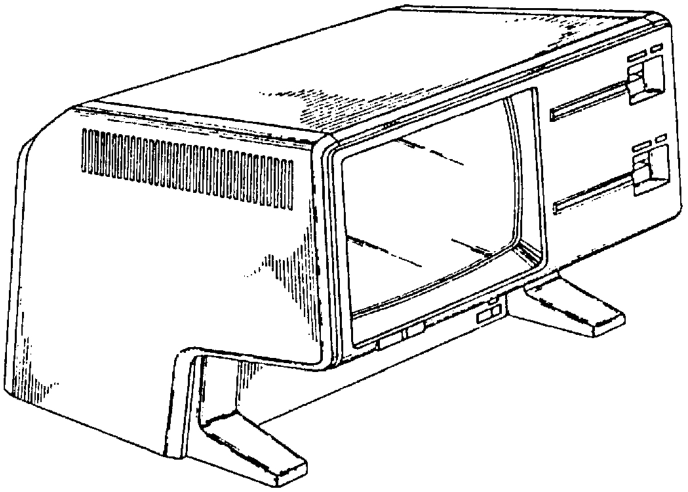
Apple Lisa Computer
Device Drivers Manual

(Alpha Draft)



11 January 1998

Table of Contents

PREFACE

CHAPTER 1
OVERVIEW OF THE DRIVER ENVIRONMENT

1.1 Introduction to Configurable Drivers
1.2 Invoking a Driver through the OS
1.3 Initializing and Loading Configurable Drivers
1.4 Communicating through Request Blocks
1.5 Communication between Drivers

CHAPTER 2
DRIVER CALLS TO THE OS

2.1 Utilities and Standard Data Structures
2.2 Memory Management
    2.2.1 Global Data Space
        2.2.1.1 GETSPACE
        2.2.1.2 RELSPACE
    2.2.2 User Data Buffers
        2.2.2.1 FREEZE_SEG
        2.2.2.2 UNFREEZE_SEG
        2.2.2.3 ADJ_IO_CNT
2.3 Real-Time Management
    2.3.1 The Microsecond Timer
        2.3.1.1 MICROTIMER
    2.3.2 Alarms
        2.3.2.1 ALARM_ASSIGN
        2.3.2.2 ALARMRELATIVE
        2.3.2.3 ALARMOFF
        2.3.2.4 ALARMRETURN
    2.3.3 Enabling and Disabling Interrupts
        2.3.3.1 INTSOFF
        2.3.3.2 INTSON
    2.3.4 Lengthy Interrupt Processing
2.4 I/O Request Block Management
    2.4.1 Request Block Initialization
    2.4.2 Request Block Updating
        2.4.2.1 UNBLK_REQ
    2.4.3 Request Block Queues
        2.4.3.1 ENQUEUE
        2.4.3.2 DEQUEUE
        2.4.3.3 CHAIN_FORWARD
2.5 Disk Driver Support
    2.5.1 USE_HDISK
    2.5.2 CALL_HDISK
    2.5.3 IODONE

        2.5.4 OKXFERNEXT
    2.6 Driver Initiated I/O Requests
        2.6.1 LINK_TO_PCB
        2.6.2 CVT_BUFF_ADDR
        2.6.3 BLK_REQ
        2.6.4 CANCEL_REQ
    2.7 Miscellaneous Subroutines
        2.7.1 SYSTEM_ERROR
        2.7.2 LOGGING
        2.7.3 LOG
        2.7.4 ALLSET
        2.7.5 CALLDRIVER
        2.7.6 DISKSYNC
        2.7.7 KEYPUSHED

CHAPTER 3
DRIVER FUNCTION CODES

- 3.1 Invoking the Driver
- 3.2 Driver Operations
    - 3.2.1 Driver Types and their Function Codes
    - 3.2.2 Error Codes
- 3.3 Function Codes Shared by Different Driver Types
    - 3.3.1 DINTERRUPT
    - 3.3.2 DINIT
    - 3.3.3 DDOWN
    - 3.3.4 DCONTROL
    - 3.3.5 REQRESTART
    - 3.3.6 DALARMS
- 3.4 Function Codes for Sequential Device Drivers
    - 3.4.1 SEQIO
    - 3.4.2 DDISCON
- 3.5 Function Codes for Disk Device Drivers
    - 3.5.1 DSKIO
    - 3.5.2 DSKUNCLAMP
    - 3.5.3 DSKFORMAT0
    - 3.5.4 HDINIT
    - 3.5.5 HDDOWN
    - 3.5.6 HDSKIO
- 3.6 Function Codes for Demultiplexing Drivers
    - 3.6.1 DATTACH
    - 3.6.2 DUNATTACH
- 3.7 Drivers for Bootable Devices

CHAPTER 4
HARDWARE INTERFACE

- 4.1 System Clock Rates
- 4.2 The Serial Communications Controller (SCC)
    - 4.2.1 SCC Timing

    - 4.2.2 Other Information Related to the SCC
        - 4.2.2.1 Cable Connections
        - 4.2.2.2 SCC Register Usage
- 4.3 The 6522 Parallel Port
    - 4.3.1 Using the 6522 Pulse Mode Handshake
    - 4.3.2 Parallel Port Differences
- 4.4 Peripheral Cards
- 4.5 Printers
- 4.6 Built-in Devices

CHAPTER 5
SPECIFYING CHARACTERISTICS FOR A NEW DRIVER

- 5.1 The Installation Disk
- 5.2 The CDCHAR Program
- 5.3 The Preferences Tool

CHAPTER 6
HOW TO MAKE A DEVICE DRIVER

- 6.1 Editing
- 6.2 Compiling and Assembling
    - 6.2.1 Assembler Language Considerations
    - 6.2.2 Pascal Considerations
- 6.3 Linking
- 6.4 Installing a New Driver
    - 6.4.1 Installing the OS
    - 6.4.2 Copying Files
    - 6.4.3 Running the Build Macro
    - 6.4.4 Installing the Driver
- 6.5 Testing and Debugging
    - 6.5.1 TRACE Function
    - 6.5.2 Getting Control during OS Startup

APPENDIX A
SAMPLE DRIVERS,
Listings of Selected I/O Support Subroutines in the OS

- A.1DRIVERDEFS
- A.2DRIVERSUBS
- A.3GENIO
- A.4HDISK
- A.5PROFILE
- A.6RS232

APPENDIX B
UPGRADING DRIVERS WRITTEN TO PREVIOUS SPECIFICATIONS

- B.1 Current Requirements
- B.2 Changes to Driver Data Structures

B.3 Future Hardware Changes

Chapter 1
OVERVIEW OF THE DRIVER ENVIRONMENT

1.1 Introduction to Configurable Drivers
1.2 Invoking a Driver through the OS
1.3 Initializing and Loading Configurable Drivers
1.4 Communicating through Request Blocks
1.5 Communication between Drivers

OVERVIEW OF THE DRIVER ENVIRONMENT

1.1 INTRODUCTION TO CONFIGURABLE DRIVERS

Configurable drivers are drivers that can be incorporated into the Operating System without rebooting it. With this feature, you simply inform the Operating System of the location of the new device, plug it in, and use it. Configurable drivers are compiled and linked as independent programs; once loaded into memory, they function as an integral part of the Operating System. This manual tells you how to write configurable drivers for the Lisa.

Every peripheral device designed for use with the Lisa must be supported by a *device driver* that communicates with the device and with the Operating System (OS). Figure 1-1 shows the Operating System environment for device drivers; the figure also indicates where in this manual or related manuals you can find information about the interfaces between the device, the driver, the Operating System, and the application program.

If the controller for a device can simultaneously support several devices by demultiplexing interrupts, it must have a *demultiplexing driver* that dispatches interrupts to the device drivers. Section 1.5 describes communication between drivers.

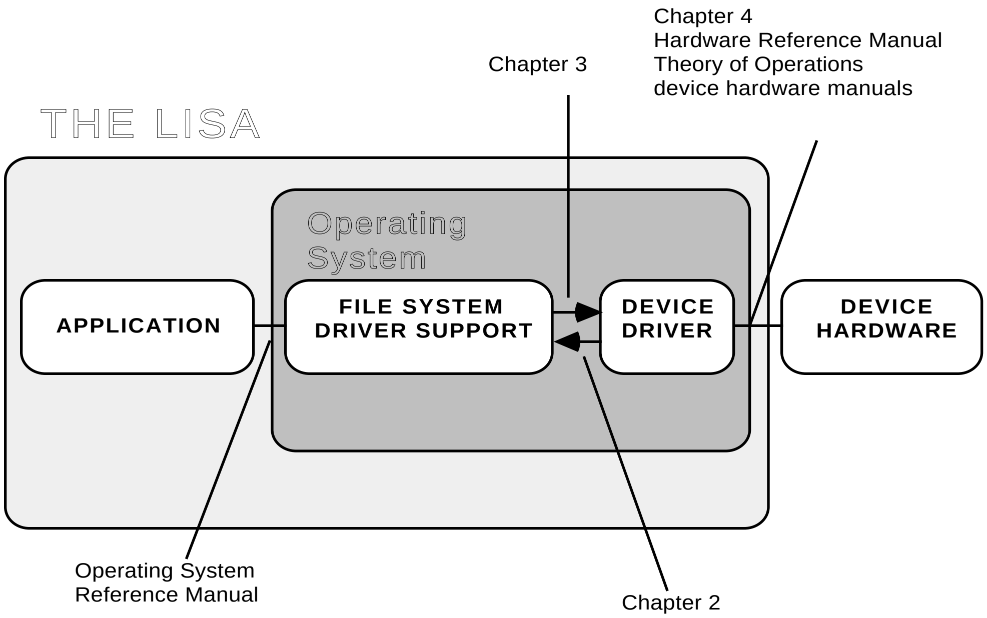

Chapter 4
Hardware Reference Manual
Theory of Operations
device hardware manuals
Chapter 3
THE LISA
Operating System
APPLICATION
FILE SYSTEM DRIVER SUPPORT
DEVICE DRIVER
DEVICE HARDWARE
Operating System Reference Manual
Chapter 2

FIGURE 1-1

Figure 1-1. Operating System Environment for Drivers

Driver operations vary with the type of device being supported. A driver for a sequential device must be able to transmit and receive byte streams of arbitrary length. For a disk device with a directory, the driver must be read and write contiguous 512-byte blocks and manage the 24-byte distributed directory record stored with every disk block. Disk drivers must also handle bad-block sparing. Demultiplexing drivers need to determine which device is the source of an interrupt and pass the interrupt to the appropriate device driver.

Because the Lisa Operating System is a multitasking system, a device may receive I/O requests from more than one process. Therefore its driver must manage I/O Request Block queues and interact with the OS Scheduler and the Memory Manager.

1.2 INVOKING A DRIVER THROUGH THE OS

When an application program requests I/O, a device driver is invoked by the OS, usually from the File System. The File System manages I/O to devices and files by calling upon device drivers and demultiplexing drivers to handle I/O requests for their devices.

A configurable driver is not linked with the OS. It is linked as if it were a standalone, executable program with an internal driver *function subroutine* which is passed a single *parameter record*. The driver function may in turn call other Pascal or assembler subroutines, including OS subroutines named in a **USES** statement, as required. The value returned by the main driver function is the error code returned by the driver. The parameter record passed to the driver contains a function code that specifies the operation to be performed by the driver. A different variant set of parameters is associated with each *function code* so that each driver operation may have its own set of parameters. (For a skeleton driver written in Pascal, see Figure 6-1.)

The Operating System may invoke the driver to handle a hardware interrupt for its device, an interrupt of an alarm that the driver set, or a Memory Management request to restart an operation begun by the driver. In each case, a specific function code is used to inform the driver which operation it must perform. Figure 1-2 shows these operations on the left and their corresponding driver function codes on the right. An application program may also request an I/O operation such as reading, writing, or mounting a device. The OS passes the application's request to the driver by means of the function codes shown in Figure 1-3. (For a description of the function codes, see Table 3-1.)

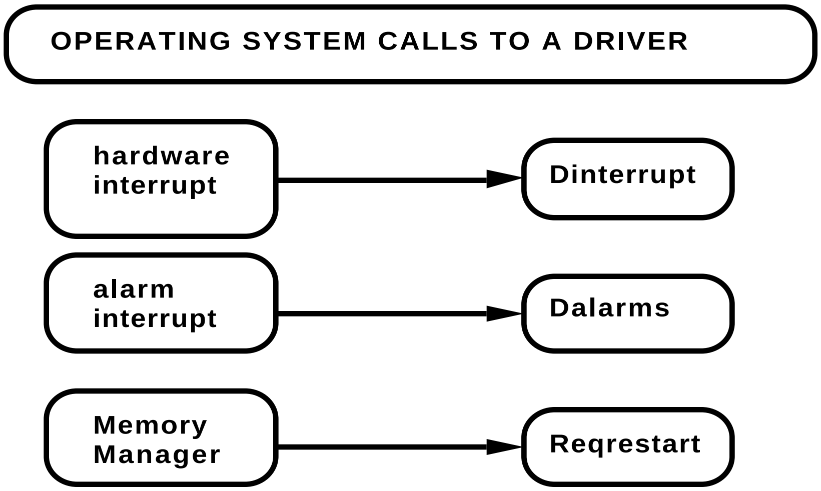


FIGURE 1-2

Figure 1-2. How a Driver is Invoked by the Operating System

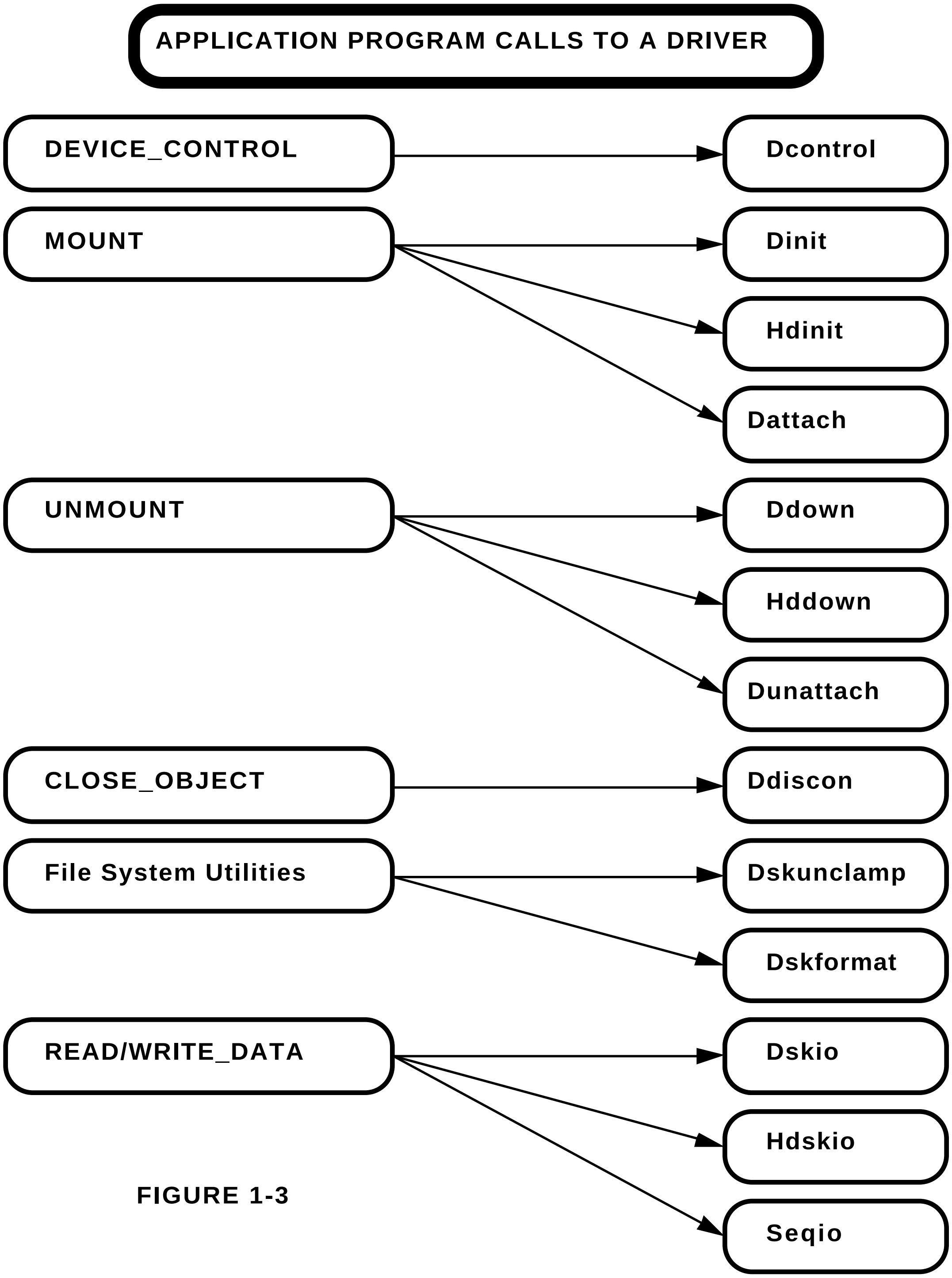

APPLICATION PROGRAM CALLS TO A DRIVER
DEVICE_CONTROL
Dcontrol
MOUNT
Dinit
Hdinit
Dattach
UNMOUNT
Ddown
Hddown
Dunattach
CLOSE_OBJECT
Ddiscon
File System Utilities
Dskunclamp
Dskformat
READ/WRITE_DATA
Dskio
Hdskio
Seqio
FIGURE 1-3

Figure 1-3. How a Driver is Invoked by an Application Program

1.3 INITIALIZING AND LOADING CONFIGURABLE DRIVERS

Configurable drivers are not usually made resident in memory until they are needed by an application. When a device is mounted, its driver is loaded from disk into memory and the driver's data structures are initialized. However, the driver writer specifies whether the driver for a device should be loaded into memory when the OS is booted (see Chapter 5, Specifying Characteristics for a New Driver).

1.4 COMMUNICATIONS THROUGH THE REQUEST BLOCKS

The Operating System creates an *I/O Request Block* to communicate information about each I/O operation. Every Request Block is linked to both the Process Control Block of the requesting process and to the Device Control Block for the device. A single device can therefore be shared among several processes. In a multitasking operating system like the Lisa's, the process that initiates an I/O transfer may not be executing when the I/O transfer actually occurs; the Request Block provides the driver with information about the I/O operation to be performed. The driver may enqueue the request if its device is busy with a previous request; otherwise it initiates the requested operation immediately. The driver then returns to the OS without waiting for the requested operation to complete.

READ_DATA and **WRITE_DATA** (File System procedures described in the *Operating System Reference Manual for the Lisa*) operate synchronously; that is, they do not return control to the application that calls them until all of the data has been read or written. Instead they call **BLK_REQ** to block the current process until their request has been completed. Thus different processes can overlap requests to the same device (because drivers are written to operate asynchronously), but read and write requests from a single process execute strictly in a synchronous fashion. An application consisting of several processes could achieve asynchronicity by allowing processing to continue in one of the non-blocked processes during each I/O transfer requested by another process.

1.5 COMMUNICATION BETWEEN DRIVERS

Demultiplexing drivers serve as interrupt demultiplexors, while device drivers actually perform I/O transfers. A hierarchy of drivers may exist: a demultiplexing driver may call either a device driver or another demultiplexing driver below it in the hierarchy. The lowest level is always a device driver.

Driver hierarchy is illustrated in Figure 1-4, using the Serial Communications Controller as an example. Interrupts are received by the Serial Communications Controller (SCC) driver at the highest level of the hierarchy. The SCC driver determines whether the interrupt is for the Asynchronous RS232 driver on Port A of the SCC or for the Applebus driver on Port B. RS232 is a sequential device; Applebus is a multi-drop, daisy-chained bus capable of controlling several different types of devices -- in this case, a printer and a disk. The Applebus driver determines which device on the bus is interrupting and invoked the driver for that device.

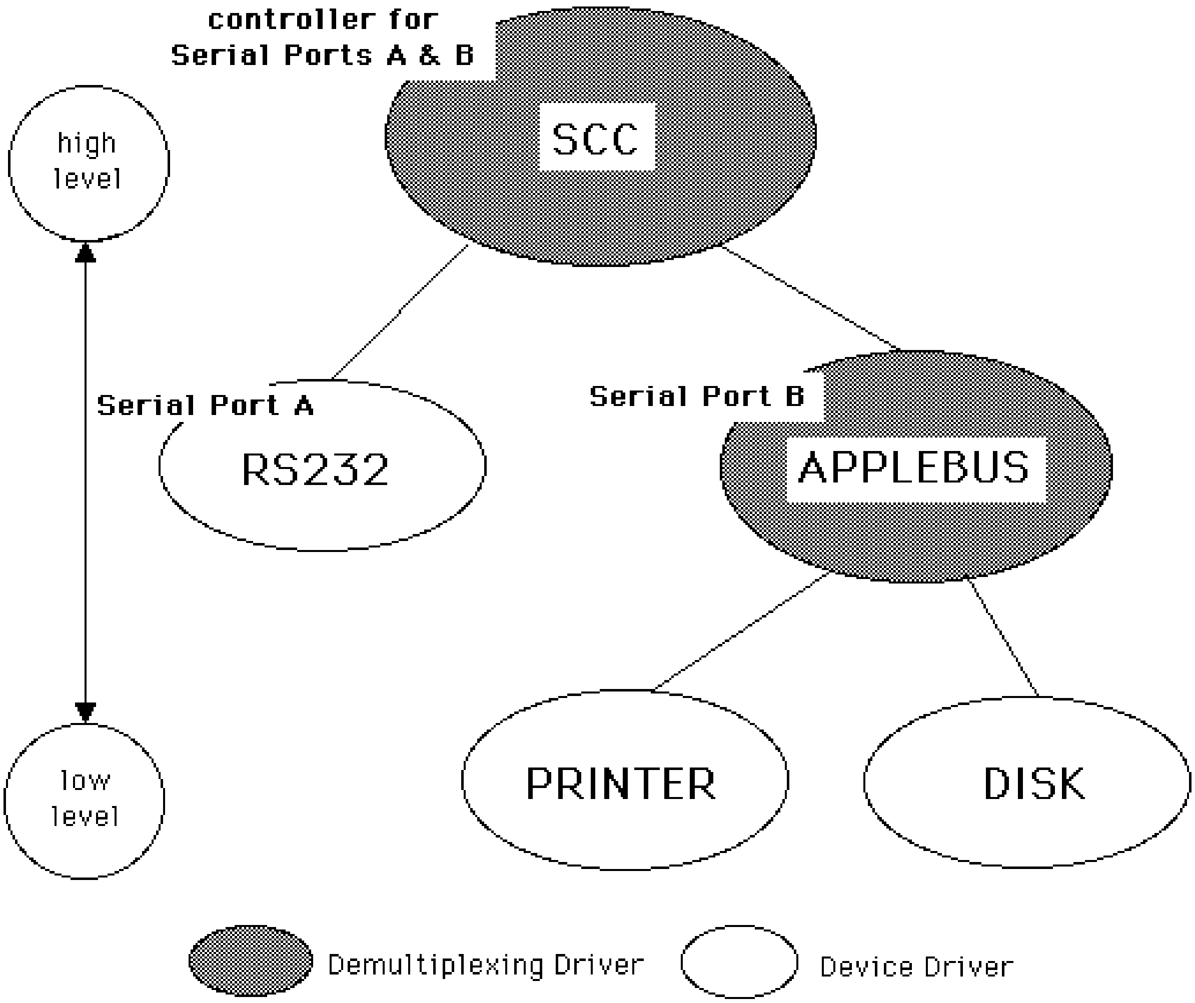

controller for
Serial Ports A & B
SCC
high
level
low
level
Serial Port A
RS232
Serial Port B
APPLEBUS
PRINTER
DISK
Demultiplexing Driver
Device Driver

Figure 1-4. Communication between Drivers.

The low-level drivers in this example are the Asynchronous RS232 driver, the printer driver, and the disk driver. Low-level drivers actually transfer data to or from the device. Higher-level drivers correspond to components in the hardware configuration that multiplex interrupts from several devices. Whenever a device driver is loaded into memory, the OS attaches it to each higher driver associated with it so that the higher driver can keep track of the devices it is responsible for.

Chapter 2
CALLS FROM A DRIVER TO THE OS

2.1 Utilities and Standard Data Structures
2.2 Memory Management
  2.2.1 Global Data Space
    2.2.1.1 GETSPACE
    2.2.1.2 RELSPACE
  2.2.2 User Data Buffers
    2.2.2.1 FREEZE_SEG
    2.2.2.2 UNFREEZE_SEG
    2.2.2.3 ADJ_IO_CNT
2.3 Real-Time Management
  2.3.1 The Microsecond Timer
    2.3.1.1 MICROTIMER
  2.3.2 Alarms
    2.3.2.1 ALARM_ASSIGN
    2.3.2.2 ALARMRELATIVE
    2.3.2.3 ALARMOFF
    2.3.2.4 ALARMRETURN
  2.3.3 Enabling and Disabling Interrupts
    2.3.3.1 INTSOFF
    2.3.3.2 INTSON
  2.3.4 Lengthy Interrupt Processing
2.4 I/O Request Block Management
  2.4.1 Request Block Initialization
  2.4.2 Request Block Updating
    2.4.2.1 UNBLK_REQ
  2.4.3 Request Block Queues
    2.4.3.1 ENQUEUE
    2.4.3.2 DEQUEUE
    2.4.3.3 CHAIN_FORWARD
2.5 Disk Driver Support
  2.5.1 USE_HDISK
  2.5.2 CALL_HDISK
  2.5.3 IODONE
  2.5.4 OKXFERNEXT
2.6 Driver Initiated I/O Requests
  2.6.1 LINK_TO_PCB
  2.6.2 CVT_BUFF_ADDR
  2.6.3 BLK_REQ
  2.6.4 CANCEL_REQ
2.7 Miscellaneous Subroutines
  2.7.1 SYSTEM_ERROR
  2.7.2 LOGGING
  2.7.3 LOG
  2.7.4 ALLSET
  2.7.5 CALLDRIVER
  2.7.6 DISKSYNC
  2.7.7 KEYPUSHED

CALLS FROM A DRIVER TO THE OS

2.1 UTILITIES AND STANDARD DATA STRUCTURES

Driversubs is a Pascal unit that enables drivers to access OS utility routines to perform the operations described in this chapter. These routines provide services such as memory management, real-time management, I/O Request Block management, and disk driver support.

Driverdefs is a Pascal unit that provides definitions of data types for use in drivers. All of the Pascal identifiers used in this manual refer to Request Blocks and other standard driver data structures are defined in **Driverdefs**. Each device in the OS has a global *Device Configuration Record*. The Device Configuration Record (**Devrec**) is defined in the **Driverdefs** unit. Each time a driver is called, the address of the appropriate Device Configuration Record is passed to the driver.

The object code for **Driverdefs** and **Driversubs** was supplied with your Lisa Operating System. For a listing of the source code for these units, see Appendix A. For information on how to make these units a part of your driver, see Chapter 6.

Figures 2-1 through 2-3 show the relationship between the data structures used by a driver. (Figure 2-1 applies to sequential device drivers; Figure 2-2 applies to disk device drivers; Figure 2-3 applies to demultiplexing drivers and non-File System drivers.) All of the figures follow the same format. The center column shows the Device Configuration Record (and, for disk drivers, its extension). This record, one per device, is built by the Operating System during startup. The right-hand column shows the Device Control Block, which is allocated by the driver during its initialization, either at startup or when the device is mounted. You design the format of the Device Control Block to meet the needs of the particular device your driver supports. The left-hand column shows the data structures that are allocated during processing when a request is made for a particular operation. The Parameter Record contains the driver function code (described in Chapter 3) and other information applicable to the specific function to be performed. The name of each data structure is shown in capital letters above its symbol. Below it in parentheses is the name by which the data structure is known in **Driverdefs**. Each black rectangle represents a pointer field; the arrow shows the data structure to which it points. The **Driverdefs** name of the pointer field is shown to the left of the rectangle.

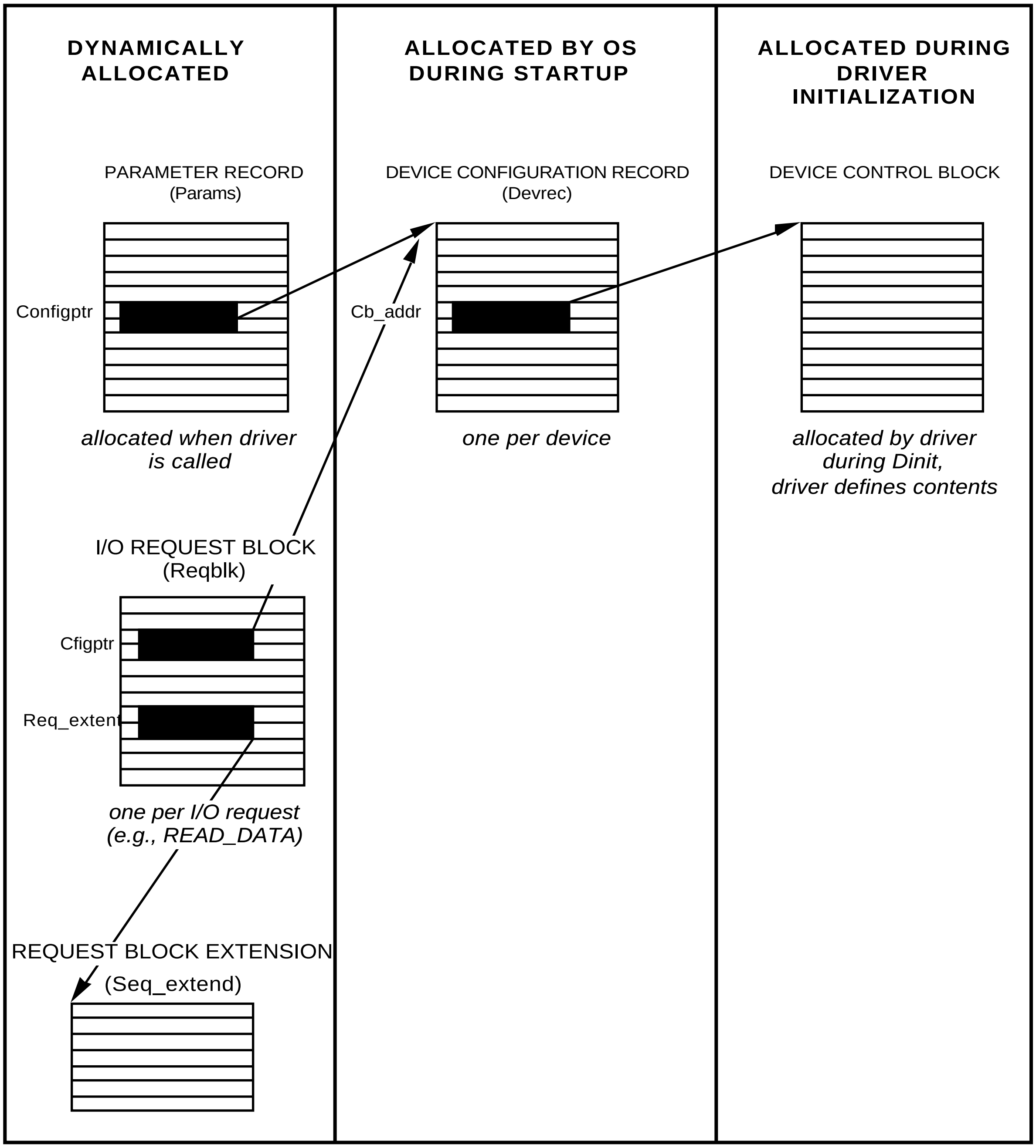
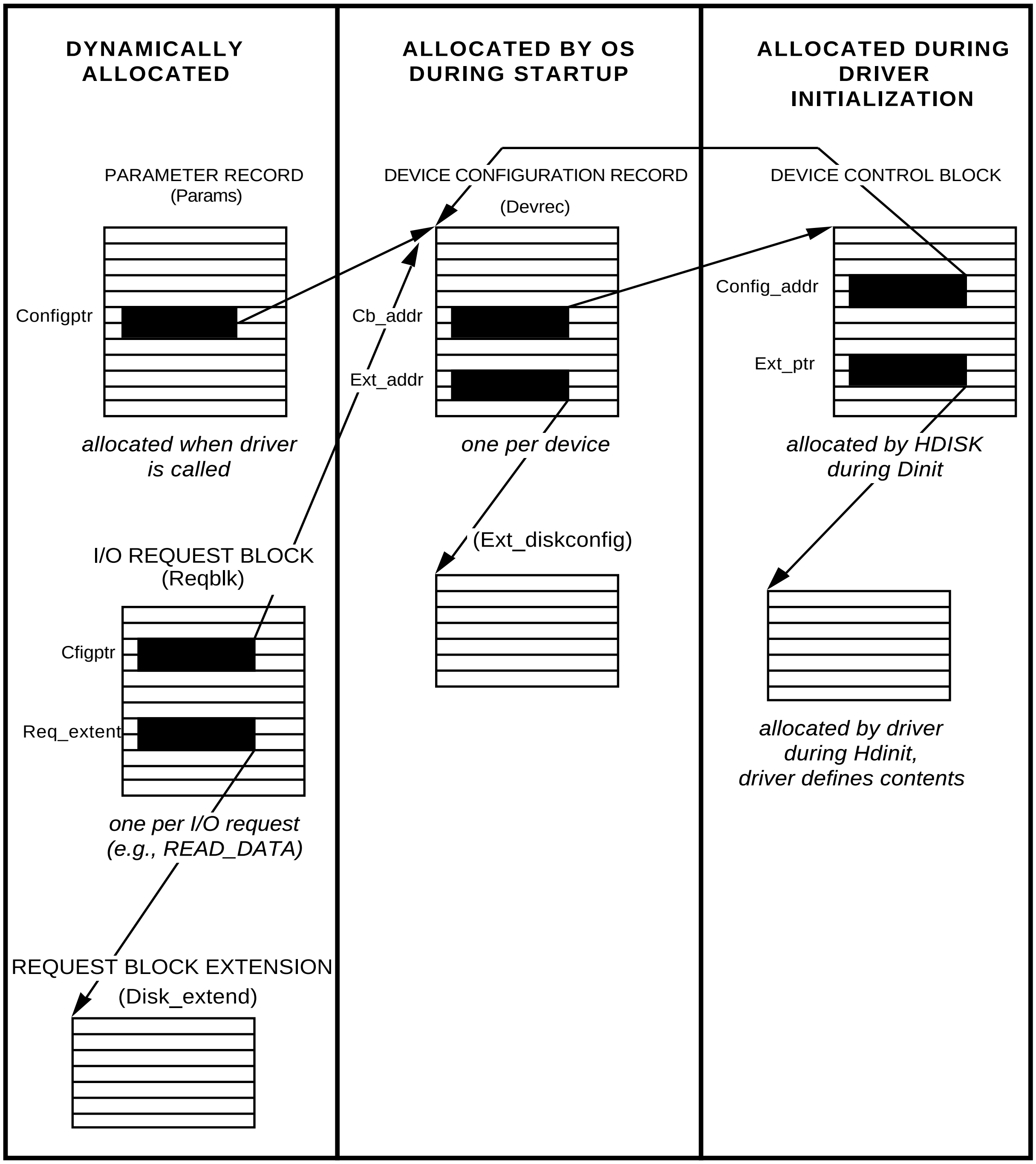

DYNAMICALLY ALLOCATED
ALLOCATED BY OS DURING STARTUP
ALLOCATED DURING DRIVER INITIALIZATION
PARAMETER RECORD (Params)
DEVICE CONFIGURATION RECORD (Devrec)
DEVICE CONTROL BLOCK
Configptr
Cb_addr
allocated when driver is called
one per device
allocated by driver during Dinit, driver defines contents
I/O REQUEST BLOCK (Reqblk)
Cfigptr
Req_extent
one per I/O request (e.g., READ_DATA)
REQUEST BLOCK EXTENSION (Seq_extend)

FIGURE 2-1

Figure 2-1. Data Structures for Sequential Device Drivers.



DYNAMICALLY ALLOCATED
ALLOCATED BY OS DURING STARTUP
ALLOCATED DURING DRIVER INITIALIZATION
PARAMETER RECORD (Params)
DEVICE CONFIGURATION RECORD
(Devrec)
DEVICE CONTROL BLOCK
Configptr
Cb_addr
Ext_addr
Config_addr
Ext_ptr
allocated when driver is called
one per device
allocated by HDISK during Dinit
(Ext_diskconfig)
I/O REQUEST BLOCK (Reqblk)
Cfigptr
Req_extent
allocated by driver during Hdinit, driver defines contents
one per I/O request (e.g., READ_DATA)
REQUEST BLOCK EXTENSION
(Disk_extend)

FIGURE 2-2

Figure 2-2. Data Structures for Disk Device Drivers.

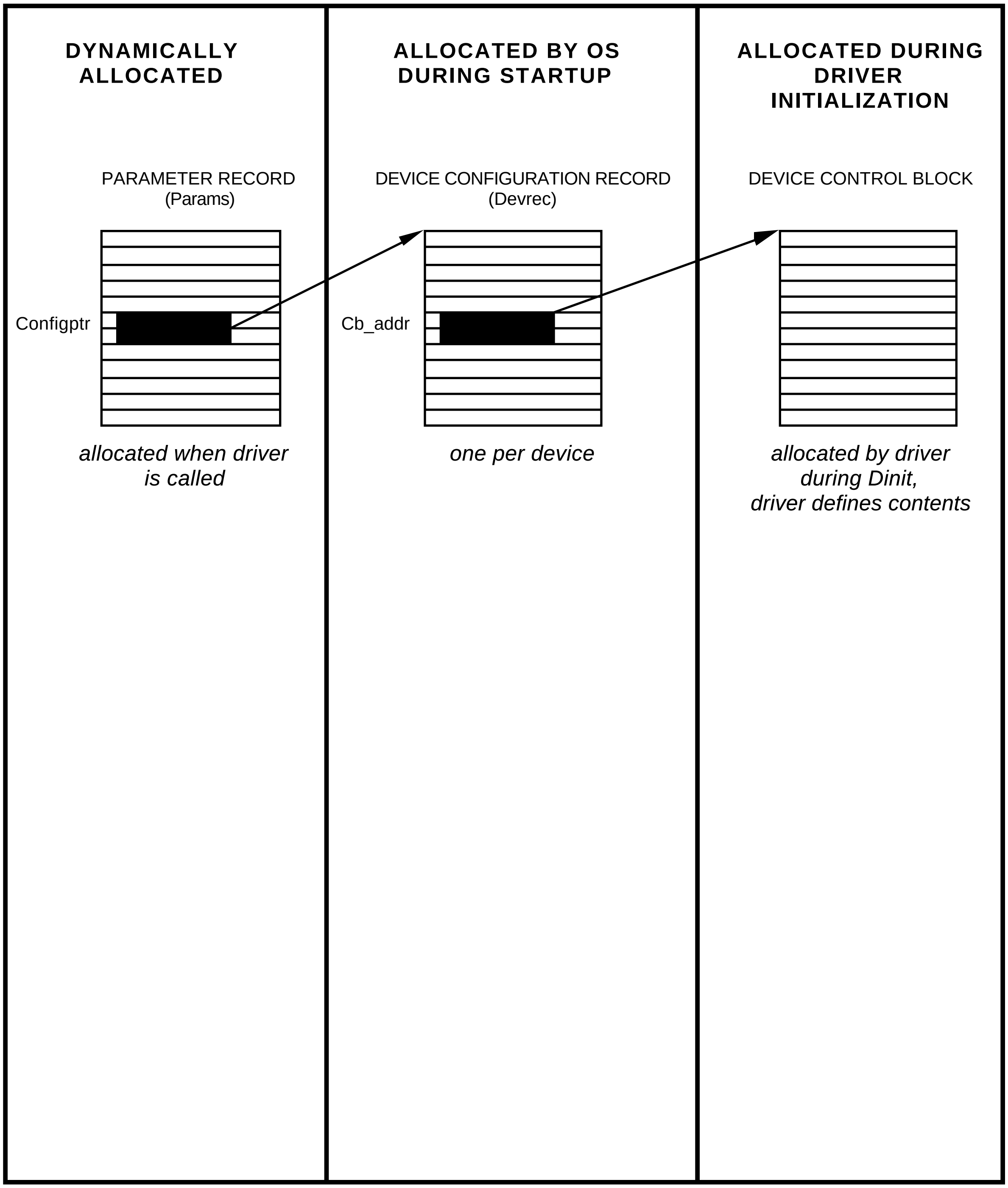

DYNAMICALLY ALLOCATED
ALLOCATED BY OS DURING STARTUP
ALLOCATED DURING DRIVER INITIALIZATION
PARAMETER RECORD (Params)
DEVICE CONFIGURATION RECORD (Devrec)
DEVICE CONTROL BLOCK
Configptr
Cb_addr
allocated when driver is called
one per device
allocated by driver during Dinit, driver defines contents

FIGURE 2-3

Figure 2-3. Data Structures for Demultiplexing and non-File System Drivers.

2.2 MEMORY MANAGEMENT

A driver is not linked with the OS, although it must execute as though it were a part of the OS. Therefore a driver cannot contain any statically allocated global variables. Instead the driver must acquire global data space from the OS Memory Manager to hold any variables it needs to keep track of its device. The Device Configuration Record is acquired by the OS during system startup. However, the Device Control Block for the device must be acquired and filled in by the driver when it is invoked with the Dinit function code.

Memory management support routines are provided to ensure that user data buffers will be accessible by the driver when an interrupt occurs.

2.2.1 Global Data Space

Each device may acquire memory space from the OS Memory Manager. One thousand bytes per device is an approximate upper limit on the amount of global data space a driver may acquire.

GETSPACE and **RELSPACE** are for dynamically acquiring and releasing OS global memory space. Drivers typically use **GETSPACE** during driver initialization (**Dinit** function code) and **RELSPACE** during driver resource deallocation (**Ddown** function code).

2.2.1.1 GETSPACE Function

function GETSPACE (Amount:int2; B_area:absptr; var Ordaddr:absptr): boolean

Input Parameters:
- Amount: Number of bytes of free space required
- B_area: Base address of the free space pool data area.

Output Parameters:
- GETSPACE: True if space was allocated;
  False if sufficient space was not available.
- Ordaddr: Address of the dynamic space allocated.

Dynamic space is allocated from a free space pool. **B_area** must contain a pointer to the free space pool header. The value for **B_area** is provided by the constant **Bsysglob**, defined in **Driverdefs**. To point to the pool, the following lines of code are recommended:

```
var
        ptrsysg: ^absptr;
...
ptrsysg := pointer(Bsysglob);
GETSPACE(Amount, ptrsysg^, Ordaddr);
...
```

Return with *error 610* if **GETSPACE** returns false.

2.2.1.2 RELSPACE Procedure

procedure RELSPACE (Ordaddr: absptr; B_area:absptr)

Input Parameters:
- Ordaddr: Address of the dynamic space allocated.
- B-area: Base address of the free space pool data area.

Ordaddr contains the address that was provided by the **GETSPACE** function. The dynamic area at **Ordaddr** will be released back into the free space pool maintained in the data area defined by the base address **B_area**. (Refer to the sample program fragment in the GETSPACE Procedure, Section 2.2.2.1.)

2.2.2 User Data Buffers

User data buffers are located in data segments managed by the OS Memory Manager. *Memory mapping* is an addressing function provided by the MMU (Memory Management Unit) hardware. The mapped address for a data segment never changes during execution of its process; however, the underlying unmapped address (that is, its real location in memory), changes as the OS Memory Manager manipulates the location of the data segment during memory compaction operations.

After a process requests I/O, its data buffers may not be mapped by the hardware MMU at the time of the interrupt. The routines in this section are provided to guarantee that when an interrupt occurs, the driver can locate the memory address for reading or writing data.

Disk drivers do not call these routines because **HDISK** calls them on behalf of the driver.

FREEZE_SEG prevents the data buffer from being moved by the Memory Manager during memory compaction; **UNFREEZE_SEG** releases the data buffer data to normal memory management. **ADJ_IO_CNT** provides a means of keeping track of the number of pending I/O requests for a given data segment.

2.2.2.1 FREEZE_SEG Procedure

procedure FREEZE_SEG (var Errnum:int2; C_sdb:absptr; Offset:int4; Ioreq_addr:absptr; var Buffaddr:absptr);

Input Parameters:
C_sdb: Data segment to be frozen.
Offset: Supplied by a prior call to CVT_BUFF_ADDR.
Ioreq_addr: Pointer to the Request Block for the data segment.

Output Parameters:
Errnum: 0 = successful freeze;
nonzero = data segment in motion, unable to freeze.
Buffaddr: Offset converted to a valid address.

The **FREEZE_SEG** procedure freezes the data segment in memory. The freeze count of data segment **C_sdb** is increased by one each time **FREEZE_SEG** is called. Once frozen, the data segment cannot be moved by the OS Memory Manager until it is unfrozen. When the data segment is frozen, **Offset** is converted to an address that can be used by the driver.

If the Memory Manager is in the process of moving the data segment at the time **FREEZE_SEG** is called, the freeze does not take place. The **Ioreq_addr** pointer is saved by the Memory Manager so that it can be returned to the driver after the data segment has been moved; the Memory Manager calls your driver with the **Reqrestart** function code after it has finished moving the data segment.

C_sdb and **Offset** must be the **Ordsdb** and **Offset** values returned by the **CVT_BUFF_ADDR** procedure. If the driver was invoked with function code **Dskio**, **Seqio**, or **Hdskio** (that is, by **READ_DATA** or **WRITE_DATA**), **CVT_BUFF_ADDR** has already been called and the values it returned were saved for you in the Request Block variables **Buff_rdb_ptr** and **Buff_offset**. If the driver was invoked with function code **Dcontrol** (that is, by a driver-initiated request for I/O from or to a user data buffer), you must call **CVT_BUFF_ADDR** yourself before calling **FREEZE_SEG**.

2.2.2. UNFREEZE_SEG Procedure

procedure UNFREEZE_SEG (c_sdb:absptr)

Input Parameters:
c_sdb: Pointer to the data segment to be unfrozen.

The freeze count of data segment **C_sdb** is decreased by one each time **UNFREEZE_SEG** is called. When the freeze count reaches zero, the data segment is no longer frozen and may be moved by the Memory Manager.

2.2.2.3 ADJ_IO_CNT Procedure
procedure ADJ_IO_CNT (Inc_io_cntF: boolean; C_sdb:absptr)

Input Parameters:
Inc_io_cntF: True if the count is to be incremented.
False if the count is to be decreased.
C_sdb: Data segment to be adjusted.

ADJ_IO_CNT adjusts the count of pending I/O requests for the data segment upward or downward depending on the value of **Inc_io_cntF**. When the I/O count reaches zero, the Memory Manager is permitted to swap the data segment to disk if necessary.

ADJ_IO_CNT is called by driver support routines in the Operating System to increase the I/O count for a device before **Seqio** or **Dskio** is called. To decrease the count, **ADJ_IO_CNT** must be called by your driver (**HDISK** calls it for disk drivers) at the time the I/O request is completed.

2.3 REAL-TIME MANAGEMENT
This section discusses critical timing considerations, interrupt handling and interrupt masking, a timer for timing driver code, and alarms for lengthy interrupt processing.

2.3.1 The Microsecond Timer
To help you determine how much time your driver spends in critical sections of code, a *microsecond timer* is available. The microsecond timer simulates a continuously running 32-bit counter that is incremented every microsecond. The timer changes sign about once every 35 minutes, and rolls over every 70 minutes. It is initialized to zero when the Lisa is booted.

2.3.1.1 MICROTIMER Function
function MICROTIMER: longint

Output Parameters:
MICROTIMER: Current value of the microsecond timer.

The microsecond timer is designed for performance measurements. It has a resolution of 2 microseconds. Calling **MICROTIMER** from Pascal takes about 135 microseconds. Note that interrupt processing will have a major effect on microsecond timings if **MICROTIMER** is called from within a routine that has interrupts enabled.

2.3.2 ALARMS
Twenty independent *alarms* are shared by all drivers. Alarms can be dynamically acquired by drivers and should be returned when no longer needed. For line protocol timeouts, an alarm can be set to "wake up" the driver after a specified interval of time. The alarms have a 10-millisecond resolution; that is, if you set an alarm to go off with a delay of 500 milliseconds, the actual delay will be between 490 and 510 milliseconds.

The routines associated with alarms are

Get an alarm -- **ALARM_ASSIGN**
Set an alarm -- **ALARMRELATIVE**
Turn off an alarm -- **ALARMOFF**
Return an alarm -- **ALARMRETURN**

After obtaining an alarm by calling **ALARM_ASSIGN**, the driver sets the alarm to a specified number of milliseconds in the future. When the millisecond timer reaches the alarm's setting, the driver's alarm handler is called by means of the **Dalarms** function code.

Interrupts at the priority levels specified by **ALARM_ASSIGN** are disabled during execution of the alarm handler.

2.3.2.1 ALARM_ASSIGN Procedure

procedure ALARM_ASSIGN (var Alarm:integer; Pdr:Ptrdevrec; Status:intson_type)

Input Parameters:
Pdr: Pointer to the Device Configuration Record associated with this alarm.
Status: Level of interrupts enabled in alarm handler.

Output Parameters:
Alarm: Number of the assigned alarm; zero if none available.

ALARM_ASSIGN returns an alarm number in **Alarm**. If an alarm is not available, **Alarm** is zero; your driver should then return with error 602.

The driver should associate the alarm with the device specified in **Pdr** by saving the alarm number returned in **Alarm** in the Device Control Block.

Interrupts from your device should be disabled while your alarm handler is executing in order to prevent data in the Device Control Block from being inconsistently modified. Status is the interrupt priority you want to be in effect when the alarm goes off. You should set **Status** to the lowest value that will disable interrupts for your device. (See the **INTSOFF** procedure, Section 2.3.3.1, for more information on possible values of **Status**.) When the alarm goes off, the driver is called using the **Dalarms** function code with the interrupt priority set to **Status**.

2.3.2.2 ALARMRELATIVE Procedure

procedure ALARMRELATIVE (Alarm:integer; Delay:longint)

Input Parameters:
Alarm: Number of the alarm to be set.
Delay: Milliseconds to wait before the alarm goes off.

ALARMRELATIVE sets an alarm to go off **Delay** milliseconds in the future. When **Delay** milliseconds have elapsed, the driver is called with the **Dalarms** function code.

If **Delay** is zero or negative, the alarm goes off immediately (at the next system alarm interrupt).

Each alarm remembers only one setting. To reset an alarm call **ALARMRELATIVE** again; this causes the previous setting to be disregarded.

2.3.2.3 ALARMOFF Procedure

procedure ALARMOFF (Alarm:integer)

Input Parameters:
Alarm: Number of the alarm to be turned off.

ALARMOFF turns off an alarm that was previously set by **ALARMRELATIVE**. That is, **ALARMOFF** can be used to cancel a request for an alarm interrupt.

2.3.2.4 ALARMRETURN Procedure

procedure AlarmReturn (Alarm:integer)

Input Parameters:
Alarm: Number of the alarm to be returned to the pool.

The **ALARMRETURN** procedure returns an alarm acquired by **ALARM_ASSIGN** to the pool of available alarms; another driver can then use it.

Alarms should be returned when they are no longer needed by your driver. This happens when the driver is called with the **Ddown** function code.

2.3.3 Enabling and Disabling Interrupts

Device Control Block variables may be in use by two or more driver functions at a time, potentially leaving the Device Control Block in an inconsistent state. To prevent this, interruptible driver functions must disable interrupts for a short period of time while updating the Device Control Block. The **INTSOFF** procedure allows you to disable some or all interrupts; the **INTSON** procedure re-enables them.

The interrupt priority mask determines which interrupts will be acknowledged by the processor. Any interrupt with a priority higher than the mask is acknowledged; any interrupt with a priority less than or equal to the mask is inhibited. (for a more detailed description of the interrupt priority mask, see the MC68000 user's manual.)

On the Lisa, the interrupt priorities are as shown in Table 2-1.

Table 2-1
Hardware Interrupt Priorities

Priority	Hardware
1	system clock, built-in floppy disk, built-in parallel port
2	keyboard, mouse, time-of-day clock/calendar
3	slot 3
4	slot 2
5	slot 1
6	built-in serial ports A and B
7	memory parity error, programmable NMI (nonmaskable interrupt) key

2.3.3.1 INTSOFF Procedure

procedure INTSOFF (Level:intsoff_type, var Status:intson_type)

Input Parameters:
Level: New interrupt priority

Output Parameters:
Status: Old interrupt priority

INTSOFF allows you to temporarily turn off some or all interrupts by specifying a new value in **Level**. Values for **Level** are

clokints, winints, or twigints (priority = 1)

slotints	(priority = 5)
rsints	(priority = 6)
allints	(priority = 7)

Before **INTSOFF** changes the interrupt priority, it saves the current priority; this is returned to you in **Status** and should be used in the **INTSON** procedure to reset the priority level to its original value.

2.3.1 INTSON Procedure

procedure INTSON (Status: intson_type)

Input Parameters:
Status: Interrupt level to be restored.

The **INTSON** procedure enables interrupts that were disabled by a previous call to **INTSOFF**. Status contains the original interrupt priority level in effect at the time **INTSOFF** was called.

2.3.4 Lengthy Interrupt Processing

An interrupt that takes more than a few milliseconds should process with all other interrupts enabled. The interrupt handler for you device can accomplish this by calling an alarm routine to perform the extended interrupt processing. The alarm mechanism is merely a convenient vehicle for scheduling a lengthy interrupt that can execute with interrupts enabled for all devices.

When your interrupt handler determines that extended processing is required, it call **ALARM_ASSIGN** to associate an alarm with the lengthy interrupt routine, specifying an interrupt level of **clkonints** to enable all interrupts during execution of the routine. The interrupt handler then calls **ALARMRELATIVE** with zero millisecond delay, so that the alarm routine will begin execution as soon as possible. The actual delay before starting will always be less than twenty milliseconds and will average about five milliseconds.

In the 68000 architecture, a device's interrupt handler (its **Dinterrupt** routine) executes with interrupts disabled for all devices of same or lower interrupt priority. The body of the lengthy-interrupt alarm routine (**Dalarms**) will execute with all interrupts enabled. Therefore it is necessary to protect the Device Control Block from simultaneous access when the same device tries to interrupt during the extended processing of a prior interrupt.

Be sending an appropriate signal to the device, the device's interrupt handler can disable (and hold pending) any further interrupts for the interrupting device until the alarm routine has completed its processing. For example, if there is an interrupt from the 6522 parallel port that requires extended processing, the interrupt handler can set a bit in the 6522's Interrupt Enable Register (IER) to zero. This prevents further interrupts from the device. When the alarm routine completes its extended processing alarm routine, the IER must be restored to the required non-zero value so that interrupts may occur again. For the serial ports, Write-Register-1 on the SCC serves the same function as the IER on the 6522. The disadvantage of this technique is that it increases the interrupt response latency on the device requiring extended processing. Depending upon the device, this may or may not be a problem.

Some devices require very short interrupt response latency (less than 70 microseconds) or long interrupt execution times (more than 10 milliseconds). Such devices (for example, laser printers and some serial communications protocols at high baud rates) require all of the processing power of the 68000 for extended periods of time. Such devices are not really suitable for use in the multi-tasking Lisa environment. Unless these devices are provided with intelligent controllers that don't force an interrupt on the transfer of each byte, the Lisa is unavailable for any other tasks while it is supporting these devices.

2.4 I/O REQUEST BLOCK MANAGEMENT

As part of its I/O operations, your driver uses and updates fields within a Request Block. When it is passed to the driver, the Request Block (and, in some cases, its extension) already contains information about the I/O request. The driver is responsible for updating the transfer count and other fields in the Request Block.

2.4.1 Request Block Initialization

When a driver is invoked with function code of **Seqio** or **Dskio**, a pre-initialized Request Block is passed to the driver. All the information needed to initialize the I/O request is passed to the driver in the Request Block. The information in the Request Block is initialized as follows:

pcb_chain: doubly-linked list to Process Control Block (PCB)

reqstatus
  reqsrvv_f: active
  reqabt_f: false

block_p_f: false

blk_in_pcb: [i_o]

hard_error: 0

operatn: 0 for write; 1 for read

cfigptr: pointer to device's configuration table entry

req_extent: address of Request Block extension

All Request Block extensions are initialized as follows:

read_flag: true if read, false if write

xfer_count: 0

buff_rdb_ptr and
buff_offset: result of calling CVT_BUFF_ADDR on data buffer

For extensions used by Seqio only, the following field is initialized:

num_bytes: desired length of transfer

For extensions used by Dskio only, the following fields are initialized:

blkno: block number to begin read or write

soft_headr: initial value for 24-byte file system "pagelabel"

last_fwd_link: fwd_link value in pagelabel of final block

last_data_used: data-used value in pagelabel of final block num_chunks: number of blocks (512 bytes, plus header) to transfer

io_mode: with_header...write using headers from soft_headr

without_header...doesn't transfer headers

raw_io...memory filled with all the headers, followed by all the data

chained_hdrs...read using headers and verify against
contents of soft_headr

2.4.2 Request Block Updating

When the driver starts up an I/O request, the following field should be changed in the Request Block:

reqstatus
reqsrv_f: in_service

When a request for I/O has been completed or an error has occurred, your driver must update the Request Block and then call UNBLK_REQ to notify the OS that your driver is finished with the request. The error field in the Request Block must be set:

hard_error: =0 for no errors, <0 for warnings, >0 for errors
(warnings and errors, pass False to UNBLK_REQ)

The driver must update the transfer count field in the Request Block extension:
xfer_count: number of bytes transferred (not including bytes in disk block headers)

For Dskio only, the driver must update the following field in the Request Block extension:

soft_headr: the last header read when using "chained_hdrs"
io_mode

2.4.2.1 UNBLK_REQ

procedure UNBLK_REQ (Reqptr:reqptr_type; Success_f:boolean)

Input Parameters:
Reqptr:Pointer to the request that is completed.
Success_f: True if the request was successfully completed;
False if Hard_error contains non-zero value.

UNBLK_REQ unblocks the process that is waiting for the request specified by **Reqptr**. **Success_f** is a flag that informs the process of the result of the I/O request.

For disk drivers the **HDISK** unit calls **UNBLK_REQ** on behalf of the driver at the end of a read or write. Other drivers must call **UNBLK_REQ** themselves.

2.4.3 Request Block Queues

A queue is a linked list, or chain. Two types of linked lists are used for Request Blocks: the first is a chain of all current Request Blocks for a particular process; the second is a chain of all Request Blocks waiting to perform I/O on a single peripheral device. Figure 2-1 illustrates the two types of Request Block queues.

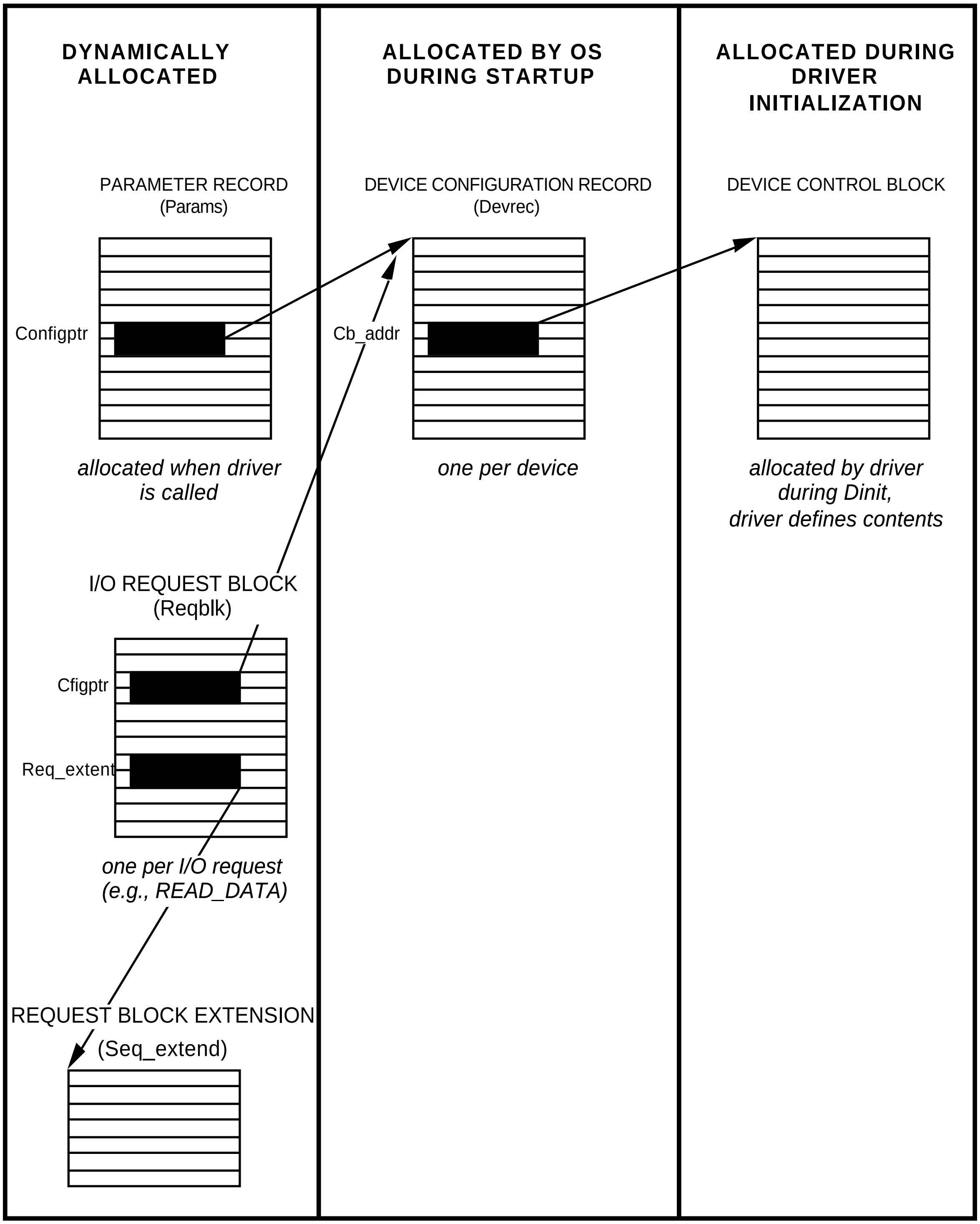

DYNAMICALLY ALLOCATED
ALLOCATED BY OS DURING STARTUP
ALLOCATED DURING DRIVER INITIALIZATION
PARAMETER RECORD (Params)
DEVICE CONFIGURATION RECORD (Devrec)
DEVICE CONTROL BLOCK
Configptr
Cb_addr
allocated when driver is called
one per device
allocated by driver during Dinit, driver defines contents
I/O REQUEST BLOCK (Reqblk)
Cfigptr
Req_extent
one per I/O request (e.g., READ_DATA)
REQUEST BLOCK EXTENSION (Seq_extend)

FIGURE 2-1

Figure 2-1

The two types of Request Block queues are circular (the end of the queue points back to the beginning) and doubly-linked (each Request Block points backward to the previous Request Block in the chain and forward to the next one). Figure 2-2 shows how a linkage record is inserted in a circular queue. The forward and backward chaining of Request Blocks is accomplished by means of the linkage data type. Linkages are not full addresses but rather offsets within the global data area.

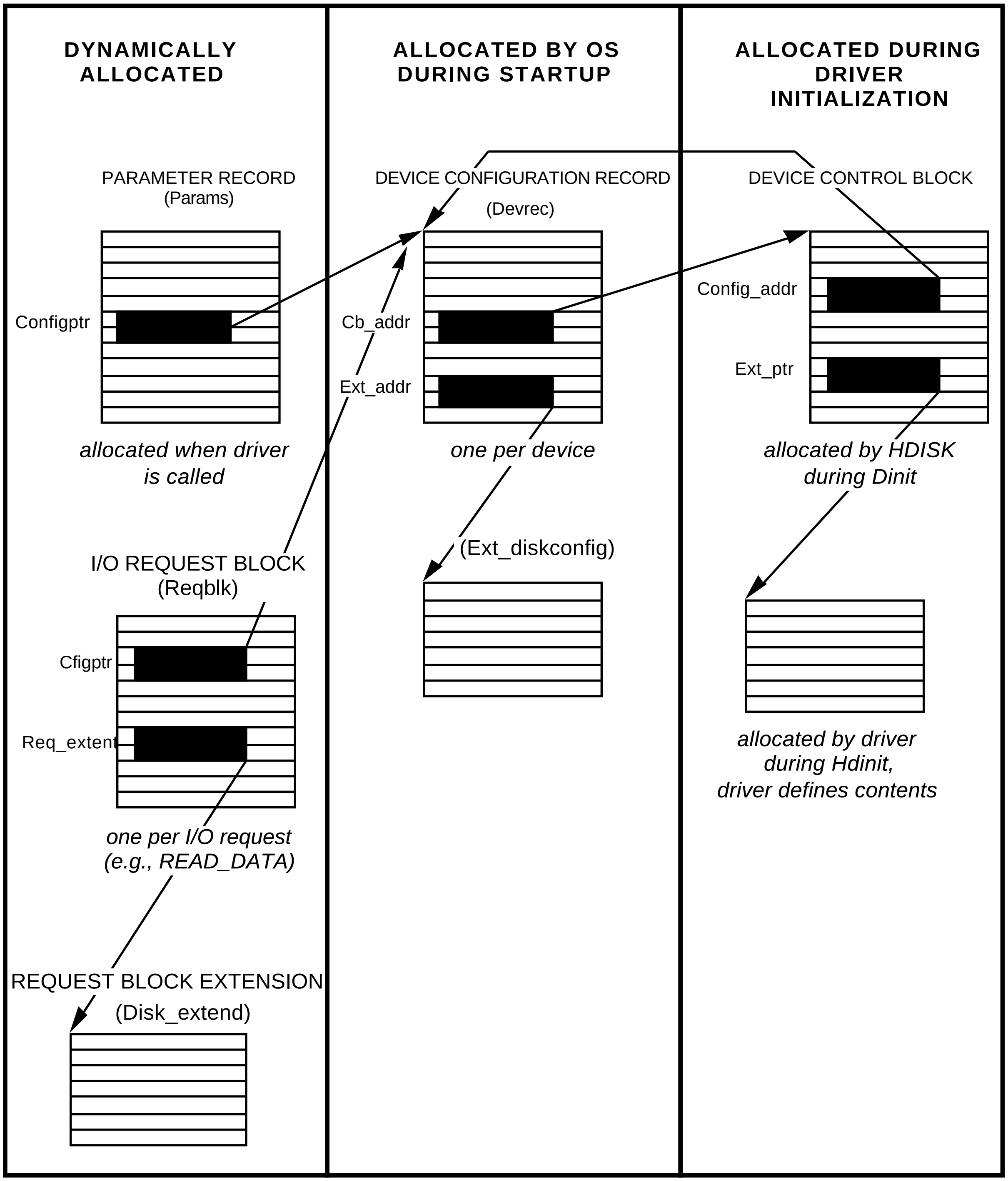

DYNAMICALLY ALLOCATED
ALLOCATED BY OS DURING STARTUP
ALLOCATED DURING DRIVER INITIALIZATION
PARAMETER RECORD (Params)
Configptr
allocated when driver is called
DEVICE CONFIGURATION RECORD
(Devrec)
Cb_addr
Ext_addr
one per device
DEVICE CONTROL BLOCK
Config_addr
Ext_ptr
allocated by HDISK during Dinit
I/O REQUEST BLOCK (Reqblk)
Cfigptr
Req_extent
one per I/O request (e.g., READ_DATA)
(Ext_diskconfig)
allocated by driver during Hdinit, driver defines contents
REQUEST BLOCK EXTENSION
(Disk_extend)

FIGURE 2-2

Figure 2-2

Request Block queues for disk **READ_DATA** and **WRITE_DATA** requests are handled by **HDISK**, a unit in the

Operating System. For all other kinds of I/O, your driver must maintain these queues. Three procedures and functions are available to help you:

- o **ENQUEUE** inserts linkages in a Request Block queue.
- o **DEQUEUE** removes linkages from a queue.
- o **CHAIN_FORWARD** points to the next Request Block in the queue.

2.4.3.1 ENQUEUE Procedure

procedure ENQUEUE (var Newlink, Leftlink:linkage; B_sysarea:absptr)

Input Parameters:
Newlink: Linkage record of the new control block.
Leftlink: Linkage record of previous block in chain.
B_sysarea: Address ordinal of the data area base.

ENQUEUE places **Newlink** next in a chain headed by **Leftlink**. **Newlink** is the linkage record of the Request Block to be added to the chain. **Leftlink** is the linkage record of the Request Block in the chain after which **Newlink** is to be added. The linkage values are relative pointers to the linkage record. **ENQUEUE** assumes that the queue has been initialized as a circular chain and that the caller has exclusive access while the chain is being modified.

B_sysarea must contain the address of the free space pool header; refer to the **GETSPACE** Function, Section 2.2.2.1, for more information.

2.4.3.2 DEQUEUE Procedure

procedure DEQUEUE (var Link:linkage; B_sysarea:absptr)

Input Parameters:
Link: Linkage record to be removed from the queue.
B_sysarea: Address ordinal of the data area base.

DEQUEUE removes a Request Block from a queue. **DEQUEUE** assumes that the queue has been initialized as a circular chain and that the caller has exclusive access while the chain is being modified.

B_sysarea must contain the address of the free space pool header; refer to the **GETSPACE** Function, Section 2.2.2.1, for more information.

2.4.3.3 CHAIN_FORWARD Function

function CHAIN_FORWARD (Current:linkage):reqptr_type

Input Parameters:
Current: Linkage record of old Request Block.

Output Parameters:
CHAIN_FORWARD: Pointer to new Request Block.

CHAIN_FORWARD uses the forward link of the linkage record to point to the next Request Block in the queue so that your driver can initiate the next I/O request for the device.

For non-disk drivers, the following steps should be performed when the current I/O request is complete:

1. Call **UNBLK_REQ** for the request you just completed.

2. Call **CHAIN_FORWARD** to get the next Request Block in the queue linked to the Device Control Block.
3. Start the next I/O request.

For disk drivers, call **IODONE** to perform these steps.

2.5 DISK DRIVER SUPPORT

The Lisa Operating System provides special utility routines for disk drivers: **USE_HDISK**, **CALL_HDISK**, **IODONE**, and **OKXFERNEXT**. These routines make subroutine calls into the **HDISK** unit (listed in Appendix A). Only disk drivers may use these routines.

o **USE_HDISK** notifies the OS that your driver wants to use **CALL_HDISK**. Your disk driver must call **USE_HDISK** once, before calling **CALL_HDISK**, each time the driver is initialized (**Dinit** function code).

o **CALL_HDISK** performs the functions associated with the **Dinit**, **Ddown**, **Dskio**, and **Reqrestart** function codes. Whenever you disk driver gets called with function code, it should call **CALL_HDISK** to perform these functions.

CALL_HDISK may be used only with a disk device with Lisa-format sectors. These are 512-byte sectors preceded by a sector header in pagelabel format (see the pagelabel format record in **Driverdefs**, listed in Appendix A).

o **IODONE** unblocks the current completed I/O request, takes the next request from the device's Request Block queue, and initiates it.

o **OKXFERNEXT** updates and verifies the sector header for each block transferred to or from the disk. Your driver calls it after transferring each 512-byte block.

2.5.1 USE_HDISK Procedure
procedure USE_HDISK (Configptr:ptrdevrec)

Input Parameters:
Configptr: Points to Device Configuration Record for the hard disk device.

USE_HDISK does initialization for a disk driver. The disk device must use Lisa-format sectors. Your disk driver should call **USE_HDISK** during the driver's **Dinit** routine.

2.5.2 CALL_HDISK Procedure
procedure CALL_HDISK (var Error:integer; Configptr:ptrdevrec; Parameters:param_ptr)

Input Parameters:
Configptr: Points to Device Configuration Record.
Parameters: Parameter list that was passed to the driver.

Output Parameters:
Error: Indicates result of CALL_HDISK operation.

CALL_HDISK calls the **HDISKIO** routine in the **HDISK** unit to support the driver's **Dinit**, **Ddown**, **Dskio**, and **Reqrestart** functions. The driver must take the **Error** value returned by **CALL_HDISK** and assign it to the driver's function result before returning.

2.5.3 IODONE Procedure
procedure IODONE (Port_ptr:hdskcb_ptr; Prev_err:integer)

Input Parameters:
Port_ptr: Points to Device Control Block.
Prev_err: Contains completion error.

Your driver must call **IODONE** after it finishes servicing an I/O request in order to finish processing the request and initiate the next request.

2.5.4 OKXFERNEXT Function
function OKXFERNEXT (Port_ptr:hdiskcb_ptr):integer

Input Parameters:
Port_ptr: Points to Device Control Block.
Output Parameters:
OKXFERNEXT: Zero if successful, else error code.

OKXFERNEXT updates the File System sector headers and data pointers and tests to see whether another sector can be transferred. If any errors are found in the sector just transferred, the transfer is terminated and **OKXFERNEXT** returns a nonzero error code. If a new transfer can be initiated, **OKXFERNEXT** returns zero.

2.6 DRIVER-INITIATED I/O REQUESTS

Your driver may not be able to complete a requested operation immediately because it takes time for the device to perform the operation. This is true when function codes **Dinit**, **Dcontrol**, and **Dskformat** are invoked and call **BLK_REQ** to wait for I/O to complete. **READ_DATA** and **WRITE_DATA** call the OS to do deferred-request management for the driver, calling **Seqio** or **Dskio** in the driver when needed; in other cases the driver has to perform these operations for itself.

To defer an I/O request, follow this procedure:

1. Call **GETSPACE** to create an I/O Request Block.

2. Call **LINK_TO_PCB** to initialize the Request Block and link it to the Process Control Block of the active process.

3. Call **CVT_BUFF_ADDR** to convert the I/O buffer address (if any) to a format the interrupt handler can use.

4. Call **BLK_REQ** to wait until the I/O request has been serviced.

5. When your request is unblocked by the Scheduler, call **CANCEL_REQ** to unlink the Request Block from the Process Control Block and return its space to the free space pool.

When managing your own Request Blocks, you will need to call **INTSOFF**, **INTSON**, **ENQUEUE**, **UNBLK_REQ**, and possibly **ADJ_IO_CNT**, **FREEZE_SEG**, and **UNFREEZE_SEG**, described in this chapter.

2.6.1 LINK_TO_PCB Procedure
procedure LINK_TO_PCB (Req_ptr:reqptr_type)

Input Parameters:

Req_ptr: Pointer to the new Request Block.

LINK_TO_PCB takes a Request Block created using **GETSPACE**, initializes several standard fields, and links the Request Block to the Process Control Block of the currently active process. (The **Reqspec_info** field is not initialized.)

2.6.2 CVT_BUFF_ADDR Procedure

procedure CVT_BUFF_ADDR (var Errnum:int2; Mem_writeF:boolean; Buffaddr:absptr; Byte_cnt:int4; var Ordsdb:absptr; var Offset:int4)

Input Parameters:

Mem_writeF: True if transfer from device to memory.
Buffaddr: Address to be converted.
Byte_cnt: Number of bytes to transfer to or from memory.

Output Parameters:

Errnum: Zero if successful conversion; else nonzero.
Ordsdb: Pointer to the data segment that includes Buffaddr.
Offset: Relative offset within the data segment.

CVT_BUFF_ADDR takes an address in the application program's address space and breaks it into two parts: a pointer to the data segment descriptor and an offset within the data segment. Interrupt handlers need this form of address to access the user's data when it is not currently mapped by the Memory Management Unit hardware. **CVT_BUFF_ADDR** also verifies that the requested I/O operation is valid for the specified data segment; for instance, you can't pass an address within a program as the destination for a read operation.

2.6.3 BLK_REQ Procedure

procedure BLK_REQ (Reqptr:reqptr_type; var First_reqptr:reqptr_type)

Input Parameters:

Reqptr: Pointer to list of requests to wait for.

Output Parameters:

First_reqptr: Pointer to list of completed results.

When **BLK_REQ** is called, the current process is blocked until any of the requests passed to **BLK_REQ** has completed. (Request completion is signaled when the driver calls **UNBLK_REQ**.)

Reqptr points to a linked list of I/O Request Blocks representing events whose completion the current process must await. **First_reqptr** points to a linked list of completed I/O requests, in the order in which they were completed.

2.6.4 CANCEL_REQ Procedure

procedure CANCEL_REQ (Req:reqptr_type)

Input Parameters:

Req: Pointer to Request Block to cancel.

CANCEL_REQ unlinks the specified Request Block from its Process Control Block and releases its space back to the free space pool.

2.7 MISCELLANEOUS SUBROUTINES

This section contains information on routines that
o report fatal errors -- **SYSTEM_ERROR**.
o detect availability of the System_Log file -- **LOGGING**.
o log data into the System_Log file -- **LOG**.
o test a byte using a bit mask -- **ALLSET**.
o call a device driver from a demultiplexing driver -- **CALLDRIVER**.
o synchronize access to the parallel port -- **DISKSYNC**.
o detect pressing of disk buttons and switches -- **KEYPUSHED**.

2.7.1 SYSTEM_ERROR Procedure

procedure SYSTEM_ERROR (Errnum:integer)

Input Parameters:
Errnum: Error code provided by the driver.

When a driver encounters an error condition so serious that it cannot continue, it should call **SYSTEM_ERROR**, which uses **Errnum** to pass an error code from the driver to the Operating System; **SYSTEM_ERROR** does not return to the driver. Do not use **SYSTEM_ERROR** if an error code can be returned to the caller.

The value of **Errnum** should be 10,000 plus the driver error number.

2.7.2 LOGGING Function

function LOGGING: boolean

Output Parameters:
LOGGING: True if system logging is operational; else false.

Before logging an entry using the **LOG** procedure, use the **LOGGING** function to determine whether system logging is available. If logging is not available, **LOG** should not be called.

2.7.3 LOG Procedure

procedure LOG (var Errnum:integer; Ptr_arr:absptr)

Input Parameters:
Ptr_arr: Pointer to 12 bytes of data to be logged.

Output Parameters:
Errnum: Error code.

The **LOG** procedure allows you to make a 12-byte entry in the **System.Log** file if logging is active. The procedure may be used to log disk errors. The data to be logged may be in any format.

Before calling **LOG**, you should determine whether logging is available by calling the **LOGGING** function.

2.7.4 ALLSET Function

function ALLSET (Source, Target:int1):boolean

Input Parameters:
Source: Mask bits.
Target: Bits to be tested.

Output Parameters:
ALLSET: True if all mask bits are set in target; else false.

ALLSET determines whether all the bits in **Source** are on (set to one) in **Target**. **Target** is a byte in which each bit has separate significance; for instance, the status register for a device. **Source** is a mask byte. If the integer value of **Target** is 7 (00000111) and the integer value of **Source** is 3 (0000011), **ALLSET** returns true. (If the value of **Source** is zero, **ALLSET** returns true.)

2.7.5 CALLDRIVER Procedure

procedure CALLDRIVER (var Errnum:integer; Config_ptr:ptrdevrec; Parameters:param_ptr)

Input Parameters:
Config_ptr: Pointer to the Device Configuration Record for the device.
Parameters: Variant record containing driver parameters.

Output Parameters:
Errnum: Depends upon Parameters.

CALLDRIVER is used by a demultiplexing driver to call a lower-level driver with an interrupt intended for its device. The device driver must previously have been associated with the demultiplexing driver by means of the **Dattach** function code.

2.7.6 DISKSYNC Procedure

procedure DISKSYNC (Busy:boolean)

Input Parameters:
Busy: True if built-in parallel port is busy; False if port is free.

The built-in parallel port (Hard Disk VIA) is shared by the disk driver, printer driver, and video contrast control routines. All device drivers using this port must synchronize their access to it by calling **DISKSYNC** before and after each use of the port. When the port is free, the contrast control routines are able to modify and then restore the registers in the 6522 VIA.

HDISK calls **DISKSYNC** for disk drivers performing reads and writes. However, when a disk driver is formatting or initializing the built-in parallel port, it must call **DISKSYNC** itself.

To gain control of the built-in parallel port, call **DISKSYNC** with **Busy** set to true before doing I/O; when the I/O has completed, call **DISKSYNC** again with **Busy** set to false. (The port is initially free.)

It is not necessary to call **DISKSYNC** when accessing a parallel port in one of the Lisa's three I/O expansion slots. Only the built-in parallel port needs this synchronization.

2.7.7 KEYPUSHED Procedure

procedure KEYPUSHED (Key:0..127)

Input Parameters:

Key: Keycode of the pseudo-key that was pressed.

The disk insertion switches, the disk eject buttons, and the power button -- collectively known as pseudo-keys -- are not detected by the normal keyboard driver. Drivers that detect these switches and buttons must call the **KEYPUSHED** procedure each time a pseudo-key is pressed. You must identify the pseudo-key that was pressed by setting Key to one of the keycode values shown below:

Keycode	Meaning
1	disk was inserted in disk 1
2	disk 1 eject button was pressed
3	disk was inserted in disk 2
4	disk 2 eject button was pressed
11	disk was inserted in built-in Sony drive
12	disk was inserted in slot 1 device
13	disk was inserted in slot 2 device
14	disk was inserted in slot 3 device

Chapter 3
DRIVER FUNCTION CODES

3.1 Invoking the Driver
3.2 DriverOperations
    3.2.1 Driver Types and their Function Codes
    3.2.2 Error Codes
3.3 Function Codes Shared by Different Driver Types
    3.3.1 DINTERRUPT
    3.3.2 DINIT
    3.3.3 DDOWN8
    3.3.4 DCONTROL
    3.3.5 REQRESTART
    3.3.6 DALARMS
3.4 Function Codes for Sequential Device Drivers
    3.4.1 SEQIO
    3.4.2 DDISCON
3.5 Function Codes for Disk Device Drivers
    3.5.1 DSKIO
    3.5.2 DSKUNCLAMP
    3.5.3 DSKFORMAT
    3.5.4 HDINIT
    3.5.5 HDDOWN
    3.5.6 HDSKIO
3.6 Function Codes for Demultiplexing Drivers
    3.6.1 DATTACH
    3.6.2 DUNATTACH
3.7 Drivers for Bootable Devices

DRIVER FUNCTION CODES

3.1 INVOKING THE DRIVER

A driver is invoked directly by the Operating System or by another driver through the **CALLDRIVER** procedure (described in Section 2.7.5). Every time the driver is invoked, it must perform a specific operation as specified in the *function code* -- the **Fnctn_code** field of the **Params** record that is passed to the driver. This chapter describes the operations that must be performed for each function code.

If your driver receives a function code that it does not recognize, it should return with *error number 656* in its function result.

The **Params** record is passed to the driver no matter which function code is invoked; therefore it is not listed in the documentation for each function code. The parameters that are shown in curly brackets under the function code are those that are passed to the driver in addition to **Params**.

The format of the **Params** record differs depending on the function code; it always contains a pointer to the *Device Configuration Record* for the device as well as the function code itself.

For device-independent applications that call **READ_DATA** and **WRITE_DATA**, standard driver function codes are provided. For nonstandard devices (those that cannot be supported through the File System) or nonstandard applications, a mechanism is provided for the application to communicate all I/O requests directly to the driver using **DEVICE_CONTROL**. In this case the driver must respond to the **Dcontrol** function code and must perform its own Request Block management. Standard drivers may on occasion call **DEVICE_CONTROL** to perform functions such as setting baud rate or acquiring status information.

To improve performance, routines that process driver function codes (for example, **Dinterrupt**, which may require fast execution) may be written in assembler language rather than Pascal.

When Apple integrates networks into the Lisa Operating System, new function codes may be added for handling network-specific functions.

3.2 DRIVER OPERATIONS

There are certain operations that are standard for each driver type: sequential, disk, demultiplexing, and non-File System. Requests for these operations are signaled to the driver by means of the driver function code. Errors the driver encounters while processing a request are reported in the driver function result.

3.2.1 Driver Types and their Function Codes

Table 3-1 is a list of driver function codes and the operations they represent by type.

Table 3-1
Driver Operations

FUNCTION CODES SHARED BY DIFFERENT DRIVER TYPES:

Dinterrupt	Handle a device interrupt
Dinit	Initialize a driver and its associated device
Ddown	Shut down a driver and its associated device
Dcontrol	Send or receive device-dependent information
Reqrestart	Restart an I/O request

Dalarms..................Handle an alarm interrupt

FUNCTION CODES FOR SEQUENTIAL DEVICE DRIVERS:
Seqio........................Input/output to a sequential device
Ddiscon...................Disconnect modem

FUNCTION CODES FOR DISK DEVICE DRIVERS:
Dskio........................Start I/O to a disk device
Dskunclamp...........Eject a disk from the disk device
Dskformat...............Format a disk
Hdinit........................Initialize a disk device driver
Hddown..................Shut down a disk driver and its associated device
Hdskio.....................Transfer data to or from a disk device

FUNCTION CODES FOR DEMULTIPLEXING DRIVERS:
Dattach.................Associate a driver with this demultiplexing driver
Dunattach............Dissociate a driver from this demultiplexing driver

---

A driver needs to handle only a subset of the standard driver operations. Table 3-2 indicates which driver types must handle each function code. The table also contains information about privilege state, interrupt disabling, and error code requirements for each function code.

Table 3-2
Characteristics of Driver Operations

(1) FUNCTION CODE	(2) PRIVILEGE STATE	(3) DISABLE INTERRUPTS	(4) ERR CODE REQUIRED	(5) DRIVER TYPE SEQ	DISK	DEMX	NON-FS
Dinterrupt	Supervisor	No	No	R	R	R	O
Dinit	User	Yes	Yes	R	R	R	R
Ddown	User	Yes	Yes	R	H	R	R
Dcontrol	User	Yes	Yes	O	O		R
Reqrestart	User	Yes	No	R	H		O
Dalarms	Supervisor	No	No	O	O	O	
Seqio	User	Yes	Yes	R			
Ddiscon	User	Yes	No	O			O
Dskio	User	Yes	Yes		H		
Dskunclamp	User	Yes	Yes		O		
Dskformat	User	Yes	Yes		O		
Hdinit	User	No	Yes		R		
Hddown	User	No	Yes		R		
Hdskio	Either	No	Yes		R		
Dattach	User	Yes	Yes		R		
Dunattach	User	Yes	No			R	

Column (5): H=HDISK, O=Optional, R=Required

The columns of Table 3-2 contain the following information:

(1) FUNCTION CODE is the identifier assigned to this operation in **Driverdefs** (see Appendix A). The function codes

are summarized in Table 3-1 and documented in this chapter.

(2) PRIVILEGE STATE indicates the setting of the S bit in the *processor's status register*. Certain assembler language instructions are privileged and may be executed only in supervisor state. Your driver should not use privileged instructions to process a function code if the operations will be performed in user state.

(3) DISABLE INTERRUPTS is Yes if the driver must call **INTSOFF** to protect the Device Control Block from being modified by another interrupt while the driver is executing. The call to **INTSOFF** should set the priority just high enough to prevent interrupts from the driver's device. Before returning, your driver must call **INTSON** to restore the interrupt status to its previous priority level.

Note that when the **Dalarms** function is invoked, the interrupt priority has already been set to the value that was established in the driver's prior call to **ALARM_ASSIGN**.

(4) ERR CODE REQUIRED is Yes if the driver must return a value in the integer function-result of the driver function. If an error code is not required, the function result need not be assigned a value.

(5) DRIVER TYPE indicates which function codes a given driver needs to support the types are the following:

SEQ	Sequential device drivers
DISK	Disk device drivers
DEMX	Demultiplexing drivers
NON-FS	Non-File System drivers that use **Dcontrol** for I/O

The letters under each driver type indicate whether the driver must support the function code:

H	The HDISK unit supports this operation
O	Optional for this device type (see detailed description of the function code for more information)
R	Required for this device type

3.2.2 Error Codes

Your driver may return an error number in the driver function result. If your driver receives a function code that it does not recognize, it should return with *error number 656* in its function result. In the case of **Dinterrupt**, **Dalarms**, and **Reqrestart**, the error number should be placed in the **Hard_error** field of the Request Block. In the case of a non-recoverable data inconsistency, the error should be signaled by calling **SYSTEM_ERROR**.

Standard error numbers are indicated throughout this chapter. The following standard error numbers apply to all device types:

660	*Cable disconnected.*
666	*Device timeout.*

You are free to use other error numbers in the range 1840 to 1879 to represent particular conditions your driver expects to encounter. *Warnings* are indicated by a negative error number in the range -1879 to -1840. A warning means that the operation was successfully completed under special circumstances. *System errors* reported by your driver must be in the range 11840 to 11879.

3.3 FUNCTION CODES SHARED BY DIFFERENT DRIVER TYPES

The function codes described in this section are used by drivers of differing type. These function codes are **Dinterrupt**, **Dinit**, **Ddown**, **Dcontrol**, **Reqrestart**, and **Dalarms**.

3.3.1 DINTERRUPT Driver Function Code

***function code* Dinterrupt**
{Intpar: integer}

Parameters:
Intpar: For Serial A & B, contains the interrupt type.

Operation: Handle a device interrupt
Drivers: SEQ (R), DISK (R), DEMX (R), NON-FS (R)
Privilege State: Supervisor
Must Disable Interrupts: No
Error Code: Not required

Your driver is called with the **Dinterrupt** function code when there is an interrupt for it to handle. For most devices, **Intpar** is zero and has no meaning. For Serial A or Serial B interrupts, **Intpar** contains a value between 0 and 3 representing the type of interrupt (refer to read-register 2B in the SCC manual).

For demultiplexing drivers, **Dinterrupt** signals you to find out from the interrupting controller which of its several devices is the source of the interrupt. You must then invoke its device driver to handle the interrupt by passing the same parameter list the demultiplexing driver was called with using the **CALLDRIVER** procedure.

Your interrupt handler for the device must:

o Signal the device to clear the interrupt.
o Perform the desired I/O transfer.
o Call **OKXFERNEXT** (disk devices only).
o Return.

The I/O request may require several I/O transfers. When the request has been completed, or if an error was detected, your interrupt handler should call **IODONE** if your driver supports a disk device; for other device types, it should:

o Call **UNBLK_REQ** to unblock the process that is waiting for I/O.
o Call **ADJ_IO_CNT** to reduce the count of pending I/O requests for the data segment.
o **DEQUEUE** the Request Block from the Device Control Block.
o If another Request Block for this device is on the queue, initiate the I/O.
o Return.

3.3.2 DINIT Driver Function Code

***function code* Dinit**

Operation: Initialize a driver
Drivers: SEQ (R), DISK (R), DEMX (R), NON-FS (R)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

For the boot device or a device that is configured to be pre-loaded, **Dinit** is invoked during startup. For other devices it is called wither at the time the device is mounted or prior to calling **Dcontrol**, **Dskunclamp**, or **Dskformat** for unmounted devices.

To initialize your device driver, you must:

- Call **GETSPACE** to acquire space for the driver's Device Control Block (its global data).
- Store the address of the Device Control Block into **Parameters^.Configptr^.Cb_addr**.
- Initialize the Device Control Block.
- Call **ALARM_ASSIGN** to reserve any alarms that will be needed while the driver is running. Save the alarm numbers that have been assigned to this device during initialization. (You must reserve space for this purpose in the Device Control Block.)
- Prepare the device hardware for operation. This may involve programming the controller's hardware registers, reading the device's version number, and so on.

If the driver encounters an error during initialization, it should reverse the initialization procedure so that the device and associated control blocks are returned to the state they were in when the driver was invoked with **Dinit** function code. The function result of the driver should be set to reflect the nature of the error:

- *Error 602* means that alarms were not available from **ALARM_ASSIGN**.
- *Error 610* means that memory was not available from **GETSPACE**.

For disk devices that are supported by the Operating System **HDISK** unit (see Section 2.5, Disk Driver Support), your driver should not perform the functions listed above. Instead:

- Call **USE_HDISK**
- Call **CALL_HDISK**

HDISK will then invoke your driver with function code **Hdinit** to request disk driver initialization.

3.3.3 DDOWN Driver Function Code

***function code* Ddown**

Operation: Shut down a device

Drivers: SEQ (R), DISK (H), DEMX (R), NON-FS (R)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

For devices that are configured to be dynamically loaded and unloaded, **Ddown** is invoked during unmounting unless the Characteristics file for the device specifies that the device is to be permanently mounted. (Chapter 5 tells you how to specify characteristics.) **Ddown** tells your driver to undo everything it did in **Dinit**.

Return with Error 60? if the driver is waiting for I/O at the time **Ddown** is invoked. Otherwise do the following:

- Call **RELSPACE** to release the space for the driver's Device Control Block.
- Set the address of the Device Control Block in **Parameters^.Configptr^.Cb_addr** to **ord(nil).**
- Call **ALARMRETURN** to release any alarms that were assigned to the driver.
- Set the device hardware so that it cannot generate an interrupt until **Dinit** is called again.

3.3.4 DCONTROL Driver Function Code

***function code* Dcontrol**
{Parptr: absptr}

Parameters:
Parptr: Address of a Device Control Record

Operation: Send or receive device-dependent information

Drivers: SEQ (O), DISK (O), NON-FS (O)

Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

For sequential and disk devices, your driver needs to support the **Dcontrol** function code only if application programs will access this device by calling the **DEVICE_CONTROL** procedure. The application program provides **DEVICE_CONTROL** with a pointer to the Device Control Record, and **DEVICE_CONTROL** passes the record to the driver with the **Dcontrol** function code.

Refer to the *Operating System Reference Manual for the Lisa* for information on the **DEVICE_CONTROL** File System call. (Note that the **Dctype** record described in that manual is the same as the **Dc_rec** shown below.) If you want your driver to be compatible with similar existing OS drivers, support all the **DEVICE_CONTROL** operations documented for that device type in the Operating System Reference Manual.

Parptr is the address of the Device Control Record, **Dc_rec**, defined below. The driver and the application program must agree upon the use of the Device Control Record; its contents are not checked by the **DEVICE_CONTROL** procedure.

```
Dc_rec = record
            Dversion: integer;
            Dcode: integer;
            Ar10: array[0..9] of longint;
     end;
```

Dversion is used for application-defined version control between the driver and the application program. **Dcode** tells your driver what status information to get from its device or what control information to send. The **Ar10** array is used to pass additional device information between the application program that calls **DEVICE_CONTROL** and the device driver.

Dcontrol may be used for configuring operational characteristics of the driver, such as the baud rate.

If **Dcode** requests I/O, your driver must perform the following operations (refer to Section 2.6, Driver-initiated I/O Requests, for more information):

- o Create its own Request Block
- o Define a Request Block Extension containing whatever information is needed.
- o Call **CVT_BUFF_ADDR**, **ADJ_IO_CNT**, and **FREEZE_SEG** to freeze the user's data buffer.
- o Call **BLK_REQ** to wait until the I/O has completed.

Return *error 623* if the contents of **Dc_rec** are not valid for the device.

3.3.5 REQRESTART Driver Function Code

function code **Reqrestart**
{Req: reqptr_type}

Parameters:
Req: Pointer to Request Block passed to FREEZE_SEG

Operation: Restart an I/O request
Drivers: SEQ (R), DISK (H), NON-FS (O)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Not required

Reqrestart is required by all drivers of sequential or disk devices and by non-File System device drivers that call **FREEZE_SEG**. **Reqrestart** is invoked by the OS Memory Manager to restart the driver from the point at which a **FREEZE_SEG** request failed because the data was being relocated in memory. If you call **FREEZE_SEG** from more than one place in your driver, you must set a flag to indicate where **Reqrestart** must restart.

If your driver uses **HDISK**, call **CALL_HDISK** when you are passed a **Reqrestart** function code.

3.3.6 DALARMS Driver Function Code

***function code* Dalarms**
{Intpar: integer}

Parameters:
Intpar: Alarm number

Operation: Handle an alarm interrupt
Drivers: SEQ (O), DISK (O), NON-FS (O)
Privilege State: Supervisor
Must Disable Interrupts: No
Error Code: Not required

Your driver needs to respond to the **Dalarms** function code only if the driver calls **ALARMRELATIVE**. **Dalarms** is invoked when an alarm set by **ALARMRELATIVE** goes off. Intpar contains the alarm number.

Alarms can be used to handle interrupts that require a long time to process. See Lengthy Interrupt Processing, Section 2.3.4.

3.4 FUNCTION CODES FOR SEQUENTIAL DEVICES

The following function-codes are called only for drivers of sequential devices. Sequential devices are accessed as a sequence of bytes; their drivers make no provision for storing and retrieving files at different locations on the device. Applications that require sequential device drivers include data communications and printing.

3.4.1 SEQIO Driver Function Code

***function code* Seqio**
{Req: reqptr_type}

Parameters:
Req: Pointer to the I/O Request Block.

Operation: Input/output to a sequential device
Drivers: SEQ (R)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

Seqio is invoked during I/O to a sequential device when **READ_DATA** or **WRITE_DATA** is called to transfer a sequence of bytes from or to a sequential device. **Req** points to an initialized I/O Request Block as described in Request Block initialization, Section 2.4.1. **Seqio** must call **ENQUEUE** to place the Request Block at the end of the Request Block queue for the device. If there are no other requests currently in progress, the driver can then initiate the I/O transfer for this request. If another I/O request for the device is in progress, the driver will initiate the I/O transfer for this request when it completes all prior requests in the queue.

3.4.2 DDISCON Driver Function Code
***function code* Ddiscon**

Operation: Modem disconnect
Drivers: SEQ (O), NON-FS (O)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Not required

Ddiscon is invoked when **CLOSE_OBJECT** is called for a sequential device or when a process aborts without closing an open sequential device. If your driver controls a device attached to a modem, disconnect the modem from the line. For other sequential devices, your driver should return immediately after receiving the **Ddiscon** function code.

Note that the driver is not called during **OPEN** for sequential devices. The application program should initialize modem control by calling **DEVICE_CONTROL** from the application program when opening the device.

3.5 FUNCTION CODES FOR DISK DEVICES

The operations described in this section are performed only by drivers of Lisa-format random-access disk devices.

Most disk devices can use the facilities of the **HDISK** unit in the Operating System to perform the standard operations required for function codes **Dinit**, **Ddown**, **Reqrestart**, and **Dskio**. (See Appendix A for a listing of **HDISK**.) These operations include managing the Request Block queue and processing the File System headers that precede each 512-byte block of data on the disk. **HDISK** contains the six routines described below.

Called directly by the disk driver (described in Chapter 2):

- **XFERINIT**
- **IODONE**

Called by the OS when the disk driver calls **CALL_HDISK** (your driver should pass to **CALL_HDISK** the same parameters with which it was called):

- **INIT** for **Dinit** function code
- **DOWN** for **Ddown** function code
- **REQRESTART** for **Reqrestart** function code
- **DSKIO** for **Dskio** function code

Standard error numbers for disk drivers are:

606	*Can't find sector (disk may be unformatted).*
614	*No disk present in drive.*
617	*Checksum error in data -- no data returned.*
616	*Cannot format disk.*
654	*Unspecified hard I/O from disk controller.*
-663	*Read with data returned -- some data may be okay, but a checksum, CRC, or parity error was detected.*
1824	*Write failed.*

3.5.1 DSKIO Driver Function Code
function code **Dskio**
{Req: reqptr_type}

Parameters:
Req: Pointer to the I/O Request Block.

Operation: Input/output to a disk device
Drivers: DISK (H)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

Dskio is invoked by **READ_DATA** and **WRITE_DATA** to access a disk file. The File System also invokes this function to maintain its internal directory structures while performing file operations such as **OPEN** and **CLOSE_OBJECT**. **Req** points to an initialized I/O Request Block.

When **Dskio** is invoked, your driver should call **CALL_HDISK**, passing it the parameters with which it was invoked. **CALL_HDISK** will then invoke **HDISK** to handle the I/O request.

3.5.2 DSKUNCLAMP Driver Function Code
function code **Dskunclamp**

Operation: Eject a disk from the disk drive
Drivers: DISK (O)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

If the disk device has a removable disk, eject the disk when **Dskunclamp** is invoked; otherwise return immediately with *error 685*.

If the device will send an interrupt to signal that the eject has completed, it may be necessary for **Dskunclamp** to create a Request Block. Refer to Section 2.6, Driver-initiated I/O Requests.

3.5.3 DSKFORMAT Driver Function Code
function code **Dskformat**

Operation: Format a disk device
Drivers: DISK (O)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

Your driver needs to handle this function code only if the disk device formats its media; otherwise the driver may return immediately. **Dskformat** may be invoked when the user mounts a disk that cannot be read or when the application receives a request to erase everything on the disk.

Dskformat initiates the format operation by signaling the device's intelligent controller. The driver must create a

Request Block for this operation and call **BLK_REQ** to wait until the format operation has completed. Refer to Section 2.6, Driver-initiated I/O Requests.

3.5.4 HDINIT Driver Function Code

function code **Hdinit**
{Req: reqptr_type}

Operation: Initialize a disk device-driver
Drivers: DISK (R)
Privilege State: User
Must Disable Interrupts: No
Error Code: Required

Hdinit is invoked by the **INIT** routine within **HDISK**. Your disk driver must perform the following operations when invoked with the **Hdinit** function code:

To initialize your device driver, you must:
- o Call **GETSPACE** to acquire space for the driver's Device Control Block extension.
- o Store the address of the extension in the **Ext_ptr** field of the Device Control Block.
- o Call **ALARM_ASSIGN** to reserve any alarms that will be needed while the driver is running. Save the alarm numbers that have been assigned to this device during initialization. (You must reserve space for this purpose in the Device Control Block.)
- o Initialize the disk controller hardware so that read and write commands can be sent.

If the driver encounters an error during initialization, it should reverse the initialization procedure so that the device and associated control blocks are returned to the state they were in when the driver was invoked with **Hdinit** function code. The function result of the driver should be set to reflect the nature of the error:
- o *Error 602* means that alarms were not available from **ALARM_ASSIGN**.
- o *Error 610* means that memory was not available from **GETSPACE**.

3.5.5 HDDOWN Driver Function Code

function code **Hddown**
{Req: reqptr_type}

Operation: Shut down a disk device
Drivers: DISK (R)
Privilege State: User
Must Disable Interrupts: No
Error Code: Required

HDISK invokes **Hddown** so that your driver can undo everything it did in **Hdinit**:

- o Call **RELSPACE** to release the space for the driver's Device Control Block extension.
- o Set the address of the Device Control Block extension in **Ext_ptr** to **ord(nil)**.
- o Call **ALARMRETURN** to release any alarms that were assigned to the driver.
- o Set the disk device controller so that it cannot generate an interrupt until **Hdinit** is invoked again.

3.5.6 HDSKIO Driver Function Code

function code **Hdskio**
{C_cmd: integer; C_sector: longint}

Parameters:
    C_cmd:
    C_sector:

Operation: Perform an I/O transfer
Drivers: DISK (R)
Privilege State: User
Must Disable Interrupts: No
Error Code: Required

HDISK invokes the disk driver with function code **Hdskio** to begin an I/O transfer; that is, **Hdskio** reads or writes a group of contiguous 512-byte blocks. Options in the **Io_mode** field of the Request Block extension allow you to transfer the blocks with or without their 24-byte headers. **Io_mode** options are described in Section 2.4.1, Request Block Initialization. Headers are described in the **Pagelabel** type in **Driverdefs**, listed in Appendix A.

During each interrupt of the I/O transfer, your driver must call **OKXFERNEXT** to update the headers; when the transfer is complete, it must call **IODONE**. (See Disk Driver Support, Section 2.5, for further information.)

3.7 FUNCTION CODES FOR DEMULTIPLEXING DRIVERS

The operations described in this section are performed only by demultiplexing drivers -- those that call lower-level drivers. See Chapter 1 for a description of driver hierarchy and the role of demultiplexing drivers.

3.6.1 DATTACH Driver Function Code

function code **Dattach**
{N_configptr: ptrdevrec}

Parameters:
N_configptr: Pointer to Device Configuration Record of new device.

Operation: Associate a driver with this demultiplexing driver
Drivers: DEMX (R)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Required

Demultiplexing drivers are drivers that dispatch interrupts for more than one device. (See Figure 1-4, Communication between Drivers.) **Dattach** enables the demultiplexing driver to service the interrupts of the devices whose drivers it attaches. In general, **Dattach** is invoked immediately after the **Dinit** call to the demultiplexing driver. Then, as each of its devices gets initialized, the demultiplexing driver is invoked with **Dattach** to attach the device driver for the new device. That is, **Dattach** is invoked once for each device associated with the demultiplexing driver.

When an interrupt for an attached device is received, the demultiplexing driver calls the lower-level driver using the **CALLDRIVER** procedure described in Chapter 2.

N_configptr points to the Device Configuration Record of a new device being attached to this driver.

3.6.2 DUNATTACH Driver Function Code

function code **Dunattach**
{O_configptr: ptrdevrec; Still_inuse: boolean}

Parameters:
O_configptr: Pointer to Device Configuration Record of new device.
Still_inuse: True if other devices remain attached; False if none.

Operation: Disassociate a driver from this demultiplexing driver
Drivers: DEMX (R)
Privilege State: User
Must Disable Interrupts: Yes
Error Code: Not required

Dunattach is invoked when a device associated with the demultiplexing driver is invoked with **Ddown** function code. **O_configptr** points to the Device Configuration Record for the device being disassociated.

Dunattach must set **Still_inuse** to **true** if other devices remain attached to the demultiplexing driver. If no devices remain attached, **Still_inuse** must be set to **false** so the calling routine can invoke **Ddown**.

3.7 DRIVERS FOR BOOTABLE DEVICES

{ To be documented by Jay Walton.

By the time we release the alpha draft (around 1/11/84), we should have a definite decision on whether configurable drivers for bootable devices will be supported for spring release.

Please advise.}

Chapter 4
HARDWARE INTERFACE

4.1 System Clock Rates
4.2 The Serial Communications Controller (SCC)
    4.2.1 SCC Timing
    4.2.2 Other Information Related to the SCC
        4.2.2.1 Cable Connections
        4.2.2.2 SCC Register Usage
4.3 The 6522 Parallel Port
    4.3.1 Using the 6522 Pulse Mode Handshake
    4.3.2 Parallel Port Differences
4.4 Peripheral Cards
4.5 Printers
4.6 Built-in Devices

HARDWARE INTERFACE

Chapter 4 supplements the *Lisa Hardware Manual* and *Lisa Theory of Operations* with hardware documentation relevant to device drivers. You need to become familiar only with those sections of the chapter that contain information about the particular device interface your driver uses.

4.1 SYSTEM CLOCK RATES

The 68000 is driven by a 20.375 MHz crystal divided by 4, yielding a 5.09375 MHz clock; the clock rate is therefore one cycle every 196.319 nanoseconds, while it may be impossible to compute exact execution times, you can use this clock to compute an approximate execution time for a given section of code.

Your calculation should round up the published 68000 instruction execution-cycle times to a value n+x (where n is the number of cycles and x is 0, 1, 2, or 3) such that n+x is a multiple of 4. Rounding is necessary because of wait states the Lisa uses to handle the video memory. The pipeline architecture of the 68000, which pre-fetches instructions, makes even this calculation imprecise. In addition, when accessing the address space of a 6522, the 68000 inserts wait states in order to synchronize with the 6522 clock.

4.2 THE SERIAL COMMUNICATIONS CONTROLLER (SCC)

The Serial Communications Controller is described in Zilog's *SCC Technical Manual*. The *Lisa Hardware Manual* (Sections 2.5.2, 3.5.1, and 6.4) also includes information about the operation of the SCC in Lisa.

4.2.1 SCC Timing

Two consecutive accesses to the control register address are necessary to read or write data to internal registers 1 through 15. Therefore you should disable SCC interrupts during consecutive accesses.

The 68000 **CLR** instruction accesses the target address twice during a single instruction. The Lisa's Assembler optimizes the **MOVE #0** instruction to **CLR**. Therefore don't use **CLR** or **MOVE #0** instructions when writing to the SCC. The Lisa's Pascal Compiler generates a **CLR** instruction when you assign zero to a variable. Therefore if you code an SCC interface in Pascal, do not assign zero directly to an SCC control register; first assign zero to a variable, then assign the variable to the SCC address.

A recovery time of 6 CPU cycles is required between consecutive accesses. The Pascal Compiler usually generates code that provides sufficient recovery time. However, if you use WITH statements to optimize SCC accesses, you may need to insert some delay between accesses.

If your driver is written in assembler language, provide recovery time by inserting one or two **NOP** instructions between contiguous instructions that access the SCC.

4.2.2 Other Information Related to the SCC

Asynchronous I/O can occur on either port A or B. The following characteristics are unique to each port:

- Port A
    - accomodates synchronous communications.
    - accomodates SDLC/HDLC.
    - provides full RS-232C modem control.

- Port B
    - accomodates Applebus.

4.2.2.1 Cable Connections

The *Lisa Hardware Manual* (Figure 3-7) shows the cable signals for the DB-25 connector. Cable signal names are the same as SCC Pin names except for the following:

CABLE SIGNAL	CONNECTS TO SCC PIN
RxC(A)	RTxCA
TEXT(A) & TxC(A)	TRxCA
DSR(A)	SYNCA (and DCDB on hardware after 1.0)
DSR(B)	CTSB/

Apple's modem eliminator cable connects the following signals (dashed lines indicate wires; arrows show the direction of information flow along the wires):

```
GROUND --------------------- GROUND
SEND -------------------------> RECEIVE
DTR ---------------------------> DSR
DSR <--------------------------- DTR
RTS    )----------------------> DCD
CTS <-)
DCD <----------------------(    RTS
(-> CTS
```

4.2.2 Register Usage

Registers 2 and 9 are shared between ports A and B. The following restrictions apply:

o Never write to register 2.

o Write to register 9 only when performing a reset operation. To reset SCC Port A or Port B, write calue $8A or $4A to register 9 during **Dinit** and **Ddown**.

The DCD-B bit indicates the state of the signal on the wire it monitors. Never enable interrupts on DCD-B because this signal belongs to Port A although it is read on port B. (Refer to the description of register 15 in the *SCC Manual*). When Port A is in synchronous or SDLC modes, it may read DCD-B to determine the state of signal DSR on Port A's connector (not available on Lisa 1.0).

Set RTS-B to 0 except when using Applebus.

4.3 THE 6522 PARALLEL PORT

You can the cycle times provided in this section to calculate timing for the pulse mode handshake and to compute data throughput rates. On Lisa 1.0, the 5.09375 MHz clock time is therefore 1963.19 nanoseconds. The same 1963.19 nanosecond clock drives the 6522s on the Sony disk-drive card and the internally released four-port card. The cycle time for the 6522s on Lisa 2.0's I/O board, and the two-port card is 785.276 nanoseconds (derived from 68000 cycle time divided by 4). To summarize:

	LISA 1.0	LISA 2.0
I/O board 6522	1963.19 ns	785.276 ns
2-port card	785.276 ns	785.276 ns
4-port card	1963.19 ns	1963.19 ns

Sony 6522	1963.19 ns	1963.19 ns

4.3.1 Using the 6522 Pulse Mode Handshake

When outgoing data is on the data bus, a pulse on a separate handshake line indicates that the data is ready for the receiver to read. The pulse mode is used to strobe outgoing data to a peripheral device or to acknowledge receipt of incoming data from a peripheral device. It is possible, however, for the 68000 to drive the 6522 faster than it can handle the pulse handshake, causing the data to be accessed before completion of the pulse from the previous access. Calculation of the minimum number of 6522 cycles allowed between read or write commands that utilize pulse mode handshake is shown below, using a Profile driver as an example.

For writing to a Profile, the number of 6522 cycles between consecutive writes is n, where n must satisfy the inequality:

n * clock-rate > T(dcw) + 1.5 * clock-rate + T(rs3) + device-latency

where T(dcw) = 300 ns, as stated in the *6522 Data Sheet*;
T(rs3) = 1000 ns, as stated in the *6522 Data Sheet*;
clock-rate = 1963.19 ns or 785.276 ns, depending on which 6522 is being used;
device-latency = 100 ns for Profile.

For reading from a Profile, the number of 6522 cycles between consecutive writes is n, where n must satisfy the inequality:

n * clock-rate > T(pcr) + 1.5 * clock-rate + T(rs1) + device-latency

where T(pcr) = 300 ns, as stated in the *6522 Data Sheet*;
T(rs1) = 1000 ns, as stated in the *6522 Data Sheet*;
clock-rate = 1963.19 ns or 785.276 ns, depending on which 6522 is being used;
device-latency = 600 ns for Profile.

Thus for both reading from and writing to a Profile, n >= 4 on the fast 6522 and n >= 3 on the slow 6522. Four cycles of the fast 6522 can be generated by separating consecutive reads or writes by between 10 and 13 cycles of the 68000 (inclusive of the read or write command). Likewise, three cycles of the slow 6522 can be generated by separating consecutive reads or writes by between 14 and 21 cycles of the 68000. Therefore to code a Profile driver that will accomodate both fast and slow 6522s, insert instructions requiring at least 14 cycles of the 68000 between consecutive reads and writes within the driver.

4.3.2 Parallel Port Differences

The Lisa contains one built-in parallel port. You may also connect a two-port card containing two parallel ports or an internally released four-port card containing three parallel ports plus a fourth non-standard port. Each parallel port is controlled by a 6522. The differences between the ports are highlighted here. The remarks below assume familiarity with the 6522 data sheet and the *Lisa Hardware Manual* (Sections 2.5.3, 3.5.2, and 6.5).

On the built-in port, the CHK signal enters as a keyboard event; on the two-port or four-port card, the CHK signal enters on the CB1 line of each port.

The diagnostic PARITY line is not accessible on the built-in parallel port but may be read on PB6 on either port of the two-port card or the first two ports of the four-port card.

When writing to a register of a 6522 parallel port, it is important to change only the bits you want to set. Therefore, the instructions **ANDI.B**, **ORI.B**, **BSET**, and **BCLR** should be used instead of **MOVE.B** whenever writing to registers **ORB**, **DDRB**, **ACR**, and **PCR**. **IFR** and **IER** are discussed below.

The following bits should never be changed on the built-in parallel port:

Register	Bits
DRB	5, 6, 7
DDRB	5, 6, 7
T1C-L	all
T1C-H	all
T1L-L	all
T1L-H	all
SR	all (never read this register either)
ACR	2, 3, 4, 6, 7
PCR	4
IFR	2, 6
IER	0, 2, 3, 4, 6

Bits 0 through 4 of a parallel port's B register are used differently, depending upon whether a ProFile disk or a Centronics-compatible dot matrix printer (such as the C.Itoh 8510A) is attached. All parallel ports treat bits 0 through 4 similarly. Following is a description of the use of these bits, as well as **CA1**, **CA2**, and **CB2** of the **IFR** register.

Bit #	Direction	ProFile	Parallel Printer
PB0	in	cable detect (1 = disconnect)	paper empty (1 = PE)
PB1	in	BSY state (0 = busy)	acknowledge (0 = ACK)
PB2	out	set = 0 always	set = 0 always
PB3	out	read (= 1)/write(= 0) dir.	set = 1 always
PB4	out	CMD (0 = CMD true)	don't care
CA1*	in	BSY latch (from PB1)	ACK latch (from PB1)
CA2	out	data strobe	data strobe
CB2	in	parity error latch	ignored

* Bit 0 of PCR controls whther CA1 will latch rising ot falling edge of PB1.

The CB2 parity error latch that will be cleared by any read or write to the B register. Therefore check disk parity *before* making any reference to the B register.

Writing a one to a bit that is already one in the 6522 **IFR** and **IER** registers may change the bit to zero. Therefore **MOVE.B** is the only instruction that should be used when writing to **IFR** or **IER**. Do not use the commands **AND**, **ANDI**, **OR**, **ORI**, **BSET**, or **BCLR**.

The 6522 that controls the Lisa's keyboard has some of the signals which belong to the built-in parallel port. Table 4-1 describes the use of the following bits on the keyboard 6522 as they pertain to the parallel port: **PB5** and **PB7** (bits 5 and 7 on the B register) and **DDRB5** and **DDRB7** (bits 5 and 7 of data direction register B). The adress pairs given in parentheses in the table result in references to the same location. This is because address bits 6, 7, and 8 were used in early Lisa prototypes but are currently ignored.

Table 4-1

Bit Settings with Disk Attached

	DDRB7	PB7	DDRB5	PB5
Built-in Parallel:				
	leave as input (FCD811/FCD911)	ignore (FCD801/FCD901)	leave as input (FCD811/FCD911)	ignore (FCD801/FCD901)
6522 Keyboard:				
	output (FCDC05/FCDD85)	set to 1 (FCDC01/FCDD81)	output (FCDC05/FCDD85)	normally 1, 0 resets parity (FCDC01/FCDD81)
Two-port/four-port Parallel:				
	output (FCXX11)	set to 1 (FCXX01)	output (FCXX11)	normally 1, 0 resets parity (FCXX01)

Bit Settings with Printer Attached

	DDRB7	PB7	DDRB5	PB5
Built-in Parallel:				
	leave as input (FCD811/FCD911)	ignore (FCD801/FCD901)	leave as input (FCD811/FCD911)	ignore (FCD801/FCD901)
6522 Keyboard:				
	output (FCDC05/FCDD85)	set to 1 (FCDC01/FCDD81)	input (FCDC05/FCDD85)	SELECT(1=online) (FCDC01/FCDD81)
Two-port/four-port Parallel:				
	output (FCXX11)	set to 1 (FCXX01)	output (FCXX11)	SELECT (1=online) (FCXX01)

To determine the value of XX in addresses FCXX01 and FCXX11 in Table 4-1, consult Table 4-2. Channel 1 refers to the bottom XX; channel 2 refers to the top XX; channel 3 is accessible with the back off on a four-port card only.

Table 4-2

	chan 1	chan 2	chan 3
slot 1 (outer)	20	28	30
slot 2 (middle)	60	68	70
slot 3 (inner)	A0	A8	B0

4.4 PERIPHERAL CARDS

The Lisa's three I/O slots support peripheral cards as well as devices. If a card can support more than one device, a demultiplexing driver is required for the card and one driver is required for each type of device supported by the card. This section describes hardware requirements for Lisa peripheral cards.

Each peripheral device (or its controller) should have a separate *interrupt-enable latch* that operates under software control. When disabled, this latch should hold pending any interrupt, without signaling the interrupt. When the latch is enabled, any previously pending interrupts should be signaled. (The Interrupt Enable Register on the 6522 VIA provides this function.) In addition, a system reset must disable the device controller so that no interrupts are possible until the driver has enabled interrupts on the device.

Any operation that may take the device more than about one millisecond should cause an interrupt upon completion. The driver should poll for this completion condition.

Writing a device driver can be greatly simplified if the device handshake sequence has only one point at which an interrupt is required. For example, it is preferable to have the driver wait for **SEEK** and **WRITE** at the same time on a disk device, rather than waiting for the **SEEK** to complete before being able to start up the **WRITE**.

Disconnecting the cable to a device should cause an interrupt that the driver can respond to. If this is not possible, the controller should provide an interval timer interrupt to allow the driver to poll for disconnection.

Any hardware configuration parameters should operate under software control.

If the device is bootable, its controller must support the conventions adopted in the Lisa Boot ROM.

Each Lisa peripheral card must be self-identifying.

4.5 PRINTERS

Printer programs execute as a shared intrinsic unit within the applications. The printer programs are responsible for creating the appropriate sequence of ASCII bytes to hand off to the OS using **WRITE_DATA**. By calling **MOUNT**, **UNMOUNT**, **OPEN**, **CLOSE_OBJECT**, **WRITE_DATA**, and **DEVICE_CONTROL**, printer programs can communicate with a serial or parallel interface port on the Lisa. *Special printer drivers that control letter quality, dot matrix, laser, and plotter printing are not within the scope of this manual.*

The serial or parallel interface driver within the Operating System is responsible only for making sure that the data is sent to the printer without overrunning the device. Handshake protocols provided by the OS include:

o Delay after CR, LF.
o Hardware handshake.
o XON/XOFF software handshake.

Through **DEVICE_CONTROL**, the handshake protocol, the modem control, parity, byte length, baud rate, and other device characteristics of the printer can be set by the printer program.

4.6 BUILT-IN DEVICES

Drivers for the mouse, speaker, interval timer, alarm, and clock/calendar are built in and are not configurable. Drivers for built-in disk drives, built-in serial ports, keyboard, and screen are configurable, since these devices can be accessed using **READ_DATA** or **WRITE_DATA**.

Chapter 5
SPECIFYING CHARACTERISTICS FOR A NEW DRIVER

5.1 The Installation Disk
5.2 The CDCHAR Program
5.3 The Preferences Tool

SPECIFYING CHARACTERISTICS FOR A NEW DRIVER

5.1 THE INSTALLATION DISK

The installation disk is used by the end user in conjunction with the Preferences tool to install a driver for a new device. The installation disk contains the driver code file, the Characteristics file, and for bootable devices only, a loader file. This software must all be contained on a single installation disk.

The *driver code file* is the configurable driver program you have written to support a new device or group of devices.

The *Characteristics file* contains information about your configurable driver that is needed by the Lisa Operating System. You create the Characteristics file by means of the CDCHAR program described in this chapter.

The *loader file* contains the program for booting the system from the new device; it should have the same name as the driver code file with extension .BT.

Figure 5-1 shows the steps necessary to make a configurable driver available on a new system.

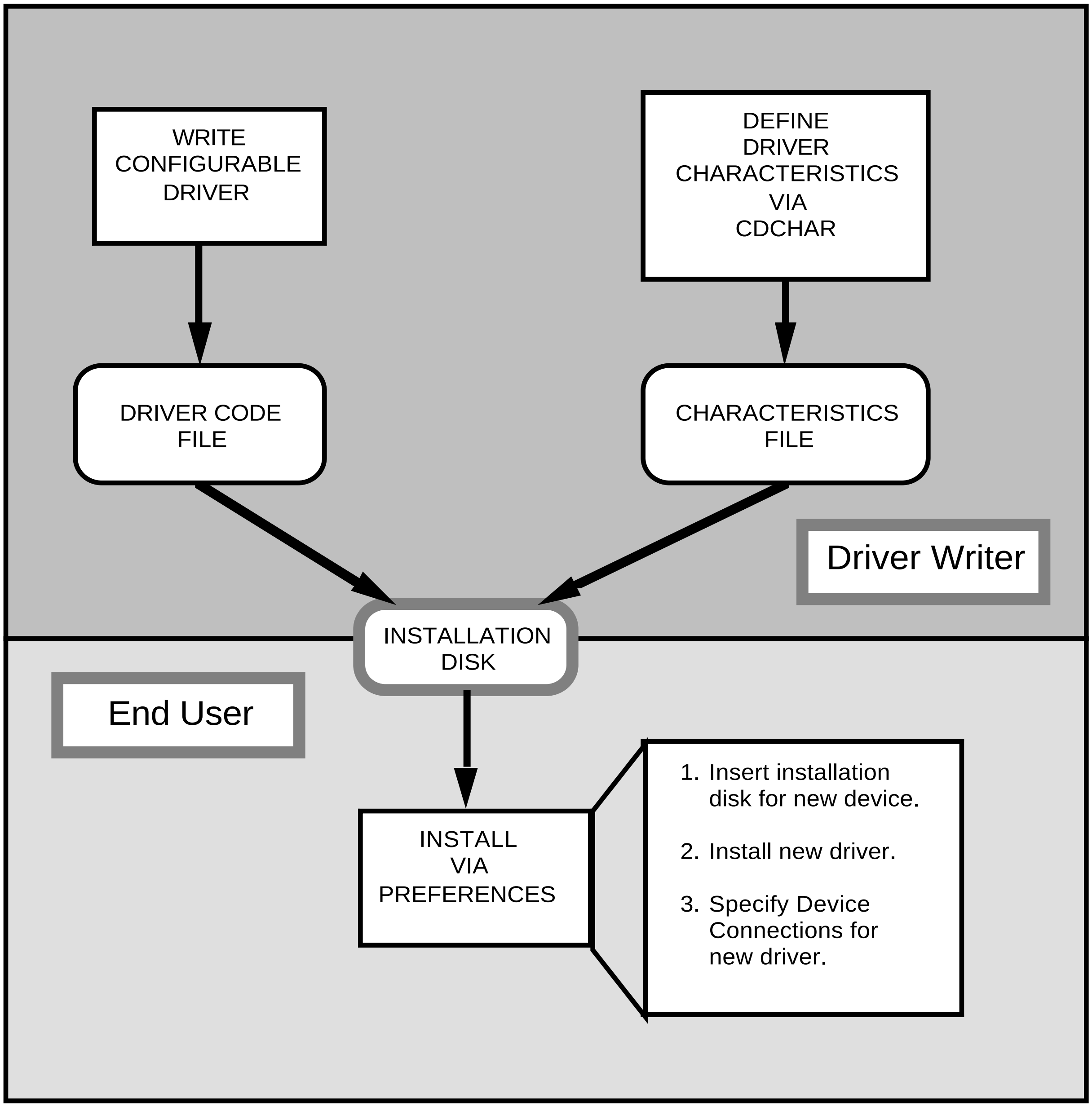

WRITE CONFIGURABLE DRIVER
DEFINE DRIVER CHARACTERISTICS VIA CDCHAR
DRIVER CODE FILE
CHARACTERISTICS FILE
Driver Writer
INSTALLATION DISK
End User
INSTALL VIA PREFERENCES
1. Insert installation disk for new device.
2. Install new driver.
3. Specify Device Connections for new driver.

FIGURE 5-1

Figure 5-1. Specifying Driver Characteristics.

5.2 THE CDCHAR PROGRAM

The CDCHAR program asks you questions about your configurable driver and saves the answers in the Characteristics file, which is used by the end user when installing the configurable driver.

The CDCHAR program is supplied with the Lisa Operating System. To run the program, use the Run command in the Workshop. The CDCHAR dialog is as follows:

Driver name?

Respond with the name you want the end user to see in the Device Connections table of the Preferences tool; for example,

Laser Printer

The driver name may contain spaces; it must not be more than 22 characters long. Note that two or more devices with the same driver name may not be configured on the same Lisa.

Unique id?

Supply the permanent identification number for your driver. This number must be assigned to you by Apple Computer. (Numbers 1 through 128 are freely available to you for the purpose of testing your driver.)

Any default info for PM's extension words?

Indicate yes or no. If yes, you are prompted for up to three words of device-specific information that will be stored in parameter memory whenever the configurable driver is loaded. Each word is a number between 0 and 65535. This information is optional and is ignored by the Operating System. The end user can override this information at installation time by means of the Preferences tool.

Device type [1 = sequential, 2 = disk, 3 = demultiplexing]?

Specify whether your device is sequential, disk, or demultiplexing.

Is the device bootable?

Indicate ues or no. If yes, supply a loader file on the installation disk; the loader file contains the program for booting the system from the new device. (This question is asked only for disk devices.)

Removable media?

Indicate yes or no. (This question is asked only for disk devices.)

Ejectable media?

Indicate yes or no. If yes, the driver program must support the Dskunclamp function code. (This question is asked only for disk devices with removable media.)

Address of boot track blocks?

Indicate the block number where the boot tracks should continue. If you specify zero, the boot tracks do not skip any blocks before beginning the first File System block. (This question is asked only for disk devices.)

Address of the first File System block?

Indicate the absolute block number of the first File System block. (This question is asked only for disk devices.)

Should the driver be preloaded?

Indicate yes or no. If the driver is preloaded -- loaded at boot tiem -- the first acess of the device will be slightly faster. (This question is nor asked for demultiplexing drivers or for disk drivers with removable disks.)

Should the driver stay resident, once loaded?

Indicate yes or no. If yes, the driver remains in memory when the device is unmounted. (This question is not asked for demultiplexing drivers or for disk drivers with removable disks.)

CDCHAR now lists the data you have entered so that you can verify it.

Is all this correct?

If no, you may enter the data again. If yes, the Characteristics file is created.

Characteristics file destination disk (e.g, -MYDISK)?

Specify the name of the installation disk.

5.3 THE PREFERENCES TOOL

The Preferences tool is used by the end user to install a configurable driver, using the installation disk described above. To install a new device, the end user must follow these steps:

1. If the device requires one or more new hardware cards, power down the Lisa and insert the cards.

2. Power up the Workshop or the Office System and open Preferences.

3. Insert the installation disk in the built-in microdisk slot.

4. Let Preferences install the driver on the installation disk.

Chapter 6
HOW TO MAKE A DEVICE DRIVER

- 6.1 Editing
- 6.2 Compiling and Assembling
  - 6.2.1 Assembler Language Considerations
  - 6.2.2 Pascal Considerations
- 6.3 Linking
- 6.4 Installing a New Driver
  - 6.4.1 Installing the OS
  - 6.4.2 Copying Files
  - 6.4.3 Running the Build Macro
  - 6.4.4 Installing the Driver
- 6.5 Testing and Debugging
  - 6.5.1 TRACE Function
  - 6.5.2 Getting Control during OS Startup

HOW TO MAKE A DEVICE DRIVER

6.1 EDITING

Use the Lisa Workshop for writing and editing your driver.

6.2 COMPILING AND ASSEMBLING

This section discusses assembler language and Pascal conventions for drivers.

6.2.1 Assembler Language Considerations

If your driver contains assembler language code, the Workshop Assembler can generate the object files needed for linking. All assembler language programs must conform to the Pascal parameter-passing and register-usage conventions described in the Assembler chapter of the Workshop manual (Sections 6.3 and 6.6). The declaration of an EXTERNAL assembler language routine in your driver must occur within the IMPLEMENTATION portion of the unit, at the outermost level (not within a subroutine). We recommend that you prefix all object-code file names with **OBJECT/**.

6.2.2 Pascal Considerations

A device driver must contain a Pascal unit like the one in Figure 6-1. If you code your driver in assembler language for efficiency, the assembler language driver must perform as though it were this Pascal program.

```
----------------------------------------------------------------
UNIT <driver unit name>;

INTERFACE
  USES
  (*$U object/driverdefs.obj*)
          driverdefs,
  (*$U object/driversubs.obj*)
          driversubs;

          function DRIVER (parameters: param_ptr): integer;

IMPLEMENTATION
  CONST ...
  TYPE ...
  <declarations of external assembler-language subroutines, if any>
  <internal Pascal subroutines go here>

          function DRIVER;
            CONST ...
            TYPE ...
            VAR ...
            <internal Pascal subroutines may go here>

          begin
                  DRIVER := 0;     (*no errors yet*)
                  case parameters^.fnctn_code of
                    dinterrupt: ...
                    dinit: ...
                    ddown: ...
                    dskunclamp: ...
```

```
                dskformat: ...
                seqio: ...
                dskio: ...
                dcontrol: ...
                reqrestart: ...
                dattach: ...
                dunattach: ...
                ddsicon: ...
                dalarms: ...
                hdinit: ...
                hddown: ...
                hdskio: ...
                otherwise DRIVER := <error code>
              end (* case *)
        end; (* driver *)
end. (* unit *)
--------------------------------------------------------------------
```

Figure 6-1. A Skeleton Driver Program.

For Pepsi release only, every driver must be linked with a dummy main program shown in Figure 6-2. This program is never executed; it is required so that the Linker will mark your configurable driver as an executable program.

```
--------------------------------------------------------------------
PROGRAM DUMMYMAIN;
  USES
        (*$Uobject/driverdefs.obj*)
          driverdefs,
        (*$Uobject/driversubs.obj*)
          driversubs,
        (*$Uobject/your-driver-unit's-file.obj*)
          your-unit-name,
  VAR
        i: integer;
        p: param_ptr;
  begin
        i := driver(p);
  end.
-------------------------------------------------------------------
```

Figure 6-2. A Dummy Main Program.

To limit the size of the Pascal library (osint.paslib.obj) linked with each driver, limit your use of Pascal functions to the following:

> **ABS, CHR, DIV, EXIT, MOD, MOVELEFT, ODD, ORD, ORD4, POINTER, PRED, SIZEOF, SUCC**

You can reduce the memory requirements of your driver by avoiding sets, console I/O, (**READ**, **READLN**, **WRITE**, and **WRITELN**), string procedures, and procedures dealing with packed array of char.

Pascal dynamic allocation (**NEW**), **real** variables, and general I/O routines provided by the Pascal language are not supported by the Pascal libraries linked with drivers.

Do not use the **$S** compiler directive. Your driver must reside entirely within the blank segment.

6.3 LINKING

Each driver is linked so that it is contained entirely, within the non-intrinsic blank segment, which is the default segment. The first object file linked with the driver is an assembler language file whose first instruction is a **JMP** to the driver function subroutine. The driver is also linked with a main Pascal program called **drivermain.obj** which, though never executed, tricks the Linker into resolving all references as thought the output of the linkage were a stand-alone, executable program.

Using the Workshop's Linker, link together the following modules: (**driverasm** must be first; the others may be in any order):

object/driverasm.obj
object/*your-driver-unit's-file*.obj
object/*your-other-driver-subroutines*.obj
object/*your-dummymain-file*.obj
object/osintpaslib.obj

6.4 INSTALLING A NEW DRIVER

This section describes the steps you must take in order to install a new configurable driver. Follow these steps in the sequence shown.

6.4.1 Installing the OS

Be sure you have version 8.1 or above of the Operating System. Then install this version on a test disk, *not on your Workshop boot disk*.

6.4.2 Copying Files

Move the following files from the test drive to your Workshop prefix volume.

build/assemb.text
build/comp.text
build/link.text
object/driverasm.obj
object/driverdefs.obj
object/driversubs.obj
object/osintpaslib.obj

Edit, rename, and copy the following files to your Workshop prefix volume.

build/rsgen.text
(example of compile/link macro for driver)
build/rslinklist.text
(example of Linker input list)
source/rs232main.text
(example of dummy main program)

6.4.3 Running the Build Macro

Run your new build macro (based on **build/rsgen.text**). A macro is a file containing commands to be executed by the Workshop.

6.4.4 Installing the Driver

Copy your driver to the test disk and boot from the test disk. Then follow these steps to replace an existing driver. An RS232 driver is given as an example. ***(Only RS232 drivers are configurable for Pepsi release.)***

If you are replacing an existing driver,

- Issue an **UNMOUNT** command for the device, RS232A.
- Rename the existing driver on the test disk, **system.cd_rs232a** to some temporary file name.
- Rename your driver to **system.cd_rs232a**, the name the existing driver had.
- Issue a **MOUNT** command for RS232A, the device your driver supports. Your new driver will be invoked.

Table 6-1 contains a list of devices supported under version 8.1 of the Lisa Operating System together with their driver name.

Table 6-1
Currently Supported Drivers (OS Version 8.1)

Device	**Driver Name**
Serial Port A	**system.cd_rs232a**
Serial Port B	**system.cd_rs232b**

6.5 TESTING AND DEBUGGING

You can debug your device driver by writing messages to the screen with **WRITELN**, by using LisaBug, or by a combination of these techniques. You can call **WRITELN** to display the contents of variables at various states in the execution of a driver.

You may want to turn these debugging features on and off. **WRITELN** commands can be *selectively executed* using the **TRACE** function described below or they can be *selectively compiled* by using compile-time conditional compilation directives described in the Workshop manual.

Other procedures and functions that are useful during program development are **MICROTIMER**, **LOG**, and **LOGGING**, discussed in Chapter 2.

6.5.1 TRACE Function

function* TRACE (Part: osportion; Level:integer):*boolean

Input Parameters:
Part: Always 'DD'.
Level: 0..99, value to compare to the trace fence.

Output Parameters:
TRACE: True if the trace fence is <= Level; else false.

Your driver can call the **TRACE** function to test a *trace fence* in the OS indicating whether **WRITELN** commands should be executed. The trace fence for device drivers is an integer located at location $220 from domain 0. (All drivers execute in domain 0.) You can change the trace fence at execution time by using the LisaBug commands Set Memory or Set Word.

The **TRACE** function returns **true** when the trace fence is set to a value less than or equal to **Level**. In debug statements, you want to appear in all or most instances, set **Level** to a high value. For device drivers, **Part** must contain the letters 'DD'.

The TRACE function is used in a Pascal statement like the following:

```
if TRACE('dd', 60) then WRITELN('debug message')
```

In the example above, if the trace fence is 60 or less, the debug message will be printed.

6.5.2 Getting Control During OS Startup

Some driver function codes are invoked during startup. To test them, you will need:

* OS version 5.4 (or later).
* CPU boot PROM $117 (or later).
* A copy of LisaBug (file **system.debug**) on your boot volume.

The OS loader chooses whether to bring the OS up in 'debug mode', based on the value of location $10000. The PROM normally initializes all of memory to -1, so a -1 in that cell means run without debug mode. In order to select debug mode, you must modify location $10000 to a value other than -1 before the loader gets control. Here's how:

1. Reboot from the PROM.

2. Press the Apple and numeric-key-pad Enter keys simultaneously at the point where you would normally request a boot device selection.

3. When the PROM presents a menu for 'restart', 'continue', or 'startup from', press the Apple and S keys simultaneously: this selects *service mode*.

4. Use the Set Memory command (by typing **2**) to store 0 in memory location $10000. The Display Memory command lets you see whether the debug-mode cell really has been changed.

5. Quit service mode by typing **7** or **9**, returning control to the menu described in step 3 above. Choose the 'startup from' option to boot from your boot volume.

6. The loader executes next. Just before it loads the main part of the OS, the loader emulates a level-7 interrupt to give you a chance to debug the loader. Type g to continue.

7. When it reaches the **DB_INIT** section of startup -- when OS symbols and labels are available -- the OS emulates another level-7 interrupt.

Now you can set breakpoints, examine system global variables, set trace fences, and perform any other LisaBug commands. Don't forget to set all breakpoints in terms of domain 0; for example:

```
>BR 0:my_label
```

Cross Reference Listing:

Beginning of file: DRIVERSUBS.TEXT

```

UNIT DRIVERSUBS;         { UNIT NEEDED BY CONFIGURABLE DRIVERS }

INTERFACE
  USES
       {$U object/driverdefs.obj}
         DRIVERDEFS;


  procedure CANCEL_REQ(req: reqptr_type);

  procedure ENQUEUE(var newlink, leftlink: linkage; b_sysarea: absptr);

  procedure DEQUEUE(var link: linkage; b_sysarea: absptr);

  procedure RELSPACE(ordaddr: absptr; b_area: absptr);

  function GETSPACE(amount: int2; b_area: absptr;
                                                             var ordaddr: absptr): boolean;

  procedure SYSTEM_ERROR(errnum: int2);

  function TRACE(part: osportion; level: integer) : boolean;

  procedure status_req(reqptr: reqptr_type;
                                                     var status: reqsts_type);

  procedure unblk_req(reqptr: reqptr_type;
                                                     success_f: boolean);

  procedure blk_req(reqptr_list: reqptr_type;
                                                     var first_reqptr: reqptr_type);

  procedure IODONE(port_ptr: hdiskcb_ptr; prev_err: integer);

  procedure FREEZE_SEG(var errnum: int2; c_sdb: absptr; offset: int4;
                                                     ioreq_addr: absptr; var buffaddr: absptr);

  procedure UNFREEZE_SEG(c_sdb: absptr);

  procedure ADJ_IO_CNT(inc_io_count: boolean; c_sdb: absptr);

  procedure CVT_BUFF_ADDR(var errnum: int2; mem_writeF: boolean; buffaddr: absptr;
                                           byte_cnt: int4; var ordsdb: absptr; var offset: int4);

  procedure CALLDRIVER(var errnum: integer; config_ptr: ptrdevrec;
                                           parameters: param_ptr);

  procedure INTSOFF(level: intsoff_type; var status: intson_type);

  procedure INTSON(status: intson_type);

  function ALLSET(b1, b2: int1): boolean;

  function LOGGING: boolean;

  procedure LOG(var errnum: integer; ptr_arr: longint);

  function CHAIN_FORWARD(current: linkage): reqptr_type;

  procedure DISKSYNC(busy: boolean);

  function MICROTIMER: longint;

  procedure ALARM_ASSIGN(var alarm: integer; pdr: ptrdevrec; status: intson_type);

  procedure ALARMRETURN(alarm: integer);

  procedure ALARMRELATIVE(alarm: integer; delay: longint);

  procedure ALARMOFF(alarm: integer);

  procedure LINK_TO_PCB(req_ptr: reqptr_type);

  procedure Use_Hdisk(configptr: ptrdevrec);

  procedure Call_Hdisk(var error: int2; configptr: ptrdevrec; parameters: param_ptr);

  function OKXFERNEXT(port_ptr: hdiskcb_ptr): integer;


IMPLEMENTATION

  procedure CANCEL_REQ;
  external;

  procedure ENQUEUE;
  external;

  procedure DEQUEUE;
  external;

  procedure RELSPACE;
  external;

  function GETSPACE;
  external;

  procedure SYSTEM_ERROR;
  external;

  function TRACE;
  external;

  procedure status_req;
  external;

  procedure unblk_req;
  external;

  procedure blk_req;
  external;

  procedure IODONE;
  external;

  procedure FREEZE_SEG;
  external;

  procedure UNFREEZE_SEG;
  external;

  procedure ADJ_IO_CNT;
  external;

  procedure CVT_BUFF_ADDR;
  external;

  procedure CALLDRIVER;
  external;

  procedure INTSOFF;
  external;

  procedure INTSON;
  external;

  function ALLSET;
  external;

  function LOGGING;
  external;

  procedure LOG;
  external;

  function CHAIN_FORWARD;
```

```
 1: 148.    external;
 1: 149.
 1: 150.    procedure DISKSYNC;
 1: 151.    external;
 1: 152.
 1: 153.    function MICROTIMER;
 1: 154.    external;
 1: 155.
 1: 156.    procedure ALARM_ASSIGN;
 1: 157.    external;
 1: 158.
 1: 159.    procedure ALARMRETURN;
 1: 160.    external;
 1: 161.
 1: 162.    procedure ALARMRELATIVE;
 1: 163.    external;
 1: 164.
 1: 165.    procedure ALARMOFF;
 1: 166.    external;
 1: 167.
 1: 168.    procedure LINK_TO_PCB;
 1: 169.    external;
 1: 170.
 1: 171.    procedure Use_Hdisk;
 1: 172.    external;
 1: 173.
 1: 174.    procedure Call_Hdisk;
 1: 175.    external;
 1: 176.
 1: 177.    function OKXFERNEXT;
 1: 178.    external;
 1: 179.
 1: 180.  end.
 1: 181.

End of File: DRIVERSUBS.TEXT


Directory of files in Cross Reference:

 1: DRIVERSUBS.TEXT
level =  1.

absptr  1:         12,   1:  14,     1:       16,   1:  16,     1:       18,   1:  19,     1:       36,   1:  37,
          1:       37,   1:  39,     1:       41,   1:  43,     1:       44.
alarm   1:         65*,  1:  67*,    1:       69*,  1:  71*.
amount  1:         18*.
b1        1:       53*.
b2        1:       53*.
b_area  1:         16*,  1:  18*.
b_sysare 1:        12*,  1:  14*.
buffaddr 1:        37*,  1:  43*.
busy    1:         61*.
byte_cnt 1:        44*.
c_sdb   1:         36*,  1:  39*,    1:       41*.
config_p 1:        46*.
configpt 1:        75*,  1:  77*.
current 1:         59*.
delay   1:         69*.
driverde 1:         7.
driversu 1:         2.
errnum  1:         21*,  1:  36*,    1:       43*,  1:  46*,    1:       57*.
error   1:         77*.
external 1:        85,   1:  88,     1:       91,   1:  94,     1:       97,   1: 100,     1: 103,   1: 106,
          1: 109,   1: 112,          1: 115,   1: 118,          1: 121,   1: 124,          1: 127,   1: 130,
          1: 133,   1: 136,          1: 139,   1: 142,          1: 145,   1: 148,          1: 151,   1: 154,
          1: 157,   1: 160,          1: 163,   1: 166,          1: 169,   1: 172,          1: 175,   1: 178.
first_re 1:        32*.
hdiskcb_ 1:        34,   1:  79.
implemen 1:        82.
inc_io_c 1:        41*.
int1    1:         53.
int2    1:         18,   1:  21,     1:       36,   1:  43,     1:       77.
int4    1:         36,   1:  44,     1:       44.
interfac 1:         4.
intsoff_ 1:        49.
```

```
intson_t 1:        49,   1:  51,     1:       65.
ioreq_ad 1:        37*.
leftlink 1:        12*.
level   1:         23*,  1:  49*.
link    1:         14*.
linkage 1:         12,   1:  14,     1:       59.
mem_writ 1:        43*.
newlink 1:         12*.
offset  1:         36*,  1:  44*.
ordaddr 1:         16*,  1:  19*.
ordsdb  1:         44*.
osportio 1:        23.
param_pt 1:        47,   1:  77.
paramete 1:        47*,  1:  77*.
part    1:         23*.
pdr      1:        65*.
port_ptr 1:        34*,  1:  79*.
prev_err 1:        34*.
ptr_arr 1:         57*.
ptrdevre 1:        46,   1:  65,     1:       75,   1:  77.
req      1:        10*.
req_ptr 1:         73*.
reqptr  1:         25*,  1:  28*.
reqptr_l 1:        31*.
reqptr_t 1:        10,   1:  25,     1:       28,   1:  31,     1:       32,   1:  59,     1:       73.
reqsts_t 1:        26.
status  1:         26*,  1:  49*,    1:       51*,  1:  65*.
success_ 1:        29*.
unit    1:          2.

Procedures

adj_io_c 1:        41*,  1: 123*.
alarm_as 1:        65*,  1: 156*.
alarmoff 1:        71*,  1: 165*.
alarmrel 1:        69*,  1: 162*.
alarmret 1:        67*,  1: 159*.
blk_req 1:         31*,  1: 111*.
call_hdi 1:        77*,  1: 174*.
calldriv 1:        46*,  1: 129*.
cancel_r 1:        10*,  1:  84*.
cvt_buff 1:        43*,  1: 126*.
dequeue 1:         14*,  1:  90*.
disksync 1:        61*,  1: 150*.
enqueue 1:         12*,  1:  87*.
freeze_s 1:        36*,  1: 117*.
intsoff 1:         49*,  1: 132*.
intson  1:         51*,  1: 135*.
iodone  1:         34*,  1: 114*.
link_to_ 1:        73*,  1: 168*.
log      1:        57*,  1: 144*.
relspace 1:        16*,  1:  93*.
status_r 1:        25*,  1: 105*.
system_e 1:        21*,  1:  99*.
unblk_re 1:        28*,  1: 108*.
unfreeze 1:        39*,  1: 120*.
use_hdis 1:        75*,  1: 171*.

Functions

allset  1:         53*,  1: 138*.
chain_fo 1:        59*,  1: 147*.
getspace 1:        18*,  1:  96*.
logging 1:         55*,  1: 141*.
microtim 1:        63*,  1: 153*.
okxferne 1:        79*,  1: 177*.
trace   1:         23*,  1: 102*.

Declaration Character : '*'
Assignment Character :  '='
```

Cross Reference Listing:

Beginning of file: GENIO.TEXT

```
UNIT genio;
                                                               { general I/O driver }


                  { By Dave Offen }
                  { Copyright 1983, Apple Computer Inc. }
                  { Modified 4/2/83 by Wendell Henry        }
                  {                Add CALLDRIVER interface          }

INTERFACE

  USES
       {$Uobject/driverdefs.obj}
         driverdefs,
       {$Uobject/hwint.obj}
         hwint,
       {$Uobject/sysglobal.obj}
         globalda,
       {$Uobject/procprims.obj}
         proc_prims,
       {$Uobject/mmprim.obj}
         mmprimitives,
       {$Uobject/asynctr.obj}
         asynctr;

  TYPE
       drvrec = record
                                 driver_id: longint;        { as a debugging aid }
                                 version: integer;                   { driver's version number }
                                 sem: semaphore;            { locks access to drvrec }
                                 addrdrivrname: absptr;     { for loading driver (ptr to pathname) }
                                 nusers: 0..32767;                   { use count for driver - 0 if unloaded }
                                 e_pt: absptr;                       { driver address - undefined if nusers=0 }
                                 kres_addr: absptr;                  { driver address if kernel-resident. nil
                                                                                         if requires loading off disk
}
                          end;

  procedure LINK_TO_PCB(req_ptr: reqptr_type);

  procedure CALLDRIVER(var errnum: integer; config_ptr: ptrdevrec;
                                                     parameters: param_ptr);

  procedure UP(var errnum: integer; config_ptr: ptrdevrec;
                                 callers_config_ptr: ptrdevrec);

  procedure DOWN(var errnum: integer; config_ptr: ptrdevrec;
                                 callers_config_ptr: ptrdevrec; enter_at: integer);

  procedure ALARM_ASSIGN(var alarm: integer; pdr: ptrdevrec;
                                 status: intson_type);

  procedure ALARM_FIRES(alarm: integer);

  procedure CANCEL_REQ(req: reqptr_type);

  procedure DRIVERCALL(var errnum: integer; dev_index: integer;
                                 parameters: param_ptr);

  procedure SEQENTIO(var errnum: integer; dev_index: integer; io_buff_addr: absptr;
                                           length: longint; read_f: boolean; pcb_ptr: ptr_pcb;
                                           var req_ptr: reqptr_type);

  procedure DISKIO(var errnum: integer; dev_index: integer; var header: pagelabel;
                                           extra_link: int4; extra_data_used: integer; io_buff_addr: absptr;
                                           length: integer; block_no: int4; read_f: boolean;
                                           mode: disk_io_type; pcb_ptr: ptr_pcb; var req_ptr: reqptr_type);

  function CHAIN_FORWARD(current: linkage): reqptr_type;

  function BADCALL(parameters: param_ptr): integer;
```

```

    function SCC(parameters: param_ptr): integer;

  IMPLEMENTATION

    {$S krgenio}

    {$IFC not debug2}
    {$R-} { rangecheck off unless debug mode }
    {$ENDC}



    CONST
         errbase = 600;
         syserrbase = 10600;

    TYPE
         ptrpathname = ^pathname;  { s/b USES }

    VAR { Yes, it's global }
         alarm_table: array[0..20] { keyed to # of alarms in HWINT } of record
                                             cfg_ptr: ptrdevrec;
                                             intstat: intson_type;
                                             filler: integer; { makes record a power of 2 - fast subscripts }
                                      end;

    procedure LOAD_DRIVER(var err: integer; var add: absptr; name: ptrpathname);
                                                     external; { s/b USES }

    procedure UNLOAD_DRIVER(var err: integer; add: absptr); external; { s/b USES }

    procedure CALLDRIVER; external;

    procedure ALRM; external;

    procedure TWIGIO(var er: integer; var req: reqblk; addr: int4); external;

    procedure CANCEL_REQ(* req: reqptr_type *);
     {***********************************************************}
     {*
*}
     {* Description: Dispose of request block & unlink from PCB *}
     {*
*}
     {* Input Parameters: Pointer to request block                          *}
     {*
*}
     {* Output Parameters: none                                                                  *}
     {*
*}
     {* Side Effects: none                                                                       *}
     {*
*}
     {* Special Conditions of Use: none                                                   *}
     {*                                                                         *}
     {* Error Conditions:                                                                          *}
     {*
*}
     {***********************************************************}


   VAR
     prevints: intson_type;
     pblk_in_pcb: ptrblk_type;

   begin
     INTSOFF(allints, prevints);

     with req^ do
     begin
            DEQUEUE(pcb_chain.header, b_sysglobal_ptr);
            INTSON(prevints);
            pblk_in_pcb := @blk_in_pcb;
            if pblk_in_pcb^ = [i_o] then
              with cfigptr^ do
```

```
                  if permreq_ptr = req then
                  begin
                    preq_avail := true;
                    EXIT(cancel_req) { permanent request doesn't get deleted }
                  end
  end;
  RELSPACE(ord(req), b_sysglobal_ptr)
end; { cancel_req }


procedure DISKIO;
        {***********************************************************}
        {*
*}
        {* Description:  Start up or queue a disk read or write        *}
        {*
*}
        {* Input Parameters: You wouldn't believe it if I told you. *}
        {*
*}
        {*
*}
        {*
*}
        {* Output Parameters: Req_ptr is the address of the newly      *}
        {*       created request block.                                                                    *}
        {*
*}
        {* Side Effects: Device dependent driver is invoked.           *}
        {*
*}
        {* Special Conditions of Use: none                                                        *}
        {*
*}
        {* Error Conditions:                                                                                        *}
        {*
*}
        {***********************************************************}

VAR
  p: params;
  d_ptr: absptr;
  ext_ptr: extdptr_type;
  pblk_in_pcb: ptrblk_type;
  ext_config: ^ext_diskconfig;
  leftlink: link_ptr;
  bytes: longint;
  prevints: intson_type;

begin
  errnum := 0;
  ext_config := pointer(configinfo[dev_index]^.ext_addr);
  if (block_no < 0) or (length <= 0) or
          ((block_no + length) > ext_config^.num_bloks) or
          ((mode = chained_hdrs) and not read_f) or
          (read_f and (mode = with_header)) then
         errnum := errbase + 8 { illegal mode, start addr or transfer length }
  else
  if not GETSPACE(sizeof(reqblk)+sizeof(disk_extend), b_sysglobal_ptr, d_ptr) then
         errnum := errbase + 10;

  if errnum = 0 then
  begin
         req_ptr := pointer(d_ptr); { point to the request }
         ext_ptr := pointer(d_ptr + sizeof(reqblk)); { point to the extension area }

         { initialize the request block }

         with req_ptr^ do
         begin
           pcb_chain.kind := reqblk_type;
           reqstatus.reqsrv_f := active;
           reqstatus.reqabt_f := false;
           pblk_in_pcb := @blk_in_pcb;
           pblk_in_pcb^ := [i_o];
           block_p_f := false;
           hard_error := 0;
           operatn := ord(read_f);
           cfigptr := configinfo[dev_index];
           req_extent := ord(ext_ptr); { point the request block to the extension }
         end;

         { transfer parameters to disk extension data block }

         with ext_ptr^ do
         begin
           if (mode = without_header) and not read_f then
                  with soft_hdr do
                  begin { zero out the header when writing w/o headers }
                    version := 0;
                    datastat := dataok;
                    filler := 0;
                    volume := 0;
                    fileid := 0;
                    dataused := 0;
                    abspage := 0;
                    relpage := 0;
                    fwdlink := 0;
                    bkwdlink := 0
                  end
           else
                  soft_hdr := header;
           last_fwd_link := extra_link;
           last_data_used := extra_data_used;
           read_flag := read_f;
           blkno := block_no + ext_config^.strt_blok;
           io_mode := mode;
           num_chunks := length;
           xfer_count := 0;
           if io_mode = raw_io then
                  bytes := 536
           else
                  bytes := 512;
           CVT_BUFF_ADDR(errnum, read_f, io_buff_addr, bytes*num_chunks,
                          buff_rdb_ptr, buff_offset);
         end;

         if errnum > 0 then
           RELSPACE(ord(req_ptr), b_sysglobal_ptr)
         else
         begin   { continue if no errors yet }
           ADJ_IO_CNT(true, ext_ptr^.buff_rdb_ptr); { prevent swapout }

           { add request to pcb chain }

           INTSOFF(allints, prevints);
           leftlink := pointer(pcb_ptr^.req_chain.header.bkwd_link + b_sysglobal_ptr);
           ENQUEUE(req_ptr^.pcb_chain.header, leftlink^, b_sysglobal_ptr);
           INTSON(prevints);

           { call appropriate disk driver }

           p.fnctn_code := dskio;
           p.req := req_ptr;
           CALLDRIVER(errnum, req_ptr^.cfigptr, @p); { call dskio }
           if errnum > 0 then
           begin
                  ADJ_IO_CNT(false, ext_ptr^.buff_rdb_ptr); { allow swapout on errors }
                  CANCEL_REQ(req_ptr);
           end
         end
  end
end; { diskio }



procedure DRIVERCALL(* var errnum: integer; dev_index: integer;
                          parameters: param_ptr *);

        {***************************************************************}
        {*
*}
```

```
        {* Description:  Device-independent driver call                          *}
        {*
*}
        {* Input Parameters: Index into configinfo, and call-                    *}
        {*                specific parameter block                                                         *}
        {*
*}
        {* Output Parameters: Only through pointers in parameter blk *}
        {*
*}
        {* Side Effects: Driver routine dependent.                                        *}
        {*
*}
        {* Special Conditions of Use: none                                                         *}
        {* 
*}
        {* Error Conditions:                                                                                     *}
        {*
*}
        {*************************************************************}

begin
  CALLDRIVER(errnum, configinfo[dev_index], parameters)
end; { drivercall }


procedure LINK_TO_PCB(* req_ptr: reqptr_type *);

        {*************************************************************}
        {*
*}
        {* Description:  link passed request to current process'        *}
        {*                ctrl blk and initialize all but respec_info fields     *}
        {*
*}
        {* Input Parameters: Pointer to request block                                     *}
        {*
*}
        {* Output Parameters: none                                                                                   *}
        {*
*}
        {* Side Effects:                                                                                             *}
        {*
*}
        {* Special Conditions of Use: none                                                         *}
        {*
*}
        {* Error Conditions:                                                                                     *}
        {*
*}
        {*************************************************************}
VAR
  prevints: intson_type;
  cur_pcb_ptr: ptr_pcb;
  leftlink: reqptr_type;
  pblk_in_pcb: ptrblk_type;

begin
  with req_ptr^ do
  begin
         pcb_chain.kind := reqblk_type;
         reqstatus.reqsrv_f := active;
         reqstatus.reqabt_f := false;
         pblk_in_pcb := @blk_in_pcb;
         pblk_in_pcb^ := [i_o];
         block_p_f := false;
         hard_error := 0;
         INTSOFF(allints, prevints);
         cur_pcb_ptr := pointer(c_pcb_ptr);
         leftlink := pointer(cur_pcb_ptr^.req_chain.header.bkwd_link +
                          b_sysglobal_ptr);
         ENQUEUE(pcb_chain.header, leftlink^.pcb_chain.header,
                          b_sysglobal_ptr);
         INTSON(prevints)
  end
end; { link_to_pcb }
```

```


 {$S fs3}

 procedure DOWN(* var errnum: integer; config_ptr: ptrdevrec;
                            callers_config_ptr: ptrdevrec; enter_at: integer *);

        {*************************************************************}
        {*
*}
        {* Description:  Undo what UP did                                                          *}
        {*
*}
        {* Input Parameters: Pointer to configinfo record of device  *}
        {*       to be DOWNed, pointer to hierarchically lower configinfo *}
        {*       (or nil if none lower); enter_at allows for several entry*}
        {*       points (0 is standard value except when called from UP         *}
        {*
*}
        {* Output Parameters: none                                                                             *}
        {*
*}
        {* Side Effects:  UNLOAD_DRIVER, dunattach and ddown may be  *}
        {*        called. Semaphore is cleared when enter_at > 1.                *}
        {*
*}
        {* Special Conditions of Use: none                                                         *}
        {*
*}
        {* Error Conditions:  only error is: I/O still in progress      *}
        {*
*}
        {*************************************************************}

 LABEL
1, 2, 3;

 VAR
err2: integer;
p: params;
pdrvrec: ^drvrec;

 begin
with config_ptr^ do
begin
  errnum := 0;
  pdrvrec := pointer(drvrec_ptr);
  if pdrvrec = nil then
         exit(DOWN); { only until all configinfos are configurable }

  { test for branch to internal entry points for backing out of UP on errors }

  if enter_at > 0 then
  begin
         if enter_at = 1 then goto 1;
         if enter_at = 2 then goto 2;
         if enter_at = 3 then goto 3;
  end;

  if entry_pt = ord(nil) then
         exit(DOWN); { device not currently "upped" }

  p.still_inuse := false;
  if callers_config_ptr <> nil then
  begin
         p.fnctn_code := dunattach;
         p.o_configptr := callers_config_ptr;
         CALLDRIVER(err2, config_ptr, @p)     { unattach the sub_driver }
  end;
  if permanent or p.still_inuse then
         exit(DOWN); { keep device up }

 1:  WAIT_SEM(pdrvrec^.sem, []);
  p.fnctn_code := ddown;
  CALLDRIVER(err2, config_ptr, @p); { down the device }
  if enter_at = 0 then
```

```
         if err2 > 0 then
         begin
           errnum := err2;
           SIGNAL_SEM(pdrvrec^.sem);
           exit(DOWN);      { I/O still in progress }
         end;

 2:  entry_pt := ord(nil);
  with pdrvrec^ do
  begin
         nusers := nusers - 1;
         if nusers = 0 then { driver no longer in use }
           if kres_addr = ord(nil) then
                  UNLOAD_DRIVER(err2, e_pt);
  end;

 3:  SIGNAL_SEM(pdrvrec^.sem);
  if required_drvr <> nil then
         DOWN(err2, required_drvr, config_ptr, 0);
end
 end; { down }



 {$S krgenio }

 procedure UP(* var errnum: integer; config_ptr: ptrdevrec;
                          callers_config_ptr: ptrdevrec *);

{**************************************************************}
{*
*}
{* Description:  Up drivers "above" this, load driver.          *}
{*   initialize driver, and attach "lower" drivers.             *}
{*
*}
{* Input Parameters: Pointer to configinfo record of device  *}
{*  to be UPed, pointer to hierarchically lower configinfo   *}
{*  (or nil if none lower)                                              *}
{*
*}
{* Output Parameters: none                                              *}
{*
*}
{* Side Effects:  may call LOAD_DRIVER, dinit and dattach       *}
{*
*}
{* Special Conditions of Use: none                                *}
{*
*}
{* Error Conditions:                                                         *}
{*
*}
{**************************************************************}


 VAR
err2: integer;
p: params;
pdrvrec: ^drvrec;


 begin
        with config_ptr^ do
        begin
          errnum := 0;

          pdrvrec := pointer(drvrec_ptr);
          if pdrvrec = nil then
                 exit(UP);           { only until all configinfos are configurable }

          if entry_pt <> ord(nil) then
          begin  { already UPed }
                 if callers_config_ptr <> nil then
                 begin
                   p.fnctn_code := dattach;
                   p.n_configptr := callers_config_ptr;
                   CALLDRIVER(errnum, config_ptr, @p)   { attach the sub-driver }
                 end;
                 exit(UP)
          end;

          if required_drvr <> nil then
          begin { bring up the required driver before this one }
                 UP(errnum, required_drvr, config_ptr);
                 if errnum > 0 then exit(UP)
          end;

          pdrvrec := pointer(drvrec_ptr);
          with pdrvrec^ do
          begin
                 WAIT_SEM(sem, []);
                 if nusers = 0 then
                   if kres_addr <> ord(nil) then
                          e_pt := kres_addr  { kernel resident driver, no need to load }
                   else
                   begin
                          LOAD_DRIVER(errnum, e_pt, pointer(addrdrivrname));
                          if errnum > 0 then
                          begin
                            DOWN(err2, config_ptr, callers_config_ptr, 3); { back out & clear sem }
                            exit(UP)
                          end
                   end;

                   entry_pt := e_pt;
                   nusers := nusers + 1;

                   p.fnctn_code := dinit;
                   CALLDRIVER(errnum, config_ptr, @p); { initialize the driver }
                   if errnum > 0 then
                   begin
                          DOWN(err2, config_ptr, callers_config_ptr, 2); { back out & clear sem }
                          exit(UP)
                   end;
                   SIGNAL_SEM(sem);
                 end;

                 if callers_config_ptr <> nil then
                 begin
                   p.fnctn_code := dattach;
                   p.n_configptr := callers_config_ptr;
                   CALLDRIVER(errnum, config_ptr, @p);  { attach the sub-driver }
                   if errnum > 0 then
                          DOWN(err2, config_ptr, callers_config_ptr, 1); { back out }
                 end
          end

 end; { up }


 procedure ALARM_FIRES(* alarm: integer *);
{***********************************************************}
{*
*}
{* Description:  driver alarms call ALRM, which calls this   *}
{*
*}
{* Input Parameters: alarm number is in alarm on entry          *}
{*
*}
{* Output Parameters: none                                                                          *}
{*
*}
{* Side Effects:                                                                                                        *}
{*
*}
{* Special Conditions of Use: none                                                   *}
{*
*}
{* Error Conditions:                                                                                           *}
{*
*}
{************************************************************}
 VAR
errnum: integer;
p: params;

 begin
with alarm_table[alarm] do
begin
  INTSON(intstat);
  p.fnctn_code := dalarms;
  p.intpar := alarm;
  CALLDRIVER(errnum, cfg_ptr, @p)
end
 end; {alarm_fires }

 procedure ALARM_ASSIGN(* var alarm: integer; pdr: ptrdevrec;
                          status: intson_type *);

{************************************************************}
{*
*}
{* Description: Call ALARMASSIGN while providing for correct *}
{*                         driver to be called when alarm fires.                  *}
{*
*}
{* Input Parameters: pointer to this devices configinfo rec. *}
{*      and desired interrupt priority during alarm execution *}
{*
*}
{* Output Parameters: alarm number (=0 when none available)  *}
{*
*}
{* Side Effects:                                                                                                     *}
{*
*}
{* Special Conditions of Use: none                                                        *}
{*
*}
{* Error Conditions:  non available when alarm = 0             *}
{*
*}
{************************************************************}

 begin
ALARMASSIGN(alarm, ord(@ALRM));
with alarm_table[alarm] do
begin
  cfg_ptr := pdr;
  intstat := status
end
 end; { alarm_assign }

 function BADCALL(* parameters: param_ptr ): integer *);
{************************************************************}
{*
*}
{* Description:  Illegal driver.                                                          *}
{*
*}
{* Input Parameters: standard parameter record                          *}
{*
*}
{* Output Parameters: errnum returned as function result        *}
{*
*}
{* Side Effects:                                                                                                     *}
{*
*}
{* Special Conditions of Use: none                                                        *}
{*
*}
{* Error Conditions:  Always causes an error                                      *}
{*
*}
{************************************************************}
```

```

 begin
if parameters^.fnctn_code = dinterrupt then
  SYSTEM_ERROR(syserrbase+9); { interrupts would ignore returned errnum }
badcall := errbase+9;
 end; { badcall }


 function CHAIN_FORWARD(* current: linkage ): reqptr_type *);
{*************************************************************}
{*
*}
{* Description: Use forward link of relptr to point to next  *}
{*                         request block                                                                    *}
{*
*}
{* Input Parameters: linkage pointing to next request block  *}
{*
*}
{* Output Parameters: pointer to the request block              *}
{*
*}
{* Side Effects:                                                                                                      *}
{*
*}
{* Special Conditions of Use: callable from drivers             *}
{*
*}
{* Error Conditions:                                                                                         *}
{*
*}
{*************************************************************}

 begin
CHAIN_FORWARD := pointer(current.fwd_link+b_sysglobal_ptr-sizeof(Rb_headT))
 end; { chain_forward }


 function SCC(* parameters: param_ptr ): integer *);
{*************************************************************}
{*
*}
{* Description:  driver for 2-port SCC                                            *}
{*
*}
{* Input Parameters: standard parameter record                           *}
{*
*}
{* Output Parameters: errnum returned as function result        *}
{*
*}
{* Side Effects:                                                                                                      *}
{*
*}
{* Special Conditions of Use: only for initialization of SCC *}
{*
*}
{* Error Conditions:                                                                                         *}
{*
*}
{*************************************************************}

 CONST
ebase = 680;

 TYPE
rscb = record { this is the control block for the SCC. 1 pointer for
                                    each channel's control block }
                   a, b: ptrdevrec;
                 end;


{$S init}

procedure SCINIT;
{**********************************************}
```

```
{* internal subroutine to initialize SCC chip *}
{**********************************************}

VAR
  cb_rec: ^rscb;
  prevints: intson_type;
  temp: longint;

begin
  INTSOFF(rsints, prevints);

  if not GETSPACE(sizeof(rscb), b_sysglobal_ptr, temp) then
         SCC := ebase+3
  else
  begin
         parameters^.configptr^.cb_addr := temp;
         cb_rec := pointer(temp);
         cb_rec^.a := nil;
         cb_rec^.b := nil;
         SCC := 0; { no errors }
  end;
  INTSON(prevints)
end; { scinit }


procedure SCATTACH;
{******************************************************}
{* internal subroutine to attach a or b driver to SCC *}
{******************************************************}

VAR
  cb_rec: ^rscb;

begin
  with parameters^ do
  begin
         cb_rec := pointer(configptr^.cb_addr);
         if n_configptr^.iochannel = 0 then
           cb_rec^.a := n_configptr
         else
           cb_rec^.b := n_configptr;
         SCC := 0;
  end
end; { scattach }

{$S fs3}

procedure SCDOWN;
{*******************************************}
{* internal subroutine to down scc devices *}
{*******************************************}

VAR
  prevints: intson_type;

begin
  INTSOFF(rsints, prevints);
  with parameters^.configptr^ do
  begin
         RELSPACE(cb_addr, b_sysglobal_ptr);
         cb_addr := ord(nil)
  end;
  INTSON(prevints);
  SCC := 0 { no errors }
end; { scdown }

procedure SCUNATTACH;
{**********************************************************}
{* internal subroutine to unattach a or b driver from SCC *}
{**********************************************************}

VAR
  cb_rec: ^rscb;

begin
  with parameters^ do
  begin
         cb_rec := pointer(configptr^.cb_addr);
         with cb_rec^ do
         begin
           if o_configptr^.iochannel = 0 then
                  a := nil
           else
                  b := nil;

           still_inuse := ((a <> nil) or (b <> nil));
         end
  end
end; { scunattach }

 {*************************************************************************}

 {$S krgenio}

 begin { scc }
case parameters^.fnctn_code of

dinit: SCINIT;  { call internal subroutine in init segment }

ddown: SCDOWN;  { call internal subroutine in non-res segment }

dattach: SCATTACH;  { call internal subroutine in non-res segment }

dunattach: SCUNATTACH;  { call internal subroutine in non-res segment }

otherwise
  SCC := ebase + 2;  { no other functions allowed for 2-line SCC }

end { case }
 end; { SCC }


 {$S nigenio}

 procedure SEQENTIO(* var errnum: integer; dev_index: integer; io_buff_addr: absptr;
                                    length: longint; read_f: boolean; pcb_ptr: ptr_pcb;
                                    var req_ptr: reqptr_type *);

{*************************************************************}
{*
*}
{* Description: Start up or queue a sequential read/write.   *}
{*
*}
{* Input Parameters: configinfo index, memory address,          *}
{*   length in bytes, read/write flag, process' pcb_ptr.        *}
{*
*}
{* Output Parameters: Req_ptr is the address of the newly       *}
{*    created request block.                                                                        *}
{*
*}
{* Side Effects: Device dependent driver is invoked.                     *}
{*
*}
{* Special Conditions of Use: none                                                         *}
{*
*}
{* Error Conditions:                                                                                         *}
{*
*}
{*************************************************************}


 VAR
d_ptr: absptr;
ext_ptr: seqextptr_type;
prevints: intson_type;
leftlink: link_ptr;
p: params;
pblk_in_pcb: ptrblk_type;

```

```
  begin
 errnum := 0;
 with configinfo[dev_index]^ do
   if preq_avail then
   begin { use pre-allocated request block }
          preq_avail := false;
          d_ptr := ord(permreq_ptr)
   end
   else
          if not GETSPACE(sizeof(reqblk)+sizeof(seq_extend), b_sysglobal_ptr, d_ptr) then
            errnum := errbase + 10;
   if errnum = 0 then
   begin
          req_ptr := pointer(d_ptr);  { return pointer to the request }
          ext_ptr := pointer(d_ptr + sizeof(reqblk));  { point to extension area }

          { initialize the request block }

          with req_ptr^ do
          begin
            pcb_chain.kind := reqblk_type;
            reqstatus.reqsrv_f := active;
            reqstatus.reqabt_f := false;
            pblk_in_pcb := @blk_in_pcb;
            pblk_in_pcb^ := [i_o];
            block_p_f := false;
            hard_error := 0;
            operatn := ord(read_f);
            cfigptr := configinfo[dev_index];
            req_extent := ord(ext_ptr);      { point the request block to the extension }
          end;

          { initialize the extension data block }

          with ext_ptr^ do
          begin
            read_flag := read_f;
            num_bytes := length;
            xfer_count := 0;
            CVT_BUFF_ADDR(errnum, read_f, io_buff_addr, length, buff_rdb_ptr,
                                     buff_offset);
          end;
          if errnum > 0 then
            with configinfo[dev_index]^ do
            if permreq_ptr = req_ptr then
                   preq_avail := true
            else
                   RELSPACE(ord(req_ptr), b_sysglobal_ptr)
          else
          begin   { continue if no errors yet }
            ADJ_IO_CNT(true, ext_ptr^.buff_rdb_ptr);  { prevent swapout }

            { add request to pcb chain }

            INTSOFF(allints, prevints);
            leftlink := pointer(pcb_ptr^.req_chain.header.bkwd_link + b_sysglobal_ptr);
            ENQUEUE(req_ptr^.pcb_chain.header, leftlink^, b_sysglobal_ptr);
            INTSON(prevints);

            { call driver to start the request }

            p.fnctn_code := seqio;
            p.req := req_ptr;
            CALLDRIVER(errnum, req_ptr^.cfigptr, @p);  { call seqio }
            if errnum > 0 then
            begin
                   ADJ_IO_CNT(false, ext_ptr^.buff_rdb_ptr);   { allow swapout on errors }
                   CANCEL_REQ(req_ptr);
            end
          end
   end
  end; { seqentio }

end.

End of File: GENIO.TEXT
```

```
Directory of files in Cross Reference:

 1: GENIO.TEXT
level =  1.

a         1: 669*,  1: 694=,         1: 715=,  1: 758=,         1: 762.
abspage 1: 227=.
absptr  1:         31,   1:  33,     1:       34,   1:  59,     1:       64,   1:  98,     1: 101,   1: 173,
          1: 815.
active  1: 203,   1: 331,   1: 843.
add       1:        98*,  1: 101*.
addr    1: 107*.
addrdriv 1:        31*,  1: 505.
adj_io_c 1: 253,   1: 269, 1: 872,   1: 888.
alarm   1:        49*,  1:  52*,    1: 560,   1: 564,          1: 591,   1: 592.
alarm_ta 1:        92*,  1: 560,     1: 592.
alarmass 1: 591.
allints 1: 132,   1: 257,   1: 337,   1: 876.
asynctr 1:         24.
b         1: 669*,  1: 695=,         1: 717=,  1: 760=,         1: 762.
b_sysglo 1: 136,   1: 147, 1: 190,   1: 250,          1: 258,   1: 259,          1: 340,   1: 342,
          1: 642,   1: 688,          1: 736,   1: 831,          1: 869,   1: 877,          1: 878.
bkwd_lin 1: 258,   1: 339, 1: 877.
bkwdlink 1: 230=.
blk_in_p 1: 138,   1: 205, 1: 333,   1: 845.
blkno   1: 237=.
block_no 1:        65*,  1: 184,     1: 185,   1: 237.
block_p_ 1: 207=,  1: 335=,          1: 847=.
buff_off 1: 246,   1: 862.
buff_rdb 1: 246,   1: 253, 1: 269,   1: 861,          1: 872,   1: 888.
bytes   1: 178*,  1: 242=, 1: 244=,  1: 245.
c_pcb_pt 1: 338.
callers_ 1:        44*,  1:  47*,    1: 402,   1: 405,          1: 481,   1: 484,          1: 508,   1: 520,
          1: 526,   1: 529,          1: 532.
cb_addr 1: 692=,  1: 713,   1: 736,   1: 737=,         1: 754.
cb_rec  1: 681*,  1: 693=, 1: 694,   1: 695,          1: 708*,  1: 713=,         1: 715,   1: 717,
          1: 749*,  1: 754=,         1: 755.
cfg_ptr 1:        93*,  1: 565,     1: 594=.
cfigptr 1: 140,   1: 210=, 1: 266,   1: 850=,         1: 885.
chained_ 1: 186.
config_p 1:        40*,  1:  43*,    1:       46*,  1: 382,     1: 406,   1: 413,          1: 433,   1: 471,
          1: 485,   1: 492,          1: 508,   1: 517,          1: 520,   1: 530,          1: 532.
configin 1: 183,   1: 210, 1: 299,   1: 824,          1: 850,   1: 865.
configpt 1: 692,   1: 713, 1: 734,   1: 754.
cur_pcb_ 1: 323*,  1: 338=,          1: 339.
current 1:         68*,  1: 642.
cvt_buff 1: 245,   1: 861.
d_ptr   1: 173*,  1: 190,   1: 195,   1: 196,          1: 815*,  1: 828=,         1: 831,   1: 835,
          1: 836.
dalarms 1: 563.
dataok  1: 222.
datastat 1: 222=.
dataused 1: 226=.
dattach 1: 483,   1: 528,   1: 778.
ddown   1: 412,   1: 776.
dequeue 1: 136.
dev_inde 1:        56*,  1:  59*,    1:       63*,  1: 183,     1: 210,   1: 299,          1: 824,   1: 850,
          1: 865.
dinit   1: 516,   1: 774.
dinterru 1: 617.
disk_ext 1: 190.
disk_io_ 1:        66.
driver_i 1:        28*.
driverde 1:        14.
drvrec  1:         27*,  1: 379,     1: 467.
drvrec_p 1: 385,   1: 475, 1: 496.
dskio   1: 264.
dunattac 1: 404,   1: 780.
e_pt    1:         33*,  1: 428,     1: 502=,  1: 505,          1: 513.
ebase   1: 664*,  1: 689,   1: 783.
enqueue 1: 259,   1: 341,   1: 878.
enter_at 1:        47*,  1: 391,     1: 393,   1: 394,          1: 395,   1: 414.
entry_pt 1: 398,   1: 422=,          1: 479,   1: 513=.
er        1: 107*.
```

```
err       1:        98*,  1: 101*.
err2     1: 377*,  1: 406,   1: 413,   1: 415,          1: 417,   1: 428,          1: 433,   1: 465*,
          1: 508,   1: 520,          1: 532.
errbase 1:         85*,  1: 188,     1: 191,   1: 619,          1: 832.
errnum  1:         40*,  1:  43*,    1:        46*,  1:  56*,    1:        59*,  1:  63*,    1: 182=,  1: 188=,
          1: 191=,  1: 193,          1: 245,   1: 249,          1: 266,   1: 267,          1: 299,   1: 384=,
          1: 417=,  1: 473=,         1: 485,   1: 492,          1: 493,   1: 505,          1: 506,   1: 517,
          1: 518,   1: 530,          1: 531,   1: 556*,         1: 565,   1: 823=,         1: 832=,  1: 833,
          1: 861,   1: 864,          1: 885,   1: 886.
exit     1: 144,   1: 387,   1: 399,   1: 409,          1: 419,   1: 477,          1: 487,   1: 493,
          1: 509,   1: 521.
ext_addr 1: 183.
ext_conf 1: 176*,  1: 183=,          1: 185,   1: 237.
ext_disk 1: 176.
ext_ptr 1: 174*,  1: 196=,  1: 211,   1: 216,          1: 253,   1: 269,          1: 816*,  1: 836=,
          1: 851,   1: 856,          1: 872,   1: 888.
extdptr_ 1: 174.
external 1:        99,   1: 101,     1: 103,   1: 105,          1: 107.
extra_da 1:        64*,  1: 235.
extra_li 1:        64*,  1: 234.
fileid  1: 225=.
filler  1:         95*,  1: 223=.
fnctn_co 1: 264=,  1: 404=,          1: 412=,  1: 483=,         1: 516=,  1: 528=,         1: 563=,  1: 617,
          1: 772,   1: 883=.
fwd_link 1: 642.
fwdlink 1: 229=.
genio   1:          1.
getspace 1: 190,   1: 688,  1: 831.
globalda 1:        18.
hard_err 1: 208=,  1: 336=,          1: 848=.
header  1:         63*,  1: 136,     1: 233,   1: 258,          1: 259,   1: 339,          1: 341,   1: 341,
          1: 877,   1: 878.
hwint   1:         16.
i_o       1: 139,   1: 206,          1: 334,   1: 846.
implemen 1:        74.
int4    1:         64,   1:  65,     1: 107.
interfac 1:        10.
intpar  1: 564=.
intsoff 1: 132,   1: 257,   1: 337,   1: 686,          1: 733,   1: 876.
intson  1: 137,   1: 260,   1: 343,   1: 562,          1: 698,   1: 739,          1: 879.
intson_t 1:        50,   1:  94,     1: 128,   1: 179,          1: 322,   1: 682,          1: 730,   1: 817.
intstat 1:         94*,  1: 562,     1: 595=.
io_buff_ 1:        59*,  1:  64*,    1: 245,   1: 861.
io_mode 1: 238=,  1: 241.
iochanne 1: 714,   1: 757.
kind    1: 202=,  1: 330=,  1: 842=.
kres_add 1:        34*,  1: 427,     1: 501,   1: 502.
label   1: 373.
last_dat 1: 235=.
last_fwd 1: 234=.
leftlink 1: 177*,  1: 258=,          1: 259,   1: 324*,         1: 339=,  1: 341,          1: 818*,  1: 877=,
          1: 878.
length  1:         60*,  1:  65*,    1: 184,   1: 185,          1: 239,   1: 859,          1: 861.
link_ptr 1: 177,   1: 818.
linkage 1:         68.
mmprimit 1:        22.
mode    1:         66*,  1: 186,     1: 187,   1: 218,          1: 238.
n_config 1: 484=,  1: 529=,          1: 714,   1: 715,          1: 717.
name    1:         98*.
num_blok 1: 185.
num_byte 1: 859=.
num_chun 1: 239=,  1: 245.
nusers  1:         32,   1: 425=,    1: 425,   1: 426,          1: 500,   1: 514=,         1: 514.
o_config 1: 405=,  1: 757.
operatn 1: 209=,  1: 849=.
p         1: 172*,  1: 264,          1: 265,   1: 266,          1: 378*,  1: 401,          1: 404,   1: 405,
          1: 406,   1: 408,          1: 412,   1: 413,          1: 466*,  1: 483,          1: 484,   1: 485,
          1: 516,   1: 517,          1: 528,   1: 529,          1: 530,   1: 557*,         1: 563,   1: 564,
          1: 565,   1: 819*,         1: 883,   1: 884,          1: 885.
pagelabe 1:        63.
param_pt 1:        41,   1:  57,     1:        70,   1:  72.
paramete 1:        41*,  1:  57*,    1:        70*,  1:  72*,    1: 299,   1: 617,          1: 692,   1: 711,
          1: 734,   1: 752,          1: 772.
params  1: 172,   1: 378,   1: 466,   1: 557,          1: 819.
pathname 1:        89.
pblk_in_ 1: 129*,  1: 138=,          1: 139,   1: 175*,         1: 205=,  1: 206=,         1: 325*,  1: 333=,
          1: 334=,  1: 820*,         1: 845=,  1: 846=.
pcb_chai 1: 136,   1: 202, 1: 259,   1: 330,          1: 341,   1: 341,          1: 842,   1: 878.
pcb_ptr 1:        60*,  1:  66*,    1: 258,   1: 877.
pdr       1:       49*,  1: 594.
pdrvrec 1: 379*,  1: 385=, 1: 386,   1: 411,          1: 418,   1: 423,          1: 431,   1: 467*,
          1: 475=,  1: 476,          1: 496=,  1: 497.
permanen 1: 408.
permreq_ 1: 141,   1: 828, 1: 866.
pointer 1: 183,   1: 195,  1: 196,   1: 258,          1: 338,   1: 339,          1: 385,   1: 475,
          1: 496,   1: 505,          1: 642,   1: 693,          1: 713,   1: 754,          1: 835,   1: 836,
          1: 877.
preq_ava 1: 143=,  1: 825, 1: 827=,  1: 867=.
prevints 1: 128*,  1: 132, 1: 137,   1: 179*,         1: 257,   1: 260,          1: 322*,  1: 337,
          1: 343,   1: 682*,         1: 686,   1: 698,          1: 730*,  1: 733,          1: 739,   1: 817*,
          1: 876,   1: 879.
proc_pri 1:        20.
ptr_pcb 1:        60,   1:  66,     1: 323.
ptrblk_t 1: 129,   1: 175, 1: 325,   1: 820.
ptrdevre 1:        40,   1:  43,     1:       44,   1:  46,     1:       47,   1:  49,     1:       93,   1: 669.
ptrpathn 1:        89*,  1:  98.
raw_io  1: 241.
rb_headt 1: 642.
read_f  1:        60*,  1:  65*,    1: 186,   1: 187,          1: 209,   1: 218,          1: 236,   1: 245,
          1: 849,   1: 858,          1: 861.
read_fla 1: 236=,  1: 858=.
relpage 1: 228=.
relspace 1: 147,   1: 250, 1: 736,   1: 869.
req       1:       54*,  1: 107*,    1: 134,   1: 141,          1: 147,   1: 265=,         1: 884=.
req_chai 1: 258,   1: 339, 1: 877.
req_exte 1: 211=,  1: 851=.
req_ptr 1:        38*,  1:  61*,    1:       66*,  1: 195=,    1: 200,   1: 250,          1: 259,   1: 265,
          1: 266,   1: 270,          1: 328,   1: 835=,         1: 840,   1: 866,          1: 869,   1: 878,
          1: 884,   1: 885,          1: 889.
reqabt_f 1: 204=,  1: 332=,          1: 844=.
reqblk  1: 107,   1: 190,  1: 196,   1: 831,          1: 836.
reqblk_t 1: 202,   1: 330, 1: 842.
reqptr_t 1:        38,   1:  54,     1:       61,   1:  66,     1:       68,   1: 324.
reqsrv_f 1: 203=,  1: 331=,          1: 843=.
reqstatu 1: 203,   1: 204, 1: 331,   1: 332,          1: 843,   1: 844.
required 1: 432,   1: 433, 1: 490,   1: 492.
rscb    1: 667*,  1: 681,  1: 688,   1: 708,          1: 749.
rsints  1: 686,   1: 733.
sem       1:       30*,  1: 411,     1: 418,   1: 431,          1: 499,   1: 523.
semaphor 1:        30.
seq_exte 1: 831.
seqextpt 1: 816.
seqio   1: 883.
signal_s 1: 418,   1: 431, 1: 523.
sizeof  1: 190,   1: 190,  1: 196,   1: 642,          1: 688,   1: 831,          1: 831,   1: 836.
soft_hdr 1: 219,   1: 233=.
status  1:        50*,  1: 595.
still_in 1: 401=,  1: 408, 1: 762=.
strt_blo 1: 237.
syserrba 1:        86*,  1: 618.
system_e 1: 618.
temp    1: 683*,  1: 688,  1: 692,   1: 693.
unit    1:         1.
version 1:        29*,  1: 221=.
volume  1: 224=.
wait_sem 1: 411,   1: 499.
with_hea 1: 187.
without_ 1: 218.
xfer_cou 1: 240=,  1: 860=.

Procedures

alarm_as 1:        49*,  1: 569*.
alarm_fi 1:        52*,  1: 539*.
alrm    1: 105*,  1: 591.
calldriv 1:        40*,  1: 103*,    1: 266,   1: 299,          1: 406,   1: 413,          1: 485,   1: 517,
          1: 530,   1: 565,          1: 885.
cancel_r 1:        54*,  1: 109*,    1: 144,   1: 270,          1: 889.
diskio  1:        63*,  1: 151*.
down    1:        46*,  1: 350*,    1: 387,   1: 399,          1: 409,   1: 419,          1: 433,   1: 508,
          1: 520,   1: 532.
driverca 1:        56*,  1: 278*.
```

```
link_to_ 1:        38*,  1: 303*.
load_dri 1:        98*,  1: 505.
scattach 1: 702*,  1: 778.
scdown  1: 724*,  1: 776.
scinit  1: 675*,  1: 774.
scunatta 1: 743*,  1: 780.
seqentio 1:        59*,  1: 791*.
twigio  1: 107*.
unload_d 1: 101*,  1: 428.
up       1:        43*,  1: 441*,    1: 477,   1: 487,         1: 492,   1: 493,         1: 509,   1: 521.

Functions

badcall 1:         70*,  1: 599*,    1: 619=.
chain_fo 1:        68*,  1: 623*,    1: 642=.
scc      1:        72*,  1: 646*,    1: 689=,  1: 696=,        1: 718=,  1: 740=,        1: 783=.

Declaration Character : '*'
Assignment Character :  '='
```

Cross Reference Listing:

Beginning of file: HDISK.TEXT

```
UNIT HDISK;


        { By Dave Offen }
        { Copyright 1983, Apple Computer Inc. }

               { For Profile compatible drives up to 32MB. For larger drives, make
                 the following changes:
                  - change assembly language routine in PROFASM
                         so that checksum is not embedded in header.
               }

        { By Wendell Henry  1/17/83 }
        { o Convert this driver from a configurable Profile driver to a resident }
        {   generic driver of hard disks of Apple format.  A device specific     }
        {   configurable driver is called by HDISK.                                                                   }

INTERFACE

  USES
         {$U object/driverdefs.obj}
                driverdefs,
         {$U object/hwint.obj}
                hwint,
         {$U object/sysglobal.obj}
                globalda,
         {$U object/procprims.obj}
                proc_prims,
         {$U object/mmprim.obj}
                mmprimitives,
         {$U object/asynctr.obj}
                asynctr,
         {$U object/genio.obj}
                genio;


  {$IFC OS15}

  procedure USE_HDISK( configptr: ptrdevrec);

  procedure CALL_HDISK( var error: int2; configptr: ptrdevrec;
                  parameters: param_ptr);

  procedure IODONE( drivecb_ptr: hdiskcb_ptr; prev_err: integer);

  function OKXFERNEXT( drivecb_ptr: hdiskcb_ptr): integer;

  {$ENDC}

  function HDISKIO( parameters: param_ptr): integer;


IMPLEMENTATION

  {$S krgenio}

  {$IFC not debug2}
  {$R-}  { rangecheck off unless debug mode }
  {$ENDC}

  CONST
       syserrbase = 10650;
       errbase = 650;     { error number base }
       premenderr = 658;           { premature end of file in chained header record }
       vfyerr = 659;               { header read verify error }
       ignorerr = 661;   { error code when get timer interrupt requiring no action }
       cserr = -663;               { data is returned but contains checksum or crc error }
       hd_err = 654;               { hard error from profile itself }
       wait_int = 1;               { wait-for-next-interrupt error code }
       readcmd = 0;                { read command }
       writecmd = 1;               { write command }
```

```


    function LOGGING: boolean; external;

    procedure LOG(var errnum: integer; ptr_arr: longint); external;

    procedure START_DISK(drivecb_ptr: hdiskcb_ptr); forward;

    procedure START_NEW_REQUEST(drivecb_ptr: hdiskcb_ptr); forward;

    procedure CALL_HDISK; external;

    procedure USE_HDISK(* configptr: ptrdevrec *);
           {************************************************************}
           {*
*}
           {* Description: The device is a hard disk with Apple format *}
           {*                         sectors and will be using HDISK to handle   *}
           {*                         sector headers.
*}
           {*
*}
           {* Input Parameters: configptr points to configinfo entry.  *}
           {*
*}
           {* Output Parameters: none                                                                         *}
           {*
*}
           {************************************************************}
         var
           ext_ptr: ^ext_diskconfig;

         begin
  ext_ptr := pointer(configptr^.ext_addr);
  ext_ptr^.hentry_pt := ord(@HDISKIO);  { HDISKIO can now be called }
end; { USE_HDISK }



 procedure FINISH_REQ( drivecb_ptr: hdiskcb_ptr; success_flag: boolean;
                          error: integer);
  {************************************************************}
  {*
*}
  {* Description: Finish current request.                                          *}
  {*
*}
  {* Input Parameters: drivecb_ptr points to control block.   *}
  {*
*}
  {* Output Parameters: none                                                                         *}
  {*
*}
  {* Side Effects: none                                                                              *}
  {*
*}
  {* Special Conditions of use: appropriate interrupts must   *}
  {*                               be off
*}
  {*
*}
  {* Error Conditions: none                                                                          *}
  {*
*}
  {************************************************************}


VAR
  errnum: integer;
  succ_f: boolean;

begin
  with drivecb_ptr^, cur_info_ptr^ do
  begin
         succ_f := success_flag;

         with req_hd_ptr^ do
           if operatn <= 1 then
           begin  { read or write request }
                  if succ_f then
                  begin
                    if io_mode = chained_hdrs then
                           xfer_count := xfer_count + soft_hdr.dataused
                    else
                           xfer_count := xfer_count + 512;
                    if worstwarning <> 0 then
                    begin
                           hard_error := worstwarning;
                           succ_f := false
                    end
                  end
                  else
                  begin { unsuccessful }
                    hard_error := error;

                    {$IFC debug2}
                    if TRACE(DD, 1) then
                           writeln('*** PROFILE ERROR #: ', hard_error);
                    {$ENDC}

                  end;

           UNFREEZE_SEG(buff_rdb_ptr);                 { allow data to move }
           ADJ_IO_CNT(false, buff_rdb_ptr);   { allow data to be swapped }
         end
         else
           hard_error := error;

         UNBLK_REQ(req_hd_ptr, succ_f);

         { remove from the top of the queue }

         DEQUEUE(req_hd_ptr^.dev_chain, b_sysglobal_ptr);
         req_hd_ptr := CHAIN_FORWARD(req_hd_ptr^.dev_chain); { new head of request list }
         cur_num_requests := cur_num_requests - 1;
         if cur_num_requests > 0 then
         begin
           if req_hd_ptr = dummy_req_ptr then
           begin { if dummy at head, skip over it and reverse its cylinder number }
                  dummy_req_ptr^.reqspec_info := dummy_req_ptr^.reqspec_info +
                                    $80000000;
                  req_hd_ptr := CHAIN_FORWARD(dummy_req_ptr^.dev_chain);
           end;
           START_NEW_REQUEST(drivecb_ptr);
         end
         else
           if int_prio = winints then
                  DISKSYNC(false);  { allow contrast changes after I/O }
  end
end; { FINISH_REQ }

 procedure NEXT_HDR( var info: disk_extend; last: boolean);
  {************************************************************}
  {*
*}
  {* Description: Update the next software header.              *}
  {*
*}
  {* Input Parameters: Info is the request block extension    *}
  {*    containing the software header to be updated. Last is  *}
  {*    true if this is the last header to be updated for this *}
  {*    request. Used by WRITEs only.                                                       *}
  {*
*}
  {* Output Parameters: The updated soft_hdr in the request   *}
  {*
*}
  {* Side Effects: none                                                                              *}
  {*
*}
  {* Special Conditions of use: none                                                        *}
  {*
*}
             {* Error Conditions: none                                                          *}
             {*
*}
             {************************************************************}

           begin
             with info.soft_hdr do
             begin
                    bkwdlink := abspage;
                    abspage := abspage + 1;
                    relpage := relpage + 1;
                    if last then
                    begin
                      fwdlink := info.last_fwd_link;
                      dataused := info.last_data_used
                    end
                    else
                      fwdlink := abspage + 1
             end
           end; { NEXT_HDR }


            function OKXFERNEXT(* drivecb_ptr: hdiskcb_ptr): integer *);
           { Called by device specific driver to prepare for transfer of next sector }
           { from the disk. Headers and data pointers will be updated as needed.     }
           { Any errors found in sector just transferred will be returned as a non-  }
           { zero return which should terminate the transfer.                                }

           var
             error: integer;

           begin
             with drivecb_ptr^, cur_info_ptr^ do
             begin
                    error := 0;
                    sect_left := sect_left - 1;

                    if io_mode = chained_hdrs then
                    begin   { reading using chained headers }
                      with soft_hdr do
                             if (version <> soft_hdr.version) or
                                     (fileid <> soft_header.fileid) or
                                     (relpage <> soft_header.relpage) then
                             begin { verify error }
                               error := vfyerr;
                             end
                             else
                             begin { verifies ok }
                               if (fwdlink = -1) then
                                      if sect_left > 1 then
                                      begin { premature end of chain }
                                        error := premenderr;
                                        xfer_count := xfer_count + dataused
                                      end
                             end
                    end; { chained headers }

                    if error = 0 then
                    begin
                      if read_flag then
                             case raw_header_ptr^.datastat of
                               datamaybe:  if worstwarning = 0 then
                                                       worstwarning := -626;
                               databad:    if worstwarning <> cserr then
                                                       worstwarning := -625;
                             end; { case raw_header }

                     if (sect_left > 0) then
                     begin { setup for next sector }
                             blkno := blkno + 1;
                             xfer_count := xfer_count + 512;
                             x_leng := xfer_count;
                             case io_mode of
                               with_header:
                                      NEXT_HDR(cur_info_ptr^, sect_left = 1);
                    raw_io:
                           raw_header_ptr := pointer(ord(raw_header_ptr) + 24);
                    chained_hdrs:
                           soft_header.relpage := soft_header.relpage + 1
                  end { case io_mode }
           end
         end;

         okxfernext := error;

  end { with drivecb_ptr, cur_info_ptr }
end; {OKXFERNEXT}

 procedure IODONE (* drivecb_ptr: hdiskcb_ptr; prev_err: integer *);
  {************************************************************}
  {*
*}
  {* Description: Continue I/O request after call to driver.  *}
  {*
*}
  {* Input Parameters: drive_cb_ptr points to control block   *}
  {*
*}
  {* Output Parameters: none                                                                           *}
  {*
*}
  {* Side Effects:                                                                                              *}
  {*
*}
  {* Special Conditions of use: appropriate interrupts must   *}
  {*                      be off                                                                                *}
  {*
*}
  {* Error Conditions:                                                                                 *}
  {*
*}
  {************************************************************}
LABEL
  1;

VAR
  parm: params;
  success_flag: boolean;
  err: integer;
  disk_log_rec: record
                                      code: int1;
                                      rd_flag: boolean;
                                      slot: int1;
                                      chan: int1;
                                      er: integer;
                                      estat: longint;
                                      dev: integer;
                                    end;

begin
  with drivecb_ptr^, cur_info_ptr^ do
  begin
         if req_hd_ptr^.operatn > 1
           then FINISH_REQ(drivecb_ptr, prev_err = 0, prev_err) { non I/O operation }
           else  { I/O operation }
 1:               case prev_err of
                  0: begin { successful operation }
                            if sect_left = 0 then
                            begin { all sectors have been transferred }
                                   FINISH_REQ(drivecb_ptr, true, 0);
                            end
                            else
                            begin { sectors still to be transferred }
                                   success_flag := true;
                                   if read_flag then
                                   begin
                                     if io_mode = chained_hdrs then
                                     begin
                                            with soft_hdr do
                                              if (version <> soft_header.version) or
                                                      (fileid <> soft_header.fileid) or
                            (relpage <> soft_header.relpage) then
                    begin { verify error }
                           prev_err := vfyerr;
                           success_flag := false
                    end
                    else { verifies ok }
                           if (fwdlink = -1) then
                             if sect_left > 1 then
                             begin  { premature end of chain }
                                    prev_err := prememderr;
                                    success_flag := false;
                                    xfer_count := xfer_count + dataused
                             end;
           end; { chained headers }

           if success_flag then
           case raw_header_ptr^.datastat of
                  datamaybe: if worstwarning = 0 then
                                              worstwarning := -625;
                  databad:   if worstwarning <> cserr then
                                              worstwarning := -625;
           end;
         end; { read_mode }

         { next complete previous request if it's done. }
         { and start up next disk I/O, if any }

         if (sect_left > 1) and success_flag then
         begin { update current request }
           sect_left := sect_left - 1;
           blkno := blkno + 1;
           xfer_count := xfer_count + 512;
           x_leng := xfer_count;
           case io_mode of
                  with_header:
                    NEXT_HDR(cur_info_ptr^, sect_left = 1);
                  raw_io:
                    raw_header_ptr := pointer(ord(raw_header_ptr) + 24);
                  chained_hdrs:
                    soft_header.relpage := soft_header.relpage + 1;
           end;
           START_DISK(drivecb_ptr);
         end
         else
           FINISH_REQ(drivecb_ptr, success_flag, prev_err);

  end; { sectors still to be transferred }
end;

wait_int: { nothing to do but wait for next interrupt }
  begin
  end;

otherwise
  begin { got error }
         restrt_count := restrt_limit; { hard error - prevent retries }
         if LOGGING then
           with disk_log_rec do
           begin
                  code := -3;
                  rd_flag := read_flag;
                  with config_addr^ do
                  begin
                    slot := slot_no;
                    chan := iochannel;
                    dev := device_no;
                  end;
                  er := prev_err;
                  estat := 0;
                  LOG(err, ord(@disk_log_rec)) { write error to log file }
           end;

         restrt_count := restrt_count + 1;
         if restrt_count <= restrt_limit then
         begin
           total_restarts := total_restarts + 1;
```

```
                                      START_DISK(drivecb_ptr);          { do it again from the top }
                                    end
                                    else { exhausted the retries available }
                                      if prev_error = cserr then
                                      begin
                                             prev_error := 0;
                                             worstwarning := cserr;
                                             goto 1
                                      end
                                      else
                                             FINISH_REQ(drivecb_ptr, false, prev_err); { quit if too }

{ many errors }

                             end;
                           end; { case }
   end;
 end; { IODONE }


  procedure START_DISK(* drivecb_ptr, hdiskcb_ptr *);
   {************************************************************}
   {*
*}
   {* Description: Start a read or write of one sector on a    *}
   {*                                hard disk.                                               *}
   {*
*}
   {* Input Parameters: drivecb_ptr points to control block    *}
   {*
*}
   {* Output Parameters: none                                                        *}
   {*
*}
   {* Side Effects:                                                                                   *}
   {*
*}
   {* Special Conditions of use: appropriate interrupts must   *}
   {*                      be off                                                                  *}
   {*
*}
   {* Error Conditions:                                                                      *}
   {*
*}
   {************************************************************}
 VAR
   parm: params;
   errnum: integer;

 begin
   with drivecb_ptr^, cur_info_ptr^ do
   begin
          with parm do
          begin
            fnctn_code := hdskio;
            if req_hd_ptr^.operatn <= 1 then
            begin { I/O operation }
                   if read_flag
                     then c_cmd := readcmd
                     else c_cmd := writecmd;
                   c_sector := blkno;
            end;
            CALLDRIVER(errnum, config_addr, @parm); { call driver to start I/O }
          end; { with }
   end;
 end; { start_disk }


  procedure START_NEW_REQUEST(* drivecb_ptr, hdiskcb_ptr *);
   {************************************************************}
   {*
*}
   {* Description: Initialize control block and make first I/O *}
   {*    call for the request at the head of the drive's queue  *}
   {*
*}
```

```
  {* Input Parameters: drivecb_ptr points to this drive's variables   *}
  {*
*}
  {* Output Parameters: none                                                                         *}
  {*
*}
  {* Side Effects:  Unit globals: num_sectors_left,              *}
  {*                                           cur_info_ptr, req_hd_ptr                     *}
  {*             Request changed from active to in_service                *}
  {*
*}
  {* Special Conditions of use: required ints must be off.    *}
  {*
*}
  {* Error Conditions:                                                                               *}
  {*
*}
  {************************************************************}

VAR
  er: integer;

begin
  with drivecb_ptr^ do
  begin

         { set up global vars for this request }

         req_hd_ptr^.reqstatus.reqsrv_f := in_service; { from active to in_service }
         cur_info_ptr := pointer(req_hd_ptr^.req_extent);
         if req_hd_ptr^.operatn > 1 then
         begin   { non I/O operation }
           START_DISK(drivecb_ptr);
         end
         else
         begin { I/O operation }
           with cur_info_ptr^ do
           begin

                  { next try to lock the data buffer in memory }

                  FREEZE_SEG(er, buff_rdb_ptr, buff_offset, ord(req_hd_ptr),
                          resolved_addr);
                  if er = 0 then { lock worked }
                  begin
                    sect_left := num_chunks;
                    restrt_count := 0;
                    worstwarning := 0;
                    raw_header_ptr := @soft_hdr;
                    raw_data_ptr := resolved_addr;
                    x_leng := xfer_count;
                    case io_mode of
                           raw_io:
                             begin
                                    raw_header_ptr := pointer(resolved_addr);
                                    if num_chunks = 1 then
                                      raw_data_ptr := resolved_addr + 24  { speed optimization }
                                    else
                                      raw_data_ptr := resolved_addr + ord4(24)*num_chunks
                             end;

                           chained_hdrs: soft_header := soft_hdr; { remember which header to match }
                    end; { case }

                    START_DISK(drivecb_ptr);
                  end
           end { with cur_info_ptr }
         end;
  end { with drivecb_ptr }
end;  { start_new_request }


 function HDISKIO(* parameters: param_ptr): integer *);
  {************************************************************}
  {*
*}
```

```
  {* Description: HDISKIO is the generic device driver for all*}
  {*                         hard disks with the Apple format. Device    *}
  {*                         specific activities will be passed on to the*}
  {*                         appropriate device driver.                                   *}
  {*
*}
  {* Input Parameters: Each function code request has its own *}
  {*                                          input parameters.                                                  *}
  {*
*}
  {* Output Parameters: none                                                                         *}
  {*
*}
  {* Side Effects: The request is added to the queue for the  *}
  {*    appropriate drive. If no request is busy, the request  *}
  {*    is started up immediately.                                                                   *}
  {*
*}
  {* Special Conditions of use: none                                                        *}
  {*
*}
  {* Error Conditions:                                                                                *}
  {*
*}
  {************************************************************}

VAR
  parm: params;
  p_extdevrec: ^ext_diskconfig;
  drivecb_ptr: hdiskcb_ptr;
  ext_ptr: extdptr_type;
  ints: intson_type;
  left_sort_ptr, right_sort_ptr, head_sort_ptr: reqptr_type;
  new_sec, left_sec, right_sec: longint;
  errnum: integer;

  {$S init}

procedure HINITIT;
{**************************************************************}
{* Internal subroutine in initialize segment for profile init *}
{**************************************************************}
TYPE
  initrec = record
                             index: integer;
                           end;

VAR
  chan: integer;
  initrec_ptr: ^initrec;
  newdrivecb, newreqblk: absptr;

begin
  errnum := 0;
  chan := parameters^.configptr^.iochannel;
  if chan >= 0 then INTSOFF(slotints, ints)
                                   else INTSOFF(winints, ints);
  drivecb_ptr := pointer(parameters^.configptr^.cb_addr);
  if drivecb_ptr = nil then
  begin  { not initialized yet }
         errnum := errbase + 3;  { insufficeint sysglobal space }
         if GETSPACE(sizeof(hdisk_cb), b_sysglobal_ptr, newdrivecb) then
           if GETSPACE(sizeof(reqblk), b_sysglobal_ptr, newreqblk) then
           begin
                  errnum := 0;
                  parameters^.configptr^.cb_addr := newdrivecb;
                  drivecb_ptr := pointer(newdrivecb);
                  with drivecb_ptr^ do
                  begin
                    config_addr := parameters^.configptr;
                    ext_ptr := ord(nil);
                    v_flag := false;  { default is don't re-read all writes }
                    total_restarts := 0;
                    cur_num_requests := 0;
                    restrt_limit := 4;  { max times an I/O is retried }
                    if chan >= 0 then int_prio := slotints
```

```
                                                   else int_prio := winints;
                  dummy_req_ptr := pointer(newreqblk);
                  req_hd_ptr := dummy_req_ptr;
                  with dummy_req_ptr^ do
                  begin
                         dev_chain.fwd_link := newreqblk - b_sysglobal_ptr +
                                            sizeof(rb_headT);
                         dev_chain.bkwd_link := dev_chain.fwd_link;
                         reqspec_info := -1;  { initialize dummy cylinder # }
                         reqstatus.reqsrv_f := active;
                         reqstatus.reqabt_f := false;
                         block_p_f := false
                  end;
                end;

                { initialize device specific driver }
                with parameters^ do
                begin   { reuse parameter block }
                  fnctn_code := hdinit;
                  CALLDRIVER(errnum, configptr, parameters);
                end; { with }
         end;
       end;
       INTSON(ints);

       {$IFC debug1}
       if TRACE(DD,1) then
         if errnum <> 0 then
         begin
                with parameters^.configptr^ do
                begin
                  writeln('Error ', errnum, ' initializing hard disk: ');
                  writeln('Slot=', slot_no, ' chan=', iochannel, ' dev=', device_no);
                end;
         end;
       {$ENDC}

       HDISKIO := errnum;
end; { hinitit }

{********************************************************************}

{$S krgenio}

begin {hdiskio }
       drivecb_ptr := pointer(parameters^.configptr^.cb_addr);

       case parameters^.fnctn_code of

       dskio: begin
         HDISKIO := 0;  { no errors }
         with parameters^.req^ do
         begin
                ext_ptr := pointer(req_extent);
                new_sec := ext_ptr^.blkno;
                reqspec_info := new_sec;
         end;

         {$IFC debug1}
         if TRACE(DD, 10) then
                if ext_ptr^.read_flag then
                  writeln('Reading Profile block# = ', new_sec)
                else
                  writeln('Writing Profile block# = ', new_sec);
         {$ENDC}


         {* Add the new request to the device queue, sorted by sector
               * Every queue contains a 'dummy' request whose sector number is either
               * smaller than any valid sector (-1) or larger than any valid sector
               * ($7FFFFFFF). The sector numbers of the requests along the device queue
               * monotonically increase or decrease until the dummy is reached. The sector
               * numbers then reverse directions and monotonically decrease or increase
               * until they reach the same sector number as the request at the head of
               * the queue. This ordering of the queue allows the interrupt handler to
               * pull the next request from the top of the queue, maintaining optimal seek
```

```
        * ordering. The drive will pick up all requests which are in the same
        * direction of head travel, before it services requests in the other direction.
        * The dummy request is at the head of the queue only when the queue is empty. *}

  with drivecb_ptr^ do
  begin
         INTSOFF(int_prio, ints);
         left_sort_ptr := req_hd_ptr;
         head_sort_ptr := left_sort_ptr;  { remember where we started }
         if cur_num_requests = 0 then  { point head to new req }
           req_hd_ptr := parameters^.req
         else
         begin
           left_sec := left_sort_ptr^.reqspec_info;
           if left_sec = new_sec then  { past dummy if same as current sector }
                  left_sort_ptr := dummy_req_ptr;
           right_sort_ptr := CHAIN_FORWARD(left_sort_ptr^.dev_chain);
           right_sec := right_sort_ptr^.reqspec_info;

           { look for insertion point }

           while (right_sort_ptr <> head_sort_ptr) and
                  ((right_sec = new_sec) or ((new_sec > right_sec) and (new_sec > left_sec))
                    or ((new_sec < right_sec) and (new_sec < left_sec))) do
           begin  { chain forward one link if not at insertion point }
                  left_sort_ptr := right_sort_ptr;
                  left_sec := right_sec;
                  right_sort_ptr := CHAIN_FORWARD(left_sort_ptr^.dev_chain);
                  right_sec := right_sort_ptr^.reqspec_info;
           end
         end;

         { insert it }

         cur_num_requests := cur_num_requests + 1;
         ENQUEUE(parameters^.req^.dev_chain, left_sort_pointer^.dev_chain,
                 b_sysglobal_ptr);
         if cur_num_requests = 1 then
         begin
           if int_prio = winints then { built-in parallel port }
                  DISKSYNC(true);  { prevent contrast changes during I/O }
           START_NEW_REQUEST(drivecb_ptr)
         end;
         INTSON(ints)
  end { with }
end; { dskio }

dinit: HINITIT;  { call internal subroutine in initialize segment }

ddown:  { down a device }
  begin
         if parameters^.configptr^.slot_no >= 0 then INTSOFF(slotints, ints)
                           else INTSOFF(winints, ints);
         if drivecb_ptr^.cur_num_requests <> 0
         then hdiskio:= errbase { + ???? }
         else
         begin { no requests outstanding - shut the device down }
           with parameters^ do
           begin { reuse parameter block }
                  fnctn_code := hddown;
                  CALLDRIVER(errnum, configptr, parameters);
           end;
           RELSPACE(ord(drivecb_ptr^.req_hd_ptr), b_sysglobal_ptr);
           RELSPACE(ord(drivecb_ptr), b_sysglobal_ptr);
           parameters^.configptr^.cb_addr := ord(nil);
           hdiskio := 0;
         end;
         INTSON(ints)
  end; { ddown }

reqrestart: begin
  HDISKIO := 0;

  {$IFC debug2}
  if TRACE(DD, 5) then
         writeln('        Profile restarting request.');
```

```
 1: 783.              {$ENDC}
 1: 784.
 1: 785.              INTSOFF(drivecb_ptr^.int_prio, ints);
 1: 786.              START_NEW_REQUEST(drivecb_ptr);  { no other requests present }
 1: 787.              INTSON(ints);
 1: 788.              end; {reqrestart}
 1: 789.
 1: 790.            otherwise
 1: 791.              HDISKIO := errbase +6; { illegal function code or not implemented yet }
 1: 792.
 1: 793.     end; { case }
 1: 794.   end; {HDISKIO}
 1: 795.
 1: 796.
 1: 797.  end.
 1: 798.
 1: 799.

End of File: HDISK.TEXT


Directory of files in Cross Reference:

 1: HDISK.TEXT
level =  1.

abspage 1: 213,   1: 214=,  1: 214,   1: 222.
absptr  1: 605.
active  1: 640.
adj_io_c 1: 162.
asynctr 1:        32.
b_sysglo 1: 171,   1: 616,  1: 617,   1: 636,          1: 743,   1: 769,          1: 770.
bkwd_lin 1: 638=.
bkwdlink 1: 213=.
blkno   1: 274=,  1: 274,   1: 379=,  1: 379,          1: 477,   1: 685.
block_p_ 1: 642=.
buff_off 1: 527.
buff_rdb 1: 161,   1: 162,  1: 527.
c_cmd   1: 475=,  1: 476=.
c_sector 1: 477=.
calldriv 1: 479,   1: 650,  1: 767.
cb_addr 1: 612,   1: 620=,  1: 676,   1: 771=.
chain_fo 1: 172,   1: 180,  1: 723,   1: 734.
chained_ 1: 140,   1: 242,  1: 282,   1: 344,          1: 387,   1: 547.
chan    1: 321*,  1: 413=,  1: 603*,  1: 609=,         1: 610,   1: 630.
code    1: 318*,  1: 408=.
config_a 1: 410,   1: 479,  1: 624=.
configpt 1:        39*,  1:  41*,    1: 100,   1: 609,          1: 612,   1: 620,          1: 624,   1: 650,
          1: 660,   1: 676,          1: 758,   1: 767,          1: 771.
cserr   1:        67*,  1: 268,     1: 368,   1: 428,          1: 431.
cur_info 1: 131,   1: 237,  1: 279,   1: 328,          1: 384,   1: 467,          1: 515=,  1: 522.
cur_num_ 1: 173=,  1: 173,  1: 174,   1: 628=,         1: 716,   1: 741=,         1: 741,   1: 744,
          1: 760.
databad 1: 268,   1: 368.
datamayb 1: 266,   1: 366.
datastat 1: 265,   1: 365.
dataused 1: 141,   1: 219=,         1: 257,   1: 360.
dd        1: 155,   1: 657,          1: 690,   1: 781.
ddown   1: 756.
dequeue 1: 171.
dev       1: 324*,  1: 414=.
dev_chai 1: 171,   1: 172,  1: 180,   1: 636,          1: 638,   1: 638,          1: 723,   1: 734,
          1: 742,   1: 742.
device_n 1: 414,   1: 663.
dinit   1: 754.
disk_ext 1: 190.
disk_log 1: 317*,  1: 406,  1: 418.
disksync 1: 186,   1: 747.
drivecb_ 1:        44*,  1:  46*,    1:       78*,  1:  80*,    1: 106*,  1: 131,          1: 182,   1: 237,
          1: 328,   1: 331,          1: 337,   1: 390,          1: 393,   1: 425,          1: 435,   1: 467,
          1: 509,   1: 518,          1: 550,   1: 584*,         1: 612=,  1: 613,          1: 621=,  1: 622,
          1: 676=,  1: 711,          1: 748,   1: 760,          1: 769,   1: 770,          1: 785,   1: 786.
driverde 1:        22.
dskio   1: 680.
dummy_re 1: 176,   1: 178,  1: 178,   1: 180,          1: 632=,  1: 633,          1: 634,   1: 722.
enqueue 1: 742.
```

```
er        1: 322*,  1: 416=,         1: 506*,  1: 527,          1: 529.
err       1: 316*,  1: 418.
errbase 1:         63*,  1: 615,     1: 761,   1: 791.
errnum  1:         76*,  1: 127*,    1: 464*,  1: 479,          1: 589*,  1: 608=,         1: 615=,  1: 619=,
          1: 650,   1: 658,          1: 662,   1: 668,          1: 767.
error   1:         41*,  1: 107*,    1: 152,   1: 165,          1: 234*,  1: 239=,         1: 249=,  1: 256=,
          1: 262,   1: 288.
estat   1: 323*,  1: 417=.
ext_addr 1: 100.
ext_disk 1:        97,   1: 583.
ext_ptr 1:         97*,  1: 100=,    1: 101,   1: 585*,         1: 625=,  1: 684=,         1: 685,   1: 691.
extdptr_ 1: 585.
external 1:        74,   1:  76,     1:       82.
fileid  1: 246,   1: 246,   1: 348,   1: 348.
fnctn_co 1: 471=,  1: 649=,          1: 678,   1: 766=.
forward 1:         78,   1:  80.
freeze_s 1: 527.
fwd_link 1: 636=,  1: 638.
fwdlink 1: 218=,  1: 222=,  1: 253,   1: 355.
genio   1:         34.
getspace 1: 616,   1: 617.
globalda 1:        26.
hard_err 1: 146=,  1: 152=,          1: 156,   1: 165=.
hd_err  1:         68*.
hddown  1: 766.
hdinit  1: 649.
hdisk   1:          1.
hdisk_cb 1: 616.
hdiskcb_ 1:        44,   1:  46,     1:       78,   1:  80,     1: 106,   1: 584.
hdskio  1: 471.
head_sor 1: 587*,  1: 715=,          1: 728.
hentry_p 1: 101=.
hwint   1:         24.
ignorerr 1:        66*.
implemen 1:        53.
in_servi 1: 514.
index   1: 599*.
info    1: 190*,  1: 211,   1: 218,   1: 219.
initrec 1: 598*,  1: 604.
initrec_ 1: 604*.
int1    1: 318,   1: 320,   1: 321.
int2    1:         41.
int_prio 1: 185,   1: 630=,          1: 631=,  1: 713,          1: 746,   1: 785.
interfac 1:        18.
ints    1: 586*,  1: 610,   1: 611,   1: 654,          1: 713,   1: 750,          1: 758,   1: 759,
          1: 774,   1: 785,          1: 787.
intsoff 1: 610,   1: 611,   1: 713,   1: 758,          1: 759,   1: 785.
intson  1: 654,   1: 750,   1: 774,   1: 787.
intson_t 1: 586.
io_mode 1: 140,   1: 242,   1: 277,   1: 344,          1: 382,   1: 537.
iochanne 1: 413,   1: 609,  1: 663.
label   1: 310.
last    1: 190*,  1: 216.
last_dat 1: 219.
last_fwd 1: 218.
left_sec 1: 588*,  1: 720=,          1: 721,   1: 729,          1: 730,   1: 733=.
left_sor 1: 587*,  1: 714=,          1: 715,   1: 720,          1: 722=,  1: 723,          1: 732=,  1: 734,
          1: 742.
mmprimit 1:        30.
new_sec 1: 588*,  1: 685=,  1: 686,   1: 692,          1: 694,   1: 721,          1: 729,   1: 729,
          1: 729,   1: 730,          1: 730.
newdrive 1: 605*,  1: 616,  1: 620,   1: 621.
newreqbl 1: 605*,  1: 617,  1: 632,   1: 636.
num_chun 1: 531,   1: 541,  1: 544.
operatn 1: 136,   1: 330,   1: 472,   1: 516.
ord4    1: 544.
p_extdev 1: 583*.
param_pt 1:        42,   1:  50.
paramete 1:        42*,  1:  50*,    1: 609,   1: 612,          1: 620,   1: 624,          1: 647,   1: 650,
          1: 660,   1: 676,          1: 678,   1: 682,          1: 717,   1: 742,          1: 758,   1: 764,
          1: 767,   1: 771.
params  1: 314,   1: 463,   1: 582.
parm    1: 314*,  1: 463*,  1: 469,   1: 479,          1: 582*.
pointer 1: 100,   1: 281,   1: 386,   1: 515,          1: 540,   1: 612,          1: 621,   1: 632,
          1: 676,   1: 684.
premende 1:        64*,  1: 256,     1: 358.
```

```
prev_err 1:       44*,  1: 331,     1: 331,   1: 333,         1: 351=,  1: 358=,        1: 393,   1: 416,
         1: 428,   1: 430=,        1: 435.
proc_pri 1:       28.
ptr_arr 1:        76*.
ptrdevre 1:       39,   1:  41.
raw_data 1: 535=,  1: 542=,        1: 544=.
raw_head 1: 265,   1: 281=,        1: 281,   1: 365,         1: 386=,  1: 386,         1: 534=,  1: 540=.
raw_io  1: 280,   1: 385,   1: 538.
rb_headt 1: 637.
rd_flag 1: 319*,  1: 409=.
read_fla 1: 264,   1: 342,  1: 409,   1: 474,         1: 691.
readcmd 1:        70*,  1: 475.
relpage 1: 215=,  1: 215,   1: 247,   1: 247,         1: 283=,  1: 283,         1: 349,   1: 349,
         1: 388=,  1: 388.
relspace 1: 769,   1: 770.
req      1: 682,   1: 717,         1: 742.
req_exte 1: 515,   1: 684.
req_hd_p 1: 135,   1: 167,  1: 171,   1: 172=,        1: 172,   1: 176,         1: 180=,  1: 330,
         1: 472,   1: 514,         1: 515,   1: 516,         1: 527,   1: 633=,        1: 714,   1: 717=,
         1: 769.
reqabt_f 1: 641=.
reqblk  1: 617.
reqptr_t 1: 587.
reqresta 1: 777.
reqspec_ 1: 178=,  1: 178,  1: 639=,  1: 686=,        1: 720,   1: 724,         1: 735.
reqsrv_f 1: 514=,  1: 640=.
reqstatu 1: 514,   1: 640,  1: 641.
resolved 1: 528,   1: 535,  1: 540,   1: 542,         1: 544.
restrt_c 1: 404=,  1: 421=,        1: 421,   1: 422,         1: 532=.
restrt_l 1: 404,   1: 422,  1: 629=.
right_se 1: 588*,  1: 724=,        1: 729,   1: 729,         1: 730,   1: 733,         1: 735=.
right_so 1: 587*,  1: 723=,        1: 724,   1: 728,         1: 732,   1: 734=,        1: 735.
sect_lef 1: 240=,  1: 240,  1: 254,   1: 272,         1: 279,   1: 335,         1: 356,   1: 376,
         1: 378=,  1: 378,         1: 384,   1: 531=.
sizeof  1: 616,   1: 617,   1: 637.
slot    1: 320*,  1: 412=.
slot_no 1: 412,   1: 663,   1: 758.
slotints 1: 610,   1: 630,  1: 758.
soft_hdr 1: 141,   1: 211,  1: 244,   1: 245,         1: 346,   1: 534,         1: 547.
soft_hea 1: 246,   1: 247,  1: 283,   1: 283,         1: 347,   1: 348,         1: 349,   1: 388,
         1: 388,   1: 547=.
succ_f  1: 128*,  1: 133=,  1: 138,   1: 147=,        1: 167.
success_ 1: 106*,  1: 133,  1: 315*,  1: 341=,        1: 352=,  1: 359=,        1: 364,   1: 376,
         1: 393.
syserrba 1:       62*.
total_re 1: 424=,  1: 424,  1: 627=.
trace   1: 155,   1: 657,   1: 690,   1: 781.
unblk_re 1: 167.
unfreeze 1: 161.
unit    1:         1.
v_flag  1: 626=.
version 1: 245,   1: 245,   1: 347,   1: 347.
vfyerr  1:        65*,  1: 249,     1: 351.
wait_int 1:       69*,  1: 398.
winints 1: 185,   1: 611,   1: 631,   1: 746,         1: 759.
with_hea 1: 278,   1: 383.
worstwar 1: 144,   1: 146,  1: 266,   1: 267=,        1: 268,   1: 269=,        1: 366,   1: 367=,
         1: 368,   1: 369=,        1: 431=,  1: 533=.
writecmd 1:       71*,  1: 476.
x_leng  1: 276=,  1: 381=,  1: 536=.
xfer_cou 1: 141=,  1: 141,  1: 143=,  1: 143,         1: 257=,  1: 257,         1: 275=,  1: 275,
         1: 276,   1: 360=,        1: 360,   1: 380=,        1: 380,   1: 381,         1: 536.

Procedures

call_hdi 1:       41*,  1:  82*.
finish_r 1: 106*,  1: 331,  1: 337,   1: 393,         1: 435.
hinitit 1: 593*,  1: 754.
iodone  1:        44*,  1: 293*.
log      1:       76*,  1: 418.
next_hdr 1: 190*,  1: 279,  1: 384.
start_di 1:       78*,  1: 390,     1: 425,   1: 444*,        1: 518,   1: 550.
start_ne 1:       80*,  1: 182,     1: 485*,  1: 748,         1: 786.
use_hdis 1:       39*,  1:  84*.

Functions
```

```
hdiskio 1:         50*,  1: 101,     1: 558*,  1: 668=,         1: 681=,  1: 761=,         1: 772=,  1: 778=,
          1: 791=.
logging 1:         74*,  1: 405.
okxferne 1:        46*,  1: 227*,    1: 288=.

Declaration Character : '*'
Assignment Character :  '='
```

Cross Reference Listing:

Beginning of file: PROFILE.TEXT

```
UNIT PROFDRVR;


                  { By Wendell Henry 1/17/83 }
                  { A device specific driver for the Profile. It uses the generic hard }
                  { disk driver HDISK }

                  { By Wendell Henry 8/23/83 }
                  { The driver has been extended to support the Widget device }


  INTERFACE

       USES
         {$U object/driverdefs.obj}
                driverdefs,
         {$U object/hwint.obj}
                hwint,
         {$U object/sysglobal.obj}
                globaldata,
         {$U object/procprims.obj}
                proc_prims,
         {$U object/mmprim.obj}
                mmprimitives,
         {$U object/synctr.obj}
                asynctr,
         {$U object/genio.obj}
                genio,
         {$U object/hdisk.obj}
                hdisk;


         function PROFILE(parameters: param_ptr): integer;

  IMPLEMENTATION

  CONST
       errbase = 650;
       countlimit = 100;         { 100 = 5 secs before timeout on 2-port }
                                                   {         = 12 secs on parallel }
       wait_int = 1;             { "wait for next interrupt" error code }
       ignorerr = 661;  { error code when get timer error requiring no action }
       hd_err = 654;
       cserr = -663;

       { Drive types }
       T_Profile = 0;
       T_Seagate = 1;
       T_Widget = 2;

       { Driver commands }
       Readcmd = 0;
       Writecmd = 1;
       Formatcmd = 2;

  TYPE
       { Extension to Drive_cb for Profile }

       (* NOTE: definitions marked with {**} have corresponding definitions in PROFASM *)

       ext_drive_cb = record     { driver info for Profile hard disk }
                  {**} hwbase: longint;              { base address of 6522 }
                  {**} hwstatus: longint;                     { addr of 6522 status register B }
                  {**} remap_interleave: boolean; { true if interleave remapped }
                  {**} command_buffer: record        { command buffer for profile }
                  {**}   case integer of
                  {**}  0: (cmd: 0..2);    { set value after storing "sector" }
                  {**}  1: (sector: longint; retry_cnt: int1; sparing_thresh:int1);
                  {**}   end;
                  {**} checksum: integer;                    { temp storage during write }
                  {**} expect_hs: int1;              { internal storage for profasm }
```

```
                    {**} drivetype: int1;             { type of disk drive }
                    {**} asm_state: integer;          { state internal to assemb lang }
                    {**} errstat: longint;            { error returned from profile }
                    {**} asmerr: integer;             { err returned from assmbly routine }
                    {**} last_hard_error: longint; { last error from errstat }
                    {**} counter: integer;            { used for timeout }
                    {**} discsize: longint;                    { device size from controller }
                    {**} cxfercnt: int1;              { current transfer count }
                    {**} csum_valid: int1;            { true if precomputed checksum valid }
                    {**} nested_bdr: int1;            { true if bad response being handled }
                    {**} extstat: array[0..15] of int1; { 16 bytes for extended status }
                    {**} cmd_buf: array[0..7] of int1;  { 8 bytes for sending widget cmds }
           end;

    ptr_ext_drive_cb = ^ext_drive_cb;

    function GP_STATUS(via_addr: longint; var counter:integer): integer; external;

    procedure PROFASM(control_block: hdiskcb_ptr); external;

    function PROF_INIT(ext_cb: ptr_ext_drive_cb; via_addr: longint;
                    hwst_irb: longint; hwst_ddrb: longint): integer; external;

    procedure START_NONIO(var errnum: integer; drivecb_ptr: hdiskcb_ptr);
         {**********************************************}
         {* Issue non-I/O request to low level driver *}
         {**********************************************}
         var
  extdrivecb_ptr: ptr_ext_drive_cb;
  reqptr: reqptr_type;
  bufaddr: longint;

begin
  extdrivecb_ptr := pointer(drivecb_ptr^.ext_ptr);
  with drivecb_ptr^, extdrivecb_ptr^, req_hd_ptr^ do
  begin
         reqstatus.reqsrv_f := in_service;
         cur_info_ptr := pointer(req_extent);
         { process format request }
         command_buffer.sector := ord(req_hd_ptr)+sizeof(reqblk); { data address }
         command_buffer.cmd := operatn;
         asm_state := 0;
         PROFASM(drivecb_ptr);
         errnum := asmerr;
         if errnum <> 0 then
           if errnum = wait_int then
                  counter := countlimit;
  end; { with }
end; { START_NONIO }


 procedure NONIO_REQ(var errnum: integer; cmd: integer; devconfig: ptrdevrec);
{******************************}
{* Build a request for Format *}
{******************************}
var
  drivecb_ptr: hdiskcb_ptr;
  reqaddr: absptr;
  reqptr: reqptr_type;
  ints: intson_type;
  leftlink: reqptr_type;
  space: integer;

begin
  drivecb_ptr := pointer(devconfig^.cb_addr);
  errnum := errbase + 3;  { insufficient sysglobal space }
  space := sizeof(reqblk);
  if cmd = formatcmd then
         space := space + 512; { extra space for spare table map }
  if GETSPACE(space, b_sysglobal_ptr, reqaddr) then
  begin
         errnum := 0;
         reqptr := pointer(reqaddr);
         with reqptr^, drivecb_ptr^ do
         begin
           cfigptr := devconfig;
           operatn := cmd;
           req_extent := ord(nil);
           with list_chain do
           begin
                  fwd_link := ord(@fwd_link) - b_sysglobal_ptr;
                  bkwd_link := fwd_link;
           end;
           LINK_TO_PCB(reqptr);

           { add to device queue }
           INTSOFF(int_prio, ints);
           reqspec_info := req_hd_ptr^.reqspec_info;
           leftlink := pointer(req_hd_ptr^.dev_chain.bkwd_link -
                          sizeof(rb_headT) + b_sysglobal_ptr);
           if req_hd_ptr = dummy_req_ptr then
                  req_hd_ptr := reqptr;     { place at head }
           cur_num_requests := cur_num_requests + 1;
           ENQUEUE(dev_chain, leftlink^.dev_chain, b_sysglobal_ptr);
           if cur_num_requests = 1 then
                  START_NONIO(errnum, drivecb_ptr);
           INTSON(ints);

           if errnum = wait_int then
           begin
                  BLK_REQ(reqptr, reqptr);
                  { process will block until the operation is done }

                  if reqptr^.reqstatus.reqsuccess_f
                    then errnum := 0
                    else errnum := hd_err;
           end;
           CANCEL_REQ(reqptr);     { this will also release data space }
         end; { with }
  end;
end; {NONIO_REQ}


 function PROFILE (*parameteres: param_ptr): integer *);
{***********************************************************}
{*
*}
{* Description: PROFILE device-dependent user I/O request     *}
{*
*}
{* Input Parameters: Rqst is the partially initialized        *}
{*   request block. Blno is the s-file block number.                   *}
{*   Rast^.reqspec_info[2] will hold the cylinder number.     *}
{*   Rqst^.reqspec_info[1] points to disk-extend record       *}
{*      which holds info necessary to do I/O to the disk.     *}
{*
*}
{* Output Parameters: none                                                                        *}
{*
*}
{* Side Effects: The request is added to the queue for the *}
{*   appropriate drive. If no request is busy, the request *}
{*   is started up immediately.                                                         *}
{*
*}
{* Special Conditions of Use: none                                                      *}
{*
*}
{* Error Conditions:                                                                                       *}
{*
*}
{***********************************************************}

 label 20;

 var
hwddrb: longint;
p_extdevrec: ^ext_diskconfig;
port_ptr: hdiskcb_ptr;
newextdrivecb: absptr;
p_extdrivecb: ptr_ext_drive_cb;
p_dc_rec: ^dc_rec;
```

```
ints: intson_type;
error: int2;
parm: params;
response: integer;
i: integer;
operation: integer;

 begin { PROFILE DRIVER }
with parameters^ do
begin

  port_ptr := pointer(configptr^.cb_addr);

  { Preliminaries are done - now process function request }
  case fnctn_code of

  dinit: { first level of device initialization }
         begin
           USE_HDISK(configptr);  { Use HDISK for handling sector headers }
           CALL_HDISK(error, configptr, parameters);
           profile := error;
         end; { dinit }

  hdinit:        { now initialize device }
         begin
           profile := errbase + 3;          { insufficient sysglobal space }
           if GETSPACE(sizeof(ext_drive_cb), b_sysglobal_ptr, newextdrivecb) then
           begin
                  p_extdrivecb := pointer(newextdrivecb);
                  with p_extdrivecb^, configptr^ do
                  begin
                    port_ptr^.ext_ptr := newextdrivecb;
                    p_extdevrec := pointer(ext_addr);
                    if iochannel >= 0 then
                    begin  { attached to multi-port card }
                           hwbase := iospacemmu * $20000 + ord4($4000)*slot_no +
                                    $2001 + $800 * iochannel;
                           hwstatus := hwbase;
                           hwddrb := hwstatus + $10;
                    end
                    else
                    begin  { attached to built-in port }
                           hwbase := iospacemmu*$20000 + $0D801;
                           hwstatus := hwbase + $400;
                           hwddrb := hwstatus + 4;
                    end;
                    nested_bdr := 0;
                    for i := 0 to 15 do extstat[i] := 0;
                    discsize := 0;
                    { assume the device is a profile }
                    with p_extdevrec^ do num_bloks := 9720;
                    last_hard_err := 0;
                    command_buffer.retry_cnt := 10;
                    command_buffer.sparing_thresh := 3;
                    remap_interleave := true;
                    if iochannel < 0 then DISKSYNC(true);
                    response := -PROF_INIT(p_extdrivecb, hwbase, hwstatus, hwddrb);
                    if iochannel < 0 then DISKSYNC(false);
                    profile := response;
                    if (response <> 0) or (discsize <= 0) or (discsize > 30000)
                           then drivetype := T_Profile { set dirvetype to profile }
                           else
                           begin
                             with p_extdevrec^ do num_bloks := discsize-strt_blok;
                             if drivetype <> 0 then
                             begin  { widget }
                                    remap_interleave := false;
                                    drivetype := T_Widget;
                                    {****************** TEMPORARY
                                    remap_interleave := true;
                                    drivetype := 1;
                                    ****************** TEMPORARY}
                             end
                             else
                             begin  { Profile or Seagate }
                                    if discsize > 9728 then drivetype := T_Seagate;  { 10MB Seagate }
```

```
                            end;
                          end;
                 end; { with }
          end;
        end; {hdinit}

 hdskio:        { start I/O to disk }
        begin
          operation := port_ptr^.req_hd_ptr^.operatn;
          if operation = formatcmd then
          begin
                 START_NONIO(error, port_ptr);
                 profile := error;
          end
          else
          begin
                 p_extdrivecb := pointer(port_ptr^.ext_ptr);
                 with p_extdrivecb^ do
                 begin
                   command_buffer.sector := c_sector;
                   command_buffer.cmd := c_cmd;  { set after sector is stored! }
                   asm_state := 0;         { initialize state machine }
                   PROFASM(port_ptr);  { call state machine }
                   profile := asmerr;  { return driver status }
                   if asmerr <> 0 then
                          if asmerr = wait_int
                            then counter := countlimit { timeout limit }
                            else IODONE(port_ptr, asmerr);  { unexpected error }
                 end;
          end;
        end; {hdskio}

 dinterrupt:    { Interrupt from device }
        begin
          with port_ptr^ do
          begin
                 p_extdrivecb := pointer(ext_ptr);
                 with p_extdrivecb^ do
                 begin
                   asmerr := GP_STATUS(hwbase, counter);  { get profile status }
                   profile := asmerr;  { return status }
                   if (cur_num_requests <> 0) then
                   begin
                          if asmerr > 0 then goto 20;  { handle error }
                          PROFASM(port_ptr);
                          profile := asmerr;  { return status }
                          if asmerr = wait_int
                            then counter := countlimit  { timeout limit }
                            else
                            begin  { unexpected error }
20:                                if asmerr <> ignorerr then
                                   begin
                                     if errstat <> 0 then
                                            if (asmerr = hd_err) or (asmerr = cserr)
                                              then last_hard_error := errstat;
                                     IODONE(port_ptr, asmerr);
                                   end;
                            end;
                   end;
                 end; { with p_extdrivecb }
          end; { with port_ptr }
        end; {dinterrupt}

 dskunclamp:    { Unclamp request }
        begin
          profile := 685;         { not an ejectable media }
        end; {dskunclamp}

 dcontrol:  { Device control }
        begin
          begin
                 p_dc_rec := pointer(parptr);
                 p_extdrivecb := pointer(port_ptr^, ext_ptr);
                 with port_ptr^, p_dc_rec^, p_extdrivecb^ do
                 begin
                   if dversion <> 2
 1: 369.                              then profile := errbase + 7  { wrong application version }
 1: 370.                              else
 1: 371.                              begin
 1: 372.                                profile := 0;
 1: 373.                                INTSOFF(int_prio, ints);
 1: 374.                                case dcode of
 1: 375.                                15: begin  { return }
 1: 376.                                                          { o last error from profile controller }
 1: 377.                                                          { o state-machine restart count             }
 1: 378.                                                ar10[0] := last_hard_error;
 1: 379.                                                ar10[1] := total_restarts;
 1: 380.                                         end;
 1: 381.                                20: begin  { get disk status }
 1: 382.                                                ar10[0] := 4;    { good disk }
 1: 383.                                                ar10[1] := 0;
 1: 384.                                                ar10[2] := 999;  { don't know how many blocks }
 1: 385.                                                                                     {  are available
}
 1: 386.                                                ar10[3] := 0;
 1: 387.                                                ar10[4] := 1;    { sparing always enabled }
 1: 388.                                                ar10[5] := ord(command_buffer.sparing_thresh
 1: 389.                                                                           <= 10);
 1: 390.                                                ar10[6] := ord(v_flag);
 1: 391.                                         end;
 1: 392.                                21: begin  { enable/disable sparing }
 1: 393.                                                if ar10[1] = 0 then  { don't rewrite sort errs }
 1: 394.                                                  command_buffer.sparing_thresh := 11
 1: 395.                                                else  { rewrite if > 30% err rate }
 1: 396.                                                  command_buffer.sparing_thresh := 3;
 1: 397.                                                v_flag := (ar10[2] <> 0);         { verify all writes }
 1: 398.                                         end;
 1: 399.                                otherwise  profile := errbase + 7;
 1: 400.                              end; { case }
 1: 401.                              INTSON(ints);
 1: 402.                       end;
 1: 403.                     end; { with port_ptr, p_dc_rec, p_extdrivecb }
 1: 404.              end;
 1: 405.            end;  {dcontrol}
 1: 406.
 1: 407.     dskformat:  { format disk }
 1: 408.            begin
 1: 409.              p_extdrivecb := pointer(port_ptr^, ext_ptr);
 1: 410.              if p_extdrivecb^.drivetype < T_Widget
 1: 411.                     then error := 0  { disk already formatted }
 1: 412.                     else NONIO_REQ(error, Formatcmd, configptr);  { Widget can be formatted }
 1: 413.              profile := error
 1: 414.            end; {dskformat}
 1: 415.
 1: 416.     otherwise { pass the function on to HDISK }
 1: 417.            begin
 1: 418.              CALL_HDISK(error, configptr, parameters);
 1: 419.              profile := error;
 1: 420.            end;
 1: 421.
 1: 422.     end; { case }
 1: 423.   end; { with }
 1: 424.    end; { PROFILE DRIVER }
 1: 425.  end.
 1: 426.
 1: 427.

End of File: PROFILE.TEXT


Directory of files in Cross Reference:

 1: PROFILE.TEXT                                                                                          level =  1.

absptr  1: 129,   1: 214.
ar10    1: 378=,  1: 379=,  1: 382=,  1: 383=,         1: 384=,  1: 386=,         1: 387=,  1: 388=,
         1: 390=,  1: 393,         1: 397.
asm_stat 1:        73*,  1: 113=,    1: 314=.
asmerr  1:        75*,  1: 115,     1: 316,   1: 317,         1: 318,   1: 320,         1: 332=,  1: 333,
         1: 336,   1: 338,         1: 339,   1: 343,         1: 346,   1: 346,         1: 348.
asynctr 1:        26.
b_sysglo 1: 141,   1: 152,  1: 161,   1: 165,         1: 243.
```

```
bkwd_lin 1: 153=,  1: 160.
blk_req 1: 172.
bufaddr 1: 102*.
c_cmd   1: 313.
c_sector 1: 312.
call_hdi 1: 236,   1: 418.
cancel_r 1: 179.
cb_addr 1: 136,   1: 228.
cfigptr 1: 147=.
checksum 1:        70*.
cmd       1:       67*,  1: 112=,    1: 123*,  1: 139,         1: 148,   1: 313=.
cmd_buf 1:         83*.
command_ 1:        65*,  1: 111,     1: 112,   1: 269,         1: 270,   1: 312,         1: 313,   1: 388,
          1: 394,   1: 396.
configpt 1: 228,   1: 235, 1: 236,   1: 246,         1: 412,   1: 418.
control_ 1:        90*.
counter 1:         77*,  1:  88*,    1: 118=,  1: 319=,        1: 332,   1: 340=.
countlim 1:        39*,  1: 118,     1: 319,   1: 340.
cserr   1:         44*,  1: 346.
csum_val 1:        80*.
cur_info 1: 109=.
cur_num_ 1: 164=,  1: 164,  1: 166,   1: 334.
cxfercnt 1:        79*.
d801    1: 259.
dc_rec  1: 216.
dcode   1: 374.
dcontrol 1: 361.
dev_chai 1: 160,   1: 165,  1: 165.
devconfi 1: 123*,  1: 136,  1: 147.
dinit   1: 233.
dinterru 1: 325.
discsize 1:        78*,  1: 265=,    1: 276,   1: 276,         1: 280,   1: 292.
disksync 1: 272,   1: 274.
drivecb_ 1:        95*,  1: 105,     1: 106,   1: 114,         1: 128*,  1: 136=,        1: 145,   1: 167.
driverde 1:        16.
drivetyp 1:        72*,  1: 277=,    1: 281,   1: 284=,        1: 292=,  1: 410.
dskforma 1: 407.
dskuncla 1: 356.
dummy_re 1: 162.
dversion 1: 368.
enqueue 1: 165.
errbase 1:         38*,  1: 137,     1: 242,   1: 369,         1: 399.
errnum  1:         95*,  1: 115=,    1: 116,   1: 117,         1: 123*,  1: 137=,        1: 143=,  1: 167,
          1: 170,   1: 176=,        1: 177=.
error   1: 218*,  1: 236,   1: 237,   1: 304,         1: 305,   1: 411=,        1: 412,   1: 413,
          1: 418,   1: 419.
errstat 1:         74*,  1: 345,     1: 347.
expect_h 1:        71*.
ext_addr 1: 249.
ext_cb  1:         92*.
ext_disk 1: 212.
ext_driv 1:        61*,  1:  86,     1: 243.
ext_ptr 1: 105,   1: 248=,  1: 309,   1: 329,         1: 365,   1: 409.
extdrive 1: 100*,  1: 105=,          1: 106.
external 1:        88,   1:  90,     1:       93.
extstat 1:         82*,  1: 264.
fnctn_co 1: 231.
formatcm 1:        54*,  1: 139,     1: 302,   1: 412.
fwd_link 1: 152=,  1: 152,  1: 153.
genio   1:         28.
getspace 1: 141,   1: 243.
globalda 1:        20.
hd_err  1:         43*,  1: 177,     1: 346.
hdinit  1: 240.
hdisk   1:         30.
hdiskcb_ 1:        90,   1:  95,     1: 128,   1: 213.
hdskio  1: 299.
hwbase  1:         62*,  1: 252=,    1: 254,   1: 259=,        1: 260,   1: 273,         1: 332.
hwddrb  1: 211*,  1: 255=,  1: 261=,  1: 273.
hwint   1:         18.
hwst_ddr 1:        93*.
hwst_irb 1:        93*.
hwstatus 1:        63*,  1: 254=,    1: 255,   1: 260=,        1: 261,   1: 273.
i         1: 221*,  1: 264=,         1: 264=.
ignorerr 1:        42*,  1: 343.
implemen 1:        35.
```

```
in_servi 1: 108.
int1    1:         68,   1:  68,     1:       71,   1:  72,     1:       79,   1:  80,     1:       81,   1:  82,
          1:       83.
int2    1: 218.
int_prio 1: 158,   1: 373.
interfac 1:        12.
ints    1: 131*,  1: 158,  1: 168,   1: 217*,         1: 373,   1: 401.
intsoff 1: 158,   1: 373.
intson  1: 168,   1: 401.
intson_t 1: 131,   1: 217.
iochanne 1: 250,   1: 253,  1: 272,   1: 274.
iodone  1: 320,   1: 348.
iospacem 1: 252,   1: 259.
label   1: 208.
last_har 1:        76*,  1: 268=,    1: 347=,  1: 378.
leftlink 1: 132*,  1: 160=,          1: 165.
link_to_ 1: 155.
list_cha 1: 150.
mmprimit 1:        24.
nested_b 1:        81*,  1: 263=.
newextdr 1: 214*,  1: 243,  1: 245,   1: 248.
num_blok 1: 267=,  1: 280=.
operatio 1: 222*,  1: 301=,          1: 302.
operatn 1: 112,   1: 148=,  1: 301.
ord4    1: 252.
p_dc_rec 1: 216*,  1: 364=,          1: 366.
p_extdev 1: 212*,  1: 249=,          1: 267,   1: 280.
p_extdri 1: 215*,  1: 245=,          1: 246,   1: 273,          1: 309=,  1: 310,          1: 329=,  1: 330,
          1: 365=,  1: 366,          1: 409=,  1: 410.
param_pt 1:        33.
paramete 1:        33*,  1: 225,     1: 236,   1: 418.
params  1: 219.
parm    1: 219*.
parptr  1: 364.
pointer 1: 105,   1: 109,   1: 136,   1: 144,          1: 160,   1: 228,          1: 245,   1: 249,
          1: 309,   1: 329,          1: 364,   1: 365,          1: 409.
port_ptr 1: 213*,  1: 228=,          1: 248,   1: 301,          1: 304,   1: 309,          1: 315,   1: 320,
          1: 327,   1: 337,          1: 348,   1: 365,          1: 366,   1: 409.
proc_pri 1:        22.
profdrvr 1:         1.
ptr_ext_ 1:        86*,  1:  92,     1: 100,   1: 215.
ptrdevre 1: 123.
rb_headt 1: 161.
readcmd 1:         52*.
remap_in 1:        64*,  1: 271=,    1: 283=.
req_exte 1: 109,   1: 149=.
req_hd_p 1: 106,   1: 111,  1: 159,   1: 160,          1: 162,   1: 163=,         1: 301.
reqaddr 1: 129*,  1: 141,   1: 144.
reqblk  1: 111,   1: 138.
reqptr  1: 101*,  1: 130*,  1: 144=,  1: 145,          1: 155,   1: 163,          1: 172,   1: 172,
          1: 175,   1: 179.
reqptr_t 1: 101,   1: 130,  1: 132.
reqspec_ 1: 159=,  1: 159.
reqsrv_f 1: 108=.
reqstatu 1: 108,   1: 175.
reqsucce 1: 175.
response 1: 220*,  1: 273=,          1: 275,   1: 276.
retry_cn 1:        68*,  1: 269=.
sector  1:         68*,  1: 111=,    1: 312=.
sizeof  1: 111,   1: 138,   1: 161,   1: 243.
slot_no 1: 252.
space   1: 133*,  1: 138=,  1: 140=,  1: 140,          1: 141.
sparing_ 1:        68*,  1: 270=,    1: 388,   1: 394=,         1: 396=.
strt_blo 1: 280.
t_profil 1:        47*,  1: 277.
t_seagat 1:        48*,  1: 292.
t_widget 1:        49*,  1: 284,     1: 410.
total_re 1: 379.
unit    1:          1.
use_hdis 1: 235.
v_flag  1: 390,   1: 397=.
via_addr 1:        88*,  1:  92*.
wait_int 1:        41*,  1: 117,     1: 170,   1: 318,          1: 339.
writecmd 1:        53*.

Procedures
```

```
nonio_re 1: 123*,  1: 412.
profasm 1:        90*,  1: 114,     1: 315,   1: 337.
start_no 1:       95*,  1: 167,     1: 304.

Functions

gp_statu 1:       88*,  1: 332.
prof_ini 1:       92*,  1: 273.
profile 1:        33*,  1: 185*,    1: 237=,  1: 242=,        1: 275=,  1: 305=,        1: 316=,  1: 333=,
          1: 338=,  1: 358=,        1: 369=,  1: 372=,        1: 399=,  1: 413=,        1: 419=.

Declaration Character : '*'
Assignment Character :  '='
```

Cross Reference Listing:

Beginning of file: RS232.TEXT

```

UNIT RS232;                         { built-in RS232 driver }

        { By Dave Offen }
        { Copyright 1983, Apple Computer Inc. }

INTERFACE

  USES
       {$U object/driverdefs.obj}
         driverdefs,
       {$U object/driversubs.obj}
         driversubs;

       function DRIVER( parameters: param_ptr): integer;

IMPLEMENTATION


  {$IFC not debug2}
  {$R-}  { rangecheck off unless debug mode }
  {$ENDC}

  CONST
       errbase = 640;
       axmit = 0;
       amodem = 1;
       arecv = 2;
       aerr = 3;

  TYPE
       send_hs = (hw_hs, xmit_xon, xmit_delay);  { transmit handshake type }
       recv_hs = (hwhs, rec_xon);  { receive handshake type }

       int1ptr = ^int1;

       int1recptr = ^int1rec;
       int1rec = record
                                  controlreg: int1;
                           end;

       portrec_ptr = ^portrec;
       portrec = record
                                  cur_req_ptr: reqptr_type;
                                  cur_info_ptr: seqextptr_type;
                                  cur_read_flag: boolean;
                                  hwdata: int1ptr;  { actual I/O address }
                                  hwcontrol: int1recptr;  { actual I/O address }
                                  nulllink: linkage;  { points to head of queue }
                                  openwr5: int1;
                                  openwr14: int1;
                                  discon_detect: boolean;  { detect disconnect on framing error }
                                  restrt_out, restrt_in: boolean;
                                  prev_char: int1;
                                  autolf: boolean;

                                  { RS232 output variables }

                                  xmt_hs: send_hs;  { handshake for transmitting }
                                  in_break: boolean;  { when T, openwr5 has bit#4 set & alarm pending }
                                  xmit_wait: boolean; { waiting for XON, modem signal or timeout }
                                  xmit_ref: integer;  { timer refnum }

                                  {xmt_hs = hw_hs or xmt_xon; (modem or xoff/xon handshake) }
                                  xmit_timeout: longint;  { hardware handshake timeout }

                                  {xmt_hs = hw_hs; (no handshake, or modem handshake) }
                                  xmtrr0: int1;    { modem controls required }
                                  xmtzrr0: int1; { modem required to = 0 }

                                  {xmt_hs = xmit_delay. ( delay after CR, LF) }
```

```
                                       waitb4next: boolean;  { cr or lf was xmitted }
                                       crlfdelay: longint;  { timer count }

                                       { RS232 input variables }

                                       rec_hs: recv_hs;  { handshake for receiving }
                                       no_block: boolean; { blocking/non-blocking read mode }
                                       last_stopped: boolean; { DTR off or XOFF last sent }
                                       size_typeah: 0..1024; { size of type-ahead input buffer }
                                       prity_char: int1; { char to replace parity errs, or 01 if disabled }
                                       num_typeah: integer; { number of chars currently in buffer }
                                       full_thresh: integer; { threshold for sending "buffer-full" }
                                       empty_thresh: integer; { threshold for sending "buffer-empty" }
                                       start_typeah, get_typeah, put_typeah: int1ptr; { type-ahead ptrs }
                                       end_typeah: int1ptr; { equals start_typeah+size_typeah }

                                       {rec_hs = rec_xon. (send xoff/xon when threshold reached) }
                                       hschar_pending: (no_char, xoff_char, xon_char);

                                end;


            procedure FINISH_REQ(port: portrec_ptr; ok_flag: boolean); forward;


            procedure BYTEO(port: portrec_ptr);
              {************************************************************}
              {*
  *}
     {* Description: Write one byte on RS232                                                    *}
     {*
*}
     {* Input Parameters: Port points to the current port in the          *}
     {*             RS232 device control block.                                                        *}
     {*
*}
     {* Output Parameters: none                                                                                  *}
     {*
*}
     {* Side Effects: Starts timer if hardware handshake requires *}
     {*
*}
     {* Special Conditions of Use: RS232 interrupts must be off  *}
     {*
*}
     {* Error Conditions:                                                                                        *}
     {*
*}
     {************************************************************}

   VAR
     data_ptr: int1ptr;
     status, err: integer;
     real_addr: longint;

   begin
     with port^.hwcontrol^ do
     begin
            status := controlreg;     { read hardware control reg 0 }

            if (hschar_pending <> no_char) then
            begin   { need to send xoff or xon }
              if ALLSET($04, status) then { output buffer is empty }
              begin { send xoff or xon now }
                     if hschar_pending = xoff_char then
                     begin

                       {$IFC debug2}
                       if TRACE(DD, 1) then
                              write('<xof>');
                       {$ENDC}

                       hwdata^ := $13; { xoff char }
                     end
                     else
                     begin   { send xon }
```

```

                    {$IFC debug2}
                    if TRACE(DD, 1) then
                           write('<xon>');
                    {#ENDC}

                    hwdata^ := $11;         { xon char }
                  end;
                  hschar_pending := no_char
           end
         end
         else
           if (xmt_hs <> hw_hs) or
                  (ALLSET(xmtrr0, status) and ALLSET(xmtzrr0, -1-status)) then
           begin { ready for xmit }
                  if ALLSET($04, status) then { xmit buffer is empty }
                    with cur_info_ptr^ do
                    begin
                           FREEZE_SEG(err, buff_rdb_ptr, buff_offset, ord(cur_req_ptr),
                                    real_addr);
                           if err <> 0 then
                           begin
                             controlreg := $28;  { clear output interrupts with $28 }
                             restrt_out := true  { reqrestart will be called to continue }
                           end
                           else
                           begin { freeze seg succeeded }
                             if autolf and
                                    ((prev_char = $0d) or (prev_char = -115{=$8D})) { bug 161 }
                             then
                                    prev_char := $0A { insert auto LF }
                             else { no automatic line feed }
                             begin
                                    data_ptr := pointer(real_addr + xfer_count);
                                    xfer_count := xfer_count + 1;
                                    prev_char := data_ptr^;
                             end;

                             {$IFC debug2}
                             if TRACE(DD, 1) then
                                    write(CHR(prev_char));
                             {$ENDC}

                             hwdata^ := prev_char;  { do the write }

                             if xmt_hs = xmit_delay then
                                    if (prev_char = $0d) or (prev_char = $0a) then
                                      waitb4next := true;     { start delay after CR or LF interrupts }

                                    UNFREEZE_SEG(buff_rdb_ptr)  { allow buffer to move again }
                           end
                    end
           end
           else { not ready for xmit }
           begin { allow delay for hardware handshake }

                  {$IFC debug2}
                  if TRACE(DD, 5) then
                    writeln(' RS-232 stop xmit for hw handshake. ');
                  {$ENDC}

                  if xmit_timeout > 0 then
                    ALARMRELATIVE(xmit_ref, xmit_timeout);
                  xmit_wait := true;
                  controlreg := $28         { clear output interrupts with $28 }
           end
  end
end;  {BYTEO}

procedureCOPYBYTES(port: portrec_ptr);
  {****************************************************************}
  {*
*}
  {* Description: Copy bytes from typeahead buffer to user buffer *}
  {*
*}
```

```
  {* Input Parameters: Port points to the current port in the              *}
  {*              RS232 device control block.                                                         *}
  {*
*}
  {* Output Parameters: none                                                                                   *}
  {*
*}
  {* Side Effects:
*}
  {*
*}
  {* Special Conditions of Use: RS232 interrupts must be off     *}
  {*
*}
  {* Error Conditions:                                                                                         *}
  {*
*}
  {*****************************************************************}

VAR
  i, errnum, cnt: integer;
  data_ptr: int1ptr;
  real_addr: longint;

begin
  with port^, cur_info_ptr^ do
         if num_typeah > 0 then
         begin
           FREEZE_SEG(errnum, buff_rdb_ptr, buff_offset, ord(cur_req_ptr),
                                    real_addr);
           if errnum > 0 then
                  restrt_in := true
           else
           begin
                  data_ptr := pointer(real_addr + xfer_count);
                  cnt := num_bytes - xfer_count;
                  if cnt > num_typeah then
                    cnt := num_typeah;

                  { transfer typed-ahead characters to user buffer }

                  i := ord(get_typeah) + cnt - ord(end_typeah);       { wraparound chars }
                  if i < 0 then { transfer all chars in single move of length "cnt" }
                    i := cnt
                  else
                  begin { transfer requires two moves }
                    moveleft(get_typeah^, data_ptr^, cnt - i);
                    data_ptr := pointer(ord(data_ptr) + cnt - i);
                    get_typeah := start_typeah
                  end
                  moveleft(get_typeah^, data_ptr^, i);
                  get_typeah := pointer(ord(get_typeah) + i);

                  UNFREEZE_SEG(buff_rdb_ptr);  { allow buffer to move }
                  num_typeah := num_typeah - cnt;
                  if last_stopped then
                    if num_typeah <= empty_thresh then { allow input }
                    begin
                           last_stopped := false;
                           if rec_hs = hwhs then
                           begin { toggle modem signals }

                             {$IFC debug2}
                             if TRACE(DD, 5) then
                                    writeln(' RS-232 DTR turned on.');
                             {$ENDC}

                             with hwcontrol^ do
                             begin
                                    controlreg := 5;
                                    i := 128;        { NOP to assure delay between accesses }
                                    controlreg := openwr5
                             end
                           end
                           else
                           begin   { send xon }
```

```
                             hschar_pending := xon_char;
                             BYTEO(port);
                           end
                    end;
                  xfer_count := xfer_count + cnt
           end;
           if (xfer_count = num_bytes) or no_block then { read complete }
                  FINISH_REQ(port, true)
         end
end; { copybytes }


procedure START_NEW_REQUEST(port: portrec_ptr);
  {****************************************************************}
  {*
*}
  {* Description: Start a new read or write from the queue.      *}
  {*
*}
  {* Input Parameters: Port points to the current port in the             *}
  {*             RS232 device control block.                                                *}
  {*
*}
  {* Output Parameters: none                                                                          *}
  {*
*}
  {* Side Effects:  Modem control signals are affected.                   *}
  {*
*}
  {* Special Conditions of Use: RS232 interrupts must be off     *}
  {*
*}
  {* Error Conditions:                                                                                *}
  {*
*}
  {****************************************************************}

begin
  with port^ do
  begin
         cur_req_ptr := CHAIN_FORWARD(nullink);
         cur_req_ptr^.reqstatus.reqsrv_f := in_service;
         cur_info_ptr := pointer(cur_req_ptr^.req_extent);
         with cur_info_ptr^ do
         begin
           cur_read_flag := read_flag;
           if discon_detect and (hwcontrol^.controlreg < 0) then
           begin  { assume disconnected cable on input framing error }
                  xmit_wait := false;  { fixes bug }
                  cur_req_ptr^.hard_error := errbase + 8;
                  FINISH_REQ(port, false)
           end
           else { start up i/o for new request }
                  if cur_read_flag then
                    COPYBYTES(port)
                  else { write }
                  begin
                    waitb4next := false;
                    if (xmt_hs = xmt_xon) and xmit_wait then { last received xoff }
                    begin
                           if xmit_timeout > 0 then
                             ALARMRELATIVE(xmit_ref, xmit_timeout)
                    end
                    else
                           BYTEO(port)
                  end
         end
  end
end; {START_NEW_REQUEST}

procedure FINISH_REQ(*port: portrec_ptr; ok_flag: boolean*);
  {****************************************************************}
  {*
*}
  {* Description: Complete the current request.                           *}
  {*
*}
  {* Input Parameters: Port points to the current port in the             *}
  {*              RS232 device control block. Ok_flag is param to          *}
  {*              unlock request.                                                                              *}
  {* Output Parameters: none                                                                                   *}
  {*
*}
  {* Side Effects:  Modem control signals are affected.                    *}
  {*
*}
  {* Special Conditions of Use: Cur_req_ptr must be valid                  *}
  {*
*}
  {* Error Conditions:                                                                                         *}
  {*
*}
  {****************************************************************}

VAR
  err: integer;
  ptrsysg: ^longint;

begin
  ptrsysg := pointer(bsysglob);  { set up pointer to b_sysglobal_ptr }
  with port^, cur_info_ptr^ do
  begin
         ADJ_IO_CNT(false, buff_rdb_ptr);  { allow swapout }
         UNBLK_REQ(cur_req_ptr, ok_flag);
         DEQUEUE(cur_req_ptr^.dev_chain, ptrsysg^);
         if xmit_wait then
           ALARMOFF(xmit_ref);
         if cur_read_flag then
           restrt_in := false  { prevent reqrestart after completion. fixes bug }
         else { write }
           if hschar_pending = no_char then
                  restrt_out := false;  { fixes bug }

         {$IFC debug2}
         if TRACE(DD, 10) then
           writeln('RS-232 Completed. Success = ', ok_flag);
         {$ENDC}

         if (nullink.fwd_link + ptrsysg^) <> ord(@nullink) then
           START_NEW_REQUEST(port)
         else
           cur_req_ptr := nil
  end   { with }
end; { finish_req }


procedure XMIT_TO_HNDL(port: portrec_ptr);
  {****************************************************************}
  {*
*}
  {* Description: Timeout handler when waited too long for       *}
  {*               resuming output to RS-232 channel A or B, or delaying *}
  {*               after transmitting CR or LF.                                    *}
  {*
*}
  {* Input Parameters: pointer to data for Port A or B                     *}
  {*
*}
  {* Output Parameters: none                                                                                   *}
  {*
*}
  {* Side Effects:  Completes the request with an error                    *}
  {*
*}
  {* Special Conditions of Use: Called with all ints off                   *}
  {*
*}
  {*
*}
  {* Error Conditions:                                                                                         *}
  {*
*}
```

```
  {****************************************************************}

VAR
  ptrsysg: ^longint;

begin
  with port^, cur_info_ptr^ do
         if in_break then
         begin   { remove from break_state }
           in_break := false;
           openwr5 := openwr5 - $10;  { disable break }
           with hwcontrol^ do
           begin
                  controlreg := 5;
                  if (rec_hs = hwhs) and last_stopped then
                    controlreg := openwr5 - (-128)  { no DTR }
                  else
                    controlreg := openwr5
           end { with }
           ptrsysg := pointer(bsysglob);
           if (nullink.fwd_link + ptrsysg^) <> ord(@nullink) then
                  START_NEW_REQUEST(port)  { start next pending request }
         end
         else
           if xmit_wait then
                  if cur_req_ptr <> nil then
                  begin
                    if xmit_hs <> xmt_xon then
                           xmit_wait := false;
                    if xmt_hs = xmit_delay then      { time up after CR or LF }
                           if xfer_count < num_bytes then
                             BYTEO(port)    { send next byte out }
                           else  {complete previous request }
                             FINISH_REQ(port, true)
                    else
                    begin  { handshake has been delayed too long }
                           cur_req_ptr^.hard_error := errbase + 7;
                           FINISH_REQ(port, false)
                    end
                  end
end; { xmit_to_hndl }


function DRIVER(* parameters: param_ptr): integer *);
  {****************************************************************}
  {*
*}
  {* Description: RS-232 external interface via DRIVERCALL       *}
  {*
*}
  {* Input Parameters: Parameters depend on function code                 *}
  {*
*}
  {* Output Parameters: none                                                                                  *}
  {*
*}
  {* Side Effects:  none                                                                                      *}
  {*
*}
  {* Special Conditions of Use: none                                                                 *}
  {*
*}
  {* Error Conditions:                                                                                        *}
  {*
*}
  {****************************************************************}

VAR
  prevints: intson_type;
  port_ptr: portrec_ptr;
  i: integer;
  char_in: int1;
  ptrsysg: ^longint;


  procedure WR_SCC(regno: int1; cal: int1);
```

```
       {****************************************************************}
       {*  Internal procedure to write to the SCC control registers   *}
       {*    Assures necessary delay between accesses.                                  *}
       {****************************************************************}
VAR
       i: integer;

begin

       {$R-} { rangecheck off }

       with port_ptr^.hwcontrol do
       begin
         controlreg := regno;
         i := 128;  { NOP to assure delay between accesses }
         controlreg := val
       end

       {$IFC debug2}
       {$R+}   { rangecheck back on if debug }
       {$ENDC}

end; {WR_SCC}


procedure INITIT;
       {****************************************************************}
       {*  Internal procedure to handle initialize "case" and reduce           *}
       {*    code size of main DRIVER procedure                                                  *}
       {****************************************************************}

VAR
       ptrsysg: ^longint;
       err, i, chan: integer;
       temp: longint;
       portacontrol, portbcontrol: int1recptr;
       p: params;
       dcrec: dc_rec;

begin
       INTSOFF(rsints, prevints);

       err := 0;
       portacontrol := pointer(iospacemmu*$20000 + $00203);
       portbcontrol := pointer(ord(portacontrol) - 2);
       ptrsysg := pointer(bsysglob);      { set up pointer to b_sysglobal_ptr }
       chan := parameters^.configptr^.iochannel;
       if not GETSPACE(sizeof(portrec), ptrsysg^, temp) then
         err := errbase + 6
       else
       begin
         parameters^.configptr^.cb_addr := temp;
         port_ptr := pointer(temp)
       end;

       { initialize either port A or B }

       if err = 0 then
       with port^ do
       begin
         cur_req_ptr := nil;

         if chan = 0 then
         begin { chan A initializations }
                xmtrr0 := $10;  { bit position for DSR signal }
                xmtzrr0 := 0;
                openwr14:= 3;    { port A uses baud rate generator }
                openwr5 := -22;
                hwcontrol := portacontrol;
                i := hwcontrol^.controlreg;  { make sure initially writes to reg 0 }
                WR_SCC(9, -118)  { = $8a, reset chan A }
         end
         else
         begin { chan B initializations }
                xmtrr0 := 0;
                xmtzrr0 := $20; { bit position for DSR signal }
                openwr14:= 1;
                openwr5 := -24;
                hwcontrol := portbcontrol;
                i := hwcontrol^.controlreg;  { make sure initially writes to reg 0 }
                WR_SCC(9, $4a)  { reset chan B }
        end;

        WR_SCC(4, $44);         { set operating mode }

        if chan = 0 then
                WR_SCC(11, $50)  { Port A uses baud rate generator }
        else
        begin
                WR_SCC(11, -48);  { = $D0 for oscillator source on Port B }
                for i := 1 to 1000 do ;  { kill time while oscillator starts }
        end;

        { set baud rate }

        p.configptr := parameters^.configptr;
        p_fnctn_code := dcontrol;
        dcrec.dversion := 2;
        dcrec.dcode := 5;
        dcrec.ar10[0] := 1200;  { set to 1200 baud }
        p.parptr := ord(@dcrec);
        i := DRIVER(@p);  { call internal device control routine }

        WR_SCC(10, 0);

        WR_SCC(3, -63);         { set receiver state to $41 or $C1 }

        WR_SCC(5, openwr5);     { set to standard modem signals }

        if chan = 0 then
                WR_SCC(15, $10)  { enable ints in SYNC A modem change }
        else
                WR_SCC(15, $20);  { or enable ints on CTS B modem change }

        WR_SCC($11, $13);  { enable desired interrupts: $13 or $17 }

        hwdata := pointer(ord(hwcontrol) + 4);
        in_break := false;
        restrt_in := false;
        restrt_out := false;
        prev_char := 0;
        autolf := false;
        nullink.fwd_link := ord(@nullink) - ptrsysg^;
        nullink.bkwd_link := nullink.fwd_link;
        discon_detect := false;
        xmit_wait := false;
        xmt_hs := hw_hs;

        { receiver initializations }

        no_block := true;
        rec_hs := hwhs;
        prity_char := 1;
        last_stopped := false;
        xmit_timeout := 20000;  { 20 seconds }
        ALARM_ASSIGN(xmit_ref, parameters^.configptr, rsints);
        if xmit_ref = 0 then
        begin
                err := 602;  { timer table full }
                RELSPACE(ord(port_ptr), ptrsysg^)
        end
        else
        begin  { initialize type-ahead buffer }
                size_typeah := 0;         { temporary value }
                p.configptr := parameters^.configptr;
                p.fnctn_code := dcontrol;
                dcrec.dversion := 2;
                dcrec.dcode := 9;
                dcrec.ar10[0] := 64;  { size of type-shead buffer }
                dcrec.ar10[1] := 16;  { low threshold }
                dcrec.ar10[2] := 32;  { high threshold }
                p.parptr := ord(@dcrec);
                { NOTE: There is code missing in the listing here. }
                {                 The missing parts are from here to the VAR }
                {                 of the next procedure.                                         }

         end
       end { with }
end; {INITIT}

procedure CONTROLIT;
       {****************************************************************}
       {*  Internal procedure to handle control "case" and reduce     *}
       {*    code size of main DRIVER procedure                                                  *}
       {****************************************************************}

VAR
       speed, i: integer;
       ptrsysg: ^longint;
       temp: longint;
       dc_rec_ptr: ^dc_rec;

begin
       with parameters^ do
       begin
         DRIVER := 0;
         if configptr^.iochannel = 0 then
                temp := 125000  { constant for baud rate computation }
         else
                temp := 115200;  { for baud rates }

         dc_rec_ptr := pointer(parptr);
         INTSOFF(rsints, prevints);
         with port_ptr^, dc_rec_ptr^ do
                if dversion <> 2 then
                  DRIVER := errbase + 0  { wrong application version }
                else
                case dcode of
                1: begin  { set parity and leave break state }
                          if configptr^.iochannel = 0 then
                                 openwr5 := -86    { = $AA, for port A }
                          else
                                 openwr5 := -88; { = $A8, port B always has RTS off }
                          in_break := false;  { removing break state }

                          case ar10[0] of
                          0: begin        { no parity, 8 data, 1 stop }
                                    WR_SCC(4, $44);
                                    WR_SCC(2, -63);         { = $C1 }
                                    openwr5 := openwr5 + $40;
                                    prity_char := 1         { disable parity err replacement }
                                  end;
                          1, 2: begin  { odd parity, 7 data, 1 stop }
                                    WR_SCC(4, $45);
                                    WR_SCC(3, $41);
                                    prity_char := 0
                                  end;
                          3, 4: begin  { even parity, 7 data, 1 stop }
                                    WR_SCC(4, $47);
                                    WR_SCC(3, $41);
                                    prity_char := -128  { = $80 }
                                  end;
                          end; { case ar10[0] }

                          if (rec_hs = hwhs) and last_stopped then
                                 WR_SCC(5, openwr5 - (-128))  { no DTR }
                          else
                                 WR_SCC(5, openwr5);

                          if odd(ar10[0]) then
                          begin
                                 WR_SCC($11, $13);  { ignore parity on input }
                                 prity_char := 1  { disable input parity err replacement }
                          end
                          else
                          begin
                                 WR_SCC($11, $17);  { enable input parity checking }
```

```
                 hwcontrol^.controlreg := $30  { ignore prior parity errors }
          end
        end;

2: begin  { set up for DSR handshake on output }
          xmit_hs := hw_hs;
          if configptr^.iochannel = 0 then
                 xmtrr0 := $10
          else
                 xmtzrr0 := $20;  { port B }
          xmit_wait := false  { clear XOFF state. fixes bug }
        end;

3: if cur_req_ptr <> nil then
          DRIVER := errbase + 9  { don't change xmit_hs during I/O }
        else
        begin  { set up for xon, xoff handshake on output }
          xmit_hs := xmt_xon;
          xmit_wait := false  { put in xon state. fixes bug }
        end;

4: begin  { set up for delay on cr, lf }
          xmt_hs := xmit_delay;
          crlfdelay := ar10[0];
          xmit_wait := false  { clear xoff state. fixes bug }
        end;

5: begin  { set baud rate }
          speed := (temp div ar10[0]) - 2;
          WR_SCC(14, 0);  { turn off baud rate generator }
          {$R-}  {rangecheck off}
          WR_SCC(12, speed);  { lsb }
          {$IFC debug2}
          {$R+}  { rangecheck back on if debug }
          {$ENDC}
          WR_SCC(13, speed div 256);  { msb }
          WR_SCC(14, openwr14)
        end;
6: no_block := (ar10[0] <> 0);  { set input blocking/not-blocking mode }

7: begin  { set input mode to hardware handshake }
          if num_typeah >= full_thresh then
          begin
                 WR_SCC(5, openwr5 - (-128));  { no DTR }
                 last_stopped := true;

                 {$IFC debug2}
                 if TRACE(DD, 5) then
                   writeln('  RS-232 DTR turned off.');
                 {$ENDC}

          end
          else
          begin
                 WR_SCC(5, openwr5);  { enable receive handshake signals }
                 last_stopped := false;

                 {$IFC debug2}
                 if TRACE(DD, 5) then
                   writeln(' RS-232 DTR turned on.');
                 {$ENDC}

          end;
          rec_hs := hwhs;
        end;

8: begin { set input mode to xon/xoff }
          rec_hs := rec_xon;
          if num_typeah >= full_thresh then
          begin
                 last_stopped := true;
                 hschar_pending := xoff_char;
                 BYTEO(port_ptr)
          end
          else
          begin
```

```
                 last_stopped := false;
                 hschar_pending := xon_char;
                 BYTEO(port_ptr)
          end
        end;

9: begin  { flush and redefine typeahead buffer }
          num_typeah := 0;
          get_typeah := put_typeah;
          hschar_pending := no_char;
          if last_stopped then
          begin
                 last_stopped := false;
                 if rec_hs = hwhs then
                 begin  { toggle modem signals }

                   {$IFC debug2}
                   if TRACE(DD, 5) then
                          writeln(' RS-232 DTR turned on.');
                   {$ENDC}

                   WR_SCC(5, openwr5)  { enable receive handshake signals }
                 end
                 else
                 begin  { send xon }
                   hschar_pending := xon_char;
                   BYTEO(port_ptr);
                 end;
          end;
          ptrsysg := pointer(bsysglob);
          if ar10[0] >= 0 then
          begin { allocate new buffer }
                 if size_typeah > 0 then
                   RELSPACE(ord(start_typeah), ptrsysg^);
                 size_typeah := ar10[0];
                 if size_typeah > 0 then
                   if not GETSPACE(size_typeah, ptrsysg^, temp) then
                          DRIVER := errbase + 6
                   else
                   begin
                          start_typeah := pointer(temp);
                          get_typeah := start_typeah;
                          put_typeah := get_typeah;
                          end_typeah := pointer(temp + size_typeah)
                   end
          end;

          if ar10[1] >= -1 then { -2 means don't change, -1 means disable }
                 empty_thresh := ar10[1];
          if ar10[2] >= -1 then { -2 means don't change, too big means disable }
                 full_thresh := ar10[2]
        end;

 10: begin { enable framing err as discon }
                 with hwcontrol^ do
                 begin
                   controlreg := 15;
                   i := 128;  { NOP to assure delay between accesses }
                   i := controlreg  { get current reg15 value }
                 end; { with }
                 if ar10[1] <> 0 then
                 begin
                   discon_detect := true;
                   if i >= 0 then
                          i := i + (-128)
                 end
                 else
                 begin
                   discon_detect := false;
                   if i < 0 then
                          i := i - (-128)
                 end
                 WR_SCC(15, i)  { set updated reg15 value }
          end;

11: begin        { set up for no handshake on output }
```

```
                               xmt_hs := hw_hs;
                               xmtrr0 := 0;
                               xmtzrr0 := 0;
                               xmit_wait := false  { clear xoff state. fixes bug }
                             end;

                    12: xmit_timeout := ar10[0] * 1000;
                             { set transmit-handshake timeout in seconds. <= 0 is infinite }

                    13:  { put into break state for specified period of time }
                             if cur_req_pre <> nil then
                               DRIVER := errbase + 9  { don't start break during I/O }
                             else
                             begin
                               ALARMRELATIVE(xmit_ref, ar10[0]);  { remain in break state until

alarm or dcontrol 1 }

                               if not in_break then
                               begin
                                      in_break := true;
                                      openwr5 := openwr5 + $10;          { enable break }
                                      if ((rec_hs = hwhs) and last_stopped) or (ar10[1] <> 0) then
                                        WR_SCC(5, openwr5 - (-128))      { no DTR }
                                      else
                                        WR_SCC(5, openwr5)  { enable receive handshake signals }
                               end;
                             end;

                    17: autolf := (ar10[0] <> 0);      { enable/disable auto LF }

                    otherwise
                      DRIVER := errbase + 9;
                    end; { case dcode }

                    INTSON(prevints)
           end { with }
    end; { controlit }

    procedure STARTIT;
           {****************************************************}
           {*  Internal procedure to handle SEQIO "case"                 *}
           {****************************************************}

    VAR
           ptrsysg: ^longint;
           leftlink: link_ptr;
           ext_ptr: seqextptr_type;

    begin
           with parameters^ do
           begin
             ext_ptr := pointer(req^.req_extent);
             if ext_ptr^.num_bytes <= 0 then
                    DRIVER := errbase + 1     { illegal parameters }
             else
             begin

                    {$IFC debug2}
                    if TRACE(DD, 10) then
                      if ext_ptr^.read_flag then
                             writeln('Reading RS-232.')
                      else
                             writeln(' Writing RS-232.');
                    {#ENDC}

                    with port_ptr^ do
                    begin   { add request to the device queue or finish the request now }
                      DRIVER := 0;  { no errors }
                      ptrsysg := pointer(bsysglob);  { set pointer to b_sysglobal_ptr }
                      INTSOFF(rsints, prevints);
                      leftlink := pointer(nullink.bkwd_link + ptrsysg^);
                      if ext_ptr^.read_flag and no_block and
                             ((num_typeah = 0) or (leftlink <> @nullink)) then
                      begin  { ok to complete request since non-blocking read w/o char }
                             ADJ_IO_CNT(false, ext_ptr^.buff_rdb_ptr);
```

```
                         if discon_detect and (hwcontrol^.controlreg < 0) then
                         begin   { assume disconnected cable on input framing error }
                           xmit_wait := false;
                           req^.hard_error := errbase + 8;
                           UNBLK_REQ(req, false)
                         end
                         else
                           UNBLK_REQ(req, true)
                  end
                  else
                  begin
                         ENQUEUE(req^.dev_chain, leftlink^, ptrsysg^);
                         if cur_req_ptr = nil then
                           if not in_break then
                                  START_NEW_REQUEST(port_ptr);
                  end;
                  INTSON(prevints)
                end { with port_ptr }
         end; {else}
       end { with parameters }
end; { startit }


procedure BYTEIN;
       {**********************************************************}
       {* Put received char in input buffer or type-ahead buffer *}
       {**********************************************************}
VAR
       i: integer;

begin
       with port_ptr^ do
       begin
         if char_in < 0 then { set "i" to char_in w/o parity bit set }
                i := char_in + $0080
         else
                i := char_in;
         if (xmit_hs = xmt_xon) and (i = $13) then
         begin { got xoff }

                {$IFC debug2}
                if TRACE(DD, 5) then
                  writeln(' RS-232 xoff received.');
                {$ENDC}

                xmit_wait := true;
                if cur_req_ptr <> nil then
                  if not cur_read_flag then { fixes bug }
                         if xmit_timeout > 0 then
                           ALARMRELATIVE(xmit_ref, xmit_timeout);
         end
         else
                if (xmit_hs = xmt_xon) and (i = $11) then
                begin
                  if xmit_wait then
                  begin  { got xon }

                         {$IFC debug2}
                         if TRACE(DD, 5) then
                           writeln(' RS-232 xon received.');
                         {$ENDC}

                         xmit_wait := false;
                         if cur_req_ptr <> nil then
                           if not cur_read_flag then { fixes bug }
                           begin
                                  ALARMOFF(xmit_ref);
                                  BYTEO(port_ptr)
                           end
                  end
                end
                else  { have real input character, put into typeahead buffer }
                  if (num_typeah + 1) <= size_typeah then
                  begin  {still room in buffer}
                         put_typeah^ := char_in;
                         put_typeah := pointer(ord(put_typeah) + 1);
```

```
                           if ord(put_typeah) = ord(end_typeah) then
                             put_typeah := start_typeah;      { wrap around }
                           num_typeah := num_typeah + 1;
                           if num_typeah = full_thresh then
                           begin
                             last_stopped := true;
                             if rec_hs = hwhs then
                             begin  { toggle modem signals }

                                    {$IFC debug2}
                                    if TRACE(DD, 5) then
                                      writeln(' RS-232 DTR turned off.');
                                    {$ENDC}

                                    WR_SCC(5, openwr5 - (-128))  { enable receive handshake signals }
                             end
                             else
                             begin  { send xoff }
                                    hschar_pending := xoff_char;
                                    BYTEO(port_ptr)
                             end
                           end;
                           if cur_req_ptr <> nil then
                             if cur_read_flag then
                                    COPYBYTES(port_ptr)
                    end
                    else  { char gets thrown away }
                    begin
                           if rec_hs = rec_xon then
                           begin { send xoff again }
                             hschar_pending := xoff_char;
                             BYTEO(port_ptr);
                             last_stopped := true;
                           end
                    end
         end; { with port_ptr }
  end;  { bytein }

{**********************************************************************}

begin { DRIVER }
  port_ptr := pointer(parameters^.configptr^.cb_addr);

  case parameters^.fnctn_code of
  seqio: STARTIT;

  dinit: INITIT;  { call subroutine to reduce size of DRIVER }

  dcontrol: CONTROLIT;  { call subroutine to reduce size }

  dalarms: XMT_TO_HNDL(port_ptr);  { alarm routine }

  dinterrupt:    begin  { ignore call version & returned function value for ints }

         with port_ptr^, hwcontrol^ do
           case parameters^.intpar of

           axmit:
                  begin
                    if hschar_pending <> no_char then
                           BYTEO(port_ptr)
                    else
                           if (cur_req_ptr = nil) or cur_read_flag or xmit_wait then
                             controlreg := $28  { ignore output interrupts }
                           else
                           with cur_info_ptr^ do
                             if waitb4next then
                             begin  { start up delay following CR or LF }
                                    waitb4next := false;
                                    ALARMRELATIVE(xmit_ref, crlfdelay);
                                    xmit_wait := true;
                                    controlreg := $28;  { start timer & kill interrupt }
                             end
                             else
                                    if xfer_count < num_bytes then
                                      BYTEO(port_ptr)         { send next byte out }
                           else
                           begin   { complete previous request }
                             controlreg := $28;  { disable transmit interrupts with $28 }
                             FINISH_REQ(port_ptr, true)
                           end
         end; {axmit}

  amodem: begin
           if cur_req_ptr <> nil then
                  if discon_detect and (controlreg < 0) then
                  begin   { assume disconnected cable on input framing error }
                    if xmit_wait then  { fixes bug }
                    begin
                           ALARMOFF(xmit_ref);
                           xmit_wait := false
                    end;
                    cur_req_ptr^.hard_error := errbase + 8;
                    FINISH_REQ(port_ptr, false)
                  end
                  else
                    if xmit_wait then
                           if xmit_hs = hw_hs then
                           begin   { we've been waiting for modem handshake to change }

                             {$IFC debug2}
                             if TRACE(DD, 5) then
                                    writeln('        RS-232 resume xmit after hw handshake. ');
                             {$ENDC}

                             xmit_wait := false;
                             ALARMOFF(xmit_ref);
                             BYTEO(port_ptr)
                           end;
                  controlreg := $10         { reset the modem interrupt state }
         end; {amodem}

  arecv:  begin
           char_in := hwdata^;     { get character }
           BYTEIN;        { put received char in input buffer or typeahead buffer }
         end;  {arecv}

  aerr: begin
           controlreg := 1;  { prepare to read register 1 with overrun error stat }
           i := 128;  { NOP to assure delay between accesses }
           i := controlreg;
           if prity_char <= 0 then          { testing for parity errs }
                  if ALLSET($10, i) then
                  begin   { parity error }
                    i := hwdata^;  { throw away real char }
                    char_in := prity_char;  { replace with parity error char: 00 or 80 }
                    BYTEIN;
                  end;
           controlreg := $30;  { reset input error }
         end; {aerr}

  end { case parameters^.intpar }

  port_ptr^.hwcontrol^.controlreg := $38;   { reset interrupt latch }
end; { dinterrupt case }

reqrestart:
  begin
         DRIVER := 0;

         {$IFC debug2}
         if TRACE(DD, 5) then
           writeln('  RS-232 restarting request.');
         {$ENDC}

         INTSOFF(rsints, prevints);

         with port_ptr^ do
         begin
           if restrt_in then
           begin
                  COPYBYTES(port_ptr);
```

```
 1:1168.                              restrt_in := false
 1:1169.                       end;
 1:1170.                       if restrt_out then
 1:1171.                       begin
 1:1172.                              BYTEO(port_ptr);
 1:1173.                              restrt_out := false
 1:1174.                       end
 1:1175.                     end;
 1:1176.
 1:1177.                     INTSON(prevints);
 1:1178.              end; {reqrestart}
 1:1179.
 1:1180.            ddown: begin
 1:1181.                     INTSOFF(rsints, prevints);
 1:1182.                     DRIVER := 0;
 1:1183.                     with port_ptr^ do
 1:1184.                     begin
 1:1185.                       if cur_req_ptr <> nil then
 1:1186.                              DRIVER := errbase { + ? }
 1:1187.                       else
 1:1188.                       begin
 1:1189.                              ptrsysg := pointer(bsysglob);      { set up pointer to b_sysglobal_ptr }
 1:1190.                              if size_typeah > 0 then
 1:1191.                                RELSPACE(ord(start_typeah), ptrsysg^);
 1:1192.                              ALARMRETURN(xmit_ref);
 1:1193.                              if parameters^.configptr^.iochannel = 0 then { reset channel }
 1:1194.                                WR_SCC(9, -118)         {=$8A}
 1:1195.                              else
 1:1196.                                WR_SCC(9, $4A);
 1:1197.                              parameters^.configptr^.cb_addr := ord(nil);
 1:1198.                              RELSPACE(ord(port_ptr), ptrsysg^);
 1:1199.                       end
 1:1200.                     end;
 1:1201.                     INTSON(prevints)
 1:1202.              end; {ddown}
 1:1203.
 1:1204.            otherwise
 1:1205.              DRIVER := errbase + 2;  { currently unimplemented w/ error }
 1:1206.     end; { case parameters^.fnctn_code }
 1:1207.   end; { DRIVER }
 1:1208.
 1:1209.  end.
 1:1210.

End of File: RS232.TEXT


Directory of files in Cross Reference:

 1: RS232.TEXT
level =  1.

a         1: 171,   1: 187,          1: 565,   1:1196.
adj_io_c 1: 374,   1: 939.
aerr    1:         29*,  1:1133.
alarm_as 1: 620.
alarmoff 1: 378,   1:1006,  1:1105,   1:1122.
alarmrel 1: 203,   1: 339,  1: 879,   1: 989,          1:1085.
alarmret 1:1192.
allset  1: 127,   1: 154,   1: 154,   1: 156,          1:1138.
amodem  1:         27*,  1:1099.
ar10    1: 584=,  1: 633=,  1: 634=,  1: 635=,         1: 681,   1: 705,          1: 736,   1: 741,
          1: 751,   1: 819,          1: 823,   1: 836,          1: 837,   1: 838,          1: 839,   1: 849,
          1: 871,   1: 879,          1: 886,   1: 893.
arecv   1:         28*,  1:1128.
autolf  1:         55*,  1: 168,     1: 606=,  1: 893=.
axmit   1:         26*,  1:1073.
bkwd_lin 1: 608=,  1: 935.
bsysglob 1: 371,   1: 435,  1: 530,   1: 818,          1: 933,   1:1189.
buff_off 1: 159,   1: 237.
buff_rdb 1: 159,   1: 190,  1: 237,   1: 262,          1: 374,   1: 939.
cal       1: 484*.
cb_addr 1: 536=,  1:1057,   1:1197=.
chain_fo 1: 318.
chan    1: 518*,  1: 531=,  1: 547,   1: 570,          1: 594.
char_in 1: 480*,  1: 973,   1: 974,   1: 976,          1:1014,   1:1129=,         1:1141=.
```

```
chr       1: 181.
cnt       1: 229*,  1: 244=,         1: 245,   1: 246=,         1: 250,   1: 252,          1: 255,   1: 256,
          1: 263,   1: 289.
configpt 1: 531,   1: 536,  1: 580=,  1: 580,          1: 620,   1: 629=,         1: 629,   1: 662,
          1: 675,   1: 719,          1:1057,   1:1193,          1:1197.
controlr 1:        39*,  1: 123,     1: 163=,  1: 205=,         1: 278=,  1: 280=,         1: 324,   1: 429=,
          1: 431=,  1: 433=,         1: 498=,  1: 500=,         1: 554,   1: 564,          1: 713=,  1: 845=,
          1: 847,   1: 940,          1:1079=,  1:1087=,         1:1094=,  1:1101,          1:1125=,  1:1134=,
          1:1136,   1:1144=,         1:1149=.
copybyte 1: 332,   1:1040,  1:1167.
crlfdela 1:        73*,  1: 736=,    1:1085.
cur_info 1:        45*,  1: 157,     1: 234,   1: 320=,         1: 321,   1: 372,          1: 422,   1:1081.
cur_read 1:        46*,  1: 323=,    1: 331,   1: 379,          1: 987,   1:1004,          1:1039,   1:1078.
cur_req_ 1:        44*,  1: 159,     1: 237,   1: 318=,         1: 319,   1: 320,          1: 327,   1: 375,
          1: 376,   1: 393=,         1: 441,   1: 452,          1: 545=,  1: 726,          1: 875,   1: 952,
          1: 986,   1:1003,          1:1038,   1:1078,          1:1100,   1:1108,          1:1185.
d         1: 169,   1: 187.
dalarms 1:1066.
data_ptr 1: 116*,  1: 174=,          1: 176,   1: 230*,         1: 243=,  1: 255,          1: 256=,  1: 256,
          1: 259.
dc_rec  1: 522,   1: 656.
dc_rec_p 1: 656*,  1: 667=,          1: 669.
dcode   1: 583=,  1: 632=,  1: 673.
dcontrol 1: 581,   1: 630,  1:1064.
dcrec   1: 522*,  1: 582,  1: 583,   1: 584,          1: 585,   1: 631,           1: 632,   1: 633,
          1: 634,   1: 635,          1: 636.
dd        1: 133,   1: 143,          1: 180,   1: 198,          1: 272,   1: 386,          1: 760,   1: 771,
          1: 806,   1: 923,          1: 981,   1: 998,          1:1026,   1:1117,          1:1157.
ddown   1:1180.
dequeue 1: 376.
dev_chai 1: 376,   1: 951.
dinit   1:1062.
dinterru 1:1068.
discon_d 1:        52*,  1: 324,     1: 609=,  1: 851=,         1: 857=,  1: 940,          1:1101.
driverde 1:        11.
driversu 1:        13.
dversion 1: 582=,  1: 631=,          1: 670.
empty_th 1:        84*,  1: 265,     1: 837=.
end_type 1:        86*,  1: 250,     1: 832=,  1:1016.
enqueue 1: 951.
err       1: 117*,  1: 159,          1: 161,   1: 367*,         1: 518*,  1: 527=,         1: 533=,  1: 542,
          1: 623=.
errbase 1:         25*,  1: 327,     1: 452,   1: 533,          1: 671,   1: 727,          1: 826,   1: 876,
          1: 896,   1: 918,          1: 943,   1:1108,          1:1186,   1:1205.
errnum  1: 229*,  1: 237,  1: 239.
ext_ptr 1: 911*,  1: 916=,  1: 917,   1: 924,          1: 936,   1: 939.
fnctn_co 1: 630=,  1:1059.
forward 1:         94.
freeze_s 1: 159,   1: 237.
full_thr 1:        83*,  1: 754,     1: 781,   1: 839=,         1:1019.
fwd_link 1: 390,   1: 436,  1: 607=,  1: 608.
get_type 1:        85*,  1: 250,     1: 255,   1: 257=,         1: 259,   1: 260=,         1: 260,   1: 797=,
          1: 830=,  1: 831.
getspace 1: 532,   1: 825.
hard_err 1: 327=,  1: 452=,          1: 943=,  1:1108=.
hschar_p 1:        89*,  1: 125,     1: 129,   1: 149=,         1: 285=,  1: 382,          1: 784=,  1: 790=,
          1: 798=,  1: 814=,         1:1034=,  1:1046=,         1:1075.
hw_hs   1:         32*,  1: 153,     1: 611,   1: 718,          1: 865,   1:1113.
hwcontro 1:        48*,  1: 121,     1: 276,   1: 324,          1: 427,   1: 496,          1: 553=,  1: 554,
          1: 563=,  1: 564,          1: 601,   1: 713,          1: 843,   1: 940,          1:1070,   1:1149.
hwdata  1:         47*,  1: 137=,    1: 147=,  1: 184=,         1: 601=,  1:1129,          1:1140.
hwhs    1:         33*,  1: 268,     1: 430,   1: 616,          1: 700,   1: 776,          1: 802,   1: 886,
          1:1022.
i         1: 229*,  1: 250=,         1: 251,   1: 252=,         1: 255,   1: 256,          1: 259,   1: 260,
          1: 279=,  1: 479*,         1: 490*,  1: 499=,         1: 518*,  1: 554=,         1: 564=,  1: 575=,
          1: 586=,  1: 653*,         1: 846=,  1: 847=,         1: 852,   1: 853=,         1: 853,   1: 858,
          1: 859=,  1: 859,          1: 861,   1: 968*,         1: 974=,  1: 976=,         1: 977,   1: 992,
          1:1135=,  1:1136=,         1:1138,   1:1140=.
implemen 1:        17.
in_break 1:        60*,  1: 423,     1: 425=,  1: 602=,         1: 679=,  1: 882,          1: 884=,  1: 953.
in_servi 1: 319.
int1    1:         35,   1:  39,     1:       50,   1:  51,     1:       54,   1:  68,     1:       69,   1:  81,
          1: 480,   1: 484,          1: 484.
int1ptr 1:         35*,  1:  47,     1:       85,   1:  86,     1: 116,   1: 230.
int1rec 1:         37,   1:  38*.
int1recp 1:        37*,  1:  48,     1: 520.
```

```
interfac 1:         7.
intpar  1:1071.
intsoff 1: 525,   1: 668,   1: 934,   1:1161,          1:1181.
intson  1: 899,   1: 956,   1:1177,   1:1201.
intson_t 1: 477.
iochanne 1: 531,   1: 662,  1: 675,   1: 719,          1:1193.
iospacem 1: 528.
last_sto 1:        79*,  1: 264,     1: 267=,  1: 430,          1: 618=,  1: 700,          1: 757=,  1: 768=,
          1: 783=,  1: 789=,         1: 799,   1: 801=,         1: 886,   1:1021=,         1:1048=.
leftlink 1: 910*,  1: 935=,          1: 937,   1: 951.
link_ptr 1: 910.
linkage 1:         49.
moveleft 1: 255,   1: 259.
no_block 1:        78*,  1: 291,     1: 615=,  1: 751=,         1: 936.
no_char 1:         89*,  1: 125,     1: 149,   1: 382,          1: 798,   1:1075.
nullink 1: 318,   1: 390,   1: 390,   1: 436,          1: 436,   1: 607,          1: 607,   1: 608,
          1: 608,   1: 935,          1: 937.
nulllink 1:        49*.
num_byte 1: 244,   1: 291,  1: 446,   1: 917,          1:1090.
num_type 1:        82*,  1: 235,     1: 245,   1: 246,          1: 263=,  1: 263,          1: 265,   1: 754,
          1: 781,   1: 796=,         1: 937,   1:1012,          1:1018=,  1:1018,          1:1019.
odd       1: 705.
ok_flag 1:         94*,  1: 375,     1: 387.
openwrl4 1: 551=,  1: 561=,          1: 749.
openwr5 1:         50*,  1: 280,     1: 426=,  1: 426,          1: 431,   1: 433,          1: 552=,  1: 562=,
          1: 592,   1: 676=,         1: 678=,  1: 685=,         1: 685,   1: 701,          1: 703,   1: 756,
          1: 767,   1: 810,          1: 885=,  1: 885,          1: 887,   1: 889,          1:1030.
openwrl4 1:        51*.
p         1: 521*,  1: 580,          1: 585,   1: 586,          1: 629,   1: 630,          1: 636.
p_fnctn_ 1: 581=.
param_pt 1:        15.
paramete 1:        15*,  1: 531,     1: 536,   1: 580,          1: 620,   1: 629,          1: 659,   1: 914,
          1:1057,   1:1059,          1:1071,   1:1193,          1:1197.
params  1: 521.
parptr  1: 585=,  1: 636=,  1: 667.
pointer 1: 174,   1: 243,   1: 256,   1: 260,          1: 320,   1: 371,          1: 435,   1: 528,
          1: 529,   1: 530,          1: 537,   1: 601,          1: 667,   1: 818,          1: 829,   1: 832,
          1: 916,   1: 933,          1: 935,   1:1015,          1:1057,   1:1189.
port_ptr 1: 478*,  1: 496,  1: 537=,  1: 624,          1: 669,   1: 785,          1: 791,   1: 815,
          1: 930,   1: 954,          1: 971,   1:1007,          1:1035,   1:1040,          1:1047,   1:1057=,
          1:1066,   1:1070,          1:1076,   1:1091,          1:1095,   1:1109,          1:1123,   1:1149,
          1:1163,   1:1167,          1:1172,   1:1183,          1:1198.
portacon 1: 520*,  1: 528=,          1: 529,   1: 553.
portbcon 1: 520*,  1: 529=,          1: 563.
portrec 1:         42,   1:  43*,    1: 532.
portrec_ 1:        42*,  1:  94,     1:       97,   1: 210,     1: 297,   1: 398,          1: 478.
prev_cha 1:        54*,  1: 169,     1: 169,   1: 171=,         1: 176=,  1: 181,          1: 184,   1: 187,
          1: 187,   1: 605=.
prevints 1: 477*,  1: 525,  1: 668,   1: 899,          1: 934,   1: 956,          1:1161,   1:1177,
          1:1181,   1:1201.
prity_ch 1:        81*,  1: 617=,    1: 686=,  1: 691=,         1: 696=,  1: 708=,         1:1137,   1:1141.
ptrsysg 1: 368*,  1: 371=,  1: 376,   1: 390,          1: 419*,  1: 435=,         1: 436,   1: 481*,
          1: 517*,  1: 530=,         1: 532,   1: 607,          1: 624,   1: 654*,         1: 818=,  1: 822,
          1: 825,   1: 909*,         1: 933=,  1: 935,          1: 951,   1:1189=,         1:1191,   1:1198.
put_type 1:        85*,  1: 797,     1: 831=,  1:1014=,         1:1015=,  1:1015,          1:1016,   1:1017=.
read_fla 1: 323,   1: 924,  1: 936.
real_add 1: 118*,  1: 160,  1: 174,   1: 231*,         1: 238,   1: 243.
rec_hs  1:         77*,  1: 268,     1: 430,   1: 616=,         1: 700,   1: 776=,         1: 780=,  1: 802,
          1: 886,   1:1022,          1:1044.
rec_xon 1:         33,   1: 780,     1:1044.
recv_hs 1:         33*,  1:  77.
regno   1: 484*,  1: 498.
relspace 1: 624,   1: 822,  1:1191,   1:1198.
req       1: 916,   1: 943,          1: 944,   1: 947,          1: 951.
req_exte 1: 320,   1: 916.
reqptr_t 1:        44.
reqresta 1:1152.
reqsrv_f 1: 319=.
reqstatu 1: 319.
restrt_i 1:        53*,  1: 240=,    1: 380=,  1: 603=,         1:1165,   1:1168=.
restrt_o 1:        53*,  1: 164=,    1: 383=,  1: 604=,         1:1170,   1:1173=.
rs232    1:         2.
rsints  1: 525,   1: 620,   1: 668,   1: 934,          1:1161,   1:1181.
send_hs 1:         32*,  1:  59.
seqextpt 1:        45,   1: 911.
seqio   1:1060.
```

```
size_typ 1:        80,   1: 628=,    1: 821,   1: 823=,         1: 824,   1: 825,          1: 832,   1:1012,
          1:1190.
sizeof  1: 532.
speed   1: 653*,  1: 741=,  1: 744,   1: 748.
start_ty 1:        85*,  1: 257,     1: 822,   1: 829=,         1: 830,   1:1017,          1:1191.
status  1: 117*,  1: 123=,  1: 127,   1: 154,          1: 154,   1: 156.
temp    1: 519*,  1: 532,   1: 536,   1: 537,          1: 655*,  1: 663=,         1: 665=,  1: 741,
          1: 825,   1: 829,          1: 832.
trace   1: 133,   1: 143,   1: 180,   1: 198,          1: 272,   1: 386,          1: 760,   1: 771,
          1: 806,   1: 923,          1: 981,   1: 998,          1:1026,   1:1117,          1:1157.
unblk_re 1: 375,   1: 944,  1: 947.
unfreeze 1: 190,   1: 262.
unit    1:          2.
val       1: 500.
waitb4ne 1:        72*,  1: 188=,    1: 335=,  1:1082,          1:1084=.
xfer_cou 1: 174,   1: 175=,         1: 175,   1: 243,          1: 244,   1: 289=,         1: 289,   1: 291,
          1: 446,   1:1090.
xmit_del 1:        32,   1: 186,     1: 445,   1: 735.
xmit_hs 1: 443,   1: 718=,  1: 730=,  1: 977,          1: 992,   1:1113.
xmit_ref 1:        62*,  1: 203,     1: 339,   1: 378,          1: 620,   1: 621,          1: 879,   1: 989,
          1:1006,   1:1085,          1:1105,   1:1122,          1:1192.
xmit_tim 1:        65*,  1: 202,     1: 203,   1: 338,          1: 339,   1: 619=,         1: 871=,  1: 988,
          1: 989.
xmit_wai 1:        61*,  1: 204=,    1: 326=,  1: 336,          1: 377,   1: 440,          1: 444=,  1: 610=,
          1: 723=,  1: 731=,         1: 737=,  1: 868=,         1: 942=,  1: 985=,         1: 994,   1:1002=,
          1:1078,   1:1086=,         1:1103,   1:1106=,         1:1112,   1:1121=.
xmit_xon 1:        32*.
xmt_hs  1:         59*,  1: 153,     1: 186,   1: 336,          1: 445,   1: 611=,         1: 735=,  1: 865=.
xmt_to_h 1:1066.
xmt_xon 1: 336,   1: 443,   1: 730,   1: 977,          1: 992.
xmtrr0  1:         68*,  1: 154,     1: 549=,  1: 559=,         1: 720=,  1: 866=.
xmtzrr0 1:         69*,  1: 154,     1: 550=,  1: 560=,         1: 722=,  1: 867=.
xoff_cha 1:        89*,  1: 129,     1: 784,   1:1034,          1:1046.
xon_char 1:        89,   1: 285,     1: 790,   1: 814.

Procedures

bytein  1: 963*,  1:1130,   1:1142.
byteo   1:         97*,  1: 286,     1: 342,   1: 447,          1: 785,   1: 791,          1: 815,   1:1007,
          1:1035,   1:1047,          1:1076,   1:1091,          1:1123,   1:1172.
controli 1: 646*,  1:1064.
finish_r 1:        94*,  1: 292,     1: 328,   1: 348*,         1: 449,   1: 453,          1:1095,   1:1109.
initit  1: 510*,  1:1062.
port    1:         94*,  1:  97*,    1: 121,   1: 210*,         1: 234,   1: 286,          1: 292,   1: 297*,
          1: 316,   1: 328,          1: 332,   1: 342,          1: 372,   1: 391,          1: 398*,  1: 422,
          1: 437,   1: 447,          1: 449,   1: 453,          1: 543.
start_ne 1: 297*,  1: 391,  1: 437,   1: 954.
startit 1: 903*,  1:1060.
wr_scc  1: 484*,  1: 555,   1: 565,   1: 568,          1: 571,   1: 574,          1: 588,   1: 590,
          1: 592,   1: 595,          1: 597,   1: 599,          1: 683,   1: 684,          1: 689,   1: 690,
          1: 694,   1: 695,          1: 701,   1: 703,          1: 707,   1: 712,          1: 742,   1: 744,
          1: 748,   1: 749,          1: 756,   1: 767,          1: 810,   1: 861,          1: 887,   1: 889,
          1:1030,   1:1194,          1:1196.
xmit_to_ 1: 398*.

Functions

driver  1:         15*,  1: 459*,    1: 586,   1: 661=,         1: 671=,  1: 727=,         1: 826=,  1: 876=,
          1: 896=,  1: 918=,         1: 932=,  1:1154=,         1:1182=,  1:1186=,         1:1205=.

Declaration Character : '*'
Assignment Character :  '='
```

```
Cross Reference Listing:


Beginning of file: RS232MAIN.TEXT

 1:      1.
 1:      2.  PROGRAM DUMMYMAIN;
 1:      3.
 1:      4.    USES
 1:      5.         {$U object/driverdefs.obj}
 1:      6.           driverdefs,
 1:      7.         {$U object/driversubs.obj}
 1:      8.           driversubs,
 1:      9.         {$U object/rs232.obj}                         { these 2 lines vary from driver }
 1:     10.           rs232;                                                       {                           to driver
         }
 1:     11.
 1:     12.    VAR
 1:     13.         i: integer;
 1:     14.         p: param_ptr;
 1:     15.
 1:     16.    begin
 1:     17.         i := driver(p);
 1:     18.    end;

End of File: RS232MAIN.TEXT


Directory of files in Cross Reference:

 1: RS232MAIN.TEXT
level =  1.

driver  1:         17.
driverde 1:         6.
driversu 1:         8.
dummymai 1:         2.
i         1:       13*,  1:  17=.
p         1:       14*,  1:  17.
param_pt 1:        14.
program 1:          2.
rs232   1:         10.

Declaration Character : '*'
Assignment Character :  '='
```

Cross Reference Listing:

Beginning of file: DRIVERDEFS.TEXT

```

UNIT DRIVERDEFS;         { UNIT NEEDED BY CONFIGURABLE DRIVERS }

        { By Dave Offen }
        { Copyright 1983, Apple Computer Inc. }

INTERFACE

{$U-} { DON'T use any iospaslib definitions in place of explicitly USE -d modules }
{$X-} { Disallow automatic stack expansion in system code }

{$SETC DEBUG:=TRUE}
{$SETC DEBUG1:=FALSE}
{$SETC DEBUG2:=FALSE}
{$SETC DEBUG3:=FALSE}
{$SETC OS1S:=TRUE}


CONST

  {**************************************************************************
  {**
  {** The delineated values are also defined in PASCALDEFS assembly language routine.
  {**
  {**************************************************************************

       { valid values in INTSOFF_TYPE }
  {**} allints = $700;   { all interrupts off}
               rsints   = $600;   { rs232 interrupts off }
               slotints = $500;   { 3 I/O slots interrupts off }
               copsints = $200;   { cops interrupts off }
               vertints = $100;   { vertical retrace interrupts off }
               twigints = $100;   { twiggy interrupts off }
               winints  = $100;   { winchester disk interrupts off }
               clokints = $100;   { clock interrupts off }
               clkonints = 0;              { clock interrupts on; lower off }

  {**} bsysglob = $200;           { memory addr of sysglobal base -- same as PASCALDEFS }
  {**} portcbofset = -24609;      { this MUST be the same as port_cb_ptrs in PASCALDEFS }

  {**} maxdev = 39;               { size of device table }

               size_rspec_info = 3; { size of specific info in reqblk }


               max_ename = 32;    { size of e_name string }

               iospacemmu = 126;           { mmu used to map the system I/O space }


{**} dinterrupt = 0;              { valid function code values }
        dinit = 1;
        ddown = 2;
        dskunclamp = 3;
        dskformat = 4;
        seqio = 5;
        dskio = 6;
        dcontrol = 7;
        reqrestart = 8;
        ddiscon = 9;

        dattach = 12;
        hdinit = 13;
        hdskio = 14;
        dunattach = 16;
        dalarms = 17;
        hddown = 18;

       { Definition of 'slot' # for Lisa Devices }
        cd_slot1 = 0;             { the basic 3 slots }
        cd_slot2 = 1;
```

```
         cd_slot3 = 2;
         cd_slot4 = 3;            { 1.75's 2 expansion slots }
         cd_slot5 = 4;
         cd_scc = 5;                        { scc chip controlling two serial channels }
         cd_soft = 6;                       { twiggy/sony or whatever else serves as built-in floppy }
         cd_hard = 7;                       { widget or whatever serves as built-in hard disk }
         cd_paraport = 8;                   { parallel port }
         cd_console = 9;                    { alternate and primary console }

                                                              { 10-14 are still available }


 TYPE
   int1 = -128..127; { 1 byte integer }
   int2 = integer;        { 2 byte integer }
   int4 = longint;        { 4 byte integer }
   relptr = int2;         { 2 byte pointer }
   absptr = int4;         { 4 byte pointer }

   OSPORTION=(MM, PM, EM, EC, AC, FS, DD, INIT, TRECORD, SPARE1, SPARE2, SPARE3, SPARE4);
   link_ptr = ^linkage;  {pointer to a linkage record }
   linkage = record
                            fwd_link: relptr;        { forward link }
                            bkwd_link: relptr;       { backward link }
                          end;

   intsoff_type = $100..$700;               { interrupts off parameter }
   intson_type = 0..$700;                            { interrupts on parameter }

{*********************************************}
{*                                                                                              *}
{* COMMON HEADER FOR REQUEST BLOCKS AND PCBS *}
{*                                                                                              *}
{*********************************************}

rb_type = (pcb_type, reqblk_type, ecm_type, spr1, spr2, spr3);
rbheader_type = ^rb_headT;
rb_headT = record
                         header: linkage;
                         kind: rb_type;
                  end;


{******************************}
{*                                                            *}
{* I/O STRUCTURES FOR DRIVERS *}
{*                                                            *}
{******************************}

{**} rec_port_cb = record
{**}                              slotx: array[0..2] of absptr;
{**}                              rs232: absptr;     { 2 channels }
{**}                              via2: absptr;      { COPS, SPEAKER, TIMER }
{**}                              via1: absptr;      { hard disk, parallel printer }
{**}                              screen: absptr;
{**}                              floppy: absptr;  { 2 drives & parameter memory }
{**}                        end;

array10ptr = ^array10;
array10 = array[0..9] of longint; { used for passing arbitrary-length parameters }

reqptr_type = ^reqblk;

e_name = string[max_ename];

{**} devtype = (diskdev, pascalbd, seqdev, bitbkt, non_io, spar1, spar2, spar3);
{**} ptrdevrec = ^devrec;

{**} devrec = record
{**}               entry_pt: absptr;        { driver address - nil if uninitialized }
{**}               cb_addr: absptr;         { ptr to control block for this device }
{**}               ext_addr: absptr;        { ptr to additional device info }
                            drvrec_ptr: absptr;      { ptr to driver-loading control record.
                                                                                  Nil means non-configurable. Defined
in genio }
                            devname: e_name;         { device name }
                            slot_no: int1;                    { 0-2 or -1 if built-in }
                            iochannel: int1;         { channel on hardware controller }
                            device_no: int1;         { device on channel }
                            devt: devtype;                    { device type }
                            blockstructured: boolean; { true if device is block structured }
                            permanent: boolean;      { don't unload when DOWNing }
                            self_ident: integer;  { from controller card - for debugging }
                            required_drve: ptrdevrec; { ptr to "higher" configinfo - nil if none }
                            permreq_ptr: reqptr_type; { ptr to pre-allocated req blk - nil if none }
                            preq_avail: boolean;  { true when permreq_ptr currently avail for use }
                          end;

{**} ext_diskconfig = record  { extension info for disk devrec }
{**}                                 hentry_pt: absptr;       { entry pt for Hdisk calls }
{**}                                 num_bloks: longint;      { size of device }
{**}                                 strt_blok: longint;      { virtual volume start address }
{**}                                 fs_strt_blok: longint;   { offset within virtual volume to }
{**}                                                                                              { file system }
                                              removable, ejectable: boolean;
{**}                               end;

{**} configtype = array[0..maxdev] of ptrdevrec;

{**} timestmp_interval = record
{**}                                         sec: longint;
{**}                                         msec: 0..999;
{**}                                       end;

       reqsts_type = record                 { request status }
                                           reqsrv_f: (active, in_service, complete);
                                           reqsuccess_f: boolean; { complete successfully or not }
                                           reqabt_f: boolean;          { abort pending or not }
                                    end;

       reqblk = record                      { request block }
                            pcb_chain: rb_headT;  { pcb chain, block header }
                            dev_chain: linkage;      { device or pipe chain }
                            list_chain: linkage;  { list of request blocks for blocking }
                            block_p_f: boolean;      { block process flag }
                            blk_in_pcb: int1;        { generic block type for future changes }
                            reqstatus: reqsts_type;          { request status }
                            operatn: int1;                    { 0=write, 1=read, 2=format, 3=unclamp... }
                            cfigptr: ptrdevrec;      { identifies to which device this request
                                                                                             belongs if it's an I/O request }
                            req_extent: absptr;      { extdptr_type when devt=diskdev
                                                                                             seqextptr_type when devt=seqdev }
                            hard_error: integer;  { error code when reqstatus.reqsuccess_f
                                                                                             = FALSE }
                            reqspec_info: longint;   { specific information such as sort
                                                                                                      key for device's queue }
                          end;

       disk_io_type = (with_header, without_header, raw_io, chained_hdrs);
       pagelabel = packed record
                                  version: integer;
                                  datastat: (dataok, datamaybe, databad);
                                  filler: -32..31;
                                  volume: int1;
                                  fileid: integer;
                                  dataused: integer;
                                  abspage: int4;
                                  relpage: int4;
                                  fwdlink: int4;
                                  bkwdlink: int4;
                                  end;

       extdptr_type = ^disk_extend;
       disk_extend = record
                                  xfer_count: longint;
                                  read_flag: boolean;
                                  buff_rdb_ptr: absptr;
                                  buff_offset: absptr;
                                  resolved_addr: absptr;      { results of freeze_seg }
                                  blkno: longint;
                                  soft_hdr: pagelabel;
                                  last_fwd_link: int4;
```

```
                           last_data_used: integer;
                           io_mode: disk_io_type;
                           num_chunks: integer;
                           misc1: integer;
                           misc2: longint;
                           end;

seqextptr_type = ^seq_extend;
seq_extend = record
                           xfer_count: longint;
                           read_flag: boolean;
                           buff_rdb_ptr: absptr;
                           buff_offset: absptr;
                           resolved_addr: absptr;      { results of freeze_seg }
                           num_bytes: longint;
                           end;

param_ptr = ^params;
params = record
                           configptr: ptrdevrec;
                           case fnctn_code: integer of
                             dskunclamp, ddown, hddown, dskformat, dinit, hdinit, ddiscon: ();
                             seqio, reqrestart, dskio: (req: reqptr_type);
                             dinterrupt, dalarms: (intpar: integer);
                             dcontrol: (parptr: absptr);
                             dattach: (n_configptr: ptrdevrec);
                             dunattach: (o_configptr: ptrdevrec; still_inuse: boolean);
                             hdskio: (c_cmd: integer; c_sector: longint);
                           end;

dc_rec = record  { for dcontrol }
                     dversion: integer;
                     dcode: integer;
                     ar10: array[0..9] of longint;
                   end;


{ Control Block for hard disk device }
hdiskcb_ptr = ^hdisk_cb;

{ NOTE: Definitions marked with ** may have corresponding definitions in }
{                 Assembly language portions of drivers.
}

hdisk_cb = record           { device/driver info for all Apple format hard disks }
          {**}     config_addr: ptrdevrec;    { config entry address - must be 1st }
          {**}     ext_ptr: absptr;                             { ptr to driver specific info }
          {**}     raw_data_ptr: longint;     { points to start of data }
          {**}     raw_header_ptr: ^pagelabel; { points to start of header }
          {**}     x_leng: longint;                             { offset to data }
          {**}     sect_left: integer;                 { for current request }
          {**}     v_flag: boolean;                             { re-read all writes? }
                            restrt_count: integer;     { the retry count for this req }
                            restrt_limit: integer;     { max number of restarts allowed }
                            total_restarts: longint;   { # of restarts of state machine }
                            soft_header: pagelabel;    { the buffer used to copy header }
                            int_prio: intsoff_type;    { the interrupt priority of this }
                                                                                           { drive }
                            cur_info_ptr: extdptr_type; { ptr to cur req's extension }
                            req_hd_ptr: reqptr_type;   { next request to be serviced }
                            dummy_req_ptr: reqptr_type; { phoney request at cyl -1 or 32767 }
                            cur_num_requests: integer;  { not counting dummy request }
                            worstwarning: integer;    { worst warning encountered in }
                                                                                           { this request }
                     end;



{*********************************}
{*                                                                       *}
{* DEFINITION OF SYSGLOBAL CELLS *}
{*                                                                       *}
{*********************************}

{ port_cb_ptrs: rec_port_cb;         port control block for drivers }
{ configinfo: configtype;            table of configured devices }
```

```
 1: 298.          { sysa5: longint;                                    A5 required for pascal }
 1: 299.
 1: 300.    IMPLEMENTATION
 1: 301.
 1: 302.    end.
 1: 303.

End of File: DRIVERDEFS.TEXT


Directory of files in Cross Reference:

 1: DRIVERDEFS.TEXT
level =  1.

abspage 1: 207*.
absptr  1:         89*,  1: 122,     1: 123,   1: 124,          1: 125,   1: 126,          1: 127,   1: 141,
          1: 142,   1: 143,          1: 144,   1: 160,          1: 191,   1: 217,          1: 218,   1: 219,
          1: 234,   1: 235,          1: 236,   1: 247,          1: 268.
ac        1:       91*.
active  1: 176*.
allints 1:         28*.
ar10    1: 256*.
array10 1: 130,   1: 131*.
array10p 1: 130*.
bitbkt  1: 137*.
bkwd_lin 1:        95*.
bkwdlink 1: 210*.
blk_in_p 1: 186*.
blkno   1: 220*.
block_p_ 1: 185*.
blockstr 1: 151*.
bsysglob 1:        38*.
buff_off 1: 218*,  1: 235*.
buff_rdb 1: 217*,  1: 234*.
c_cmd   1: 250*.
c_sector 1: 250*.
cb_addr 1: 142*.
cd_conso 1:        79*.
cd_hard 1:         77*.
cd_parap 1:        78*.
cd_scc  1:         75*.
cd_slot1 1:        70*.
cd_slot2 1:        71*.
cd_slot3 1:        72*.
cd_slot4 1:        73*.
cd_slot5 1:        74*.
cd_soft 1:         76*.
cfigptr 1: 189*.
chained_ 1: 199.
clkonint 1:        36*.
clokints 1:        35*.
complete 1: 176.
config_a 1: 267*.
configpt 1: 242*.
configty 1: 168*.
copsints 1:        31*.
cur_info 1: 280*.
cur_num_ 1: 283*.
dalarms 1:         66*,  1: 246*.
databad 1: 202.
datamayb 1: 202*.
dataok  1: 202*.
datastat 1: 202*.
dataused 1: 206*.
dattach 1:         62*,  1: 248*.
dc_rec  1: 253*.
dcode   1: 255*.
dcontrol 1:        58*,  1: 247*.
dd        1:       91*.
ddiscon 1:         60*,  1: 244.
ddown   1:         53*,  1: 244*.
dev_chai 1: 183*.
device_n 1: 149*.
devname 1: 146*.
devrec  1: 138,   1: 140*.
```

```
devt    1: 150*.
devtype 1: 137*,  1: 150.
dinit   1:         52*,  1: 244*.
dinterru 1:        51*,  1: 246*.
disk_ext 1: 213,   1: 214*.
disk_io_ 1: 199*,  1: 224.
diskdev 1: 137*.
driverde 1:         2.
drvrec_p 1: 144*.
dskforma 1:        55*,  1: 244*.
dskio   1:         57*,  1: 245*.
dskuncla 1:        54*,  1: 244*.
dummy_re 1: 282*.
dunattac 1:        65*,  1: 249*.
dversion 1: 254*.
e_name  1: 135*,  1: 146.
ec       1:        91*.
ecm_type 1: 107*.
ejectabl 1: 165*.
em       1:        91*.
entry_pt 1: 141*.
ext_addr 1: 143*.
ext_disk 1: 159*.
ext_ptr 1: 268*.
extdptr_ 1: 213*,  1: 280.
fileid  1: 205*.
filler  1: 203.
floppy  1: 127*.
fnctn_co 1: 243*.
fs       1:        91*.
fs_strt_ 1: 163*.
fwd_link 1:        94*.
fwdlink 1: 209*.
hard_err 1: 193*.
hddown  1:         67*,  1: 244*.
hdinit  1:         63*,  1: 244*.
hdisk_cb 1: 261,   1: 266*.
hdiskcb_ 1: 261*.
hdskio  1:         64*,  1: 250*.
header  1: 110*.
hentry_p 1: 160*.
implemen 1: 300.
in_servi 1: 176*.
init    1:         91*.
int1    1:         85*,  1: 147,     1: 148,   1: 149,         1: 186,   1: 188,         1: 204.
int2    1:         86*,  1:  88.
int4    1:         87*,  1:  89,     1: 207,   1: 208,         1: 209,   1: 210,         1: 222.
int_prio 1: 278*.
interfac 1:         7.
intpar  1: 246*.
intsoff_ 1:        98*,  1: 278.
intson_t 1:        99*.
io_mode 1: 224*.
iochanne 1: 148*.
iospacem 1:        48*.
kind    1: 111*.
last_dat 1: 223*.
last_fwd 1: 222*.
link_ptr 1:        92*.
linkage 1:         92,   1:  93*,    1: 110,   1: 183,         1: 184.
list_cha 1: 184*.
max_enam 1:        46*,  1: 135.
maxdev  1:         41*,  1: 168.
misc1   1: 226*.
misc2   1: 227*.
mm       1:        91*.
msec    1: 172.
n_config 1: 248*.
non_io  1: 137*.
num_blok 1: 161*.
num_byte 1: 237*.
num_chun 1: 225*.
o_config 1: 249*.
operatn 1: 188*.
osportio 1:        91*.
pagelabe 1: 200*,  1: 221,  1: 270,   1: 277.
```

```
param_pt 1: 240*.
params  1: 240,   1: 241*.
parptr  1: 247*.
pascalbd 1: 137*.
pcb_chai 1: 182*.
pcb_type 1: 107*.
permanen 1: 152*.
permreq_ 1: 155*.
pm        1:        91*.
portcbof 1:        39*.
preq_ava 1: 156*.
ptrdevre 1: 138*,  1: 154,  1: 168,   1: 189,          1: 242,   1: 248,          1: 249,   1: 267.
raw_data 1: 269*.
raw_head 1: 270*.
raw_io  1: 199*.
rb_headt 1: 108,   1: 109*,          1: 182.
rb_type 1: 107*,  1: 111.
rbheader 1: 108*.
read_fla 1: 216*,  1: 233*.
rec_port 1: 121*.
relpage 1: 208*.
relptr  1:         88*,  1:  94,     1:       95.
removabl 1: 165*.
req       1: 245*.
req_exte 1: 191*.
req_hd_p 1: 281*.
reqabt_f 1: 178*.
reqblk  1: 133,   1: 181*.
reqblk_t 1: 107*.
reqptr_t 1: 133*,  1: 155,  1: 245,   1: 281,          1: 282.
reqresta 1:        59*,  1: 245*.
reqspec_ 1: 195*.
reqsrv_f 1: 176*.
reqstatu 1: 187*.
reqsts_t 1: 175*,  1: 187.
reqsucce 1: 177*.
required 1: 154*.
resolved 1: 219*,  1: 236*.
restrt_c 1: 274*.
restrt_l 1: 275*.
rs232   1: 123*.
rsints  1:         29*.
screen  1: 126*.
sec       1: 171*.
sect_lef 1: 272*.
self_ide 1: 153*.
seq_exte 1: 230,   1: 231*.
seqdev  1: 137*.
seqextpt 1: 230*.
seqio   1:         56*,  1: 245*.
size_rsp 1:        43*.
slot_no 1: 147*.
slotints 1:        30*.
slotx   1: 122*.
soft_hdr 1: 221*.
soft_hea 1: 277*.
spar1   1: 137*.
spar2   1: 137*.
spar3   1: 137.
spare1  1:         91*.
spare2  1:         91*.
spare3  1:         91*.
spare4  1:         91.
spr1    1: 107*.
spr2    1: 107*.
spr3    1: 107.
still_in 1: 249*.
string  1: 135.
strt_blo 1: 162*.
timestmp 1: 170*.
total_re 1: 276*.
trecord 1:         91*.
twigints 1:        33*.
unit    1:          2.
v_flag  1: 273*.
version 1: 201*.
```

```
vertints 1:        32*.
via1    1: 125*.
via2    1: 124*.
volume  1: 204*.
winints 1:         34*.
with_hea 1: 199*.
without_ 1: 199*.
worstwar 1: 284*.
x_leng  1: 271*.
xfer_cou 1: 215*,  1: 232*.

Declaration Character : '*'
Assignment Character :  '='
```

### THE END ###