33h
byte 34h
byte 35h
ILB_reserved_7 (note 11)
byte 36h

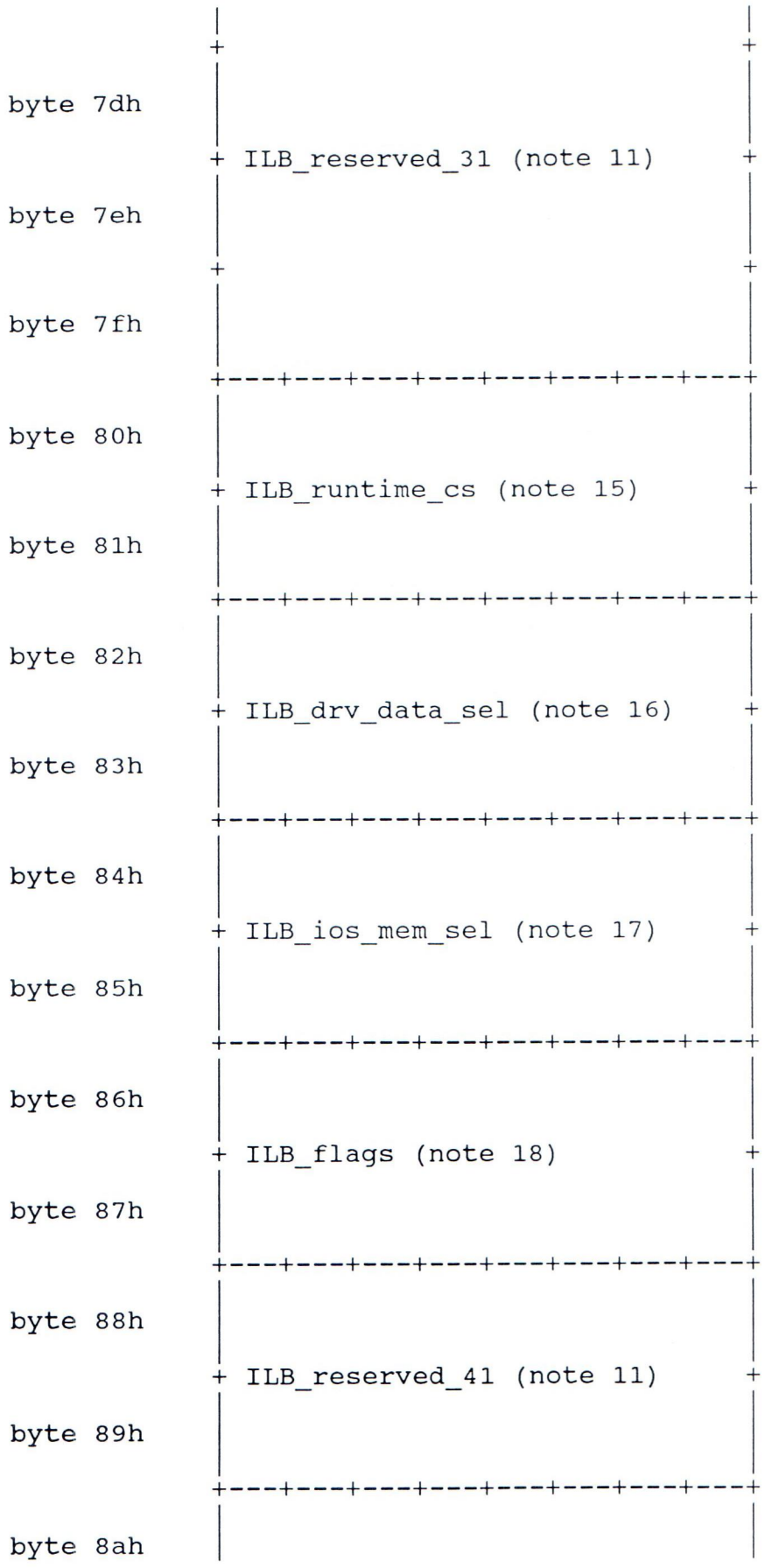

byte 37h
byte 38h
byte 39h
ILB_reserved_8 (note 11)
byte 3ah
byte 3bh
byte 3ch
byte 3dh
ILB_reserved_9 (note 11)
byte 3eh
byte 3fh
byte 40h
byte 41h
ILB_reserved_10 (note 11)
byte 42h
byte 43h
byte 44h


byte 45h
ILB_reserved_11 (note 11)
byte 46h
byte 47h
byte 48h
byte 49h
ILB_reserved_12 (note 11)
byte 4ah
byte 4bh
byte 4ch
byte 4dh
ILB_reserved_13 (note 11)
byte 4eh
byte 4fh
byte 50h
byte 51h
ILB_reserved_14 (note 11)
byte 52h


byte 53h
byte 54h
byte 55h
ILB_reserved_15 (note 11)
byte 56h
byte 57h
byte 58h
byte 59h
ILB_reserved_16 (note 11)
byte 5ah
byte 5bh
byte 5ch
byte 5dh
ILB_reserved_17 (note 11)
byte 5eh
byte 5fh
byte 60h


byte 61h
ILB_reserved_18 (note 11)
byte 62h
byte 63h
byte 64h
byte 65h
ILB_reserved_19 (note 11)
byte 66h
byte 67h
byte 68h
byte 69h
ILB_reserved_20 (note 11)
byte 6ah
byte 6bh
byte 6ch
byte 6dh
ILB_devhlp (note 12)
byte 6eh


byte 6fh
byte 70h
byte 71h
ILB_trace_rtn (note 13)
byte 72h
byte 73h
byte 74h
byte 75h
ILB_dvt (note 14)
byte 76h
byte 77h
byte 78h
byte 79h
ILB_reserved_30 (note 11)
byte 7ah
byte 7bh
byte 7ch


byte 7dh
ILB_reserved_31 (note 11)
byte 7eh
byte 7fh
byte 80h
ILB_runtime_cs (note 15)
byte 81h
byte 82h
ILB_drv_data_sel (note 16)
byte 83h
byte 84h
ILB_ios_mem_sel (note 17)
byte 85h
byte 86h
ILB_flags (note 18)
byte 87h
byte 88h
ILB_reserved_41 (note 11)
byte 89h
byte 8ah

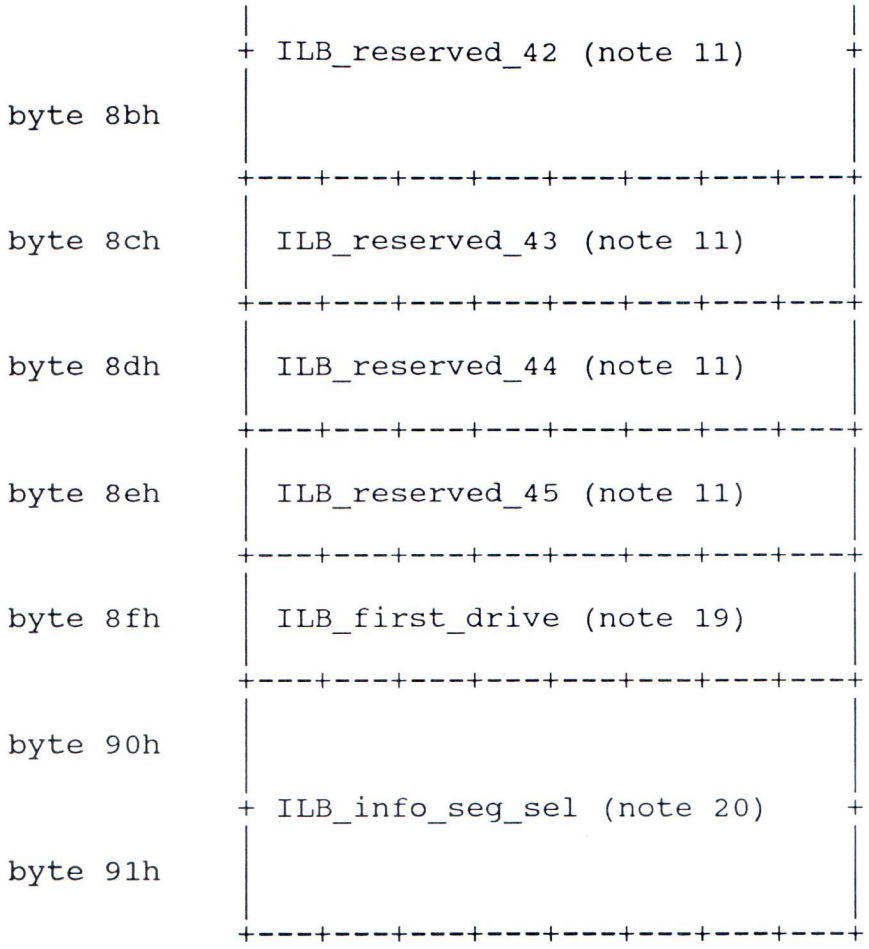


Figure 20. IOS Linkage Block (ILB) Format

Notes:

1. **ILB_strat1_block** During registration, IOS stores into this field a 16:16 pointer to IOS's STRAT1 request submission routine for block devices.

2. **ILB_strat2_block** During registration, IOS stores into this field a 16:16 pointer to IOS's STRAT2 request submission routine for block devices.

3. **ILB_strat1_char** During registration, IOS stores into this field a 16:16 pointer to IOS's STRAT1 request submission routine for character devices.

4. **ILB_strat2_char** During registration, IOS stores into this field a 16:16 pointer to IOS's STRAT2 request submission routine for character devices.

5. **ILB_queue_srb** During registration, IOS stores into this field a 16:16 pointer to IOS's SRB queuing routine.

   When subsequently called by the driver, the SRB

queuing routine expects the SI register to contain an IOS:16 pointer to the SRB and the DI register to contain an IOS:16 pointer to the DCB.

6. **ILB_service_rtn** During registration, IOS stores into this field a 16:16 pointer to IOS's general purpose service routine.

7. **ILB_dprintf_rtn** During registration, IOS stores into this field a 16:16 pointer to IOS's DPRINTF routine.

8. **ILB_eoi_rtn** During registration, IOS stores into this field a 16:16 pointer to IOS's end of interrupt routine.

9. **ILB_PhysToGDTSel_rtn** During registration, IOS stores into this field a 16:16 pointer to an IOS routine which is functionally identical to the OS/2 Kernel PhysToGDTSel DEVHLP.

10. **ILB_AllocGdtSel_rtn** During registration, IOS stores into this field a 16:16 pointer to an IOS routine which is functionally identical to the OS/2 Kernel AllocGdtSel DEVHLP.

11. **Reserved Fields** These fields are reserved for future use.

12. **ILB_devhlp** During registration, IOS stores into this field a 16:16 pointer to the OS/2 Kernel's DEVHLP routine.

13. **ILB_trace_rtn** During registration, IOS stores into this field a 16:16 pointer to IOS's debug trace routine.

14. **ILB_dvt** During registration, IOS stores into this field an IOS:32 pointer to the DVT for this instance of the driver.

15. **ILB_runtime_cs** During registration, IOS stores into this field the 16 bit code selector which the driver will use when it is running at ring zero after its initialization is complete.

16. **ILB_drv_data_sel** During registration, IOS stores into this field the 16 bit selector for the drivers data group.

17. **ILB_ios_mem_sel** During registration, IOS stores into this field the 16 bit selector for its memory pool.

18. **ILB_flags** During registration, IOS stores into this field flags which are defined as follows:

    ILB_286_mode (0001 hex) current cpu is a 286

(fffe hex) reserved for future use

19. **ILB_first_drive** During registration, IOS stores into this field an eight bit value which corresponds to the lowest drive letter serviced by this instance of the driver.

20. **ILB_info_seg_sel** During registration, IOS stores into this field the 16 bit selector for the OS/2 Kernel's Global Info segment.

IOS Service Packet - ISP

IOS service packets (ISP) are generated by a driver and passed to IOS to request some IOS service to be performed.

An ISP may reside anywhere in memory and is not necessarily contiguous. It is referenced via a 16:32 address.

The ISP has a variable format which is dependant on the IOS service being requested.

It is declared in ISP.H.

IOS Vector Table - IVT

The IOS vector table (IVT) is assembled into IOS's code segment. During IOS initialization it is initialized with configuration information and addresses which IOS will need at a later time. It is created by IOS during driver registration. The IVT is private to IOS.

It is declared in IVT.H.

Memory Element Descriptor - MED

Memory element descriptors (MED) exist in IOS's memory pool. They are created, changed, and deleted by IOS's memory management routines as needed to control use of the memory pool. MED's are private to IOS and contain a checksum to detect tampering and damage.

MED's always reside in IOS's memory pool, are contiguous, and are referenced via an IOS:32 address.

They are declared in MED.H.

Request Control Block - RCB

Request control blocks (RCB) are created by an IOS service routine at the behest of a driver. They contain pointers and control information needed to manage the processing of an I/O request. An RCB may be shared between IOS and several drivers.

RCB's always reside in IOS's memory pool, are contiguous, and are referenced via an IOS:32 address.

RCB's are declared in RCB.H.

STRAT1 Packet - RP

A STRAT1 packet (RP) is generated either by the kernel or a file system (FSD) and passed to a driver via the kernel and volume manager to request that an I/O operation be performed. In conjunction with pointers contained within it, it contains all the information required to precisely specify an I/O operation which is to be performed. RP's are typically contained in FSD or kernel owned space.

An RP may reside anywhere in memory and is not necessarily contiguous. It is referenced via a 16:16 address.

RP's have a variable format which is dependant on the I/O operation being performed. The disk, tape, cd-rom, and printer TSD specifications define the various RP formats.

For LADDR compliant drivers, the RP is declared in RP.H. It is also defined in other places (for use by non-compliant drivers, FSD's and the kernel) but these other definitions should not be used by compliant drivers.

STRAT2 Packet - RLH

A STRAT2 packet (RLH) is generated either by the kernel, a file system (FSD), IOS, or a driver to request that an I/O operation be performed. An RLH may or may not be passed through the kernel and the volume manager. In conjunction with pointers contained within it, it contains all the information required to precisely specify an I/O operation which is to be performed. RLH's are typically contained in FSD or kernel owned space, or in IOS's memory pool.

In 286 systems, RLH's are not used outside of IOS.

An RLH may reside anywhere in memory and is not necessarily contiguous.

In release 1.21, RLH's are referenced via a 16:32 address. In subsequent releases it is normally referenced by a 16:16

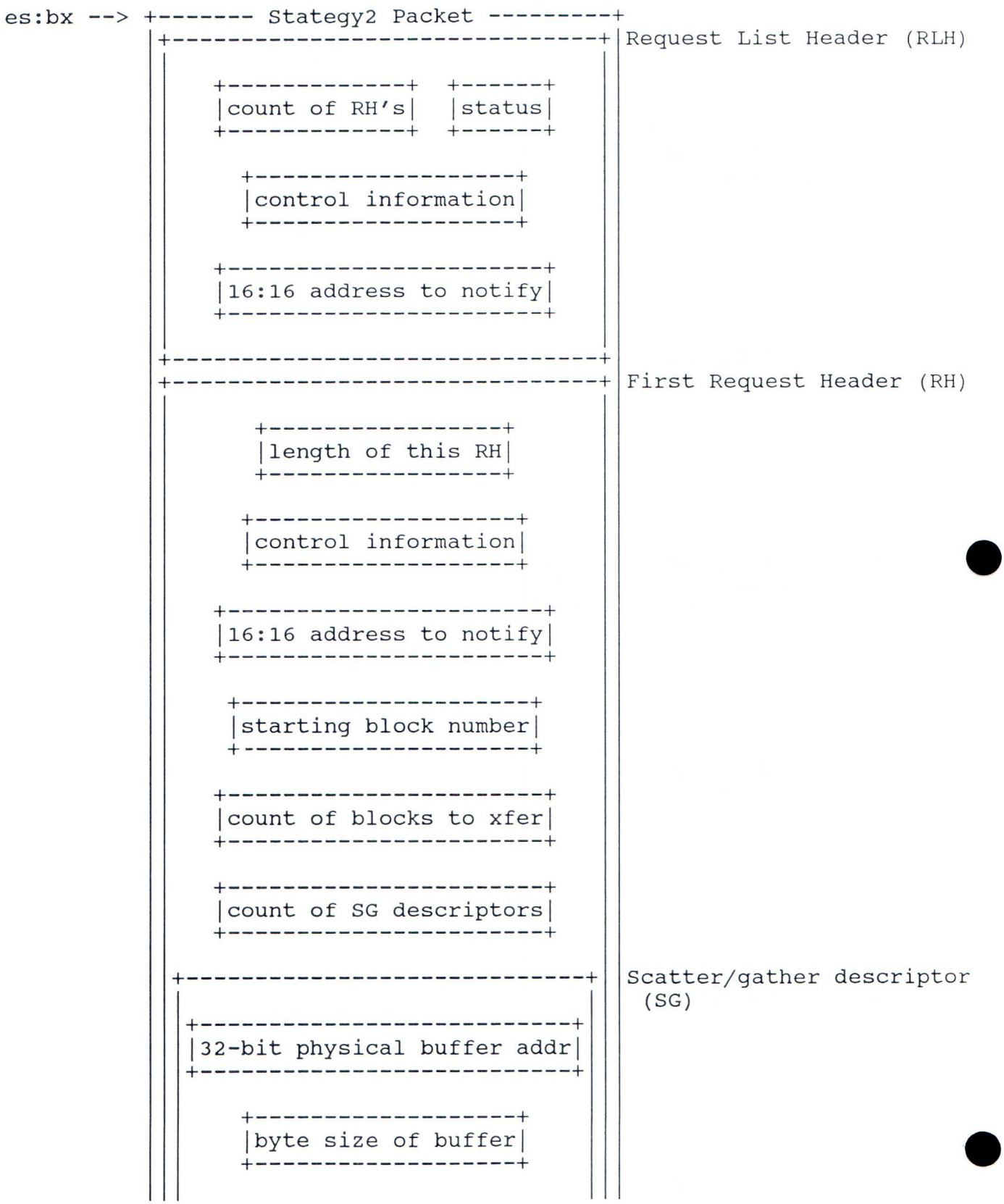
address, but may, be referenced by a 16:32 address - typically a 16:16 address which has been zero extended to 16:32 format.

RLH's have a segmented structure which results in their appearance varying in responce to changes in the degree of dicontiguity of data area on the media and the buffer area in memory.

For LADDR compliant drivers, the RLH is declared in RLH.H. It is also defined in other places (for use by non-compliant drivers, FSD's and the kernel) but these other definitions should not be used by compliant drivers.

The RLH is hierarchically structured with elements of the packet nested within other elements in a fixed order.

The following illustrates the relationship of the various elements of the RLH packet, with selected fields within each element shown.



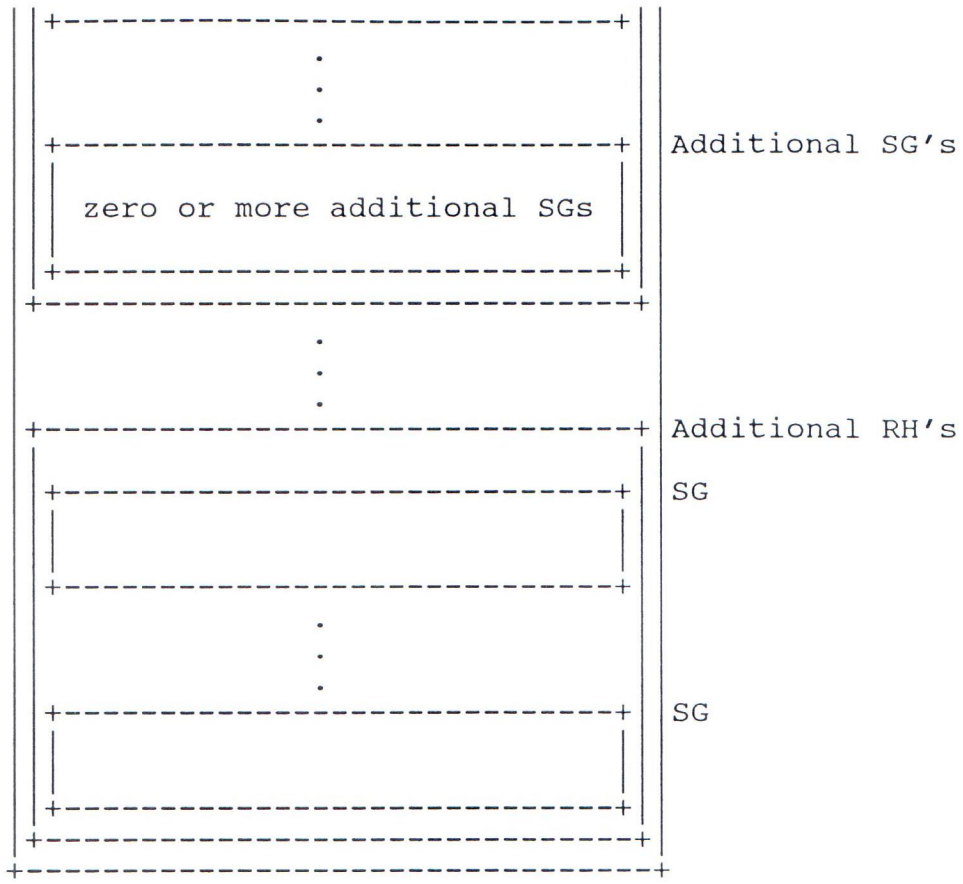


Figure 21. Hierarchical Structure of the RLH

The following notes provide some basic guidelines for accessing the various parts of the RLH.

1. The RLH is best viewed as a packed structure that consists of a fixed-size header, also called the RLH, followed by one or more variable length sub-structures, called RH's.

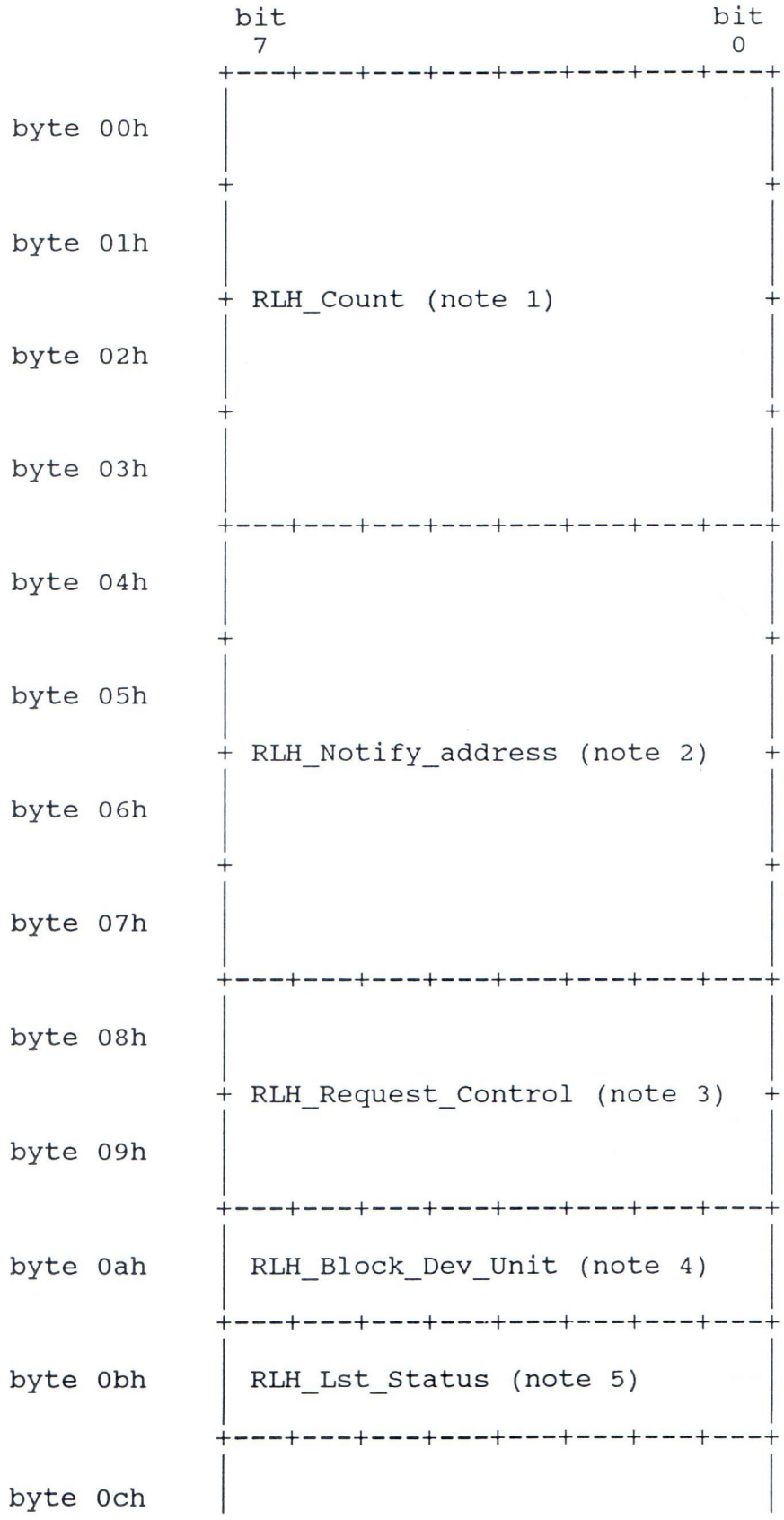
   RH's are, in turn, packed structures that comprise a fixed-size section, called the RH and one or more fixed size sections, called SG's.

2. The first RH starts at a fixed displacement from the RLH. Use "size Req_List_header" to obtain the displacement.

3. Given the address of an RH, the address of the RLH can be computed by subtracting RH_Head_Offset from the address of the RH.

4. The address of the first SG can be computed by adding "size Req_Header" to the address of the RH. Adding "size SG_Descriptor" to the address of one SG will give the address of the next SG.

The format of the RLH is as follows:



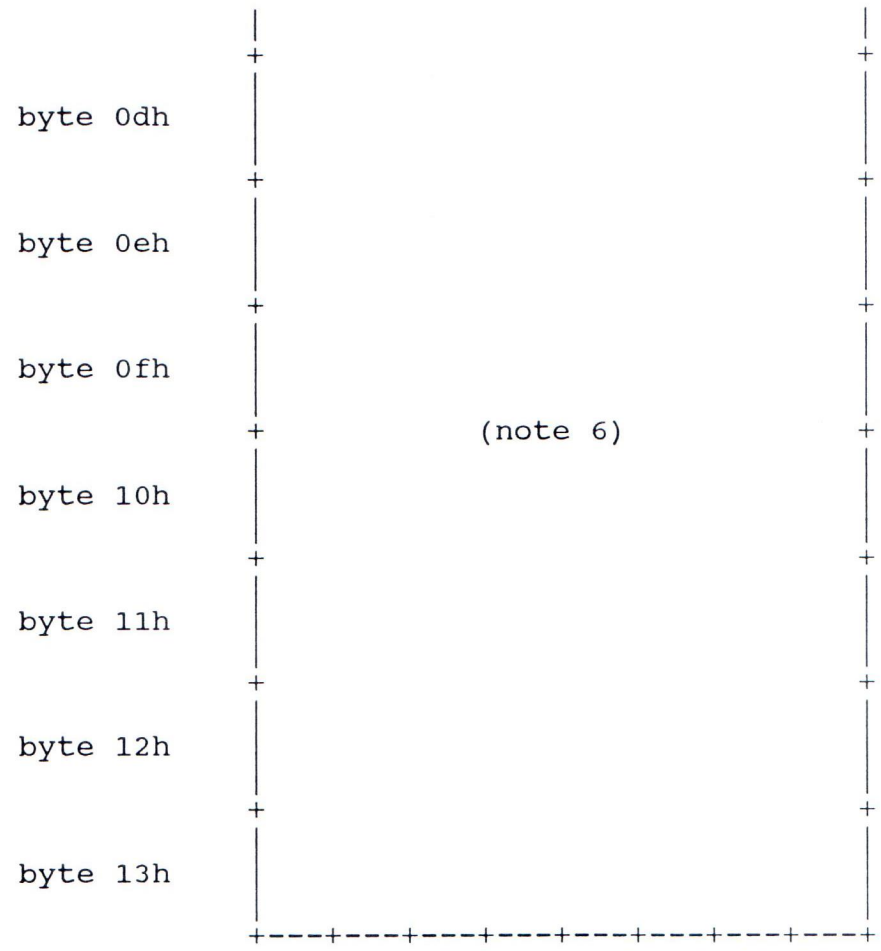


Figure 22. Request List Header (RLH) Format

Notes:

1. RLH_Count This field contains the count of RH's contained in this RLH.

The requestor must set it to a value in the range 1 through 32767.

2. RLH_Notify_address This field must either contain zero, or it must contain a 16:16 address of executable code which is present and locked down.

If bits RLH_NOTIFY_ERR and RLH_NOTIFY_DONE are both zero, this field is not used and may contain either a null or a valid address.

If bit RLH_NOTIFY_ERR is nonzero, this field must contain a non-null valid address that will gain control if any RH completes with an unrecoverable error.

If bit RLH_NOTIFY_DONE is nonzero, this field must contain a non-null valid address that will

gain control when this RLH completes

Note that if both bits RLH_NOTIFY_ERR and RLH_NOTIFY_DONE are nonzero and one or more RH's completes with an unrecoverable error, the subroutine pointed to by RLH_Notify_Address will gain control once for each RH that completes in error, and once for the completion of the RLH.

If both bits RLH_NOTIFY_ERR and RLH_NOTIFY_DONE are nonzero, the subroutine pointed to by RLH_Notify_Address should differentiate between the two possible cases where it gains control by examining RLH_Lst_Status. When RLH_Lst_Status contains RLH_ALL_REQ_DONE, the sub-routine gained control as the result of the request completing; otherwise control was gained as the result of an unrecoverable error.

This subroutine will gain control for all failing RH's before it gains control for the RLH completion.

Note: In the event that RLH_Notify_Address does not contain a valid address and a need exists to call that address, the call -- at the option of the device driver's author -- may or may not actually be attempted. In either case, system failure or a system hang will probably result.

3. **RLH_Request_Control** This field contains flags that control the processing of the request contained in the packet.

The requestor must set all the reserved flags to zero, and must set the other flags appropriately.

These flags are defined as follows:

RLH_Req_From_PB (0001 hex) request came directly from HPFS

RLH_Single_Req (0002 hex) single request in list

RLH_Exe_Req_Seq (0004 hex) RHs to be executed in sequence

RLH_Abort_Err (0008 hex) abort on error

RLH_Notify_Err (0010 hex) notify immediately on error

RLH_Notify_Done (0020 hex) notify when all RHs have completed

(ffc0 hex) reserved

4. **RLH_Block_Dev_Unit** The field contains the logical unit number upon which the I/O is to be performed. Unit 1 is "A:", unit 2 is "B:", etc.

   The requestor must set this field to a valid logical unit number for a disk partition which supports scatter/gather operations.

5. **RLH_Lst_Status** This field contains the overall status for the packet. The right (low) nibble indicates the degree of completion of the packet, defined as follows:

   RLH_No_Req_Queued (x0 hex) no request queued

   RLH_Req_Not_Queued (x1 hex) some, but not all, requests queued

   RLH_All_Req_Queued (x2 hex) all requests queued

   RLH_All_Req_Done (x4 hex) all request done or aborted

   RLH_Seq_In_Progress (x8 hex) requests being processed in sequence

   RLH_Abort_Pending (x8 hex) abort list processing in progress

   The left (high) nibble indicates the error recovery status of the packet, defined as follows:

   RLH_No_Error (0x hex) no error detected

   RLH_Rec_Error (1x hex) recoverable error has occured

   RLH_Unrec_Error (2x hex) unrecoverable error has occured

   RLH_Unrec_Error_Retry (3x hex) unrecoverable error after retry

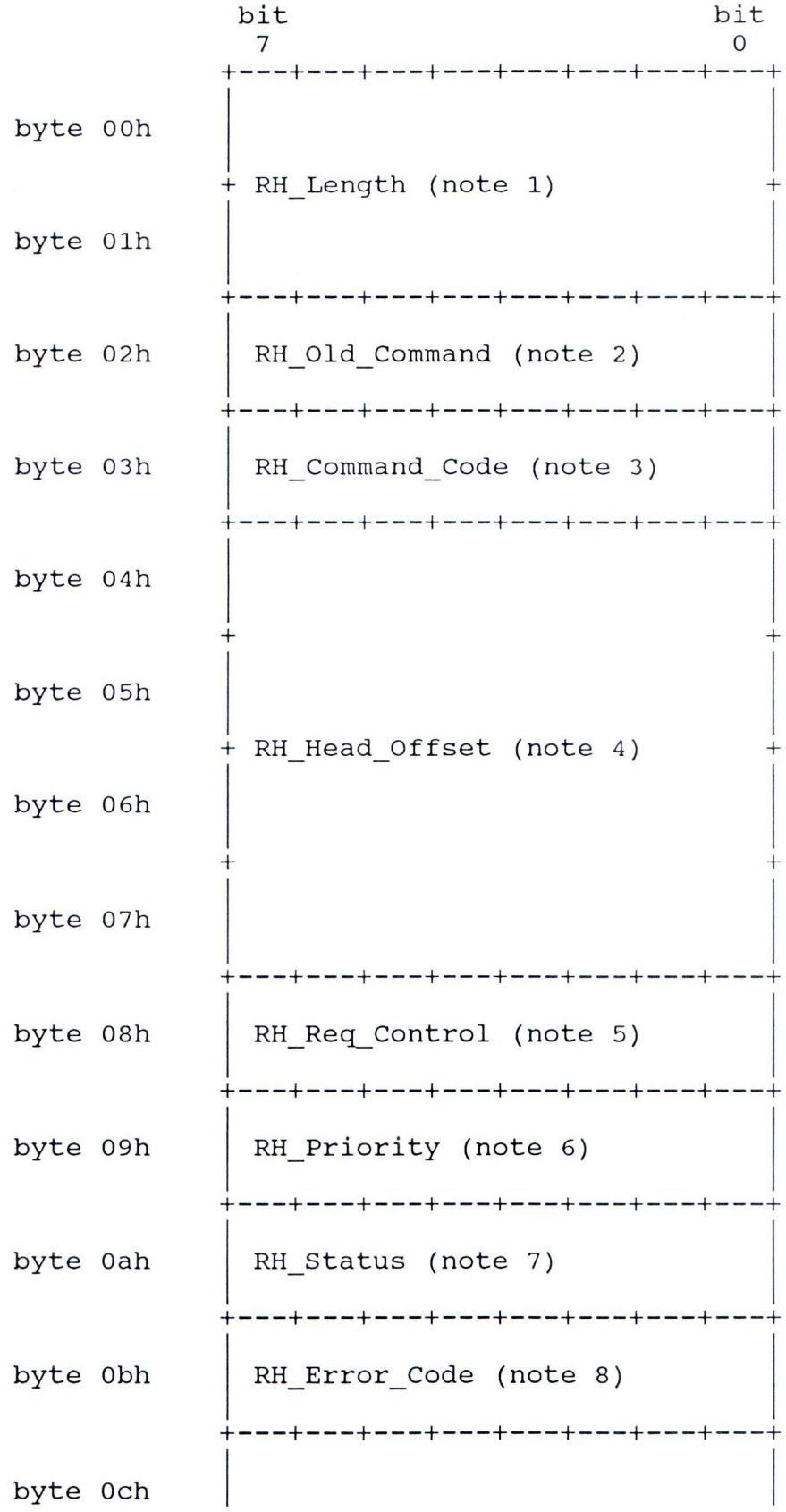
   This field is maintained by the device driver, which initially sets it to zero and then adjusts it as appropriate as packet processing proceeds.

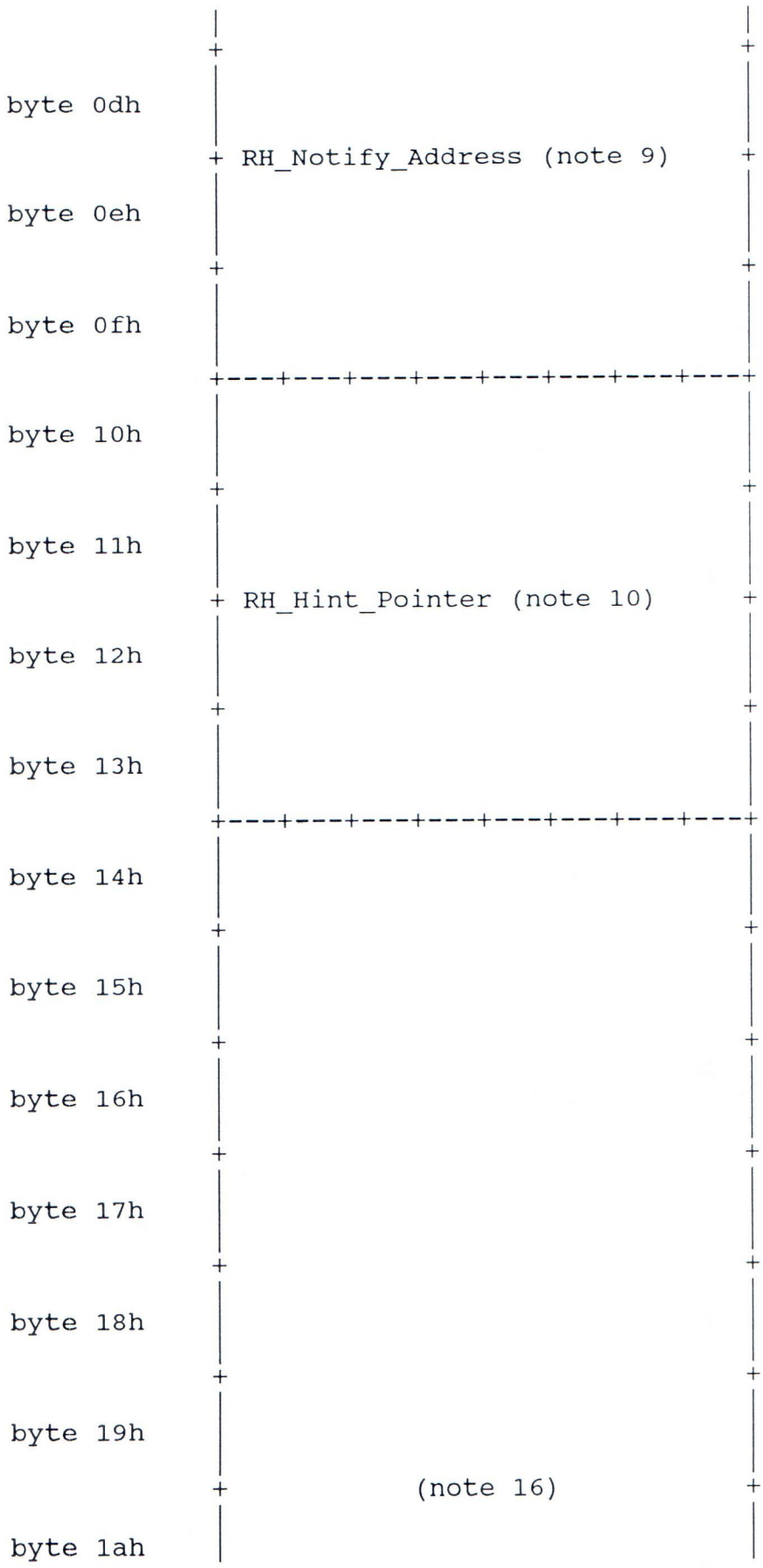
   Note: The values specified above are 4-bit values; not single bit values. Consequently, new values cannot simply be or'd into the field; the value previously stored in the relevant nibble must first be anded out, and then the new value can be or'd in.

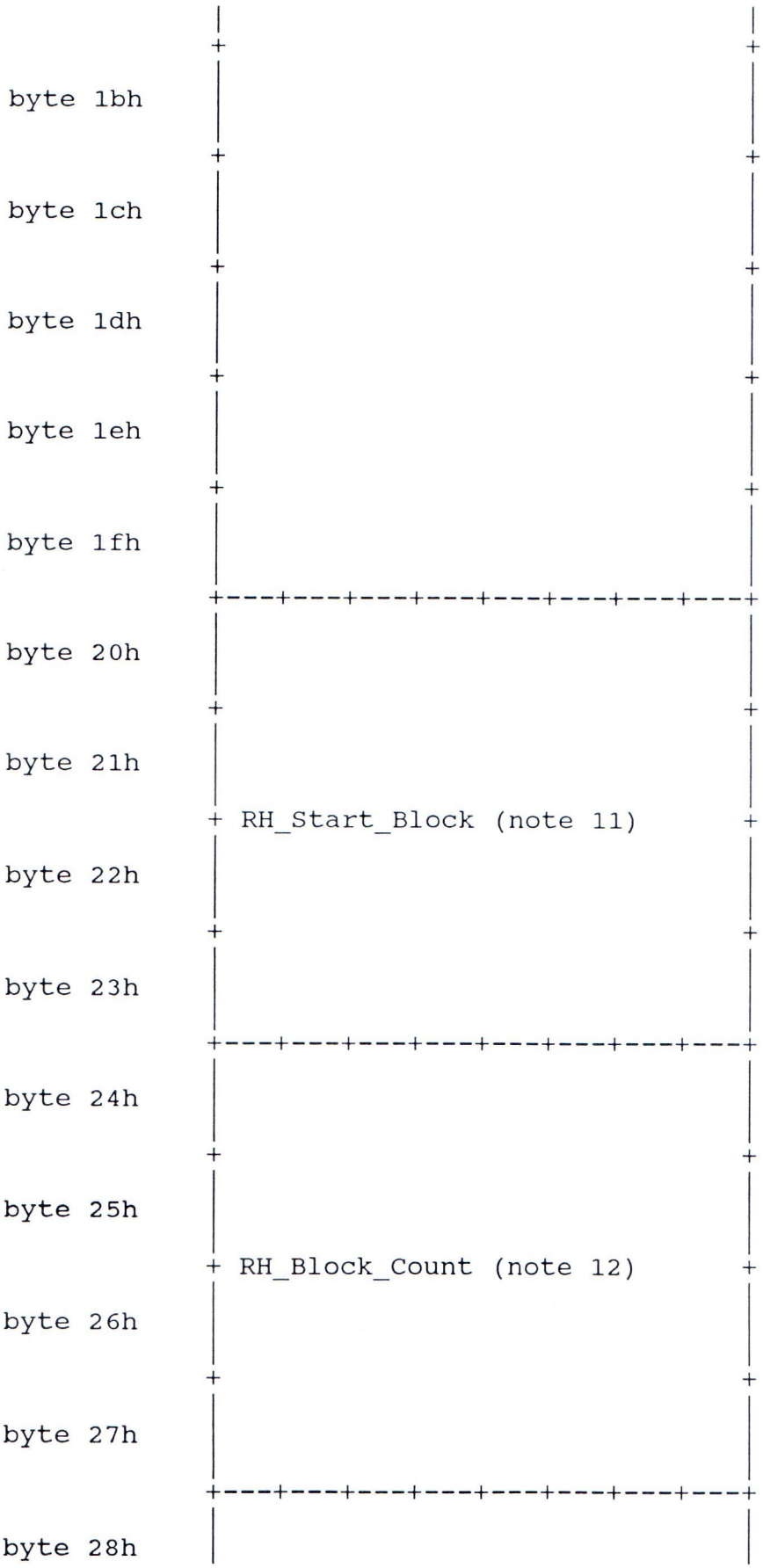
6. **This field is reserved for use by the device driver.** The

requestor should neither depend on the content of this field being preserved, nor try to interpret the content of this field.

The format of the Request Header - the RH - is as follows:




byte 0dh
RH_Notify_Address (note 9)
byte 0eh
byte 0fh
byte 10h
byte 11h
RH_Hint_Pointer (note 10)
byte 12h
byte 13h
byte 14h
byte 15h
byte 16h
byte 17h
byte 18h
byte 19h
(note 16)
byte 1ah


byte 1bh
byte 1ch
byte 1dh
byte 1eh
byte 1fh
byte 20h
byte 21h
RH_Start_Block (note 11)
byte 22h
byte 23h
byte 24h
byte 25h
RH_Block_Count (note 12)
byte 26h
byte 27h
byte 28h

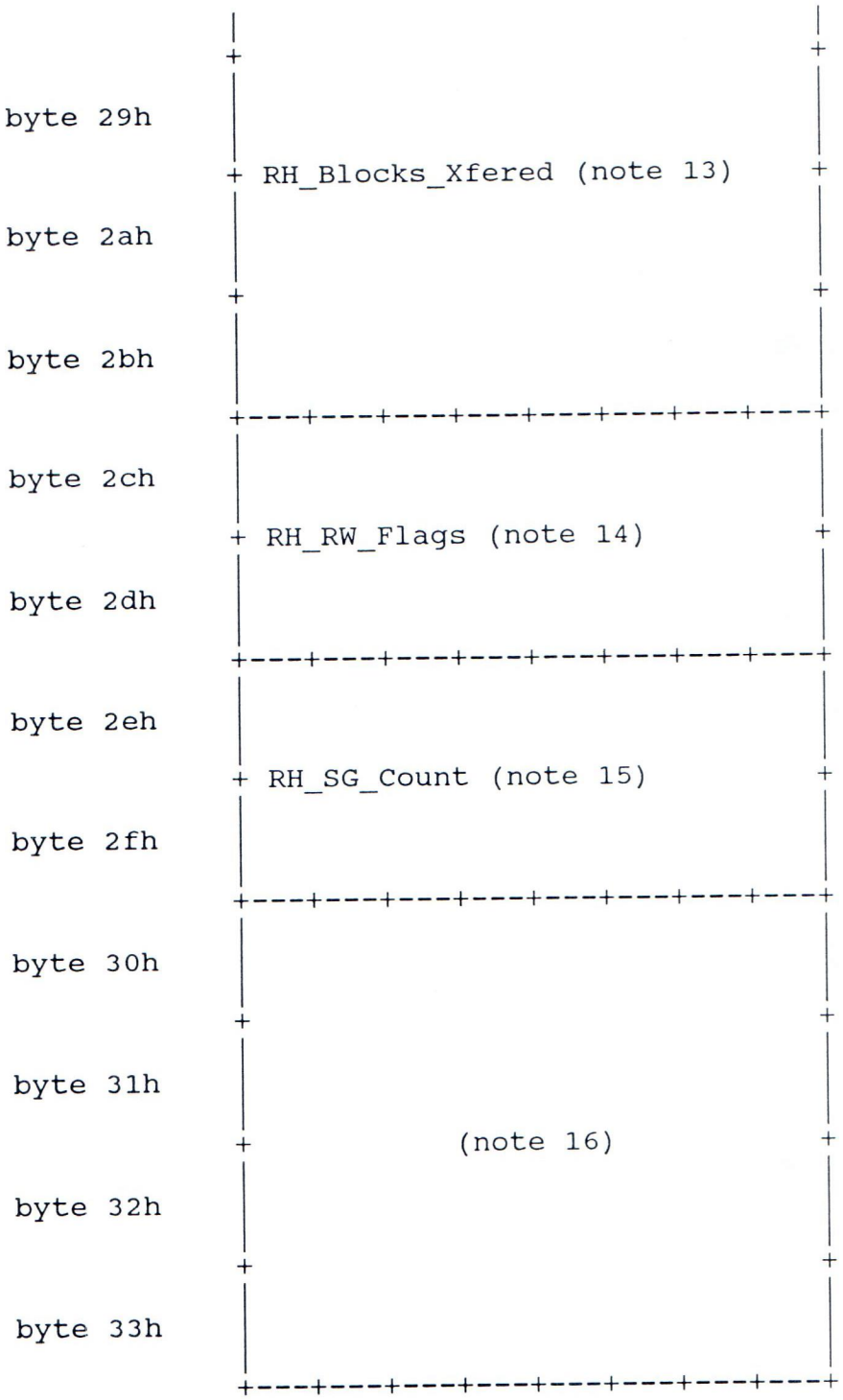


Figure 23. Request Header (RH) Format

The RH is defined in STRAT2.H and in STRAT2.INC.

Notes:

1. RH_Length This field contains a value that either indicates that this is the last RH, or is the delta to the next RH.

When this field contains ffff (hex), this is the last RH.

When this field contains a value other than ffff (hex), the value, when added to the address of this RH, provides the address of the next RH.

Note that this field is unsigned.

2. **RH_Old_Command** This field must contain 1c (hex).

3. **RH_Command_Code** This field contains a value indicating the operation to be performed. The following values are defined:

   RH_READ_X (1e hex) read

   RH_WRITE_X (1f hex) write

   RH_WRITEV_X (20 hex) write/verify

   RH_PREFETCH_X (21 hex) pre-fetch read.

   Note: This operation is not currently supported.

4. **RH_RH_Head_Offset** This field contains an unsigned value, that when subtracted from the address of this RH, generates the address of the RLH.

   Note: This field is unsigned.

5. **RH_Req_Control** This field contains control flags, defined as follows:

   RH_PB_Request (01 hex) request came directly from HPFS

   RH_NOTIFY_ERROR (10 hex) call RH_Notify_Address if this RH completes with an unrecoverable error

   RH_NOTIFY_DONE (20 hex) call RH_Notify_Address if this request completes

   Note: All other bits must be zero.

6. **RH_Priority** This field contains the priority of this RH, defined as one of the following:

   PRIO_PEFETCH (00 hex) prefetch requests

   PRIO_LAZY_WRITE (01 hex) lazy writer generated writes

   PRIO_PAGER_READ_AHEAD (02 hex) pager low

priority read ahead

PRIO_BACKGROUND_USER (04 hex) synchronous I/O from a background user

PRIO_FOREGROUND_USER (08 hex) synchronous I/O from a foreground user

PRIO_PAGER_HIGH (10 hex) pager high priority I/O

PRIO_URGENT (80 hex) extremely high priority, such as cache writebacks triggered by a power loss

7. RH_Status This field contains the current status of this RH, defined as follows:

RH_NOT_QUEUED (x0 hex) not yet queued

RH_QUEUED (x1 hex) queued and waiting

RH_PROCESSING (x2 hex) in process

RH_DONE (x4 hex) done

RH_NO_ERROR (0x hex) no error

RH_RECOV_ERROR (1x hex) recoverable error occured

RH_UNREC_ERROR (2x hex) an unrecoverable error occured

RH_UNREC_ERROR_RETRY (3x hex) "an unrecoverable error with retry"

RH_ABORTED (4x hex) aborted

Note: The values specified above are 4-bit values; not single bit values. Consequently, new values cannot simply be or'd into the field; the value previously stored in the relevant nibble must first be anded out, and then the new value can be or'd in.

8. RH_Error_Code If the RH completes with an unrecoverable error, this field contains an error code, as defined in error.inc

9. RH_Notify_Address This field must either contain zero, or it must contain a 16:16 address of a requestor provided subroutine which is present and locked down.

If bits RH_NOTIFY_ERROR and RH_NOTIFY_DONE are both zero, this field is not used and may contain

either a null or a valid address.

If bit RH_NOTIFY_ERROR is nonzero, this field must contain a non-null valid address that will gain control if this RH completes with an unrecoverable error.

If bit RH_NOTIFY_DONE is nonzero, this field must contain a non-null valid address that will gain control if this RH completes

Note that if both bits RH_NOTIFY_ERROR and RH_NOTIFY_DONE are nonzero and this RH completes with an unrecoverable error, the subroutine pointed to by RH_Notify_Address will gain control twice for this RH.

If both bits RH_NOTIFY_ERROR and RH_NOTIFY_DONE are nonzero, the subroutine pointed to by RH_Notify_Address should differentiate between the two possible cases where it gains control by examining RH_Status. When RH_Status contains RH_DONE, the sub-routine gained control as the result of the request completing; otherwise control was gained as the result of an unrecoverable error.

Once this subroutine has gained control because an RH has completed, it will not be given control again for that same RH. That is to say, if the subroutine is to be given control twice for a particular RH (once because of an unrecoverable error and once because of completion), the first time it gains control will be because of the unrecoverable error and RH_Status will not contain RH_DONE; and the second time will be because of the completion and RH_Status will contain RH_DONE.

If an RH is aborted, it is considered to have neither encountered an error nor to have completed. Consequently, for an RH which has been aborted, the subroutine pointed to by RH_Notify_Address does not gain control, regardless of the setting of RH_NOTIFY_ERROR and RH_NOTIFY_DONE.

Note: In the event that RH_Notify_Address does not contain a valid address and a need exists to call that address, the call -- at the option of the device driver's author -- may or may not actually be attempted. In either case, system failure or a system hang will probably result.

10. RH_Hint_Pointer This field is not currently supported and should be set to zero.

11. **RH_Start_Block** This field contains the logical volume (aka partition) relative address of the first sector to be transferred.

12. **RH_Block_Count** This field contains the number of sectors to be transferred.

13. **RH_Blocks_Xfered** This field is provided for use by requestor error recovery logic to determine how much, if any, data was transferred without error.

    If the transfer completed without encountering an unrecoverable error, the disk driver may set this field to zero, set this field equal to RH_Block_Count, or leave this field unchanged. It is recommended that the disk driver set this field to zero.

    If the transfer did encounter an unrecoverable error, the disk driver may either set this field to zero, or it may set it to a count of how many sectors -- starting from RH_Start_Block -- were transferred without error. It is recommended that the disk driver set this field to zero.

    Requestors may use this field to reduce the amount of data that they must recover by their own means. However, because of the difficulty of maintaining this field accurately, it is recommended that requestors re-process transfers that encounter unrecoverable errors one sector at a time to accurately determine which sectors -- if any -- must be recovered by their own means.

14. **RH_RW_Flags** This field contains flags that control the processing of this RH, defined as follows:

    RH_Cache_WriteThru (01 hex) for writes, this disk driver must complete writing data to the media before it reports this request as complete. For reads, the disk driver should ignore this bit.

    RH_Cache_Req (10 hex) for writes, disk drivers that support caching, should cache this request and should report the request as complete as soon as the data has been transferred to the cache. For writes non-caching disk drivers should ignore this bit; for reads all disk drivers should ignore this bit.

    Note: All other bits must be zero.

15. RH_SG_Count This field contains the number of SG's that follow.

The format of the Scatter/Gather descriptor - the SG - is as follows:

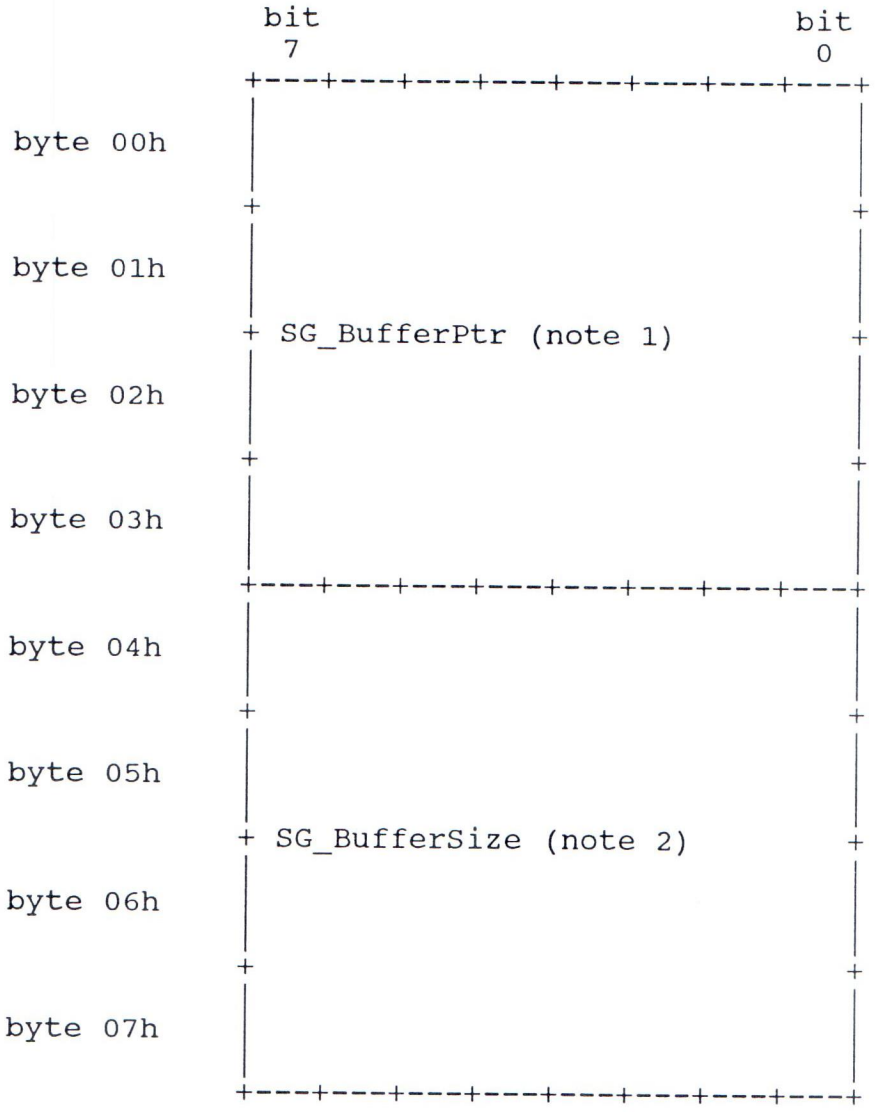


Figure 24. Scatter/Gather Descriptor (SG) Format

The SG is defined in STRAT2.H and in STRAT2.INC, which is built from STRAT2.H.

Notes:

1. SG_BufferPtr This field contains the 32-bit physical memory address of the buffer for the data transfer.

2. SG_BufferSize The field contains the size in bytes of the continuous buffer for the data transfer.

The sum of all SG_BufferSize's for a given RH

should be a precise multiple of the disk's block size.

SCSI Request Block - SRB

SCSI request blocks (SRB) are created by an IOS service routine in response to a request from a driver. They contain pointers, parameters, and control information needed to perform an I/O operation on a SCSI device. An SRB may be shared between IOS and several drivers.

SRB's always reside in IOS's memory pool, are contiguous, and are referenced via an IOS:32 address.

SRB's are declared in SRB.H.

Index

A

B

C

D

Index

F

G

I

Index

Index