General Technical Information

Mass Storage

CORVUS

LIMITED WARRANTY

Corvus warrants its hardware products against defects in materials and workmanship for a period of 180 days from the date of purchase from any authorized Corvus Systems dealer. If Corvus receives notice of such defects during the warranty period, Corvus will, at its option, either repair or replace the hardware products which prove to be defective. Repairs will be performed and defective parts replaced with either new or reconditioned parts.

Corvus software and firmware products which are designed by Corvus for use with a hardware product, when properly installed on that hardware product, are warranted not to fail to execute their programming instructions due to defects in materials and workmanship for a period of 180 days. If Corvus receives notice of such defects during the warranty period, Corvus does not warrant that the operation of the software, firmware or hardware shall be uninterrupted or error free.

Limited Warranty service may be obtained by delivering the product during the 180 day warranty period to Corvus Systems with proof of purchase date. YOU MUST CONTACT CORVUS CUSTOMER SERVICE TO OBTAIN A "RETURN AUTHORIZATION CODE" PRIOR TO RETURNING THE PRODUCT. THE RAC (RETURN AUTHORIZATION CODE) NUMBER ISSUED BY CORVUS CUSTOMER SERVICE MUST APPEAR ON THE EXTERIOR OF THE SHIPPING CONTAINER. ONLY ORIGINAL OR EQUIVALENT SHIPPING MATERIALS MUST BE USED. If this product is delivered by mail, you agree to insure the product or assume the risk of loss or damage in transit, to prepay shipping charges to the warranty service location and to use the original shipping container. Contact Corvus Systems or write to Corvus Customer Service, 2100 Corvus Drive, San Jose, CA, 95124 prior to shipping equipment.

ALL EXPRESS AND IMPLIED WARRANTIES FOR THIS PRODUCT, INCLUDING THE WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO A PERIOD OF 180 DAYS FROM DATE OF PURCHASE, AND NO WARRANTIES, WHETHER EXPRESS OR IMPLIED, WILL APPLY AFTER THIS PERIOD. SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

IF THIS PRODUCT IS NOT IN GOOD WORKING ORDER AS WARRANTED ABOVE, YOUR SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. IN NO EVENT WILL CORVUS SYSTEMS BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE SUCH PRODUCT, EVEN IF CORVUS SYSTEMS OR AN AUTHORIZED CORVUS SYSTEMS DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH MAY VARY FROM STATE TO STATE.

CORVUS

December 16, 1985

Dear Customer,

The artwork for several of our user manuals have been damaged beyond use and we have been unable to replace them in time to fill your order. As a temporary replacement, we have made a copy of an existing manual for your use. Unfortunately it did not copy well and some of the line art is not legible.

We expect new manuals from new artwork to be available in March.

Please fill out the attached form and mail it back to us. We will send you a replacement manual just as soon as they are available.

Quality Assurance Department
Corvus Systems, Inc.
P.O. Box 2428
White City, Oregon 97503

* *

Name ____

Address ____

City ____________________ State __________ Zip ________

Manuals required:

_________ Consetllation 7100-05944-01

_________ Mass Storage 7100-05945-01

_________ Omninet 7100-06614-01

CORVUS MASS STORAGE SYSTEMS
GENERAL TECHNICAL INFORMATION

Part Number: 7100-05945-01
Release Date: October 1984
Revision: A

TABLE OF CONTENTS

LIST OF FIGURES

SCOPE

This manual describes the command protocols used by Corvus mass storage systems. It covers the disk commands and the Omninet protocols used to send those commands. It also describes how to use the various features provided by the commands. It is meant to be used in conjunction with the following manuals:

> Omninet Local Area Network General Technical Information, Corvus P/N 7100-02040
>
> Constellation Software General Technical Information, Corvus P/N 7100-05944-01
>
> Omninet Protocol Book

CONVENTIONS

Hexadecimal values are suffixed with an **h**. For example, FFh, 02h.

When not otherwise qualified, a **sector** is 512 bytes. A **block** is always 512 bytes.

All program examples are given in psuedo-Pascal and are not necessarily syntactically correct. The examples are meant to serve as guidelines to you in implementing your own programs.

In command and table descriptions, **lsb** means least significant byte or least significant bit, depending on context. Similarly, **msb** means most significant byte or most significant bit.

The TYPE column used in describing commands, protocols, and tables has the following meanings:

Type	Meaning
BYTE	An unsigned 8 bit value.
WORD	An unsigned 16 bit value; msb, lsb format.
FWRD	An unsigned 16 bit value; lsb, msb format; a byte-flipped WORD.
ADR3	An unsigned 24 bit value; msb..lsb format.
FAD3	An unsigned 24 bit value; lsb..msb format; a byte-flipped ADR3.
DADR	A 3-byte field, called Disk address; interpretation is shown in Chapter 1, section titled Logical sector address decoding.
BSTR	A string of 1 or more characters, padded on the right with blanks (20h).
NSTR	A string of 1 or more characters, padded on the right with NULs (00h).
FLAG	A byte with bits numbered 7..0; msb..lsb format.
ARRY	An array of 1 or more BYTEs.

CONTROLLER FUNCTIONS | 1

Corvus currently supports three mass storage devices: the Revision B/H Series drives, the OmniDrive (TM) mass storage system, and The Bank (TM) mass storage system. Each of these devices may be attached to a Corvus network. The Rev B/H drives may be attached to a Corvus multiplexer, or through a disk server to an Omninet (TM) local area network. The OmniDrive and The Bank have built-in Omninet interfaces.

Although these devices have very different hardware characteristics, the software interface to each is very similar. For example, one software disk driver can interface to all these devices.

This chapter describes the functions supported by Corvus mass storage devices. Each section describes the function and lists the relevant commands. Where needed, additional explanatory text follows.

The commands are described as a string of bytes to be sent to the device, and a string of bytes that is the expected reply. The format used to describe commands is shown in the following example:

Command Name: Read a sector (256 byte sector)

Command Length: 4 bytes
Result Length: 257 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 2h
1 / 3	DADR	sector number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 256	ARRY	contents of sector

In this example, the command described is the Read a sector command. As you can see, the command length is 4 bytes, and the expected result length is 257 bytes. This means that you send 4 bytes to the drive, and expect to receive 257 bytes in reply. Each field of the command and result is described by its starting offset in the string of bytes (indexed starting at 0), the length of the field, and its type. Then a verbal description of the contents of the field is given.

The first byte of any command is always the command code; the value of the command code is given in the description column. In this case, the command code for Read a sector is 2h. Whenever a field has a fixed value, its value is given in the description column.

In the case of an error, normally only one byte, the disk result code, is received. Disk result codes are summarized in Appendix B.

Chapter 2 describes the Omninet protocols used to send the commands. Chapter 3 gives examples of sending commands over Omninet and over flat cable.

Command name	Code:Modifier	Length	Length
Read/Write Commands:			
Read Sector (256 bytes)	02h	4	257
Write Sector (256 bytes)	03h	260	1
Read Sector (128 bytes)	12h	4	129
Read Sector (256 bytes)	22h	4	257
Read Sector (512 bytes)	32h	4	513
Read Sector (1024 bytes-Bank)	42h	4	1025
Write Sector (128 bytes)	13h	132	1
Write Sector (256 bytes)	23h	260	1
Write Sector (512 bytes)	33h	516	1
Write Sector (1024 bytes-Bank)	43h	1028	1
Record Write (Bank)	16h	2	1
Semaphore Commands:			
Semaphore Lock	0Bh:01h	10	12
Semaphore Unlock	0Bh:11h	10	12
Semaphore Initialize	1Ah:10h	5	1
Semaphore Status	1Ah:41h	5	257
Pipe Commands:			
Pipe Read	1Ah:20h	5	516
Pipe Write	1Ah:21h	517	12
Pipe Close	1Ah:40h	5	12
Pipe Status 1	1Ah:41h	5	513
Pipe Status 2	1Ah:41h	5	513
Pipe Status 0	1Ah:41h	5	1025
Pipe Open Write	1Bh:80h	10	12
Pipe Area Initialize	1Bh:A0h	10	12
Pipe Open Read	1Bh:C0h	10	12
Active User Table Commands:			
AddActive	34h:03h	18	2
DeleteActiveUsr (Rev B/H)	34h:00h	18	2
DeleteActiveNumber(OmniDrive)	34h:00h	18	2
DeleteActiveUsr (OmniDrive)	34h:01h	18	2
FindActive	34h:05h	18	17
ReadTempBlock	C4h	2	513
WriteTempBlock	B4h	514	1

Figure 1.1: Summary of Disk Commands by Function
(continued on next page ...)

Command name	Code:Modifier	Command Length	Result Length
Miscellaneous Commands:			
Boot	14h	2	513
Read Boot Block	44h	3	513
Get Drive Parameters	10h	2	129
Park heads (Rev H)	11h	514	1
Park heads (OmniDrive)	80h	1	1
Echo (OmniDrive,Bank)	F4h	513	513
Put Drive in Prep Mode:			
Prep Mode Select	11h	514	1
Prep Mode Commands:			
Reset Drive	00h	1	1
Format Drive (Rev B/H)	01h	513	1
Format Drive (OmniDrive)	01h	1	1
Fill Drive (OmniDrive)	81h	3	1
Format Tape (Bank)	01h:01h	8	1
Reformat Track (Bank)	01h:02h	8	2
Verify (Rev B/H,OmniDrive)	07h	1	variable
Non-destructive Verify (Bank)	07h:02h	6	10
Destructive Verify (Bank)	07h:01h	6	10
Read Corvus Firmware	32h	2	513
Write Corvus Firmware	33h	514	1

Figure 1.1: Summary of Disk Commands by Function (cont.)

READ-WRITE COMMANDS

Five sets of read-write commands are supported, each set specifying a different sector size. Data can be read or written in sectors of 128 bytes, 256 bytes, 512 bytes, or 1024 bytes. There are two sets of commands that support 256 byte sectors; they are identical.

The Rev B/H controller and the OmniDrive controller use a physical sector size of 512 bytes. When a host sends a write of a sector size other than 512 bytes to the drive, the controller first reads the entire physical sector, overlays the written data onto the appropriate chunk of the physical sector, and then writes the physical sector. It is therefore recommended that hosts, where possible, use a write command of 512 bytes to minimize overhead when writing to the drive.

The Bank physical sector size is 1024 bytes. When a host sends a write of a sector size other than 1024 bytes to The Bank, the data is buffered until the whole sector is received; then the data is written to the media. If any other commands are received before this buffer is full, or if another sector is to be written to, the controller performs as described above; that is, it reads the whole physical sector, overlays the written data onto the appropriate chunks of the physical sector, and then writes the physical sector. It is therefore recommended that hosts, where possible, use a write command of 1024 bytes to minimize overhead when writing to The Bank.

The fact that The Bank buffers write commands has one other ramification: the controller always returns 0 as the disk result code, indicating a successful write. When it comes time for the Bank to actually write the sector and an error is encountered, no error status is reported to the host.

The read function always reads the whole physical sector and returns the appropriate chunk of data. Unlike the write mode, no performance penalty is paid when using any particular sector size.

All of the read-write commands decribed below use a three byte sector number as the disk address. The interpretation of sector number (DADR) is described in the next section.

Command Name: Read a sector (256 byte sector)

Command Length: 4 bytes
Result Length: 257 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 2h
1 / 3	DADR	sector number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 256	ARRY	contents of sector

Command Name: Write a sector (256 byte sector)

Command Length: 260 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 3h
1 / 3	DADR	sector number
4 / 256	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Read a sector (128 byte sector)

Command Length: 4 bytes
Result Length: 129 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 12h
1 / 3	DADR	sector number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 128	ARRY	contents of sector

Command Name: Write a sector (128 byte sector)

Command Length: 132 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 13h
1 / 3	DADR	sector number
4 / 128	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Read a sector (256 byte sector)

Command Length: 4 bytes
Result Length: 257 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 22h
1 / 3	DADR	sector number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 256	ARRY	contents of sector

Command Name: Write a sector (256 byte sector)

Command Length: 260 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 23h
1 / 3	DADR	sector number
4 / 256	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Read a sector (512 byte sector)

Command Length: 4 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 32h
1 / 3	DADR	sector number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 512	ARRY	contents of sector

Command Name: Write a sector (512 byte sector)

Command Length: 516 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 33h
1 / 3	DADR	sector number
4 / 512	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Read a sector (1024 byte sector) (Bank only)

Command Length: 4 bytes
Result Length: 1025 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 42h
1 / 3	DADR	sector number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1024	ARRY	contents of sector

Command Name: Write a sector (1024 byte sector)(Bank only)

Command Length: 1028 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 43h
1 / 3	DADR	sector number
4 / 1024	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

LOGICAL SECTOR ADDRESS DECODING

On the Rev B/H drives, the three byte sector number specified in a read or write command is decoded into a 4-bit drive number and a 20-bit address. The decoding is described below:

byte 1	byte 2	byte 3
d	lsb	msb

Byte 1, upper nibble, is the most significant nibble of the address.
Byte 1, lower nibble, is the drive number (1 through 15).
Byte 2 is the least significant byte of the address.
Byte 3 is the middle byte of the address.

Thus to write to drive 1, address 02D348h, the host should send to the controller these bytes:

21h, 48h, D3h

A 20-bit address allows the controller to address approximately 1 million sectors per drive, or 512MB using 512 byte sectors. Virtual drives can be used to extend the addressing capabilities of the Rev B/H controller; see the section titled "Virtual Drive Table" later in this chapter.

For OmniDrive and The Bank, the three byte sector number is treated as a 24-bit address; all three bytes are used to indicate the address. The OmniDrive and Bank controllers can thus address 16 times more data than the Rev B/H controller, or approximately 8 gigabytes using 512 byte sectors. The three byte address is decoded as follows:

byte 1	byte 2	byte 3
d	lsb	msb

Byte 1, upper nibble, is bits 17-20 of the address.
Byte 1, lower nibble, is decremented by 1, and becomes bits 21-24 of the address.
Byte 2 is the least significant byte of the address.
Byte 3 is the middle byte of the address.

Thus to write to an address, say 32D348h, the host should send to the controller these bytes:

24h, 48h, D3h

The controller flips the nibbles in byte d, subtracts 10h from the result and uses this value as the most significant byte of the address. Byte 2 is used as the least significant byte and byte 3 the middle byte.

Note that for addresses of 20 bits or less, the two addressing schemes are equivalent. For example, to write to drive 1, address 2D348h, the host sends these bytes:

```
        21h, 48h, D3h
```

The address specified in the Read-Write commands is a sector address, where the size of the sector is specified by the command. For example, to read block 8 of the device, any of the following commands can be used:

Command string	Meaning
02h, 01h, 10h, 00h	sector 16 (256-byte sector)
12h, 01h, 20h, 00h	sector 32 (128-byte sector)
22h, 01h, 10h, 00h	sector 16 (256-byte sector)
32h, 01h, 08h, 00h	sector 8 (512-byte sector)
42h, 01h, 04h, 00h	sector 4 (1024-byte sector; Bank only)

WRITE VERIFY OPTION

The OmniDrive provides the option of specifying write-verify or non-write-verify. If the write-verify option is chosen, the controller, after each write to the media, performs a read operation of that sector to verify that the sector can be read with a correct CRC. If the non-write-verify option is specified, there is no read after write.

The tradeoff is between performance and reliability. The write-verify costs at least an extra revolution of the disk but it verifies that the data is recorded properly on the media. The other provides higher performance without the assurance of data integrity.

The option is represented by one byte in the firmware area. The standard firmware release has this byte set to non-write-verify. The option can be changed using the Corvus diagnostic program.

Rev B/H drives always use write-verify. The Bank always uses non-write-verify.

FAST TRACKS (BANK ONLY)

A Bank Tape (TM) cartridge can be configured to use fast-track or non-fast-track mode. In fast-track mode, a read completes much faster than in non-fast-track mode. However, a write takes much longer in fast-track mode than in non-fast-track mode. Fast-track mode is therefore recommended for applications which require heavy look-up of data, but little or no modification of the data the data.

In fast-track mode, the first 16 tracks of the user data area (4MB) are redundantly recorded. For a 200MB tape, the controller records each sector of data 8 times, once on each of 8 tracks; each succeeding track has the data skewed 1/8 around the tape loop. For a 100MB tape, the controller records each sector of data 4 times on 4 tracks; each succeeding track has the data skewed 1/4 around the tape loop.

When a sector is read, the controller determines where on the track its head is, and reads from the closest sector. Thus, the average read access time is 1/8 (or 1/4) that of the non-fast-track mode.

There are two types of write to the fast tracks area: normal write and record write. For normal write, the controller updates all the redundant sectors in one pass. Thus, it takes an entire revolution to complete one write. For record write, the host can specify the redundant sector to be written. The sector specified is used for all succeeding Write commands, until the next Record Write command is received. This feature allows the host to write to a whole track, then repeat the process for the redundant tracks.

To turn record write on or off, use the Record Write command.

Command Name: Turn on Record Write (Bank only)

Command Length: 2 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 16h
1 / 1	BYTE	sector number*

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

* For a 200MB tape, valid sector numbers are 80h-87h, specifying sector 0 through 7; for a 100MB tape, valid sector numbers are 80h-83h, specifying sector 0 through 3.

Command Name: Turn off Record Write (Bank only)

Command Length: 2 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 16h
1 / 1	BYTE	00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

When using normal write, updating 100 sectors requires 100 tape revolutions, one for each sector write. When updating many consecutive sectors, it may be faster to use record write. Let's assume you want to update sectors 100 to 199 on a 200MB tape. You first issue a Record Write command for redundant sector 0 (80h), and then 100 sector write commands, one for each sector 100 to 199. Depending on the interleaving, this should take only 1 tape revolution. Next you issue a Record Write command for redundant sector 1 (81h), and then the same 100 sector write commands. Repeat this sequence for redundant sectors 2 through 7, and you should complete the update in only 8 tape revolutions, as opposed to the 100 revolutions used in normal write.

SEMAPHORES

Semaphores provide an indivisible test and set operation for use by application programs. See chapter 5 for examples of how to use semaphores.

The semaphore commands are listed below:

- Semaphore Lock
- Semaphore Unlock
- Initialize Semaphore Table
- Semaphore Status

Any host can, at any time, request to lock a semaphore. If the specified semaphore is not already locked, the controller locks the semaphore. If a semaphore is already locked, the application program using the semaphores can continue to poll the semaphore table by resending the Lock command until the desired semaphore is no longer locked.

The Semaphore Unlock command always unlocks the semaphore.

The status of the semaphore prior to each operation is also returned to provide for a full test-set or test-clear operation.

A semaphore can be any 8-byte name, except for 8 bytes of 20h (ASCII space character). There is no limit on the number of semaphores that may exist in a given application or network; however, only 32 semaphores may be locked at any one time (on each server).

Two semaphores are equivalent only if each character in the name is exactly the same. For example, semaphore 'CORVUS11' is different than semaphore 'corvus11', which is different than 'Corvus11'. The characters do not have to be printing characters; eight bytes of 10h (ASCII LF character) is a legal semaphore name.

OmniDrive and The Bank support a wild card character in semaphore names. The character 00h (ASCII NUL character) matches any other character in semaphore lock and unlock operations.

The Initialize Semaphore Table command clears the semaphore table, which is equivalent to unlocking all the semaphores. The semaphore table can be initialized by any processor, but this should only be performed on system-wide initialization or for recovery from error conditions.

The Semaphore Status command returns the semaphore table, which can then be examined to see which semaphores are locked.

Command Name: Semaphore lock

Command Length: 10 bytes
Result Length: 12 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 0Bh
1 / 1	BYTE	01h
2 / 8	ARRY	semaphore name

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	semaphore result
2 / 10	ARRY	unused (no meaning)

Command Name: Semaphore unlock

Command Length: 10 bytes
Result Length: 12 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 0Bh
1 / 2	BYTE	11h
2 / 8	ARRY	semaphore name

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	semaphore result
2 / 10	ARRY	unused (no meaning)

Command Name: Initialize semaphore table

Command Length: 5 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	10h
2 / 3	ARRY	don't care - use 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Semaphore status

Command Length: 5 bytes
Result Length: 257 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	41h
2 / 1	BYTE	03h
3 / 2	ARRY	don't care - use 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 256	BYTE	semaphore table

Semaphore results

Value		Meaning
0	0h	Semaphore Not Set/no error
128	80h	Semaphore Set
253	FDh	Semaphore table full
254	FEh	Error on semaphore table read/write
255	FFh	Semaphore not found

Implementation Details For Semaphores

The semaphores are implemented using a lookup table containing an 8-byte entry for each of the 32 possible semaphores. A used entry in the table indicates that the semaphore is locked. Unused table entries are represented by 8 bytes of 20h (ASCII space character).

When a Lock command is received, the controller searches the table for a matching entry. If one is found, a Semaphore Set status (80h) is returned. Otherwise, the semaphore is written over the first empty entry, and a status of Semaphore Not Set (0) is returned.

When an Unlock command is received, the controller searches the table for a matching entry. If one is found, it is overwritten with blanks, and a status of Semaphore Set (80h) is returned. Otherwise, a status of Semaphore Not Set (0) is returned.

The format of the semaphore table is shown below. See Appendix A for the location of the semaphore table.

```
 Table layout                           Entry layout
+--------------+ byte 0      +--< +---------------+
|semaphore #1  |             |    |   1st byte    |
+--------------+             |    +-             -+
|semaphore #2  |<------------+    |   2nd byte    |
+--------------+             |    +-             -+
|              |             |    |               |
=              =             |    =               =
|              |             |    |               |
+--------------+             |    +-             -+
|semaphore #31|              |    |   7th byte    |
+--------------+             |    +-             -+
|semaphore #32|              |    |   8th byte    |
+--------------+ byte 255    +--< +---------------+
```

For Rev B/H drives, the semaphore table is initialized to blanks only when the firmware is rewritten or when an Initialize Semaphore Table command is received. For OmniDrives and Banks, the semaphore table is initialized at power up or when an Initialize Semaphore Table command is received.

Performance Considerations When Using Semaphores

For Rev B/H drives, a semaphore operation causes 2 disk reads, and 0 or 1 disk writes. First the semaphore block must be read from the firmware area. If the Lock or Unlock is successful, then the semaphore table must be written back to the disk. Finally, the dispatcher code must be reloaded from the firmware area.

For OmniDrives and Banks, a semaphore operation causes no disk I/O, as the semaphore table is maintained in the controller RAM. The table is not saved when the device is powered off.

PIPES

Pipes provide synchronized access to a reserved area of the disk. Any computer can use the pipes commands to read or write data to the pipes area at any time, and not worry about conflicting with another computer's read or write to the pipes area. See chapter 6 for examples of how to use pipes.

The pipe commands are listed below:

- Pipe Open for Write
- Pipe Open for Read
- Pipe Write
- Pipe Read
- Pipe Close
- Pipe Purge
- Pipe Status
- Pipe Area Initialize

The pipes area must be initialized before any other pipe commands are used.

The Pipe Area Initialize command specifies the pipe area starting block number and the length in number of blocks. Note that the block size is 512 bytes for the Bank as well as the OmniDrive and Rev B/H drives. The pipes area must be entirely within the first 32k blocks of the tape or disk; the starting block number plus the number of blocks must be less than 32k. The Pipe Area Initialize command does not actually write anything to the pipes area, other than the pipes tables.

The normal sequence of events in using the pipes area is as follows:

One host opens the pipe for write. It then uses Pipe Write commands to write blocks to the pipe. When it has written all the data, it uses the Pipe Close command to close the pipe.

Later on, either the same host or some other host issues a Pipe Open for Read command. It uses Pipe Read commands to read data from the pipe. When done reading, it issues a Pipe Close command. If the pipe is empty (i.e., all of the data has been read), it is deleted. If data is still remaining, the host can open the pipe again later to finish reading the data.

Each time a pipe is opened for write, a new pipe is created. When a Pipe Open for Read command is received, the lowest numbered closed pipe with the specified name is opened.

The Pipe Purge command can be used to purge any unwanted pipes.

The Pipe Status command is used to view the state of the internally managed pipe tables.

Command Name: Pipe Open for Write

Command Length: 10 bytes
Result Length: 12 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Bh
1 / 1	BYTE	80h
2 / 8	BSTR	pipe name

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	pipe result
2 / 1	BYTE	pipe number (1-62)
3 / 1	FLAG	pipe state - see below
4 / 8	ARRY	unused (no meaning)

Command Name: Pipe Open for Read

Command Length: 10 bytes
Result Length: 12 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Bh
1 / 1	BYTE	C0h
2 / 8	BSTR	pipe name

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	pipe result
2 / 1	BYTE	pipe number (1-62)
3 / 1	FLAG	pipe state - see below
4 / 8	ARRY	unused (no meaning)

Command Name: Pipe Read

Command Length: 5 bytes
Result Length: 516 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	20h
2 / 1	BYTE	pipe number
3 / 2	FWRD	data length - 00h, 02h (512 bytes)

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	pipe result
2 / 2	FWRD	number of bytes read - 00h, 02h (512 bytes)
4 / 512	ARRY	data

Command Name: Pipe Write

Command Length: 517 bytes
Result Length: 12 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	21h
2 / 1	BYTE	pipe number
3 / 2	FWRD	data length - 00h, 02h (512 bytes)
5 / 512	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	pipe result
2 / 2	FWRD	number of bytes written - 00h, 02h (512 bytes)
4 / 8	ARRY	unused (no meaning)

Command Name: Pipe Close, Pipe Purge

Command Length: 5 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	40h
2 / 1	BYTE	pipe number
3 / 1	BYTE	FEh - close write FDh - close read 00h - purge
4 / 1	BYTE	don't care - use 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	pipe result

Command Name: Pipe Status

Command Length: 5 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	41h
2 / 1	BYTE	01h - Pipe Name table 02h - Pipe Pointer table
3 / 2	ARRY	don't care - use 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 512	ARRY	contents of specified table

Command Name: Pipe Status

Command Length: 5 bytes
Result Length: 1025 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Ah
1 / 1	BYTE	41h
2 / 1	BYTE	00h
3 / 2	ARRY	don't care - use 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 512	ARRY	contents of Pipe Name table
513 / 512	ARRY	contents of Pipe Pointer table

This is the only command which returns more than 530 bytes. If you are using a general purpose command buffer for sending device commands, you may wish to use the version of the Pipe Status command which returns either the Pipe Name table or the Pipe Pointer table, so that you do not have to declare a 1025-byte buffer.

Command Name: Pipe Area Initialize

Command Length: 10 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 1Bh
1 / 1	BYTE	A0h
2 / 2	FWRD	starting block number
4 / 2	FWRD	length in blocks
6 / 4	ARRY	don't care - use 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	pipe result

Starting block number + Length in blocks must be less than 32k.

Pipe state flag (returned on Pipe Open)

Bit #	Meaning
bit 7	1=contains data / 0=empty
bit 1	1=open for read
bit 0	1=open for write

Pipe results

	Value	Meaning
0	00h	No error.
8	08h	Tried to read an empty pipe.
9	09h	Pipe not open for read or write.
10	0Ah	Tried to write to a full pipe.
11	0Bh	Tried to open an open pipe.
12	0Ch	Pipe does not exist.
13	0Dh	Pipe buffer full.
14	0Eh	Illegal pipe command.
15	0Fh	Pipes area not initialized.

Implementation Details For Pipes

Internally, the pipes area is managed by two tables: a Pipe Name Table and a Pipe Pointer Table. These tables are stored in different areas on the various disk devices; see Appendix A. The host can retrieve these tables by sending a Pipe Status command.

The Pipe Name Table contains 64 entries of 8 bytes each. The first and last names in the table are reserved for system use. The first name is WOOFWOOF and the last name is FOOWFOOW. An entry of all blanks (20h) indicates an unused entry.

The format of the Pipe Name Table is shown below:

```
                +------------+ byte 0
pipe number 0  |  WOOFWOOF  |
                +------------+ byte 8
pipe number 1  |            |
                =            =
pipe number 62 |            |
                +------------+ byte 504
pipe number 63 |  FOOWFOOW  |
                +------------+
```

The Pipe Pointer Table also contains space for 64 entries of 8 bytes each, each entry being formatted as shown below:

```
 Rev B/H                                 OmniDrive/Bank
+------------------+                   +------------------+
| pipe number      |   byte 0          | pipe number      |
+------------------+                   +------------------+
| starting   (msb) |   byte 1          | starting   (0)   |
+-                -+                   +-                -+
| byte             |                   | block      (msb) |
+-                -+                   +-                -+
| address    (lsb) |                   | address    (lsb) |
+------------------+                   +------------------+
| ending     (msb) |   byte 4          | ending     (0)   |
+-                -+                   +-                -+
| byte             |                   | block      (msb) |
+-                -+                   +-                -+
| address    (lsb) |                   | address    (lsb) |
+------------------+                   +------------------+
| pipe state       |   byte 7          | pipe state       |
+------------------+                   +------------------+
```

While the format of the Pipe Pointer table on the disk is different for the Rev B/H drives than it is for OmniDrive and Bank, the table returned by the Pipe Status command always has the Rev B/H format. That is, the OmniDrive and Bank convert the disk format to the Rev B/H format for the Pipe Status command.

Pipe number (byte 0) is an index into the Pipe Name Table. A pipe number of 0 indicates the first entry in the Pipe Name Table, and a pipe number of 63 indicates the last entry in the Pipe Name table.

Entries in the Pipe Pointer Table are ordered by starting address. Unlike the Pipe Name table, where unused entries are interspersed with used entries, all of the unused entries in the Pipe Pointer table occur at the end of the table. The entry with pipe number 63 marks the end of the used entries.

For the Rev B/H drives, the starting and ending byte addresses are absolute disk byte addresses. Each should be divided by 512 to get an absolute block address.

The Pipe State is a flag which is interpreted as shown below:

bit #	Meaning
bit 7	1=contains data / 0=empty
bit 1	1=open for read
bit 0	1=open for write

The first entry in the Pipe Pointer Table always looks like the following, which corresponds to the WOOFWOOF entry in the Pipe Name Table:

```
 Rev B/H                                OmniDrive/Bank
+-------------------+                  +-------------------+
|  pipe number = 0  |   byte 0         |  pipe number = 0  |
+-------------------+                  +-------------------+
| starting byte     |   byte 1         | starting block    |
+-                 -+                  +-                 -+
| address of pipes  |                  | address of pipes  |
+-                 -+                  +-                 -+
| area              |                  | area              |
+-------------------+                  +-------------------+
| starting byte     |   byte 4         | same as bytes     |
+-                  +                  +-                  +
| address of pipes  |                  | 1 through 3       |
+-                 -+                  +-                 -+
| area + 1024       |                  |                   |
+-------------------+                  +-------------------+
| pipe state = 80h  |   byte 7         | pipe state = 80h  |
+-------------------+                  +-------------------+
```

The last entry in the Pipe Pointer Table always looks like the following, which corresponds to the FOOWFOOW entry in the Pipe Name Table):

```
     Rev B/H                                  OmniDrive/Bank
    +------------------+                      +------------------+
    | pipe number = 63 |   byte 0             | pipe number = 63 |
    +------------------+                      +------------------+
    | ending byte      |   byte 1             | ending block     |
    +-                -+                      +-                -+
    | address of pipes |                      | address of pipes |
    +-                -+                      +-                -+
    | area             |                      | area             |
    +------------------+                      +------------------+
    | same as bytes    |   byte 4             | same as bytes    |
    +-                 +                      +-                 +
    | 1 through 3      |                      | 1 through 3      |
    +-                -+                      +-                -+
    |                  |                      |                  |
    +------------------+                      +------------------+
    | pipe state = 80h |   byte 7             | pipe state = 80h |
    +------------------+                      +------------------+
```

Whenever a Pipe Area Initialize command is received, the pipes tables are initialized with the entries for pipes 0 and 63 shown above, and all other entries unused. The pipes area can be deleted by rewriting the firmware.

The following example shows a typical state of the pipe tables. It shows 3 existing pipes, two called PRINTER and one called FASTLP.

```
 Pipe Pointer table      offset    Pipe Name table
+---------------------+           +-----------------+
| entry for pipe 0    |     0     | WOOFWOOF        |
+---------------------+           +-----------------+
| entry for pipe 1    |     1     | PRINTER         |
+---------------------+           +-----------------+
| entry for pipe 6    |     2     | FASTLP          |
+---------------------+           +-----------------+
| entry for pipe 2    |     3     | blanks          |
+---------------------+           +-----------------+
| entry for pipe 63   |     4     | blanks          |
+---------------------+           +-----------------+
| 0's                 |     5     | blanks          |
+---------------------+           +-----------------+
| 0's                 |     6     | PRINTER         |
+---------------------+           +-----------------+
|                     |           |                 |
=                     =           =                 =
|                     |           |                 |
+---------------------+           +-----------------+
| 0's                 |    63     | FOOWFOOW        |
+---------------------+           +-----------------+
```

Individual Pipe Disk Space Allocation

The pipes area consists of used space and holes (unused space). There are two kinds of holes:

Active hole -- a contiguous area of unused pipe space bounded on the low address end by an open for writing pipe.

```
=                 =
|                 |
+-----------------+
|    open for     |
|    writing      |
|    pipe         |
+-----------------+
|    active       |  the open pipe in front of the hole
|    hole         |  can grow into this region.
+-----------------+
|    pipe         |
=                 =
```

Inactive hole -- a contiguous area of unused pipe space bounded on the low address end by the end of a closed pipe or the end of an open for reading pipe.

```
=                  =
|                  |
+------------------+
|    open for      |
|    reading or    |
|    closed pipe   |
+------------------+
|    inactive      |  the pipe in front of the hole
|    hole          |  cannot grow.
+------------------+
|    pipe          |
=                  =
```

New pipe allocations are made by examining all the holes in the pipe area. The allocator looks for the larger of: (1) the largest inactive hole or (2) half the size of the largest active hole. A new pipe starts at the beginning of an inactive hole or at the midpoint of an active hole. All pipes grow in the same direction, by increasing address.

When an open for writing pipe hits the end of a hole (that is, it bumps into an existing pipe), the error code, tried to write to a full pipe (0Ah), is returned. This can happen even if there is space remaining in other holes.

Performance Considerations When Using Pipes

On a Rev B/H drive, a Pipe Write results in 2 disk reads, and 2 disk writes. First, the pipes code is overlayed into the controller RAM; then the data is written and the Pipe Pointer Table rewritten; finally, the dispatcher code is reloaded. A Pipe Read is similar, only there are 3 disk reads and 1 disk write. Since the controller code is located in the firmware area, and the pipes area is in the user area of the drive, a pipe operation can cause considerable head movement.

For OmniDrives and Banks, the pipes controller code is loaded at power-on time, and does not have to be swapped in and out. Also, the Pipe Name Table and the Pipe Pointer Table are located in the firmware area. For the OmniDrive, the tables are written back to the drive only when a pipe is closed, so a Pipe Read is 1 disk read operation, and a Pipe Write is 1 disk write operation. For the Bank, the pipe tables are only written to the media when the Bank is ready to turn off the motor (see section titled "Changing Bank Tapes" later in this chapter).

ACTIVE USER TABLE

The Active User Table is used by Corvus applications software to keep track of the active devices on the network. At any given time, it should contain a list of those users who are connected to the network. See the section titled "Active User Table" in Chapter 2 for more explanation.

The Bank does not support the Active User Table.

There are six commands supported:

- AddActive
- DeleteActiveUsr
- DeleteActiveNumber (OmniDrive only)
- FindActive
- ReadTempBlock
- WriteTempBlock

The AddActive command adds a user to the table. The host specifies the user name, the Omninet address, and the device type. See Appendix B for a list of device types.

The DeleteActiveUsr command deletes a user from the table. Note that the command code for DeleteActiveUsr is different for the Rev B/H drives than it is for the OmniDrive.

The DeleteActiveNumber command deletes all users with the specified Omninet address from the table (OmniDrive only).

The FindActive command returns the Omninet address and the device type of the user with the specified name.

The ReadTempBlock command can be used to read the entire Active User Table, and the WriteTempBlock can be used to initialize the Active User Table.

Command Name: Add Active

Command Length: 18 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 34h
1 / 1	BYTE	03h
2 / 10	BSTR	name
12 / 1	BYTE	host Omninet address
13 / 1	BYTE	host device type
14 / 4	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	table result

Command Name: Delete Active User (Rev B/H drives only)

Command Length: 18 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 34h
1 / 1	BYTE	00h
2 / 10	BSTR	name
12 / 6	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	table result

Command Name: Delete Active User (OmniDrive only)

Command Length: 18 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 34h
1 / 1	BYTE	01h
2 / 10	BSTR	name
12 / 6	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	table result

Command Name: Delete Active Number (OmniDrive only)

Command Length: 18 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 34h
1 / 1	BYTE	00h
2 / 10	ARRY	unused - use 0's
12 / 1	BYTE	host Omninet address
13 / 5	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	table result

Command Name: Find Active

Command Length: 18 bytes
Result Length: 17 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 34h
1 / 1	BYTE	05h
2 / 10	BSTR	name
12 / 6	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 1	BYTE	first byte of name, or table result
2 / 9	BSTR	remaining bytes of name
11 / 1	BYTE	host Omninet address
12 / 1	BYTE	host device type
13 / 4	ARRY	unused

Command Name: Read Temp Block

Command Length: 2 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - C4h
1 / 1	BYTE	block number - 0 to 6 for Rev B/H, 0 to 3 for OmniDrive

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 512	ARRY	contents of block

Command Name: Write Temp Block

Command Length: 514 bytes
Result Length: 1 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - B4h
1 / 1	BYTE	block number - 0 to 6 for Rev B/H, 0 to 3 for OmniDrive
2 / 512	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Table results

Value	Meaning
0	Ok.
1	No room to add.
2	Duplicate name.
3	User not found.

Implementation Details For The Active User Table

The Active User Table implementation is similar to semaphores, in that an unused entry is indicated by blanks. When an AddActive command is received, the controller searches the table for an entry with a matching name. If one is found, the entry is overwritten with the new data, and a table result of duplicate name (2) is returned. If no matching entry is found, the first entry with blanks is overwritten with the specified data, and a status of Ok (0) is returned.

For DelectActiveUsr, the first entry with a matching name is overwritten with blanks. For DeleteActiveNumber, all entries with matching Omninet addresses are overwritten with blanks.

The table consists of four blocks, located in the firmware area. The blocks are numbered 0 to 3. Each table entry is 16 bytes long, as shown below:

```
   Table layout
+----------------+ block 0
|    entry #1    |
+----------------+
|                |                     Entry layout
=                =              +--< +----------------+
|                |              |    |      name      | byte 0
+----------------+              |    +-              -+
|    entry #32   |              |    |                |
+----------------+ block 1      |    =                =
|    entry #33   |              |    |                |
+----------------+              |    +-              -+
|                |              |    |                | byte 9
=                = <------------+    +----------------+
=                =              |    |Omninet address| byte 10
|                |              |    +----------------+
+----------------+ block 3      |    | device type    | byte 11
|    entry #97   |              |    +----------------+
+----------------+              |    |     unused     | byte 12
|                |              |    =     (0's)      =
=                =              |    |                | byte 15
|                |              +--< +----------------+
+----------------+
|    entry #128  |
+----------------+
```

Omninet address is 0 to 63. Device types are listed in Appendix B.

The normal initialization of the Active User table is described in the section titled "Active User Table" in Chapter 2. The table can also be initialized by rewriting the firmware, or by issuing Write Temp Block commands.

BOOTING

There are two commands which provide a boot function. The purpose of these commands is to provide a machine independent means of booting a host computer.

The first boot command, called the Boot command (14h), was Corvus' first attempt to provide a boot function. The Boot command was not flexible enough, so a second boot command, the Read Boot Block command (44h), was added.

The first Boot command is used by Corvus to support Apple II (TM) computers and Corvus Concept (TM) workstations. The Read Boot Block command is used to support all other computers. Each computer is assigned a computer number by Corvus. See Appendix B for a list of the currently assigned computer numbers.

Both boot commands return a block of 512 bytes to the host computer. This block normally contains boot code for the computer, but can be used for whatever the particular computer requires.

In order to use the boot commands, an application program must be written which sets up the data structures used by the boot commands. Corvus provides such an application program, called BOOTMGR, with its Constellation II software. Refer to the manual titled Constellation Software General Technical Information for more information on how Corvus software uses the boot commands.

Command Name: Boot

Command Length: 2 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 14h
1 / 1	BYTE	boot block number (0-7)

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 512	ARRY	contents of block

Command Name: Read Boot Block

Command Length: 3 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 44h
1 / 1	BYTE	computer number (See Appendix B)
2 / 1	BYTE	block number

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result*
1 / 512	BYTE	contents of block

* If the disk result = FFh, the block could not be found.

Implementation Details For Boot Commands

For the Boot command, the boot blocks are located in the firmware area (see Appendix A for exact locations). Blocks 0 through 3 contain 6502 code for the Apple II, and blocks 4 through 7 contain 68000 code for the Corvus Concept. These blocks are included in the firmware files distributed by Corvus.

For the Read Boot Block command, the following data structures are used:

Block 8, bytes 36 - 39 contain the absolute block address of the Corvus volume. The Boot Table is located 6 blocks past this location. The format of the Boot Table is described below:

```
 Table format
+-----------------+
|   entry #0      |  block 0
+-----------------+                     Entry format
|                 |             +-< +-----------------+
=                 =             |   | address (msb)   |  byte 0
|                 | <-----------+   +-               -+
+-----------------+             |   | address (lsb)   |  byte 1
|   entry #127    |             +-< +-----------------+
+-----------------+
|   entry #128    |  block 1
+-----------------+
|                 |
=                 =
|                 |
+-----------------+
|   entry #255    |
+-----------------+
```

The address is a relative block address which is added to the Boot Table address. The result is the block number of the 0th block of boot code. The block number specified in the Read Boot Block command is added to this result to get the absolute block address of the data to be returned. Thus, the block address of the data returned is computed as follows:

```
Boot Table address     + boot code address + boot block #
(contents of block 8,    (from Boot Table)    (from Read Boot
 bytes 36-39, + 6)                             Block command)
```

DRIVE PARAMETERS

The Get Drive Parameters command can be used by application programs to find out the user-accessible size of the drive (device capacity) and other device specific information. The format given differs slightly from that used for other commands: the first page shows the information that is returned from all devices and the second page shows the device specific information.

Command Name: Get drive parameters

Command Length: 2 bytes
Result Length: 129 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 10h
1 / 1	BYTE	drive number (starts at 1)

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 32	BSTR	firmware message
33 / 1	BYTE	ROM version
34 / 4	ARRY	track information (see below)
38 / 3	FAD3	capacity in 512 byte blocks
41 / 16	ARRY	unused (no meaning)
57 / 1	BYTE	interleave factor
	ARRY	Table information (see below)
58 / 12		MUX parameters
70 / 6		pipes information
76 / 14		virtual drive table
90 / 16		LSI-11 information
106 / 1	BYTE	physical drive number
107 / 3	FAD3	capacity of physical drive
110 / 1	BYTE	drive type (see below)
111 / 6	ARRY	tape information (see below)
117 / 2	WORD	media id (see below)
119 / 1	BYTE	maximum number of bad tracks (see below)
120 / 8	ARRY	unused (no meaning)

The table below shows the meanings of the status bytes that are different for the various device types.

Offset/Len	Type	Rev B/H Drives	OmniDrive	Bank
35 / 1	BYTE	sectors/track	sectors/track	sectors/track (lsb,msb)
36 / 1	BYTE	tracks/cylinder	tracks/cylinder	
37 / 2	FWRD	cylinders/drive	cylinders/drive	tracks/tape
58 / 12	ARRY	MUX parameters	unused	unused
70 / 2	FWRD	pipe name tbl ptr	pipe area ptr	pipe area ptr
72 / 2	FWRD	pipe pointer tbl ptr	pipe area size	pipe area size
74 / 2	FWRD	pipe area size	unused	unused
76 / 14	ARRY	Virtual drive tbl	unused	unused
90 / 8	ARRY	LSI-11 VDO table	unused	unused
98 / 8	ARRY	LSI-11 spared tbl	unused	unused
110 / 1	BYTE	unused	drive type	drive type (82H)
111 / 3	FAD3	unused	unused	*tape life (# of minutes)
114 / 2	FWRD	unused	unused	start/stop count
116 / 1	FLAG	unused	unused	fast track flag (=1 fast tracks on)
117 / 2	WORD	unused	media id	media id
119 / 2	BYTE	unused	max # of bad tracks	reserved

* The tape life is specified at 500 hours and 2000 start/stops

PARKING THE HEADS

Rev B drives do not require parking of heads.

The Rev H and OmniDrives provide a firmware command that allows a host to instruct a drive to park its heads in a landing zone or cylinder. This command is used in preparing the drive for shipping.

The landing (or parking) cylinder is a reserved cylinder for Rev H drives; for OmniDrives, the landing cylinder is specified in the disk parameter block of each drive. Some drives automatically park the heads during power off; the landing cylinder in this case is specified as 0FFFFh. No actual movement of the heads is performed when a park command is sent to one of these drives.

The park command only positions the heads over the landing cylinder; it does not turn off the motor. When the drive is parked, it is offline to the network, and no host can communicate with it. The drive stays parked until it is reset.

Command Name: Park the heads (Rev H Drive ONLY)

Command Length: 514 bytes
Result Length: 1 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 11h
1 / 1	BYTE	drive number (starts at 1)
2 / 11	ARRY	all 0's
13 / 2	WORD	C3h, C3h
15 / 499	ARRY	all 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

This is really a special Prep block.

Command Name: Park the heads (OmniDrive ONLY)

Command Length: 1 byte
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 80h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

CHANGING BANK TAPES OR POWERING OFF The Bank

The Bank Tape is continuously looping. While the motor is on, the tape cannot be removed. If the tape is not accessed for about 1 minute 15 seconds, The Bank goes into a "shut down" mode. The controller flushes tape information back to the firmware area, seeks to track 0, then turns off the motor. At this point, the tape can be removed.

There is a reset switch on The Bank which can be used to force the "shut down" sequence. However, this switch should only be used when absolutely necessary.

CHECKING DRIVE INTERFACE

The Echo command can be used to check the interface to the drive. The host sends 512 bytes to the drive, and expects to get the same 512 bytes back.

Command Name: Echo (OmniDrive/Bank ONLY)

Command Length: 513 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - F4h
1 / 512	ARRY	data to be echoed

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result
1 / 512	ARRY	data from command vector

PREP MODE

The host can put the drive into prep mode by sending a prep command with 512 bytes of executable controller code. The controller loads this code over the RAM-resident dispatcher whose function is to interpret the command bytes sent to the controller. Thus in effect, the prep block can be considered as a specialized dispatcher. Some applications requiring direct control of the hardware can utilize this feature (e.g., burn-in program). The standard prep block shipped by Corvus supports the following functions:

format the drive or tape
verify the drive (Rev B/H, OmniDrives only)
read from the firmware area
write to the firmware area

fill the drive with a pattern (OmniDrive only)

reformat a track (Bank only)
destructive verify a track (Bank only)
non-destructive verify a track (Bank only)

All prep blocks should support a reset function in order to take the drive out of prep mode and back to the normal mode. This is done through a reset command (command code = 00h) in prep mode. Also, when the controller is put in prep mode, the front panel LED's are set as a visual indication of this mode. For Rev B/H drives, the FLT and RDY lights are turned off and the BSY light is turned on. For OmniDrives and Banks, the opposite is true; i.e., the FLT and RDY lights are turned on and the BSY light is turned off.

Rev B/H drives can use only one prep block at a time (maximum 512 bytes of code). OmniDrives and Banks, however, use a maximum of 4 prep blocks (2K of code). The first prep command puts the drive into prep mode. Any additional prep command blocks are loaded after the previous block. After the fourth block has been received, any additional block overlays the fourth block.

Prep blocks are hardware dependent. Prep blocks for Rev B/H drives contain Z80 code, whereas prep blocks for OmniDrives and Banks contain 6801 code.

Command Name: Put drive in prep mode

Command Length: 514 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 11h
1 / 1	BYTE	drive number (starts at 1)
2 / 512	ARRY	prep block

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Reset drive (take drive out of prep mode)

Command Length: 1 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 00h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

FORMAT DRIVE

In prep mode using the Corvus prep block, the host can send a format command to the controller. The controller lays down on the media the sector format, and the data fields are filled with whatever is specified by the Format command. OmniDrives use the pattern FFFFh.

A Format command destroys ALL information on the drive, including the firmware itself. The spared track table, the virtual drive table, and the pipes tables, as well as the polling parameters, interleave factor, read after write flag, etc., are all destroyed by Format. You would not normally format a drive until this information is written down, so that it may be manually restored after formatting.

For Rev B/H drives, the controller refuses the Format command if the Format switch (beneath the front panel LED's, second from right) is set to the left. You must set this switch to the right in order to format the drive.

Drives shipped from Corvus have been formatted, burned-in, bad tracks logged in the spare table, and the firmware written. If you must format the drive, you should always verify the drive after formatting, and spare any bad tracks found. See the section titled "Verify," later in this chapter, for more information.

Command Name: Format drive (Rev B/H drives ONLY)
(drive in prep mode)

Command Length: n bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 01h
2 / n-1	ARRY	format pattern

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

The Corvus diagnostic programs send 513 bytes and use pattern 76h or E5h.

Command Name: Format drive (OmniDrives ONLY)
(drive in prep mode)

Command Length: 1 byte
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 01h

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Command Name: Fill the drive (OmniDrives ONLY)
(drive in prep mode)

Command Length: 3 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 81h
1 / 2	WORD	fill pattern

Result

Offset/Len	Type	Description
0 / 1	BYTE	disk result

Note: The recommended fill pattern is B6D9h.

FORMAT TAPE (BANK)

In prep mode using the Corvus prep blocks, the host can send a tape format command to The Bank. With this command, the host specifies whether fast tracks are to be used, the tape type (100MB or 200MB), and the interleave factor to be used.

The interleave factor must be an odd number between 1 and 31. The controller automatically increases by 1 any specified even interleave. Any interleave greater than 31 is set to 31.

After receiving the format command (full tape format only), the controller sends back a success status immediately to acknowledge that the format command has been received. It then turns off interrupts, thus taking The Bank offline. During this time, no devices can communiate with The Bank. After formatting the media, the controller fills the tape with a pattern (B6D9h). It then attempts to verify the tape by reading all sectors. Any bad sectors are spared automatically. The results of the format are written to firmware block 2.

Any tracks reported as bad have more than 4 bad sectors, and should not be used. If any bad tracks are reported, the tape should either be discarded, or dummy volumes allocated over the bad tracks. See the section titled "Physical Versus Logical Addressing" later in this chapter for more information on mapping track numbers to block addresses.

The prep block also allows the host to send a command to reformat one track. The tape is assumed to have been formatted, so the controller uses the current interleave and tape parameters. This feature is provided in case one track has read-write problems and needs to be reformatted.

The command to reformat one track returns the number of bad sectors on the track. If the number of bad sectors is greater than 4, the track is bad. You should use the Get Drive Parameters command to check the tape life. Tapes are rated for 500 hours and 2000 start-stops. If either of these numbers is exceeded, the tape should be discarded. Otherwise, you should allocate a dummy volume over the bad track. See the section titled "Physical Versus Logical Addressing" later in this chapter for information on mapping track numbers to block addresses.

Command Name: Format tape (Bank ONLY)
(Bank in prep mode)

Command Length: 8 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 01h
1 / 1	BYTE	01h
2 / 3	ARRY	unused - use 0's
5 / 1	FLAG	fast track flag (01h = fast tracks on)
6 / 1	BYTE	tape size (01h = 200MB; 00h = 100MB)
7 / 1	BYTE	interleave factor (odd number 1 to 31)

Result

Offset/Len	Type	Description
0 / 1	BYTE	result

An even interleave factor is automatically increased by 1. Interleave greater than 31 is set to 31.

The results are recorded in firmware block 2 in the following format:

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 1	BYTE	bad track count (=n)
2 / 2*n	ARRY	bad track list (each entry is lsb,msb)

Command Name: Reformat one track (Bank ONLY)
(Bank in prep mode)

Command Length: 8 bytes
Result Length: 2 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 01h
1 / 1	BYTE	02h
2 / 2	FWRD	track number to format
4 / 3	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 1	BYTE	number of bad sectors

Track number range is 0-100. The firmware track (track 1) contains sparing information for the whole tape; if this track is reformatted, the sparing information for the rest of the tape will be lost.

MEDIA VERIFY (CRC)

The verify command is a prep mode command. For Rev B/H drives, the verify is performed as follows: The controller reads each sector on the disk. If it is unable to read a particular sector, it tries again to read the sector. If it can read the sector within 10 retries, it reports a soft error. If it cannot read the sector, it rewrites the sector with the data it read, which is probably bad, and reports a bad sector.

For OmniDrives, each sector is read only once, and a hard error is reported if the sector is bad. The sector is not rewritten.

Marginal sectors may be reported on one execution of the Verify command, yet not show up on the next. Any sector which is ever reported as bad should be spared. Each media has a maximum number of tracks that may be spared. If the Verify command reports more than this number, the media is bad, and should not be used.

A list of spared tracks should be maintained on paper near the drive. Then if it is ever necessary to reformat the drive or rewrite the entire firmware area, the appropriate tracks can be respared.

A list of bad sectors is returned to the host. The sector numbers are physical sector numbers, and are converted to track numbers with the following algorithm:

    track # = [(cylinder #) * (number of heads)] + (head #)

Note that those sectors which are already spared may be reported as bad.

For The Bank, the prep block provides two verify features: a non-destructive verify and a destructive verify. These commands work on one track at a time. The non-destructive track verify reads all the sectors on the specified track and reports the number of bad sectors found and the sector numbers of the first four bad sectors. The destructive verify fills the track with the input pattern (2 bytes) first and then verifies the track as described for non-destructive verify.

See the section titled "Physical Versus Logical Addressing" later in this chapter for information on mapping track numbers to block addresses.

Command Name: Verify drive (OmniDrive, Rev B/H ONLY)
(Drive in prep mode)

Command Length: 1 byte
Result Length: 2+4*n bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 07h

Result

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 1	BYTE	number of bad sectors
2 / 4	ARRY	head, cylinder, sector of 1st bad sector
6 / 4	ARRY	head, cylinder, sector of 2nd bad sector
n*4-2 / 4	ARRY	head, cylinder, sector of nth bad sector

The 4 bytes per sector are interpreted as follows:

Offset/Len	Type	Description
0 / 1	BYTE	head number
1 / 2	FWRD	cylinder number
3 / 1	BYTE	sector number

Command Name: Non-destructive track verify (Bank ONLY)
(Bank in prep mode)

Command Length: 6 bytes
Result Length: 10 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 07h
1 / 1	BYTE	02h
2 / 2	FWRD	track number
4 / 2	ARRY	unused - use 0's

Result

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 1	BYTE	number of bad sectors
2 / 2	WORD	sector number of 1st bad sector
8 / 2	WORD	sector number of 4th bad sector

The sector number is interpreted as msb = head number and lsb = sector number. Since there are 256 sectors per section, this value is also an absolute sector number.

Command Name: Destructive track verify (Bank ONLY)
(Bank in prep mode)

Command Length: 6 bytes
Result Length: 10 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 07h
1 / 1	BYTE	01h
2 / 2	FWRD	track number
4 / 2	WORD	fill pattern

Result

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 1	BYTE	number of bad sectors
2 / 2	WORD	sector number of 1st bad sector
8 / 2	WORD	sector number of 4th bad sector

The recommended fill pattern is B6D9h.

TRACK SPARING

When the drive is formatted, it is filled with a pattern. A burn-in can then be performed to find the marginal tracks. These can be recorded in the firmware track sparing block to make them invisible.

Each type of mechanism has a different number of spared tracks allowed. This number is returned by the Get Drive Parameters command to let the host know the maximum number of tracks it can spare out. Rev B drives allow 7 spared tracks; Rev H drives allow 31 spared tracks; OmniDrives allow from 7 to 64 spared tracks, depending on the drive type (see Appendix A).

Internally, the spared tracks are recorded in the firmware area; see Appendix A for a complete description of the spared track table. You should also maintain a list of the spared tracks on a piece of paper near the drive, so that if the firmware is ever overwritten you can respare the proper tracks.

Tracks are spared by updating the firmware blocks containing the spared track table. The Corvus Diagnostic program provides this capability.

For Banks, when a tape is formatted, it is also verified and all the bad sectors are logged in the firmware area. Each track has four sectors reserved for use as spared tracks.

Since only four sectors are reserved, any track with five or more bad sectors should not be used. The firmware has no capability to skip these tracks. Therefore it is recommended that the tape be discarded or dummy volumes be located over this track. A dummy Constellation volume can be allocated to this track to skip it. See the next section for information on converting sector numbers to block numbers.

PHYSICAL VERSUS LOGICAL ADDRESSING

The physical layout of each media is shown below.

	Rev B/H	OmniDrives	Bank
Firmware	tracks 0 - (m-1)	tracks 0 - 3	track 1
User area	tracks m - n	tracks 4 - n	tracks 2 - z
Unused	tracks n+1 - z	tracks n+1 - z	

where m = (# of heads/drive) * 2 (see Appendix A)

z = total number of tracks - 1

x = maximum number of spared tracks allowed

n = z - x + number of tracks currently spared

The unused area is used up as tracks are spared.
Track 0 on The Bank is reserved for a landing area.

For Rev B/H drives and OmniDrives, the drive is viewed as a series of consecutive physical tracks, where a track is identified by a head number and a cylinder number (head number varies fastest). Logical tracks are mapped onto the physical tracks one-to-one, skipping over spared tracks and the firmware area. A typical layout of a hypothetical drive is shown below. This example assumes a 4 track firmware area, 120 tracks total, with 16 maximum spared tracks allowed. The drive has 4 heads and 20 sectors per track. Two tracks, tracks 34 and 67, are spared:

```
                 Physical      Head,Cyl   Logical
                 --------      --------   -------
              +---------------+          +---------------+
      ^       |  track 0      | 0,0      | firmware area |
firmware area =               =          =               =
      v       |  track 3      | 3,0      |               |
              +---------------+          +---------------+
      ^       |  track 4      | 0,1      |  track 0      |
      |       +---------------+          +---------------+
      |       |  track 5      | 1,1      |  track 1      |
              =               =          =               =
              |  track 33     | 1,8      |  track 29     |
              +---------------+          +---------------+
    user      |  track 34     | 2,8      |  spared track |
    area      +---------------+          +---------------+
              |  track 35     | 3,8      |  track 30     |
              +---------------+          +---------------+
              |               |          |               |
              =               =          =               =
              +---------------+          +---------------+
              |  track 67     | 3,16     |  spared track |
      |       +---------------+          +---------------+
      |       =               =          =               =
      v       |  track 103    | 3,25     |  track 97     |
              +---------------+          +---------------+
      ^       |  track 104    | 0,26     |  track 98     |
  reserved    +---------------+          +---------------+
  for spared  |  track 105    | 1,26     |  track 99     |
  tracks      =               =          +---------------+
      v       |  track 119    | 3,29     =  unused       =
              +---------------+          +---------------+
```

When a track is spared, the user data following the spared track is still there, but is no longer accessible, since the data is now located at a different logical address.

The algorithm for converting block numbers to physical sector numbers would be as shown below, if it were not for the firmware area and spared tracks. The real algorithm is explained immediately following the simplified form.

sector # = (block #) modulo (sectors per track)
track # = (block #) div (sectors per track)
head # = (track #) modulo (number of heads)
cylinder # = (track #) div (number of heads)

Note that the track number is a temporary result and is not a directly addressable entity in the drive; a given block is addressed physically by sector number, head number and cylinder number.

The real algorithm for converting block numbers to physical sector numbers is shown below:

```
sector # =      (block #) modulo (sectors per track)
logical track # =  (block #) div    (sectors per track)
physical' track # = (logical track #) plus (firmware
                    area offset)
physical track # = (physical' track #) plus (one for
                    every spared track preceding).
head # =        (physical track #) modulo (number of heads)
cylinder # =    (physical track #) div    (number of heads)
```

Continuing with the example given above, let's convert block number 1308 to a physical sector address.

```
     sector # =           1308 mod 20        = 8
     logical track # =    1308 div 20    = 65
     physical' track # =65 + 4           = 69
Tracks 34 and 67 are spared, so add 2
     physical track # = 69 + 2           = 71
     head # =             71 mod 4           = 1
     cylinder # =         71 div 4           = 17
```

Alternatively, suppose you have run the Verify Drive command, and it reported a bad track at head 2, cylinder 12, sector 10. You want to compute the range of blocks that the bad sector lies within. You must apply the above algorithm in reverse:

```
     physical track # = 2 + (12*4)           = 50
Track 34 is already spared, so subtract 1
     physical track #'= 50 - 1               = 49
     logical track # =  49 - 4               = 45
     starting sector # = 45 * 20             = 900
     ending sector #   = 900 + 20 - 1        = 919
```

Thus, the bad sector lies somewhere between sector 900 and sector 919. You must apply the interleave factor (see next section) to determine exactly which sector is bad.

For Banks, the tape is viewed as a series of tracks numbered 0 to 100. Each track consists of a number of sections; a 200MB tape has 8 sections per track, while a 100MB tape has 4 sections per track. Each section contains 256 sectors, and a sector contains 1024 bytes. On a Bank tape, each track has four sectors reserved for sparing, so a given block number always falls within the same track. The track number of the track in which a given block is located is computed as follows:

```
sector # =  (block #) div 2
logical track #  =  (sector #) div (sectors per track)
physical track # = logical track # + 2
```

To compute which blocks lie within a given track, use the following algorithm:

```
blocks per track =  (sectors per track - 4) * 2
starting block # =  (track # - 2 ) * (blocks per track)
ending block #   =  (starting block #) + (blocks per track) - 1
```

Thus, if track 17 is reported as bad (more that 4 bad sectors) by the Track Verify command, you compute the bad blocks as follows (assuming a 200MB tape):

```
blocks per track = (2048 - 4) * 2       = 4090
starting block # = (17-2) * 4090        = 81350
ending block #   = 81350 + 4090 - 1     = 85439
```

In order to "spare" the track, you should allocate an unused volume starting at block 81350 that is 4090 blocks in length.

INTERLEAVE

Interleaving provides a way of improving disk performance on reading sequential sectors. The interleave factor specifies the distance between logical sectors within a given track. For example, if we assume 20 sectors per track, an interleave factor of 1 specifies that the sectors are numbered logically 1 to 20. An interleave factor of 2 specifies that the sectors are numbered 1, 11, 2, 12, ..., 10, 20. An interleave factor of 5 specifies that the sectors are numbered 1, 5, 9, 13, 17, 2, 6, 10, 14, 18, 3 ...

As you can see, the interleave factor specifies how far apart sequential sectors are located. If the interleave factor is optimal, a sequential read operation is able to read more than one sector per disk revolution. Note that different interleave factors are optimal for different applications. You will have to decide if changing the interleave factor will significantly enhance the speed of one application without penalizing other users of the drive.

The interleave is specified in the drive information block of the firmware area. When the firmware is first updated, it uses the standard interleave specified in the firmware file. Legal values are given below:

	min	max	default
Rev B/H	1	19	9
OmniDrive	1	17	9
Bank	1	31	11

Interleave for The Bank must be odd.

If the media has information recorded, a change of interleave effectively scrambles the information. Changing the interleave back to the old value restores all information. When the interleave is changed, the sparing information is preserved since it is physical track information. Also, the firmware blocks are not interleaved.

The interleave is changed by updating the firmware block containing it. This capability is provided in the Corvus Diagnostic program.

READ-WRITE FIRMWARE AREA

Each mass storage device has a designated firmware area which is not accessible to normal read-write commands, and is not counted in reporting the usable blocks on the drive. To access this area, the host must put the drive in prep mode and send firmware read-write commands. There is no interleaving performed on the firmware area, nor may this area have any bad sectors.

For Rev B/H drives, the firmware file currently consists of 40 blocks. (Some old firmware files were 60 blocks.) The firmware file occupies the first 2 tracks of cylinder 0; a duplicate firmware file is located in the first 2 tracks of cylinder 1. The remaining tracks of the first 2 cylinders are unused. The user area starts at cylinder 2.

The read-write firmware commands require a head and sector as the address, rather than a block number. The head-sector number is a byte field: the head number occupies the upper 3 bits of the byte, and the sector number occupies the lower 5 bits. Firmware blocks 0-19 are head 0, sectors 0-19, and blocks 20-39 are head 1, sectors 0-19. For example, firwmare block 16 is addressed as 10h, and firmware block 32 is addressed as 2Ch.

For OmniDrives, the firmware file consists of 36 blocks, thus occupying two entire tracks. A total of four tracks are reserved on the media so that a duplicate copy of the firmware can be maintained. The user area starts at track 4.

The firmware blocks are numbered from 0 to 35. The read-write firmware commands require a block number as the address. Note that this is different from the Rev B/H drives where a physical head and sector are specified instead.

For The Bank, track 1 of the tape has the first 38 sectors designated as the firmware area; only the first 512 bytes of each physical sector are used. The first three sectors contain identical information and are called the boot blocks (triple redundancy for safety). The firmware blocks are numbered 0 to 35, and a block number is used as the address for the firmware read-write commands.

Command Name: Read a block of Corvus firmware (Rev B/H ONLY)
(Drive in prep mode)

Command Length: 2 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 32h
1 / 1	BYTE	head (bits 7-5), sector (bits 4-0)

Result

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 512	ARRY	contents of specified firmware block

Command Name: Write a block of Corvus firmware (Rev B/H ONLY)
(Drive in prep mode)

Command Length: 514 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 33h
1 / 1	BYTE	head (bits 7-5), sector (bits 4-0)
2 / 512	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	result

Command Name: Read a block of Corvus firmware (OmniDrive/Bank)
(Drive in prep mode)

Command Length: 2 bytes
Result Length: 513 bytes

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 32h
1 / 1	BYTE	block number

Result

Offset/Len	Type	Description
0 / 1	BYTE	result
1 / 512	ARRY	contents of specified firmware block

Command Name: Write a block of Corvus firmware (OmniDrive/Bank)
(Drive in prep mode)

Command Length: 514 bytes
Result Length: 1 byte

Command

Offset/Len	Type	Description
0 / 1	BYTE	command code - 33h
1 / 1	BYTE	block number
2 / 512	ARRY	data to be written

Result

Offset/Len	Type	Description
0 / 1	BYTE	result

VIRTUAL DRIVE TABLE (REV B/H DRIVES)

The Virtual Drive Table was implemented to avoid rewriting drivers which had a 16MB addressing limitation.

The controller maintains a table of virtual drives in the firmware area. This 14 byte table provides for the definition of up to 7 virtual (logical) drives per physical drive. The format for the virtual drive table is shown below:

```
+--------------------+
|  track offset (lsb)|
+-  of 1st virtual  -+
|  drive        (msb)|
+--------------------+
|  track offset (lsb)|
+-  of 2nd virtual  -+
|  drive        (msb)|
+--------------------+
|         .          |
+-        .         -+
|         .          |
+--------------------+
|  track offset (lsb)|
+-  of 7th virtual  -+
|  drive        (msb)|
+--------------------+
```

An entry with a track offset equal to FFFFh indicates the absence of the corresponding virtual drive.

The track offset is a logical track number, and is simply multiplied by the number of sectors per track to obtain a block offset. When a drive number is specified in a Read-Write command, the controller examines its virtual drive table. If an entry exists for that drive, the track offset is multiplied by 20 (the number of sectors per track), and the result is added to the address.

For instance, on a 20MB Rev B drive, which has a user capacity of 38460 blocks, the Constellation I Apple software creates a virtual drive table with 0 as the entry for the first drive, and 947 as the entry for the second drive. Virtual drive 1 consists of blocks 0 to 18939, and virtual drive 2 consists of blocks 18940 (20*947) to 38459.

The controller does not check whether an address exceeds the capacity of a virtual drive. I.e., if virtual drive 2 starts at track 100 (address 2000 on a Rev B/H drive), then block 2010 can be addressed as drive 1, block 2010, or as drive 2, block 10. This allows hosts that do not need the artificial disk division to share the same disk with those that do.

The Virtual Drive Table is updated by editing the firmware block containing it. The Corvus Diagnostic program provides this capability.

The settings used by Corvus for Apple II Constellation I systems are listed below:

Drive	Total blocks	Drive 2 offset	Drive 1 blocks	Drive 2 blocks
Rev B 20MB	38460			
DOS only		976	19520	18940
Pascal/Basics		947	18940	19520
Rev H 20MB	35960			
DOS only		911	18220	17640
Pascal/Basics		896	17920	17940

CONSTELLATION PARAMETERS

The Constellation parameters are used when a Rev B/H drive is connected to a master MUX, and the MUX switch (second from left under the front panel LED's) is set to the right. The parameters specify what kind of host is connected to each slot in the MUX; a host cannot communicate with the drive if this table is not set up properly. Note that the table must be set up BEFORE the MUX is installed.

The format of the table is shown below:

```
+----------------+
|value for slot 1|  byte 0
+----------------+
|value for slot 2|
+----------------+
|                |
=                =
|                |
+----------------+
|value for slot 8|  byte 7
+----------------+
| poll param 1   |  byte 8
+----------------+
| poll param 2   |  byte 9
+----------------+
| poll param 3   |  byte 10
+----------------+
| poll param 4   |  byte 11
+----------------+
```

The slots on the MUX are numbered as shown below:

```
    5     4
    6     3
    7     2
    8     1
       X
```

where the flat cable connects at X.

Valid slot values are shown below:

Values	Meaning
0	Nothing
1	MUX
2	LSI-11
128	Computer

Each slot value is set to 1 (MUX) by default. It is possible to have a computer connected to a slot with a value of 1; and it is possible to have a MUX connected to a slot with a value of 128; however, this is not recommended because performance of the network suffers.

The meaning of each polling parameter is given below:

poll param 1: Time scale factor for timing out on a host. This is the total time the MUX will stay at one slot, regardless of the number of transactions completed. This prevents a user from hogging the network.

poll param 2: Time scale factor for timing out on a potential host. This determines how long the multiplexer waits for the first request at a particular slot.

poll param 3: The maximum number of transactions that will be accepted from a host before the multiplexer switches to the next slot.

poll param 4: unused

The default values for the polling parameters are:

```
poll param 1:  180
poll param 2:   16
poll param 3:   32
poll param 4:    0
```

The Constellation parameters are updated by editing the firmware block containing them. The Corvus Diagnostic program provides this capability.

This page intentionally left blank.

OMNINET PROTOCOLS | 2

This chapter describes the Omninet functions of the OmniDrive mass storage system, The Bank mass storage system, and the disk server for Rev B/H drives. It describes how disk commands are sent over an Omninet local area network.

A brief review of the Omninet Local Area Network General Technical Information Manual, chapter 3, will help you understand the material presented here. In that manual, the Omninet command vectors used to send and receive messages are described. The two commands that are relevant to this discussion are repeated below:

Send Message
Command vector

Offset/Len	Type	Description
0 / 1	BYTE	Command code = 40h
1 / 3	ADR3	Result record address
4 / 1	BYTE	Destination socket
5 / 3	ADR3	Data address
8 / 2	WORD	Data length
10 / 1	BYTE	User control length
11 / 1	BYTE	Destination host

Result record

Offset/Len	Type	Description
0 / 1	BYTE	Return code - values are: 00-7Fh - message sent successfully 80h - message not acknowledged 81h - message too long 82h - message sent to unitialized socket 83h - control length mismatch 84h - invalid socket number 85h - invalid destination address
1 / 3	BYTE	Unused
4 / n	ARRY	User control information

Setup Receive Message
Command vector

Offset/Len	Type	Description
0 / 1	BYTE	Command code = F0h
1 / 3	ADR3	Result record address
4 / 1	BYTE	Socket number
5 / 3	ADR3	Data address
8 / 2	WORD	Data length
10 / 1	BYTE	User control length

Result record

Offset/Len	Type	Description
0 / 1	BYTE	Return code - values are: FFh - initial value (set by user) FEh - socket set up succesfully 84h - invalid socket number 85h - socket already set up 00h - message received
1 / 1	BYTE	Source host
2 / 2	WORD	Data length
4 / n	ARRY	User control information

Any message exchange on Omninet consists of setting up a receive socket with a Setup Receive command, sending the message with a Send command, and waiting for the reply to be received. You always need at least 4 buffers for this task:

1) a command vector
2) a data buffer
3) a result record for the Setup Receive message,
4) a result record for the Send message.

You can use two separate command vectors: one for Setup Receive and one for Send, but you don't have to. You can also use separate data buffers. You MUST use separate result records.

The disk servers on Omninet currently provide two functions: the execution of disk commands, and a name service. In the future, they and other servers, developed by Corvus or other software developers, will provide many more services. In order for a server to distinguish which service is being requested, Corvus has defined a message format which includes a protocol identifier (protocol ID) as the first 2 bytes of each message. This protocol ID identifies what type of service is being requested or provided. For more information on protocol IDs, refer to the Omninet Protocol Book.

CONSTELLATION DISK SERVER PROTOCOLS

The Disk Server Protocol is used to exchange commands and data between Corvus disk devices on Omninet and the host computers which they support. The disk commands were defined in Chapter 1. The Disk Server Protocol defines the format of Omninet messages which contain disk commands, data, and control information. It also describes the mechanism for exchanging those messages. In general, the Disk Server Protocol is a two way conversation between a client and a server. The server is usually a Corvus disk device and the client is usually a personal computer. It is possible for a personal computer to run a program which enables it to act as a Corvus disk device. Corvus OmniShare for the IBM-PC, and Corvus DisketteShare for the Apple II, are two examples of such a program.

The Disk Server Protocol is a transaction based protocol; in other words, for each message sent, a reply is expected. There are two basic types of transactions: short commands and long commands. Short commands (4 bytes or less) involve the exchange of two messages, while long commands require four messages to complete a transaction. A disk read is a short command and a disk write is a long command.

The general message exchange for data transfer is shown in Figure 2.1. For a short command, the Disk Request message contains the first four or fewer bytes of the command, and the Results message contains the results of the command. For a long command, the Disk Request message contains the first four bytes of the command. After sending the Disk Request message, the host waits for a Go message from the server. After receiving the Go message, the host sends the remaining bytes of the command with a Last message. The server finally sends the results of the command with the Results message.

```
     Short command                        Long command

Client             Server          Client                Server

   Disk Request                       Disk Request
o---------------------------->     o---------------------------->

               Results                                     Go
<----------------------------o     <----------------------------o

                                      Last
                                   o---------------------------->

                                                      Results
                                   <----------------------------o
```

Figure 2.1: Message exchange for Disk Server Protocol

There are two versions of Disk Server Protocol: old and new. These are described in detail in the sections "Old Disk Server Protocol," and "New Disk Server Protocol." The new protocol follows the protocol guidelines established in the <u>Omninet Protocol Book</u>, supports more operations than the old, and uses different sockets. The operations supported are listed below:

	old	new	originator
Disk request (send disk command)	x	x	client
Last (remainder of disk command)	x	x	client
Abort request		x	client
Go	x	x	server
Results (of disk command)	x	x	server
Cancel request		x	server
Restart request		x	server

An example is probably in order. Let's look at the process of sending both a short and long command. This example uses the Old Disk Server protocol. You may wish to refer ahead to the section "Old Constellation Disk Server Protocol" for further explanation of the message contents.

Sending A Short Command

This section contains an example of sending a short command. We will use the Read a Sector (512-byte sector) command to read sector 0 from drive 1 on server 1. Recall that this command is 4 bytes long: command code is 32h, and the sector address is 01h, 00h, 00h.

First, we must issue a Setup Receive command to the transporter. The fields marked with - will contain the indicated data upon receipt of the Results message.

```
 Command vector                        Receive Result Record
+---------------------+   +------->+--------------------+
|command code =   F0h |   |        | return code =  FFh |
+---------------------+   |        +--------------------+
|  result             |---+        | - (source address) |
+-                   -+            +--------------------+
|  record             |            | - (user data       |
+-                   -+            +-                  -+
|  address            |            | -   length)        |
+---------------------+          --+--------------------+--
|socket number =  B0h |            | - (user control    |
+---------------------+            +-                  -+
|  user               |---+        | -   information)   |
+-                   -+   |        +-                  -+
|  data               |   |        |                    |
+-                   -+   |        +--------------------+
|  address            |   |
+---------------------+   |
| user data       02h |   |         User data buffer
+-                   -+   +------->+--------------------+
| length = 512    00h |            | - (512 bytes of    |
+---------------------+            +-                  -+
|control len =    03h |            | -   data)          |
+---------------------+            =                    =
                                   | -                  |
                                   +--------------------+
```

When the return code field in the Receive Result Record changes to FEh, the socket has been successfully set up. We can now proceed to send the Disk Request message.

```
  Command vector                           Send Result Record
 +-----------------------+    +------->+---------------------+
 |command code =     40h |    |        | return code =   FFh |
 +-----------------------+    |        +---------------------+
 |  result               |----+        | unused              |
 +-                     -+             +-                   -+
 |  record               |             |                     |
 +-                     -+             +-                   -+
 |  address              |             |                     |
 +-----------------------+          ---+---------------------+--
 |socket number =    B0h |             | send length     00h |
 +-----------------------+             +-                   -+
 |  user                 |---+         | length = 4      04h |
 +-                     -+   |         +---------------------+
 |  data                 |   |         | receive         02h |
 +-                     -+   |         +-                   -+
 |  address              |   |         | length = 512    00h |
 +-----------------------+   |         +---------------------+
 | user data         00h |   |
 +-                     -+   |
 | length = 4        04h |   |         User data buffer
 +-----------------------+   +------->+---------------------+
 |control len =      04h |             | read            32h |
 +-----------------------+             +-                   -+
 |destination =      01h |             | command         01h |
 +-----------------------+             +-                   -+
                                       |                 00h |
                                       +-                   -+
                                       |                 00h |
                                       +---------------------+
```

When the return code field of the Send Result Record changes to less than 80h, the message has been successfully sent. Now you must wait for the return code field of the Receive Result Record to change to 00h, indicating that a message has been received. If there are no errors, the Receive Result Record and the User Data Buffer will look like this:

```
     Receive Result Record
    +--------------------+
    | return code =  00h |
    +--------------------+
    | source addr =  01h |
    +--------------------+
    | user data      02h |
    +-                  -+
    | length = 512   00h |
  --+--------------------+--
    | length of      02h |
    +-                  -+
    | response=513   01h |
    +--------------------+
    | disk rslt      00h |
    +--------------------+

     User data buffer
    +--------------------+
    | contents of disk   |
    +-                  -+
    | sector 0, 512      |
    =                    =
    | bytes              |
    +--------------------+
```

Sending A Long Command

This section contains an example of a long command. We will use the Write a Sector (512-byte sector) to write sector 0 to drive 1 on server 1. Recall that this command is 516 bytes long: command code is 33h, and the sector address is 01h, 00h, 00h, followed by 512 bytes of data.

First, we must set up a socket to recevie the Go message. The fields marked with - will contain the indicated data upon receipt of the Go message.

```
  Command vector                         Receive Result Record
 +-----------------------+   +------->+---------------------+
 |command code =    F0h  |   |        | return code =   FFh |
 +-----------------------+   |        +---------------------+
 |  result               |---+        | - (source address)  |
 +-                     -+            +---------------------+
 |  record               |            | - (user data        |
 +-                     -+            +-                   -+
 |  address              |            | -   length)         |
 +-----------------------+            +---------------------+
 |socket number =   B0h  |
 +-----------------------+
 |  user                 |---+
 +-                     -+   |         User data buffer
 |  data                 |   +------->+---------------------+
 +-                     -+            | - (2 bytes of data) |
 |  address              |            +-                   -+
 +-----------------------+            | -                   |
 | user data        00h  |            =                     =
 +-                     -+            |                     |
 | length = 2       02h  |            +---------------------+
 +-----------------------+
 |control len =     00h  |
 +-----------------------+
```

When the return code field in the Receive Result Record changes to FEh, the socket has been successfully set up. We can now proceed to send the Disk Request message.

```
     Command vector                       Send Result Record
    +----------------------+   +------->+---------------------+
    |command code =    40h |   |        | return code =   FFh |
    +----------------------+   |        +---------------------+
    |  result              |---+        | unused              |
    +-                    -+            +-                   -+
    |  record              |            |                     |
    +-                    -+            +-                   -+
    |  address             |            |                     |
    +----------------------+          --+---------------------+--
    |socket number =   B0h |            | send            02h |
    +----------------------+            +-                   -+
    |  user                |---+        | length = 516    02h |
    +-                    -+   |        +---------------------+
    |  data                |   |        | receive         00h |
    +-                    -+   |        +-                   -+
    |  address             |   |        | length = 0      00h |
    +----------------------+   |        +---------------------+
    | user data        00h |   |
    +-                    -+   |
    | length = 4       04h |   |         User data buffer
    +----------------------+   +------->+---------------------+
    |control len =     04h |            | 1st four        33h |
    +----------------------+            +-                   -+
    |destination =     01h |            | bytes of        01h |
    +----------------------+            +-                   -+
                                        | write           00h |
                                        +-                   -+
                                        | command         00h |
                                        +---------------------+
```

When the return code field of the Send Result Record changes to less than 80h, the message has been successfully sent. Now you must wait for the return code field of the Receive Result Record to change to 00h, indicating that a message has been received. If there are no errors, the Receive Result Record and the User Data Buffer will look like this:

```
 Receive Result Record
+--------------------+
| return code =  00h |
+--------------------+
| source addr =  01h |
+--------------------+
| user data      00h |
+-                  -+
| length = 2     02h |
+--------------------+

 User data buffer
+--------------------+
| 'G'            47h |
+-                  -+
| 'O'            4Fh |
=                    =
|                    |
+--------------------+
```

After the Go message has been recevied, we are ready to send the Last message, but first we must set up to receive the Results message. There will be no user data received, since the Write command returns only a disk return code, but we will specify a data buffer anyway.

```
     Command vector                      Receive Result Record
   +-----------------------+    +------->+--------------------+
   |command code =    F0h  |    |        | return code =  FFh |
   +-----------------------+    |        +--------------------+
   |  result               |----+        | - (source address) |
   +-                     -+             +--------------------+
   |  record               |             | - (user data       |
   +-                     -+             +-                  -+
   |  address              |             | -  length)         |
   +-----------------------+           --+--------------------+--
   |socket number =   B0h  |             | - (user control    |
   +-----------------------+             +-                  -+
   |  user                 |----+        | -  information)    |
   +-                     -+    |        +-                  -+
   |  data                 |    |        |                    |
   +-                     -+    |        +--------------------+
   |  address              |    |
   +-----------------------+    |
   | user data        02h  |    |         User data buffer
   +-                     -+    +------->+--------------------+
   | length = 512     00h  |             |                    |
   +-----------------------+             +-                  -+
   |control len =     03h  |             |                    |
   +-----------------------+             =                    =
                                         |                    |
                                         +--------------------+
```

When the return code field in the Receive Result Record changes to FEh, the socket has been successfully set up. We can now proceed to send the Last message. Note that the socket number is A0h.

```
 Command vector                                Send Result Record
+-----------------------+    +------->+---------------------+
|command code =     40h |    |        | return code =   FFh |
+-----------------------+    |        +---------------------+
|  result               |----+        | unused              |
+-                     -+             +-                   -+
|  record               |             |                     |
+-                     -+             +-                   -+
|  address              |             |                     |
+-----------------------+             +---------------------+
|socket number =    A0h |
+-----------------------+
|  user                 |----+
+-                     -+    |         User data buffer
|  data                 |    +------->+---------------------+
+-                     -+             | 512 bytes of data   |
|  address              |             +-                   -+
+-----------------------+             | to be written       |
| user data         02h |             =                     =
+-                     -+             |                     |
| length = 512      00h |             +---------------------+
+-----------------------+
|control len =      00h |
+-----------------------+
|destination =      01h |
+-----------------------+
```

When the return code field of the Send Result Record changes to less than 80h, the message has been successfully sent. Now you must wait for the return code field of the Receive Result Record to change to 00h, indicating that a message has been received. If there are no errors, the Receive Result Record and the User Data Buffer will look like this:

```
     Receive Result Record
    +--------------------+
    | return code =  00h |
    +--------------------+
    | source addr =  01h |
    +--------------------+
    | user data      00h |
    +-                  -+
    | length = 0     00h |
  --+--------------------+--
    | length of      00h |
    +-                  -+
    | response=1     01h |
    +--------------------+
    | disk rslt      00h |
    +--------------------+


     User data buffer
    +--------------------+
    | nothing            |
    +-                  -+
    |                    |
    =                    =
    |                    |
    +--------------------+
```

For the example above, the sequence of message exchange using the new protocol would be exactly the same; only the contents of the User Control and the User Data buffers and the socket usage would differ.

As you can see from the above example, the disk server protocol uses the transporter's message splitting feature. The disk server protocol always knows what packet is expected next, so it can specify the user's buffer when it sets up a receive. The control information always goes to a separate data area managed by the driver. This feature cuts down on the amount of data movement that must take place, by putting the command results directly into the user's buffer.

The concept of short and long commands is used because of limited buffer space in the disk server. The disk server is capable of queuing one request for each network device. When it is ready for the Last portion of the disk command, it sends the Go message. The disk server emulates the Constellation multiplexer in that once the server services a particular host, it accepts up to 32 commands before going on to the next host. See Chapter 3 for more information on disk server service times.

The OmniDrive and Bank controllers support both the old and the new protocols, while the disk server for Rev B/H drives supports only the old protocol. All the hosts on the network are treated separately, i.e. the OmniDrive and Bank can support one protocol for one host and a different protocol for another host. The protocol to be used is derived from the type of Omninet message format received by the controller. It will be used for only that command.

OLD DISK SERVER PROTOCOL

(The Old Disk Server Protocol was written before the idea of protocol IDs was finalized; therefore it does not abide by the current protocol guidelines.)

Name: Disk request

User Control Length: 4

User Data Length: 4 or less

Protocol ID: -

Message Type: -

Socket Usage: B0h

User Control Format:

Field Name	Offset/Len	Type	Description
M	0 / 2	WORD	Number of bytes in command. If M>4, then this is a long command.
N	2 / 2	WORD	Maximum number of return bytes excluding the disk return code.

User Data Format:

Field Name	Offset/Len	Type	Description
DATA	0 / n	-	First 4 or fewer bytes of disk command.

This message is used to send the first four bytes of a disk command to the server.

If M > 4, then a Go message is expected next, otherwise a Results message is expected.

Name: Last Protocol ID: -

User Control Length: 0 Message type: -

User Data Length: depends on command Socket Usage: A0h

User Data Format:

Field Name	Offset/Len	Type	Description
DATA	0 / n	WORD	M minus 4 bytes of disk command

The Last message is used to send the last M-4 bytes of a long command to the server. This message is sent in response to a Go message from the server. M is the M from the Disk Request message.

If there are no errors, the next message from the server should be the Results message.

This command is always sent to socket A0h.

Name: Go

Protocol ID: -

User Control Length: 0

Message type: -

User Data Length: 2

Socket Usage: B0h

User Data Format:

Field Name	Offset/Len	Type	Description
GO	0 / 2	WORD	'GO' - 474Fh

The Go message is sent by the server in response to a Disk Request message. It tells the client that the server is ready to receive the Last message.

If the most significant bit of the first byte of the GO Field (i.e., the 'G' byte) is on, the disk has been reset and the operation should be restarted.

Name: Results Protocol ID: -

User Control Length: 3 Message type: -

User Data Length: depends on command Socket Usage: B0h

User Control Format:

Field Name	Offset/Len	Type	Description
NACTUAL	0 / 2	WORD	Number of bytes actually returned including the disk return code.
RETCODE	2 / 1	BYTE	Disk return code

User Data Format:

Field Name	Offset/Len	Type	Description
DATA	0 / n	ARRY	Results of disk command (NACTUAL-1 bytes).

This message contains the results of a disk command.

If the most significant bit of the first byte of the NACTUAL field is on, the disk has been reset and the operation should be restarted.

Name: Find a server

Protocol ID: 01FEh

User Control Length: 0

Message type: 01h

User Data Length: 8 bytes

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FEh
MSGTYP	2 / 1	BYTE	Message type - 01h
M	3 / 2	WORD	Length of command - 0001h
N	5 / 2	WORD	Expected length of result - 0000h
COMMAND	7 / 1	BYTE	Illegal command code

This message is used to broadcast an illegal disk command. The disk server and the OmniDrive respond to this message with a Results message; The Bank does not respond to this message.

Some host systems using this protocol broadcast an illegal disk command during power on to find servers on the network. They try to boot from the first server that replies. To prevent host systems from booting from The Bank, The Bank controller ignores the illegal command opcode FFh and does not return any status. Other illegal commands are acknowledged.

NEW SERVER PROTOCOL

Disk servers with PROM versions DS8A.A or DSD18A do not support the new disk server protocol.

Disk servers with PROM version DSD9B1D and later, OmniDrives, and Banks support the old disk server protocol as well as the new disk server protocol.

The new disk server protocol is similar to the old in basic message exchange; that is, for a short command the client sends a Disk Request message and expects a Results message; for a long command, the client sends a Disk Request message, the server replies with a Go message, the client sends a Last message, and the server replies with a Results message. However, the new protocol uses different sockets than the old, and includes more information with each message. The new protocol also includes three new messages: Abort, Cancel and Restart.

With the new disk server protocol, the client always sends the Disk Request message to socket 80h of the server, and the server always sends the Go message to socket 80h of the client. For the Last and Results messages, the server and the client respectively specify to which socket (A0h or B0h) to send the message. All asynchronous messages (Cancel, Restart, and Abort) are sent to socket 80h.

The new disk server protocol requires that a media ID be sent along with each Disk Request. This is to prevent the case when the media is swapped and the host unknowingly attempts to write to the wrong tape. During power up, the controller generates a random number to be used as the media ID of the tape. This number is based on the value of the free running counter of the 6801 clocks; it is random and has a value between 0-0FFFFh.

The host can obtain the current media ID by issuing a Get Drive Parameters command with a media ID of zero. A media ID of zero is honored by the controller regardless of the current ID. The current media ID is one of the parameters returned by the Get Drive Parameters command.

The controller broadcasts a Cancel message during power up to inform all hosts on the network about a media change. If a host does not receive or act upon the Cancel message, it will receive a Wrong Media ID error message when it tries to access the tape. The host can recover by reissuing a Get Drive Parameters command with an ID of zero in order to obtain the new media ID number.

The new disk server protocol also requires that a request ID be sent along with each disk command. This is done so that either the disk server or the host can cancel, abort, or restart a particular command. The request ID is selected by the host, and can simply be an integer which is incremented for each request.

Any Cancel, Restart, or Abort message includes a field which indicates the reason for the abnormal condition. The possible reason codes are summarized below:

Value	Meaning
01h	Timed out - either the disk server timed out waiting for a Last message, or the host timed out waiting for a Go or Results message. See chapter 3 for more information on timeouts.
02h	Offline - the disk device is currently offline for backup or reformatting.
03h	Out of synch - the server has received a Last message when it was not expecting one.
04h	Wrong media - the MEDIAID in the Disk Request message does not match the current media ID.
05h	Rebooted - the server has just come online.

Name: Disk request

Protocol ID: 01FFh

User Control Length: 0

Message Type: 0001h

User Data Length: 18

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - 0001h
RQSTID	4 / 2	WORD	Request ID
MEDIAID	6 / 2	WORD	Media ID
RESHOST	8 / 1	BYTE	Result host
RESSOCK	9 / 1	BYTE	Result socket - A0h or B0h
M	10 / 2	WORD	Number of bytes in command. If M>4, then this is a long command.
N	12 / 2	WORD	Maximum number of return bytes excluding the disk return code.
DCMD	14 / 4	ARRY	First 4 or fewer bytes of disk command.

This message is used to send the first four bytes of a disk command to the server. It tells the server to which host (ResHost) and to which socket (ResSock) to send the reply.

The host selects the request ID. The media ID was established during the first message exchange between the host and this server. If the media ID does not match the server's current media ID (because someone has switched Bank tapes, for example), then the server will not respond to the Disk Request message, but will send a Cancel message instead. The Cancel message includes the current media ID.

If M > 4, then a Go message is expected next, otherwise a Results message is expected.

Name: Last

Protocol ID: 01FFh

User Control Length: 12

Message Type: 0002h

User Data Length: depends on command

Socket Usage: A0h or B0h

User Control Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - 0002h
RQSTID	4 / 2	WORD	Request ID
reserved	6 / 2	WORD	Reserved - use 0's
reserved	8 / 2	WORD	Reserved - use 0's
reserved	10 / 2	WORD	Reserved - use 0's

User Data Format:

Field Name	Offset/Len	Type	Description
DATA	0 / n	ARRY	M minus 4 bytes of disk command

The Last message is used to send the last (M-4) bytes of a long command to the server, where M is the M from the Disk Request message. This message is sent in response to a Go message from the server. Last messages are sent to socket A0h or B0h, whichever was specified in the Go message.

If there are no errors, the next message from the server should be the Results message.

Name: Abort

Protocol ID: 01FFh

User Control Length: 0

Message Type: 0003h

User Data Length: 8

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - 0003h
RQSTID	4 / 2	WORD	Request ID
REASON	6 / 2	WORD	Reason for abort: 01h = timed out waiting for disk server response

This message tells the server to abort request RQSTID. If the RQSTID is 0 then abort any requests from this host.

Name: Go

Protocol ID: 01FFh

User Control Length: 0

Message Type: 0100h

User Data Length: 8

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - 0100h
RQSTID	4 / 2	WORD	Request ID
reserved	6 / 1	BYTE	Reserved - use 0
LASTSOCK	7 / 1	BYTE	Socket number to which Last message should be sent (A0h or B0h)

The Go message is sent by the server in response to a Disk Request message. It tells the client that the server is ready to receive the Last message for request RQSTID.

Name: Results

Protocol ID: 01FFh

User Control Length: 12

Message Type: 0200h

User Data Length: depends on command

Socket Usage: A0h or B0h

User Control Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - 0200h
RQSTID	4 / 2	WORD	Request ID
NACTUAL	6 / 2	WORD	Number of bytes acutally returned, including the disk return code.
reserved	8 / 1	BYTE	Reserved - use 0
RETCODE	9 / 1	BYTE	Disk return code
reserved	10 / 2	WORD	Reserved - use 0's

User Data Format:

Field Name	Offset/Len	Type	Description
DATA	0 / n	ARRY	Results of disk command (NACTUAL-1 bytes)

This message contains the results of a disk command. It is sent to socket A0h or B0h, whichever was specified in the Disk Request message.

Name: Cancel

Protocol ID: 01FFh

User Control Length: 0

Message Type: 0300h

User Data Length: 10

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - 0300h
RQSTID	4 / 2	WORD	Request ID
REASON	6 / 2	WORD	Reason for cancel: 02h - disk device has gone offline 04h - the MEDIAID in the Disk request message does not match the current MEDIAID
MEDIAID	8 / 2	WORD	Current Media ID

This is the server's mechanism for cancelling a request. RQSTID identifies the request which was cancelled.

Name: Restart

Protocol ID: 01FFh

User Control Length: 0

Message Type: FF00h

User Data Length: 10

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FFh
MSGTYP	2 / 2	WORD	Message type - FF00h
RQSTID	4 / 2	WORD	Request ID
REASON	6 / 2	WORD	Reason for restart: 05h - server has been rebooted 03h - out of synch: a Last message was received when one was not expected. 01h - timed out: Last message not received after Go was sent
MEDIAID	8 / 2	WORD	Current Media ID

This is the server's mechanism for telling the host to restart a request. This tells the client to send request RQSTID again. If RQSTID is zero then the client should restart any requests pending to that server.

MEDIAID is the current media ID. If it does NOT match the MEDIAID of the pending request, then the the media was changed (e.g., changing a Bank tape) while the server was offline.

CONSTELLATION NAME LOOKUP PROTOCOL

The Constellation name lookup protocol is used to identify devices on the network by name. It is currently supported by disk servers DSD18A, DSD9B1D, and later, all OmniDrives, and all Banks. It is NOT supported by disk server DS8A.A.

The messages are summarized below:

Hello
Goodbye
Who Are You
Where Are You
My ID Is

The Hello and Goodbye messages are broadcast during power up and power down respectively, to announce the presence or absence of a device. The Who Are You and Where Are You messages can either be broadcast or directed; a My ID Is message is expected in response.

Each device on the network can be identified by its name, its Omninet address, or its device type. Using the name lookup protocol, you can find the answers to such questions as, What are the addresses of all the disk servers on the network? and What is the address of the disk server named RDSERVER?

Each device is assigned one or more device types which are used to identify the types of services it supports. There are two kinds of device types: generic and specific. Generic device types define a class of Omninet hosts, while specific device types define a specific service. The currently assigned device types are listed in Appendix B.

As always, there are a few exceptions to the rules; the device types for disk devices are listed below. As you can see, the disk server and the Bank each respond to only one device type.

	Generic	Specific
Rev B/H disk server	1	1
OmniDrive	1	6
Bank	-	5

For example, the following algorithm finds all (booting) disk servers on the network:

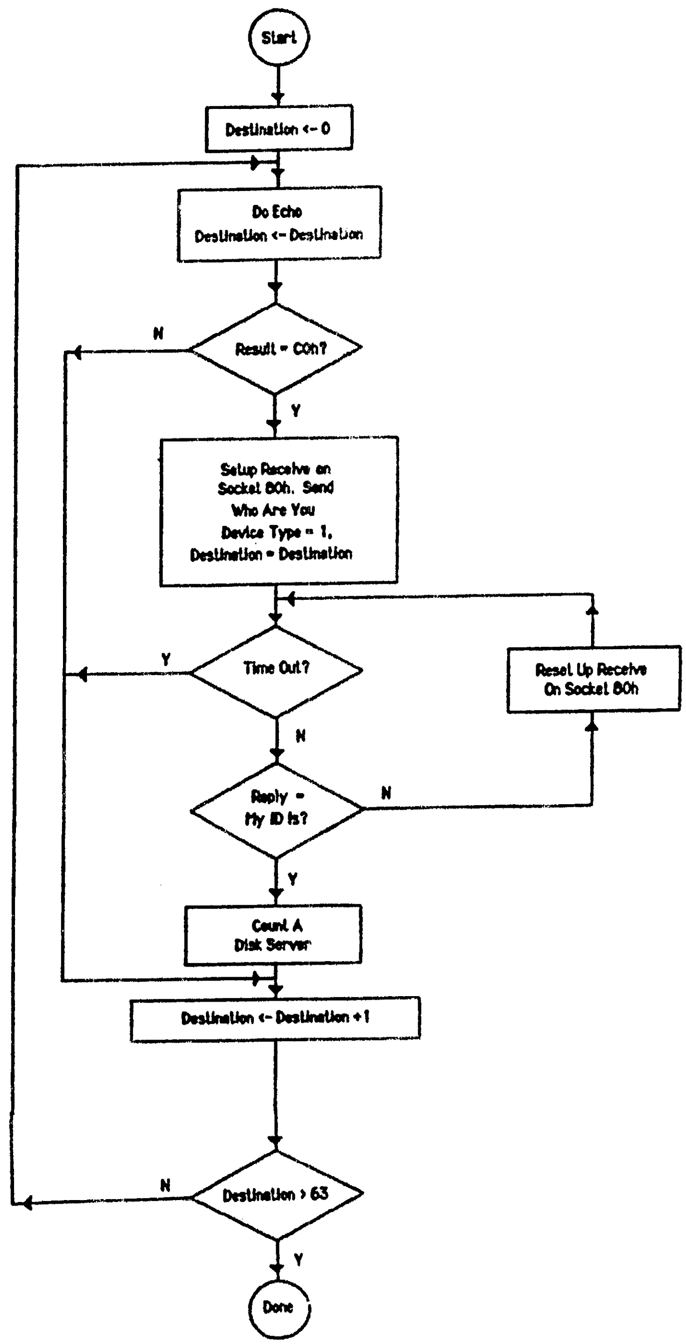

Figure 2.2a: Find all disk servers using directed messages

You could also use the following algorithm, but it is not quite as reliable since it uses a broadcast command and timeouts:

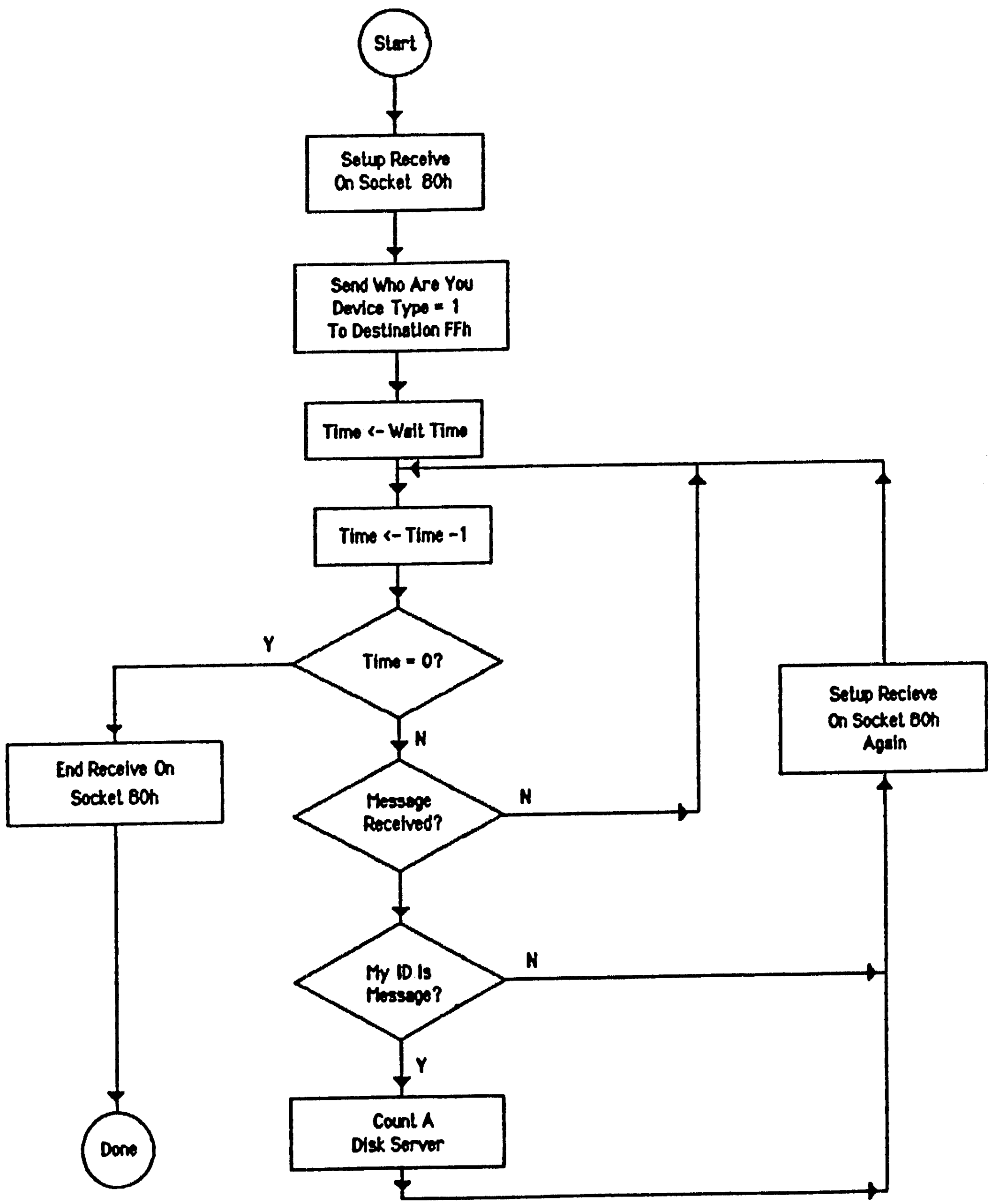

Figure 2.2b: Find all disk servers using broadcast messages

The following algorithm is used to reply to Who Are You and Where Are You messages:

1. Respond to all device types that apply.

2. If the device type is FFh, the device responds with its most specific device type.

3. If the device type is generic, and it is one of the generic types assigned to this device, then the device responds with the same generic device type. For example, if the OmniDrive receives a Who Are You, device type = 01h, it replies with a My ID Is, device type = 01h.

4. If the device type is specific, then the device responds with the same device type.

Name: Hello

Protocol ID: 01FEh

User Control Length: 0

Message Type: 0000h

User Data Length: 18

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FEh
MSGTYP	2 / 2	WORD	Message type - 0000h
SOURCE	4 / 2	WORD	Omninet address of device
DEVTYPE	6 / 2	WORD	Device type
NAME	8 / 10	BSTR	Device name

This message should be broadcast whenever a host "logs onto" the network.

Whenever a disk server receives one of these messages, it adds the device to its Active User Table. If DEVTYPE is 1, indicating that the Hello message came from some other disk server, then the receiving disk server sends back a My ID Is message to the originator of the Hello message. See the discussion of the Active User Table in the next section.

Name: Goodbye

Protocol ID: 01FEh

User Control Length: 0

Message Type: FFFFh

User Data Length: 18

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FEh
MSGTYP	2 / 2	WORD	Message type - FFFFh
SOURCE	4 / 2	WORD	Omninet address of device
DEVTYPE	6 / 2	WORD	Device type
NAME	8 / 10	BSTR	Device name

This message should be broadcast whenever a host "logs off" the network.

Name: Who Are You

Protocol ID: 01FEh

User Control Length: 0

Message Type: 0200h

User Data Length: 8

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FEh
MSGTYP	2 / 2	WORD	Message type - 0200h
SOURCE	4 / 2	WORD	Omninet address of deivce
DEVTYPE	6 / 2	WORD	Device type

This message can be directed or broadcast. Only devices which are assigned the specified DEVTYPE will respond. If DEVTYPE = FFh, all devices will respond.

The expected response is a My ID Is message.

Name: Where Are You

Protocol ID: 01FEh

User Control Length: 0

Message Type: 0300h

User Data Length: 18

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FEh
MSGTYP	2 / 2	WORD	Message type - 0300h
SOURCE	4 / 2	WORD	Omninet address of device
DEVTYPE	6 / 2	WORD	Device type
NAME	8 / 10	BSTR	Device name

This message is broadcast. Only devices with the specified name and device type will respond.

The expected response is a My ID Is message.

Name: My ID Is

Protocol ID: 01FEh

User Control Length: 0

Message Type: 1000h

User Data Length: 18

Socket Usage: 80h

User Data Format:

Field Name	Offset/Len	Type	Description
PID	0 / 2	WORD	Protocol ID # - 01FEh
MSGTYP	2 / 2	WORD	Message type - 1000h
SOURCE	4 / 2	WORD	Omninet address of device
DEVTYPE	6 / 2	WORD	Device type
NAME	8 / 10	BSTR	Device name

This message is sent in reponse to a Who Are You or a Where Are You message.

ACTIVE USER TABLE

It is not practical to implement the Constellation name protocol on all hosts, because the name lookup protocol requires that a host respond to an asynchronous message. Not all processors or operating systems support asynchronous events. Therefore, Corvus provides a rudimentary name service with the Active User Table. The contents of this table were described in Chapter 1. The Active User Table commands are repeated below:

AddActive
DeleteActiveUsr
DeleteActiveNumber
FindActive
ReadTempBlock
WriteTempBlock

An Active User Table is maintained on each disk device on the network. Whenever a disk device receives a Hello message, it adds the user to its Active User Table with an AddActive command. Similarly, whenever a disk device receives a Goodbye message, it deletes the user with a DeleteActiveUsr command.

If all the hosts on the network broadcast a Hello message on boot up, and broadcast a Goodbye message as part of the shut-down procedure, then the Active User Table will usually contain a list of which hosts are currently active on the network.

However, since the Hello and Goodbye messages are normally broadcast, it is possible that a disk device may miss a Hello or Goodbye message, and that an Active User Table may not reflect the actual state of the network. It is also possible, in a multiple disk server network, that the Active User Table on one disk device may not be the same as that on another disk device.

Each disk device is responsible for initializing its Active User Table. Here is the sequence of events that occurs when a disk server is powered on:

1. The disk server broadcasts a Hello message with a device ID of 1.

2. If another server is present on the network, it will add the new server to its Active User table, and send a My ID Is message back to the new server.

3. If the new server receives a My ID Is message, it reads the Active User table from the server that sent the message, and uses it to initialize its own table.

4. If the new server does not receive a My ID Is message, then there are no other disk servers on the network, so it initializes its Active User table to blanks.

The OmniDrive goes through a process similar to the one detailed above, with one difference. The OmniDrive broadcasts a Hello message with a device ID of 1, so that the old disk server PROM will recognize it as a disk device. The OmniDrive then broadcasts another Hello message with a device ID of 6, so that the Active User Table will contain device ID 6 instead of 1.

Also for the sake of compatability, the OmniDrive replies to a Hello message with a My ID Is message of device type 1. For the Who Are You and Where Are You messages, the OmniDrive replies with device type 6.

The Bank has an Omninet device type of 5. This number is used for the Hello message during power on and for response to the Who Are You message. The Bank does not implement the Active User Table.

OUTLINE OF A DISK DRIVER | 3

This chapter outlines a simple disk driver that interfaces to any Corvus mass storage device. If written properly, the same Omninet driver can support a disk server, an Omnidrive mass storage system, or The Bank mass storage system. A flat cable driver can support a Rev B/H drive directly, or one connected via a MUX.

When writing a disk driver, you should remember that the Corvus disk merely supports absolute disk sector reads-writes. It knows nothing about which computers are connected to it, nor whether it is connected over flat cable or Omninet. It knows nothing about volumes or users or file systems. In a network environment, the drive merely knows which command came from which computer, so that it can send the reply to the proper computer. Thus, a disk driver for a Corvus device resides at the BIOS level of the operating system. This is different from other network implementations, where references to the disk may be intercepted at the file level.

A typical BIOS level interface for a disk driver has at least three entry points: Driver Initialization, Device Read, and Device Write. These are the only functions discussed here.

The Device Read and Write entry points generally have the following parameters:

Device number: this number is used as an index into a table of device characteristics, such as device type, device location, device size, etc.

Sector number: this is the sector number to be read or written. Disk devices consist of n sectors, numbered 0 to n-1.

Number of sectors: this is the number of sectors to be read or written.

Buffer: this is the address of a buffer where the data is to be read into or written from.

Result code: this value is returned. It either indicates a successful operation, or indicates the nature of the failure.

The Device Read portion of the driver sends a Corvus disk Read Sector command, and returns the data in the user's buffer. The Device Write portion sends a Write Sector command along with the data in the user's buffer. The sector command used (128, 256, 512, or 1024 bytes) depends upon the sector size used by the operating system. The examples below assume a 512 byte sector size. Any information that depends on sector size is marked.

For the purposes of this chapter, it is assumed that the disk driver treats the entire disk as one device. See the Constellation Software General Technical Information Manual for information on how a Constellation disk driver treats a disk as more than one device.

There are several types of errors that the driver can encounter: timeout errors (device does not respond), disk errors (controller errors), hardware errors (Omninet transporter errors). Your driver must map these errors into the codes that your operating system defines.

OMNINET

You may want to refer to the following manuals while reading this section:

> Omninet Local Area Network General Technical Information, Chapter 3, pages 31-38, which describes the Omninet commands Setup Receive, Send, etc.
>
> Chapter 2 of this manual, which describes the disk server protocols.
>
> Chapter 1 of this manual, which describes the sector read and write commands.

The disk driver described here is simplified in two ways. First, this description assumes that the disk driver is the only user of the tranporter (TM) interface card; that is, the disk driver expects to be able to use the transporter at will and it throws away messages it does not recognize. In reality, the transporter functions should be handled by a transporter driver, and the disk driver should call on the transporter driver to do transporter functions. Corvus is currently developing a specification of a transporter driver and software which uses such a driver.

Secondly, the description of the disk driver given here ignores whether the transporter is buffered or unbuffered. A driver which handles a buffered transporter will naturally be more complicated since it must manage the buffer space and move data to and from user memory. Of course, if a transporter driver existed which the disk driver could use, then the transporter driver would handle the buffering, and the disk driver would not have to worry about whether the transporter were buffered or not. This is another reason for having a transporter driver.

However, as mentioned above, the driver described here does not assume the existence of a transporter driver.

The driver is described by the data structures, flowcharts and notes on the next few pages. The flowcharts cover how to send short and long commands and describe timeout recovery procedures. Many systems have no recourse when a timeout error occurs. A driver written for one of these systems should implement the timeout recovery described here, but instead of reporting a timeout error, restart the operation from the appropriate point.

Figure 3.1 reviews the flow of data for a read (short) command, and for a write (long) command, and shows the areas where timeouts can occur.

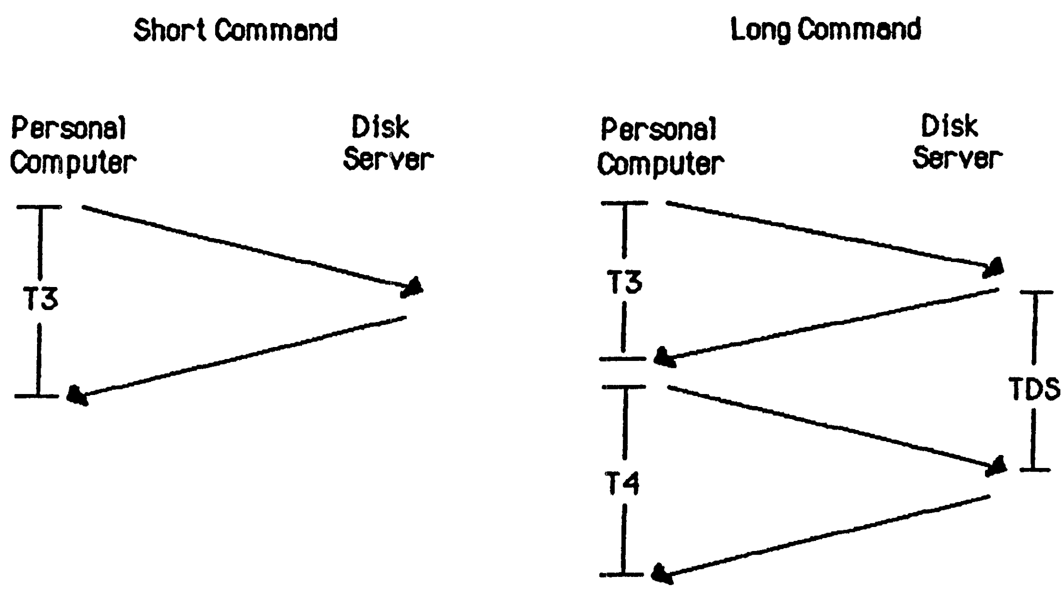

Figure 3.1 Timeouts for short and long command exchanges

There are two types of events which would cause a driver to time out: waiting for a response from the local transporter, and waiting for a disk server response. These can be broken down further as follows:

Transporter timeouts

T0: The time between a command strobe and the next ready. Recommended timeout value: 10ms.

T1: The time between strobing a receive command and the receive result changing from FFh to FEh. This is very fast, ususally within 200 microseconds. However, an incoming receive could happen during the processing of the Setup Receive, so the elapsed time could be several milliseconds. Recommended timeout value: 10ms.

T2: The time between strobing a Send command and its result changing. The result for a Send command does not change until an acknowledgement is received or the transporter gave up after sending 10 retransmissions. This can produce a very long delay (in computer time), since 11 transmissions are possible and the transporter will accept messages for any receives which are set up. Recommended timeout value: 100ms.

Disk Server timeouts (refer to figure 3.1)

T3: The time between the completion of the Send of the Disk Request message and the receipt of the Results or Go message. This interval could be as long as 3 minutes for a disk, and 11 hours for a Bank. Recommended timeout value: see below.

T4 The time between the completion of the Send of the Last message and the receipt of the Results message. Recommended timeout value: 150ms for a disk, 20 seconds for a Bank.

The disk server itself will timeout between sending a Go message and receiving the Last message. This timeout value is 768ms. This time is indicated in figure 3.1 by TDS.

Most systems do not use the transporter timeouts (T0, T1, and T2) since there is nothing they can do if the transporter is not working reliably.

All systems must support the disk server timeouts (T3 and T4) in order to work reliably in a multiple server environment. The timeout value for T3 must be variable, since a 3 minute or 11 hour timeout is not practical.

The recommended approach to implementing the T3 timeout is to use an adaptable timeout. Since different devices have different timing characteristics, the timeout value must depend upon the device type. Also, as more servers are added to a network, the response times will lengthen. Therefore, the timeout value must also adapt to the network environment.

The flow chart in figure 3.4 shows a very simple method for adapting the timeout values. The timeout value should start out relatively short (3 seconds for a disk, 20 seconds for a Bank), and increase only when a long delay is encountered.

The Old Disk Server Protocol is described first, and then the New Disk Server Protocol is described.

OLD DISK SERVER PROTOCOL

This section describes the old disk server protocol.

```
; Sample data structures for a disk server driver using Old Disk
; Server Protocol
;
; First the data structure is declared, then a list of offsets
; into the structure are declared.
;
; Transporter command vector (see Omninet GTI, pgs. 32,33)
;   It is not necessary to have more than one command vector,
;   although it is sometimes more convenient to use separate
;   records which are preinitialized as Send and Setup receive
;   commands.

TCmd    .BYTE 0          ; OpCode - command code
        .BYTE 0          ; ResAdr  - high order byte of result address
        .WORD 0          ;         - low order word of result address
        .BYTE 0          ; Sock - socket number
        .BYTE 0          ; DatAdr  - high order byte of data address
        .WORD 0          ;         - low order word of data address
        .WORD 0          ; DataLen - data length
        .BYTE 0          ; CrtlLen - user control length
        .BYTE 0FFh       ; Dest - destination host number
                         ; offsets
OpCode .EQU 0            ; offset to OpCode
ResAdr .EQU 1            ; offset to ResAdr
Sock   .EQU 4            ; offset to socket number
DatAdr .EQU 5            ; offset to DatAdr
DataLen.EQU 8            ; offset to data length
CrtlLen.EQU 10           ; offset to user control length
Dest   .EQU 11           ; offset to destination host number (Send only)
```

```
; Sample data structures for a disk server driver using Old Disk
; Server Protocol (cont.)
;
; Result record definitions (see section 2.2)
;   Every driver must have 2 separate result records, one for
;   sends, and one for receives.
                       ; Send result record
SndRes .BYTE 0         ; transporter return code
       .BYTE 0         ; unused
       .WORD 0         ; unused
SndUC  .WORD 0         ; M - the number of data bytes to send to drive
       .WORD 0         ; N - the maximum number of data bytes
                       ;     expected on receive
                       ; offsets
RCode  .EQU 0          ; offset to transporter return code
M      .EQU 0          ; offset to M
N      .EQU 2          ; offset to N

                       ; Receive result record
RcvRes .BYTE 0         ; transporter return code
       .BYTE 0         ; Src - source host number
       .WORD 0         ; Len - actual length of data received
RcvUC  .WORD 0         ; DLen - number of bytes actually returned from driv
       .BYTE 0         ; DCode - disk return code
                       ; offsets
Src    .EQU 1          ; offset to Src
Len    .EQU 2          ; offset to Len
DLen   .EQU 0          ; offset to DLen
DCode  .EQU 2          ; offset to DCode
;
;
; Data area buffers
;
GoData .BYTE 0FFh      ; this is where we receive the 'GO' packet
       .BYTE 0FFh

DCmd   .WORD 0         ; space for the disk command
       .WORD 0
```

```
; Sample data structures for a disk server driver using Old Disk
; Server Protocol (cont.)
;
; DrvRet is a global variable in the driver which each routine
;  sets.  It is the value that will be returned to the operating
;  system upon completion of the driver call.
;
DrvRet .BYTE 0          ; Driver return code

; DrvRet values:
;   The codes which are marked with an asterisk (*) are those
;   which may be returned to the caller of the driver.  All
;   others are used internally.  The codes which are marked with
;   a T are transporter return codes.
;
OkCode .EQU 0           ; *T
GiveUp .EQU 128         ; T - gave up after n retries
TooLong.EQU 129         ; T - message too long
NoSock .EQU 130         ; T - socket not initialized
BadHdr .EQU 131         ; T - header length mismatch - should never happen
SndErr .EQU 140         ; * - unable to send messages to disk server
TOErrDS.EQU 252         ;   - timed out waiting for disk server response
TOErrTR.EQU 253         ; * - timed out waiting for transporter
                        ;     (hardware error)
;
; The following global variables are set on each read or
;   write, to the values specified for the device.

TimeOut.WORD 0          ; used to control disk server wait loop
DSNum  .BYTE 0          ; disk server number
```

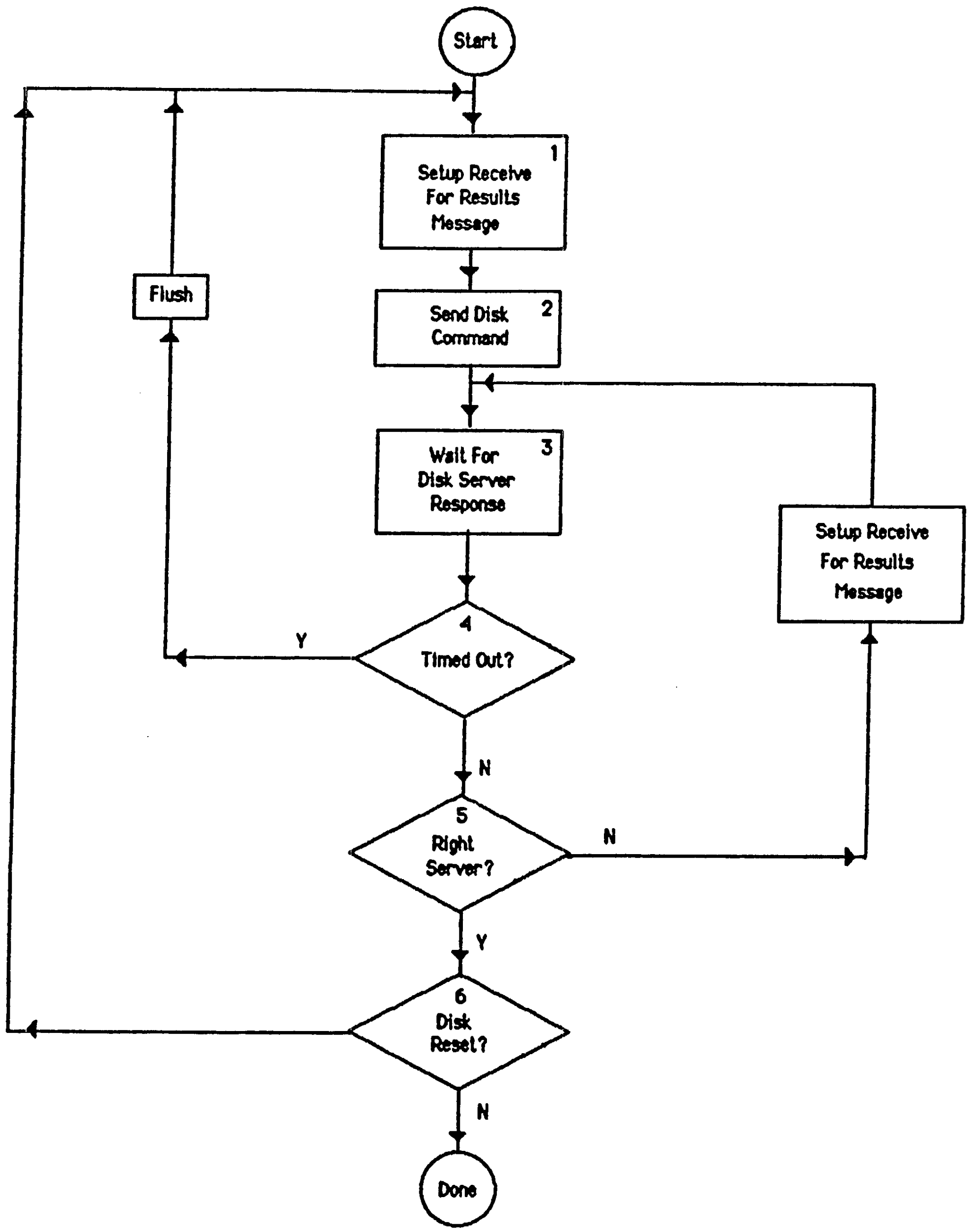

Figure 3.2: Flowchart of a short (read) command
Old Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions on the following pages.

1. Setup receive for results.

```
TCmd+OpCode   <- F0h (Setup Receive command)
TCmd+ResAdr   <- address of RcvRes
TCmd+Sock     <- B0h
TCmd+DatAdr   <- address of user's buffer
TCmd+DataLen  <- 512 (use appropriate sector size)
TCmd+CrtlLen  <- 3

RcvRes+Rcode  <- FFh (must initialize result code)
```

If transporter result code (RcvRes+Rcode) does not change within 10 ms, report a hardware error (TOErrTR) and exit.

2. Send disk command.

```
TCmd+OpCode   <- 40h (Send command)
TCmd+ResAdr   <- address of SndRes
TCmd+Sock     <- B0h
TCmd+DatAdr   <- address of DCmd buffer
TCmd+DataLen  <- 4 (4 byte read command)
TCmd+CrtlLen  <- 4
TCmd+Dest     <- DSNum

SndRes+Rcode  <- FFh (initialize result code)
SndUC +M      <- 4
SndUC +N      <- 512 (use appropriate sector size)

DCmd+0        <- 32h (use appropriate read command)
DCmd+1        <- sector address byte d
DCmd+2        <- sector address lsb
DCmd+3        <- sector address msb
```

If transporter result code (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

3. Wait for disk server response.

   This is a loop which is checking the transporter return code in the receive buffer (RcvRes+Rcode). When this value goes to zero, the disk read has completed. See figure 3.4 and accompanying notes.

4. If a timeout error occurred, try to recover. See figure 3.5 for a description of the recovery procedure.

5. Check the responding disk server (RcvRes+Src). If it does not match the destination disk server (DSNum) the message received is irrelevant. Setup the receive again, and wait for another response.

6. Check the first byte of the User Control Data (RcvUC +DLen). If the most significant bit is on, the disk has been reset. Start the entire sequence over.

   Check the disk result (RcvUC+Dcode). If the most significant bit is on, report an error.

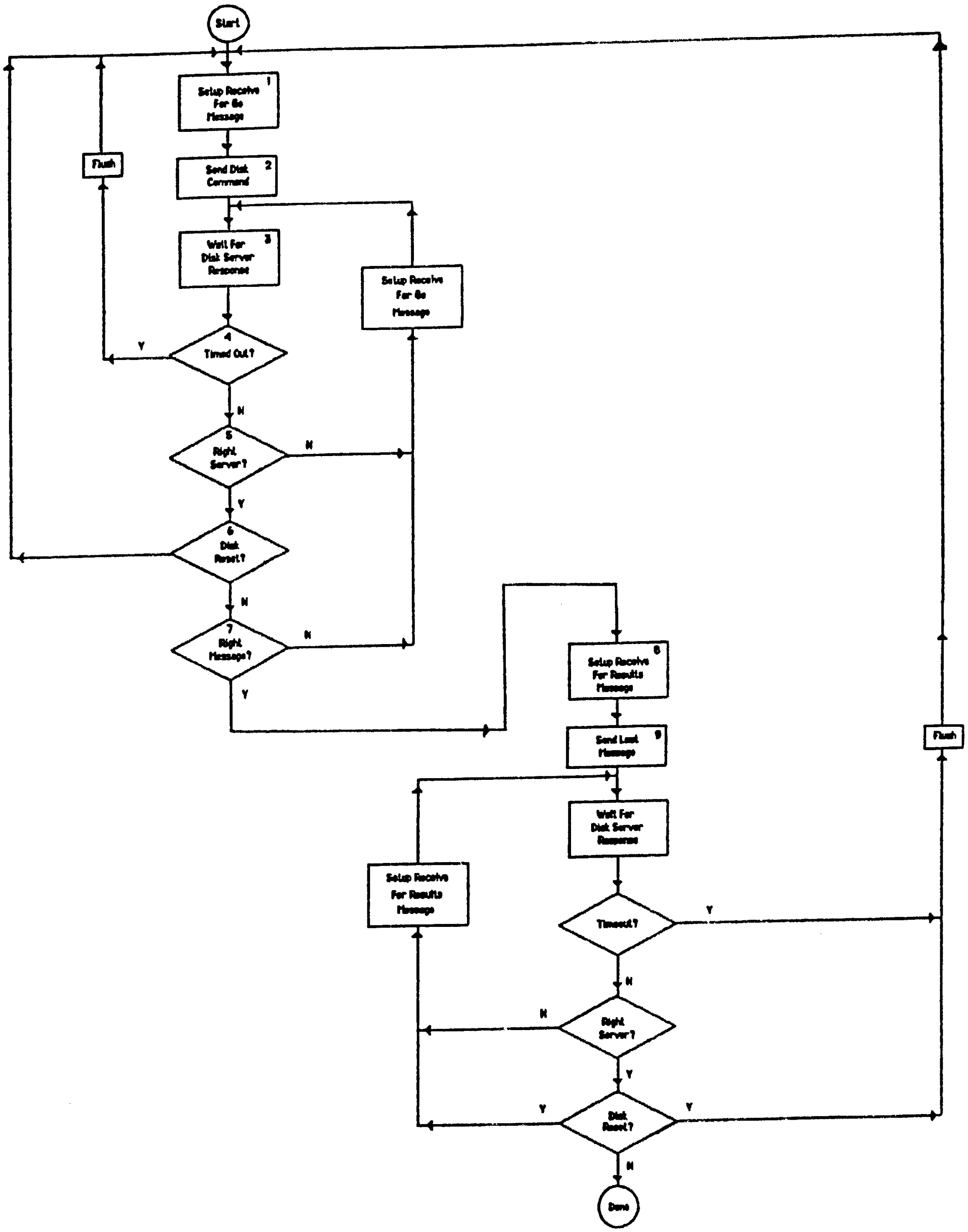

Figure 3.3: Flowchart of a long (write) command
Old Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions on the following pages.

1. Setup receive for the 'GO' command.

```
TCmd+OpCode  <- F0h (Setup Receive command)
TCmd+ResAdr  <- address of RcvRes
TCmd+Sock    <- B0h
TCmd+DatAdr  <- address of GoData
TCmd+DataLen <- 2
TCmd+CrtlLen <- 0

RcvRes+Rcode <- FFh (must initialize the result code)
```

If transporter result code (RcvRes+Rcode) does not change within 10 ms, report a hardware error (TOErrTR) and exit.

2. Send the first 4 bytes of the write command.

```
TCmd+OpCode  <- 40h (Send command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- B0h
TCmd+DatAdr  <- address of DCmd buffer
TCmd+DataLen <- 4
TCmd+CrtlLen <- 4
TCmd+Dest    <- DSNum

SndRes+Rcode <- FFh (initialize result code)
SndUC +M     <- 516 (use appropriate sector size)
SndUC +N     <- 0

DCmd+0       <- 33h (use appropriate read command)
DCmd+1       <- sector address byte d
DCmd+2       <- sector address lsb
DCmd+3       <- sector address msb
```

If transporter result code (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

3. Wait for disk server response.

   This is a loop which is checking the transporter return code (SndRes+Rcode). When this value goes to zero, the 'GO' message has been received. See figure 3.4 and accompanying notes.

4. If a timeout error occurred, try to recover. See figure 3.5 for a description of the recovery procedure.

5. Check the responding disk server (RcvRes+Src). If it does not match the destination disk server (DSNum) the message received is irrelevant. Setup the receive again, and wait for another response.

6. Check the first byte of the data buffer (GoData). If the most significant bit is on, the disk server has been reset, and you should restart the sequence from the beginning.

7. If the data received is anything but the 2 bytes 'GO', the message is irrelevant. Setup the receive again, and wait for another response.

8. Set up another receive to get the results of the next Send.

```
TCmd+OpCode   <- F0h (Setup Receive command)
TCmd+ResAdr   <- address of RcvRes
TCmd+Sock     <- B0h
TCmd+DatAdr   <- address of DCmd buffer
TCmd+DataLen  <- 4
TCmd+CrtlLen  <- 3

RcvRes+Rcode  <- FFh (must initialize the result code)
```

If transporter result code (RcvRes+Rcode) does not change within 10 ms, report a hardware error (TOErrTR) and exit.

9. Send the rest of the Write command. Note that the socket number is A0h, not B0h as for the previous commands.

```
TCmd+OpCode   <- 40h (Send command)
TCmd+ResAdr   <- address of SndRes
TCmd+Sock     <- A0h
TCmd+DatAdr   <- address of user's buffer
TCmd+DataLen  <- 512 (use appropriate sector size)
TCmd+CrtlLen  <- 0
TCmd+Dest     <- DSNum

SndRes+Rcode  <- FFh (initialize result code)
```

User's buffer contains the data to be written.

If transporter result code (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

If the transporter result code is 82h (uninitialized socket), then the disk server has timed out waiting for the second half of the disk command. You should restart the operation from the beginning.

10. Check the first byte of the User Control Data (RcvUC +DLen). If the most significant bit is on, the disk has been reset. Start the entire sequence over.

Check the disk result code (RcvUC+Dcode). If the most significant bit is on, report an error.

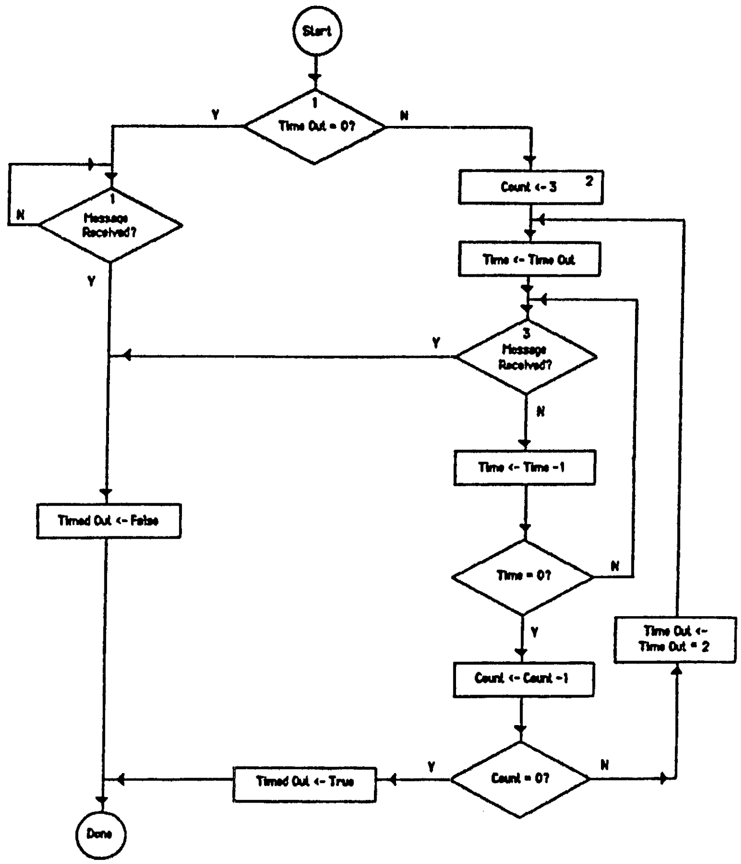

Figure 3.4: Wait for disk server response Old Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions below.

1. The timeout value should be set to whatever is specifed in the device table for this device. If the timeout value is 0, the driver loops forever, waiting for a response. A timeout value of 0 should be used only for Mirror and Prep mode commands.

2. The count of 3 is arbitrary. It is basically a retry count.

3. The loop terminates when the transporter return code goes to 0 (message received), or when the timeout value is reached.

4. If the number of retries is exceeded, report a timeout error and exit.

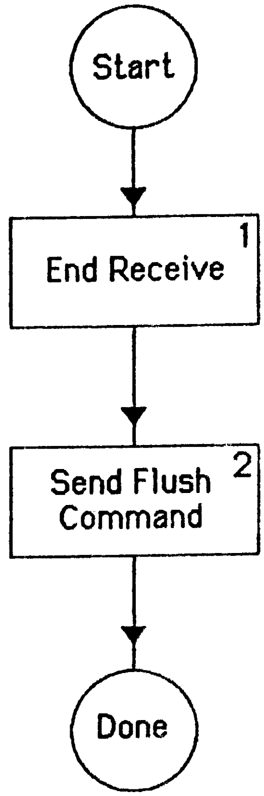

Figure 3.5: Flush
Old Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions below.

1. Do an End Receive on socket B0h.

```
TCmd+OpCode   <- 10h (End receive command)
TCmd+ResAdr   <- address of SndRes
TCmd+Sock     <- B0h

SndRes+Rcode <- FFh (initialize result code)
```

   If transporter result (SndRes+Rcode) does not change within 10ms, report a hardware error (DrvRet <- TOErrTR) and exit.

   If transporter result (SndRes+Rcode) is not 0, report a hardware error (DrvRet <- TOErrTR) and exit.

2. Send a Flush command.

```
TCmD+OpCode   <- 40h (Send command)
TCmD+ResAdr   <- address of SndRes
TCmD+Sock     <- B0h
TCmD+DatAdr   <- address of DCmd buffer
TCmD+DataLen  <- 4
TCmD+CrtlLen  <- 4
TCmD+Dest     <- DSNum

SndRes+Rcode  <- FFh (initialize result code)
SndUC +M      <- 0
SndUC +N      <-0
```

If transporter result (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

NEW DISK SERVER PROTOCOL

The description of the New Disk Server Protocol is very similar to that of the Old Disk Server Protocol, but there are two important differences. The first is that the driver must be prepared to generate request IDs and use media IDS. The second is that the driver must be prepared to receive a Cancel or Restart message at any time. The flowcharts for Wait for Disk Server Response (figure 3.9) and Flush (figure 3.10) are therefore more complicated. The flowcharts for the Short (figure 3.6) and Long (figure 3.7) commands look similar to those for the Old Disk Server Protocol (figures 3.2 and 3.3), but the explanations differ.

The new disk server protocol requires that you specify to which socket, A0h or B0h, the server should send the Results message. The server tells you to which socket you should send the Last message.

You will also see that some of the fields in the declarations are described in three places: as part of the RcvUC record, as part of the SndUc record, and as part of the Dcmd record. This is because the protocol information is sometimes included in the User Data portion of the message, and sometimes in the User Control portion.

```
; Sample data structures for a disk server driver using New Disk
;   Server Protocol
;
; First the data structure is declared, then a list of offsets
; into the structure are declared.
;
;
; Transporter command vector (see Omninet GTI, pgs. 32,33)
;   It is not necessary to have more than one command record,
;   although it is sometimes more convenient to use separate
;   records which are preinitialized as Send and Setup receive
;   commands.

TCmd    .BYTE 0         ; OpCode - command code
        .BYTE 0         ; ResAdr  - high order byte of result address
        .WORD 0         ;           - low order word of result address
        .BYTE 0         ; Sock - socket number
        .BYTE 0         ; DatAdr  - high order byte of data address
        .WORD 0         ;           - low order word of data address
        .WORD 0         ; DataLen - data length
        .BYTE 0         ; CrtlLen - user control length
        .BYTE 0FFh      ; Dest - destination host number
                        ; offsets
OpCode .EQU 0           ; offset to OpCode
ResAdr .EQU 1           ; offset to ResAdr
Sock   .EQU 4           ; offset to socket number
DatAdr .EQU 5           ; offset to DatAdr
DataLen.EQU 8           ; offset to data length
CrtlLen.EQU 10          ; offset to user control length
Dest   .EQU 11          ; offset to destination host number (Send only)
```

```
; Sample data structures for a disk server driver using New Disk
;   Server Protocol (cont.)
;
; Result record definitions (see section 2.3)
;   Every driver should have 2 separate result records, one for
;   sends, and one for receives.
;
                            ; Send result record
SndRes .BYTE 0              ; transporter return code
       .BYTE 0              ; unused
       .WORD 0              ; unused
SndUC  .WORD 0              ; ProtoID - Protocol ID
       .WORD 0              ; MsgTyp - message type
       .WORD 0              ; RqstID - request ID
       .WORD 0              ; M - the number of data bytes to send to drive
       .WORD 0              ; N - the maximum number of data bytes
                            ;     expected on receive
                            ; offsets
RCode   .EQU 0              ; offset to transporter return code
ProtoID.EQU 0               ; offset to ProtoID
MsgTyp .EQU 2               ; offset to MsgTyp
RqstID .EQU 4               ; offset to RqstID
Reason .EQU 6               ; offset to Reason (for Cancel and Restart)
MediaI2.EQU 8               ; offset to MediaID (for Cancel and Restart)

                            ; Receive result record
RcvRes .BYTE 0              ; transporter return code
       .BYTE 0              ; Src - source host number
       .WORD 0              ; Len - actual length of data received
RcvUC  .WORD 0              ; ProtoID - Protocol ID
       .WORD 0              ; MsgTyp - message type
       .WORD 0              ; RqstID - request ID
       .WORD 0              ; NActual - number of bytes returned from drive
       .BYTE 0              ; reserved
       .BYTE 0              ; DCode - disk return code
       .WORD 0              ; reserved
                            ; offsets
Src    .EQU 1               ; offset to Src
Len    .EQU 2               ; offset to Len
NActual.EQU 6               ; offset to NActual
DCode  .EQU 9               ; offset to DCode
                            ; Second receive result record for Cancel or Restart
Rcv80  .BYTE 0              ; transporter return code
       .BYTE 0              ; Src - source host number
       .WORD 0              ;
```

```
; Sample data structures for a disk server driver using New Disk
;   Server Protocol (cont.)
;
; Data area buffers
;
DCmd    .WORD 0           ; ProtoID
        .WORD 0           ; MsgTyp
        .WORD 0           ; RqstID
        .WORD 0           ; MediaID
        .BYTE 0           ; ResHost
        .BYTE 0           ; ResSock
        .WORD 0           ; M
        .WORD 0           ; N
        .WORD 0           ; space for the disk command (4 bytes)
        .WORD 0
                          ; offsets
MediaID.EQU 6             ; offset to MediaID
ResHost.EQU 8             ; offset to ResHost
ResSock.EQU 9             ; offset to ResSock
M       .EQU 10           ; offset to M
N       .EQU 12           ; offset to N
Cmd     .EQU 14           ; offset to start of command

                          ; space for socket 80h messages (Go, Cancel or Resta
S80Msg  .WORD 0           ; ProtoID
        .WORD 0           ; MsgTyp
        .WORD 0           ; RqstID
        .WORD 0           ; Reason, LastSock
        .WORD 0           ; MediaID
                          ; offsets
LstSock.EQU  7            ; Last socket for Go message
```

```
; Sample data structures for a disk server driver using New Disk
;   Server Protocol (cont.)
;
; DrvRet is a global variable in the driver which each routine
;  sets.  It is the value that will be returned to the operating
;  system upon completion of the driver call.
;
DrvRet .BYTE 0          ; Driver return code

; DrvRet values:
;   The codes which are marked with an asterisk (*) are those
;   which may be returned to the caller of the driver.  All
;   others are used internally.  The codes which are marked with
;   a T are transporter return codes.
;
OkCode .EQU 0           ; *T
GiveUp .EQU 128         ; T - gave up after n retries
TooLong.EQU 129         ; T - message too long
NoSock .EQU 130         ; T - socket not initialized
BadHdr .EQU 131         ; T - header length mismatch should never happen
SndErr .EQU 140         ; * - unable to send messages to disk server
TOErrDS.EQU 252         ;   - timed out waiting for disk server response
TOErrTR.EQU 253         ; * - timed out waiting for transporter
                        ;     (hardware error)

;
; The following global variables are set on each call from the
;   values specified for the device.
;
TimeOut.WORD 0          ; used to control disk server wait loop
DSNum  .BYTE 0FFh       ; disk server number
Media  .WORD 0          ; media id

; The following global variables are set on each call.
;
UseSock.BYTE 0          ; which socket to use (A0h or B0h)
Request.WORD 0          ; bumped by 1 on each call

; The following global variables are set at driver
;   initialization
;
MyAddr .BYTE 0          ; this computer's transporter address
```

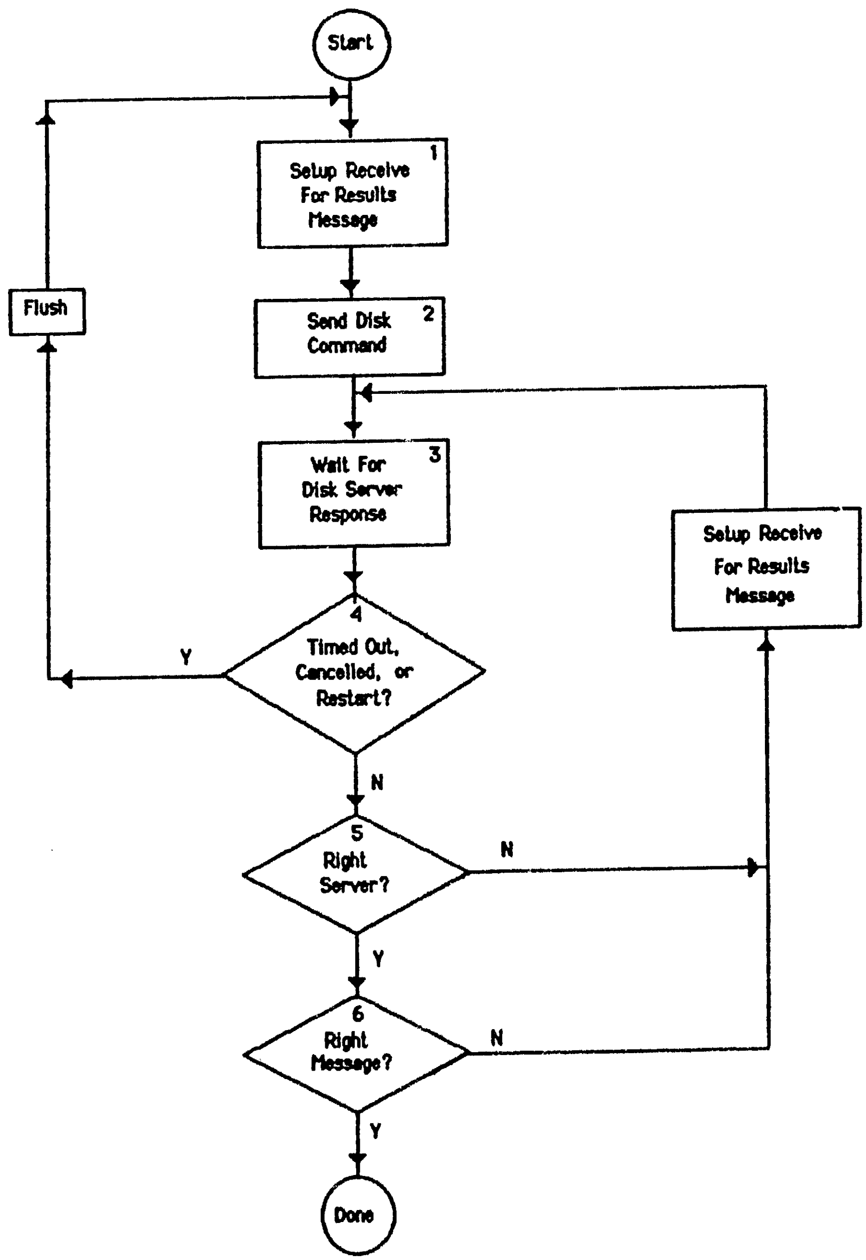

Figure 3.6: Flowchart of a short (read) command
New Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions on the following pages.

1. Setup receive for results.

```
TCmd+OpCode   <- F0h (Setup Receive command)
TCmd+ResAdr   <- address of RcvRes
TCmd+Sock     <- UseSock
TCmd+DatAdr   <- address of user's buffer
TCmd+DataLen  <- 512 (use appropriate sector size)
TCmd+CrtlLen  <- 12

RcvRes+Rcode  <- FFh (must initialize result code)
```

If transporter result code (RcvRes+Rcode) does not change within 10 ms, report a hardware error (TOErrTR) and exit.

Setup receive for possible socket 80h message (Cancel or Restart):

```
TCmd+OpCode   <- F0h (Setup Receive command)
TCmd+ResAdr   <- address of Rcv80
TCmd+Sock     <- 80h
TCmd+DatAdr   <- address of S80Msg
TCmd+DataLen  <- 8
TCmd+CrtlLen  <- 0

Rcv80+Rcode   <- FFh (must initialize result code)
```

2. Send disk command.

```
TCmd+OpCode   <- 40h (Send command)
TCmd+ResAdr   <- address of SndRes
TCmd+Sock     <- 80h
TCmd+DatAdr   <- address of DCmd buffer
TCmd+DataLen  <- 18
TCmd+CrtlLen  <- 4
TCmd+Dest     <- DSNum

SndRes+Rcode  <- FFh (initialize result code)
SndUc +M      <- 4
SndUc +N      <- 512 (use appropriate sector size)

DCmd+ProtoID  <- 01FFh
DCmd+MsgTyp   <- 0001h (Disk request)
DCmd+RqstID   <- Request
DCmd+MediaID  <- Media
DCmd+ResHost  <- MyAddr
DCmd+ResSock  <- UseSock
DCmd+M        <- 4 (4 byte read command)
DCmd+N        <- 512 (use appropriate sector size)
DCmd+Cmd      <- 32h (use appropriate read command)
DCmd+Cmd+1    <- sector address byte d
DCmd+Cmd+2    <- sector address lsb
DCmd+Cmd+3    <- sector address msb
```

If transporter result code (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

3. Wait for disk server response.

   This is a loop which is checking the transporter return code in the receive buffer (RcvRes+Rcode). When this value goes to zero, the disk read has completed. See figure 3.8 and accompanying notes.

   This loop must also check whether a Cancel or Restart message has been received. See figure 3.9 and accompanying notes.

4. If a timeout error or cancellation occurred, try to recover. See figure 3.10 for a description of the recovery procedure.

5. Check the responding disk server (RcvRes+Src). If it does not match the destination disk server (DSNum) the message received is irrelevant. Setup the receive again, and wait for another response.

6. Check the User Control Data (RcvUC). Ensure the ProtoID is 1FFh, and that MsgTyp is 0200h. If not, the message received is irrelevant. Setup the receive again, and wait for another response.

   Check the disk result (RcvUC+Dcode). If the most significant bit is on, report an error.

   Do an End Receive on socket 80h.

```
TCmd+OpCode  <- 10h (End Receive command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- 80h

SndRes+Rcode <- FFh (initialize result code)
```

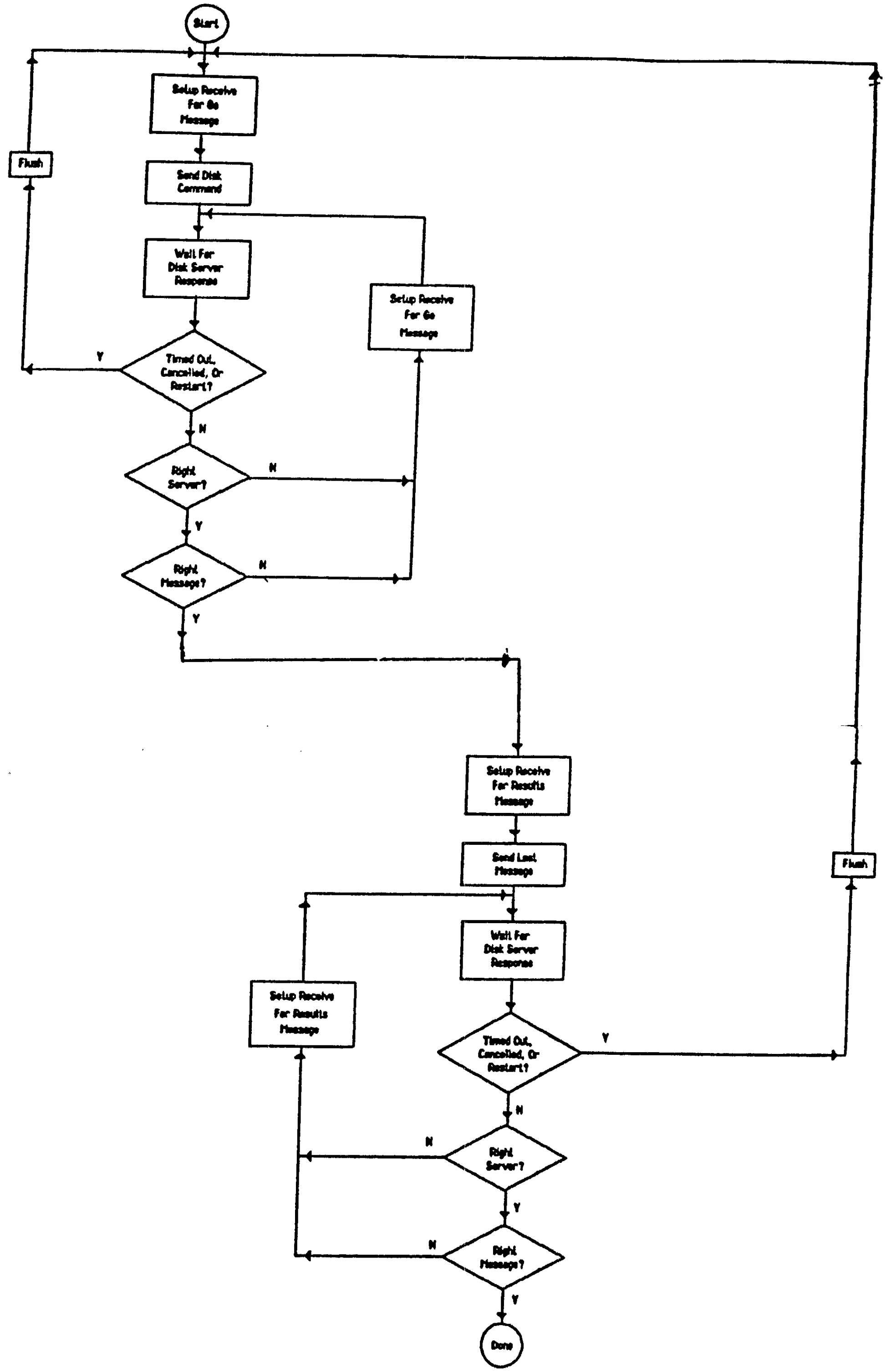

Figure 3.7: Flowchart of a long (write) command
New Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions on the following pages.

1. Setup receive for the Go message. The Go message is sent to socket 80h.

```
TCmd+OpCode  <- F0h (Setup Receive command)
TCmd+ResAdr  <- address of RcvRes
TCmd+Sock    <- 80h
TCmd+DatAdr  <- address of S80Msg
TCmd+DataLen <- 8
TCmd+CrtlLen <- 0

Rcv80+Rcode  <- FFh (must initialize result code)
```

If transporter result code (RcvRes+Rcode) does not change within 10 ms, report a hardware error (TOErrTR) and exit.

2. Send the first 4 bytes of the write command.

```
TCmd+OpCode  <- 40h (Send command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- 80h
TCmd+DatAdr  <- address of DCmd buffer
TCmd+DataLen <- 18
TCmd+CrtlLen <- 4
TCmd+Dest    <- DSNum

SndRes+Rcode <- FFh (initialize result code)

DCmd+0       <- 1FFh (protocol id)
DCmd+2       <- 001h (message type = Disk request)
DCmd+4       <- request id
DCmd+6       <- media id
DCmd+8       <- FFh
DCmd+9       <- UseSock
DCmd+10      <- 516 (use appropriate sector size)
DCmd+12      <- 1
DCmd+14      <- 33h (use appropriate read command)
DCmd+15      <- sector address byte d
DCmd+16      <- sector address lsb
DCmd+17      <- sector address msb
```

If transporter result code (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

3 Wait for disk server response.

This is a loop which is checking the transporter return code. Since the Go message will be received on socket 80h, the driver must check Rcv80+Rcode, not RcvRes+Rcode, as in all the other cases. When this value goes to zero, a message has been received. See figure 3.8 and accompanying notes.

This loop must also check whether a Cancel or Restart message has been received. See figure 3.9 and accompanying notes.

4. If a timeout or cancellation error occurred, try to recover. See figure 3.10 for a description of the recovery procedure.

5. Check the responding disk server (Rcv80+Src). If it does not match the destination disk server (DSNum) the message received is irrelevant. Setup the receive again, and wait for another response.

6. No box.

7. If the data received is anything but the Go message (S80Msg+ProtoID=01FFh, S80Msg+MsgTyp=0100h), the message is irrelevant. Setup the receive again, and wait for another response.

8. Set up another receive to get the results of the next Send.

```
TCmd+OpCode  <- F0h (Setup Receive command)
TCmd+ResAdr  <- address of RcvRes
TCmd+Sock    <- UseSock
TCmd+DatAdr  <- address of DCmd buffer
TCmd+DataLen <- 4
TCmd+CrtlLen <- 12

RcvRes+Rcode <- FFh (must initialize result code)
```

If transporter result code (RcvRes+Rcode) does not change within 10 ms, report a hardware error (TOErrTR) and exit.

Setup receive for possible socket 80h message (Cancel or Restart):

```
TCmd+OpCode  <- F0h (Setup Receive command)
TCmd+ResAdr  <- address of Rcv80
TCmd+Sock    <- 80h
TCmd+DatAdr  <- address of S80Msg
TCmd+DataLen <- 8
TCmd+CrtlLen <- 0

Rcv80+Rcode  <- FFh (must initialize result code)
```

9. Send the rest of the Write command.

```
TCmd+OpCode  <- 40h (Send command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- specified in Go message (S80Msg+LstSock)
TCmd+DatAdr  <- address of user's buffer
TCmd+DataLen <- 512 (use appropriate sector size)
TCmd+CrtlLen <- 12
```

```
TCmd+Dest     <- DSNum

SndRes+Rcode <- FFh (initialize result code)
SndUC +ProtoId<-1FFh
SndUC +Msgtyp<- 002h (Last message)
SndUC +RqstId<- ReqestId
SndUC +Reser1<- 0
SndUC +Reser2<- 0
SndUC +Reser3<- 0
```

User's buffer contains the data to be written.

If transporter result code (SndRes+Rcode) does not change within 100 ms, report a hardware error (TOErrTR) and exit.

If the transporter result code is 82h (uninitialized socket), then the disk server has timed out waiting for the second half of the disk command. You should restart the operation from the beginning.

10. Check that the Results message was received (RcvUC+ProtoID = 1FFh; RcvUC+MsgTyp = 0200h). If not, the message received is irrelevant. Setup the receive again, and wait for another response.

    Check the disk result (RcvUC+Dcode). If the most significant bit is on, report an error.

    Do an End Receive on socket 80h.

```
TCmd+OpCode  <- 10h (End Receive command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- 80h

SndRes+Rcode <- FFh (initialize result code)
```

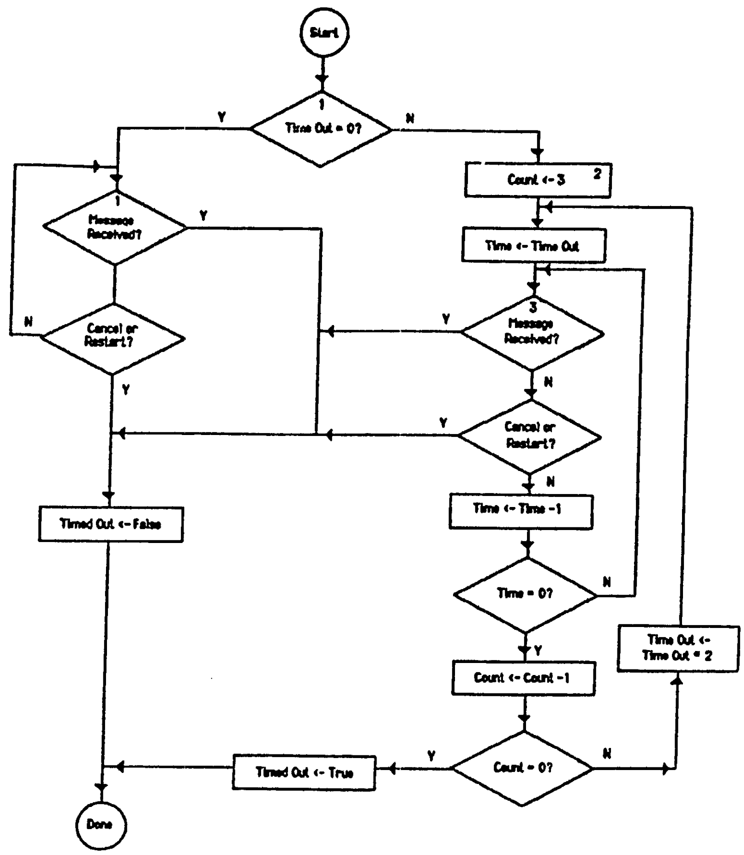

Figure 3.8: Wait for disk server response
New Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions below.

1. The timeout value should be set to whatever is specifed in the device table for this device. If the timeout value is 0, the driver loops forever, waiting for a response. A timeout value of 0 should be used only for Mirror and Prep mode commands.

2. The count of 3 is arbitrary. It is basically a retry count.

3. The loop terminates when the transporter return code goes to 0 (message received), when a Cancel or Restart message is received, or when the timeout value is reached.

   See figure 3.9 for the Cancel and Restart check.

4. If the number of retries is exceeded, report a timeout error and exit.

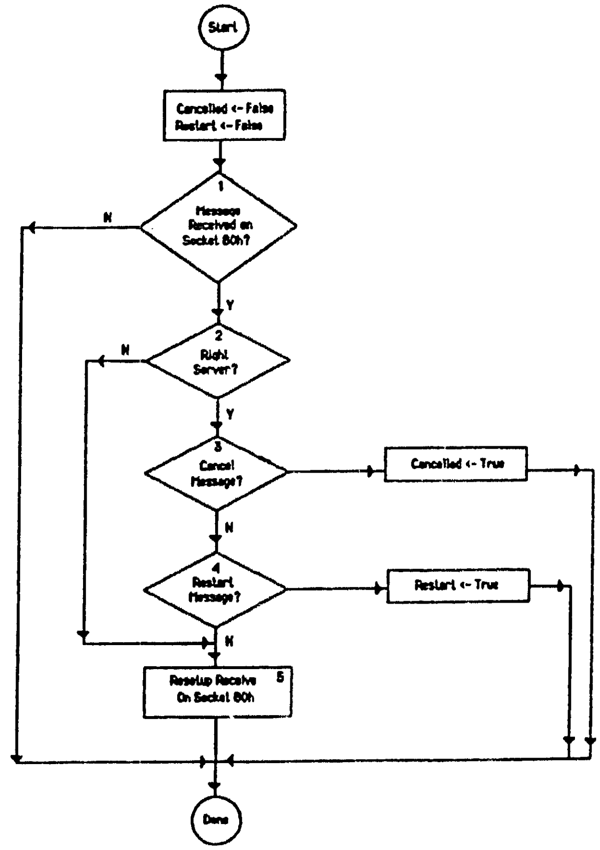

Figure 3.9: **Check for Cancel or Restart New Disk Server Protocol**

The numbers in the flowchart boxes refer to text descriptions below.

1. Has a message been received on socket 80h (Rcv80+Rcode=00h)? If not, continue waiting for disk server response.

2. Is the message from our server (Rcv80+Src=DSNum)? If not, ignore the message, resetup the receive on socket 80h, and go back to waiting.

3. Is the message a Cancel message (S80Msg+ProtoID=01FFh, S80Msg+MsgTyp=0300h)? If so, set Cancelled flag, and exit the wait for response loop.

4. Is the message a Restart message (S80Msg+ProtoID=01FFh, S80Msg+MsgTyp=FF00h)? If so, set Restart flag, and exit the wait for response loop.

5. The message is not a Cancel or Restart, so ignore it. Resetup the receive, and go back to waiting.

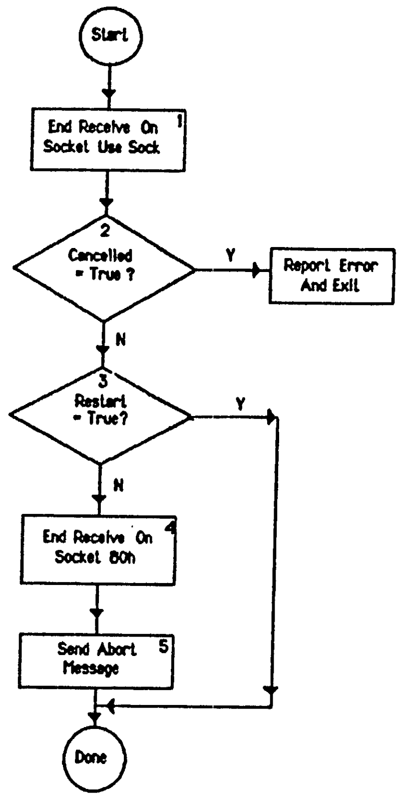

Figure 3.10: **Flush**
New Disk Server Protocol

The numbers in the flowchart boxes refer to text descriptions below.

1. Do an End Receive on socket UseSock.

```
TCmd+OpCode   <- 10h (End receive command)
TCmd+ResAdr   <- address of SndRes
TCmd+Sock     <- UseSock.

SndRes+Rcode <- FFh (initialize result code)
```

   If transporter result (SndRes+Rcode) does not change within 10ms, report a hardware error (DrvRet <- TOErrTR) and exit.

   If transporter result (SndRes+Rcode) is not 0, report a hardware error (DrvRet <- TOErrTR) and exit.

2. Check the Cancelled flag. If set, report an error and exit.

3. Check the Restart flag. If set, restart from the beginning.

4. End receive on socket 80h, in preparation for restart.

```
TCmd+OpCode  <- 10h (End receive command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- 80h

SndRes+Rcode <- FFh (initialize result code)
```

5. Send an Abort command.

```
TCmd+OpCode  <- 40h (Send command)
TCmd+ResAdr  <- address of SndRes
TCmd+Sock    <- 80h
TCmd+DatAdr  <- address of DCmd buffer
TCmd+DataLen <- 8
TCmd+CrtlLen <- 0
TCmd+Dest    <- DSNum

SndRes+Rcode <- FFh (initialize result code)

Dcmd+ProtoID <- 1FFh
Dcmd+MsgTyp  <- 0003h (Abort message)
Dcmd+RqstID  <- Request
Dcmd+Reason  <- 01h (Timedout)
```

If transporter result (SndRes+Rcode) does not change within 100ms, report an error (TOErrTR) and exit.

FLAT CABLE

You may want to refer to the following manuals while reading this section:

Chapter 1 of this manual, which describes the sector read and write commands.

Appendix A of this manual, which describes the flat cable interface bus.

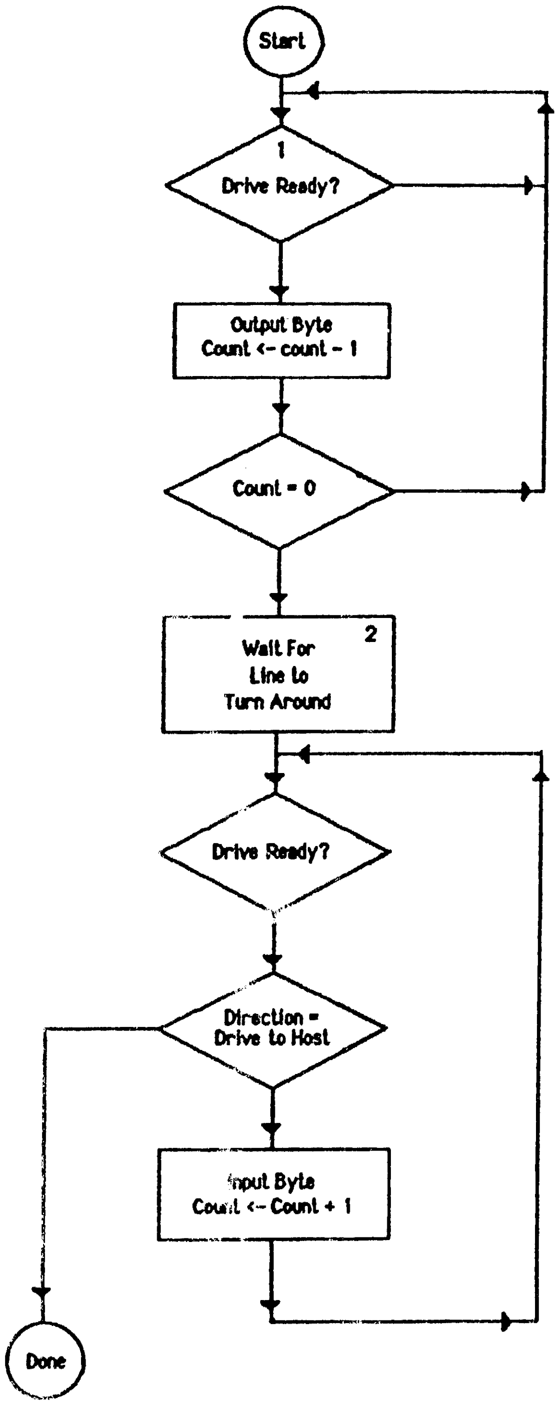

Figure 3.11
Flat cable command sequence

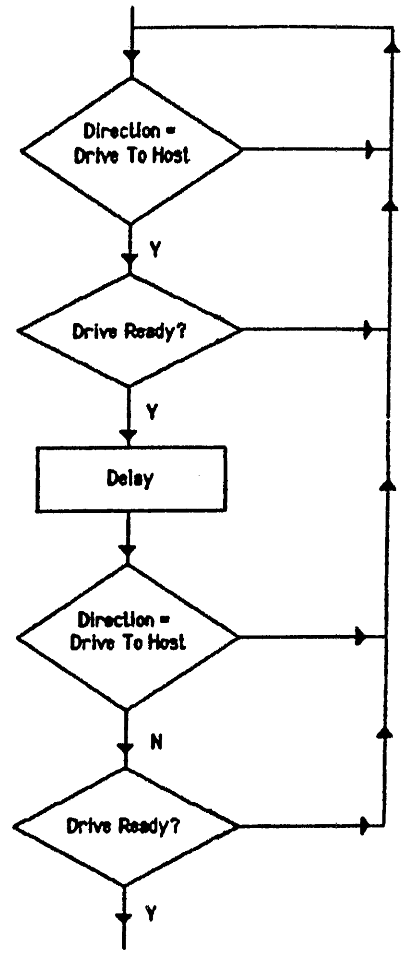

Figure 3.12
Flat cable turnaround routine

Refer to the interface signal descriptions at the end of Appendix A.

Disk read:

1. Send out read command (4 bytes). For each byte, check that drive is ready (READY line high), then output byte. See note below.

2. Wait for bus to turn around (READY line high and DIRC line low).

3. Receive results until drive stops sending. For each byte, wait for READY line to go high. Then check the DIRC line. If it is high, the drive has stopped sending; if it is low, read the data byte and increment the count of bytes received. In our example, we expect to receive 512 bytes; you should expect to receive the number of bytes specified by the read command (128, 256, 512, or 1024).

4. Check first byte received. If the most significant bit is on, an error occurred.

Disk write:

1. Send out write command. In our example, we send out 516 bytes. You should send out the appropriate number for the write command that you are using (132, 260, 516, or 1028). For each byte, check that drive is ready (READY line high), then output byte. See note below.

2. Wait for bus to turn around (READY line high and DIRC line low).

3. Receive results until drive stops sending. For each byte, wait for READY line to go high. Then check the DIRC line. If it is high, the drive has stopped sending; if it is low, read the data byte and increment the count of bytes received. In our example, we expect to receive 1 byte.

4. Check first byte received. If the most significant bit is on, an error occurred.

Note: Some care must be exercised in sending out at least the first byte of a command if a multiplexer is being used. There is a potential timing problem if the system software can be interrupted during the send of this first byte. On a multiplexer network, the individual computers must respond within approximately 50 microseconds after the READY line goes high, or the multiplexer will switch to the next slot. (It will first wait for a while after dropping the READY line -- a period controlled by the second polling parameter.) If your driver is interrupted after it detects that the READY line is high, and before it sends the first byte, then by the time it is ready to send the first byte, the multiplexer may have already switched to the next slot.

This problem can be avoided by turning off the interrupt system during part of the send loop to insure that if your driver finds the drive ready, it can send out the byte without being interrupted. See the sample 8086 driver in Appendix E for an example of this sequence.

SENDING OTHER DISK COMMANDS | 4

The Corvus mass storage devices support more operations than just read and write. Semaphores, pipes, mirror operations, etc., can all be invoked by application programs. This chapter discusses how these commands may be used by application programs.

This chapter merely describes how to send the command bytes and receive the results. The functionality of the commands is described in other chapters (Chapter 5: Semaphores, Chapter 6: Pipes).

The interface for sending a drive command generally consists of specifying the number of bytes to send, the maximum number of bytes expected to be received, and 2 buffers, one which contains the bytes to be sent and one which will contain the results.

```
    PROCEDURE SendCom( SendLen: INTEGER; VAR RecvLen: INTEGER;
                       VAR SendBuf, RecvBuf: Dbuf );
```

After a call to SendCom, RecvLen contains the number of bytes actually received, and RecvBuf contains the data.

For example, the code to send a semaphore lock command would look something like this (the semaphore name is 'S   '):

```
    TYPE Dbuf:  PACKED ARRAY [1..530] OF 0..255;

    VAR SendBuf, RecvBuf: Dbuf;
        SendLen, RecvLen: INTEGER;


    BEGIN

    SendLen := 10;   { semaphore lock sends 10 bytes }
    RecvLen := 530;  { the size of RecvBuf }
    SendBuf[1] := 11;  SendBuf[2] := 1;  { command code and subop }
    SendBuf[3] := ORD('S');              { semaphore name }
    SendBuf[4] := ORD(' ');
    ...
    SendBuf[10] := ORD(' ');

    SendCom( SendLen, RecvLen, SendBuf, RecvBuf);
```

```
{ now check resuls }
IF RecvBuf[1] > 127 THEN { disk error ... } ELSE
IF RecvBuf[2] = 0   THEN { semaphore successfully locked } ELSE
CASE RecvBuf[2] OF       { couldn't lock, report error }
  128: { already locked }
  253: { table full }
  254: { table read-write error }
  END;

...
END.
```

Corvus provides a version of the SendCom procedure for each operating system it supports. The next sections describe each implementation in detail. Often, there are several layers of interface, and the application developer can pick the level of interface desired. Generally, the highest level interface is the most flexible, but also the most costly in terms of execution time and memory space required.

Of course, you as a software developer may choose to ignore any software provided by Corvus, and develop your own interface which talks directly to the transporter or flat cable card. The flowcharts given in Chapter 3, "Disk Drivers," should be helpful in this case. If you do choose to develop your own interface, you must consider the impact on other software developers. As mentioned in the section on Omninet in Chapter 3, the receipt of unknown messages and the use of buffer space in buffered transporters must be considered.

The same example, a semaphore lock, is used in each description below, but the procedures described may be used tc send any disk command.

The implementation of the SendCom procedure takes one of two forms: 1) the SendCom procedure calls an entry point in the disk driver to do the actual send of the command, or 2) the SendCom procedure is a stand-alone procedure, which does not require the disk driver to be present.

The advantages and disadvantages of form 1, where the SendCom procedure calls the driver, are summarized below:

Advantages: the send-receive need only be coded once, and it becomes part of the operating system. Application programs then do not have to change when they are ported from one hardware environment to another.

Disadvantages: the application program cannot run unless the driver is installed. Drivers become part of the resident operating system, and therefore occupy memory, leaving less memory available to those applications which do not use the feature.

The advantages and disadvantages of form 2, where the SendCom procedure is a stand-alone procedure, are summarized below:

Advantages: the driver need not be installed, leaving more memory available to the application.

Disadvantages: each application which uses the interface must be relinked if the interface changes, either because of bugs or hardware changes.

Most of the early Corvus implementations, including Apple (R) Constellation I and CP/M 80 (TM), use form 2, a stand-alone procedure, to send drive commands. The later implementations, including MS(TM)-DOS Constellation II, use form 1.

In most of the Corvus implementations, the procedure SendCom is usually coded as two separate procedures: CDSEND and CDRECV (the reason for this is historical). A call to CDSEND must always be followed immediately by a call to CDRECV. Also, in most of the Corvus implementations, the SendBuf and RecvBuf are the same buffer; i.e., the results of a command overlay the command itself.

Corvus Concept operating system:

Direct communication with the Corvus drive is handled by the two procedures CDSEND and CDRECV. Any command described in Chapter 1 may be sent to the Corvus drive using these routines. These procedures are contained in the unit CCDRVIO, which is in the library C2LIB. C2LIB is included in the standard release of Concept software.

Please refer to the Pascal Library User Guide (Corvus P/N 7100-04978). You will need to look at Chapter 14, "Corvus Disk Interface Unit" (ccDRVIO).

CDSEND and CDRECV each have two parameters described by the following type declarations, which appear in the interface section of unit ccDrvio:

```
const SndRcvMax = 530;

type CDaddr   = RECORD
   SlotNo:    byte;        { slot number }
   Kind:      SlotTypes;   { OmninetDisk or LocalDisk (defined in CCDefn)
   NetNo:     byte;        { unused }
   Stationno: byte;        { Omninet server address }
   Driveno:   byte;        { drive number }
   BlkNo:     LONGINT;     { block number }
```

```
type SndRcvStr= RECORD
   sln:     INTEGER;   { length of command to be sent }
   rln:     INTEGER;   { maximum number of bytes to be returned }
   CASE INTEGER OF
     2: (c:   PACKED ARRAY [1..SndRcvMax] OF CHAR);
     1: (b:          ARRAY [1..SndRcvMax] OF byte);
   END;
```

Calls to these procedures occur in pairs. That is, a call to CDSEND is followed immediately by a call to CDRECV. The same variables are normally used for both calls.

The unit ccDRVIO must be initialized by calling the procedure ccDrvIoInit BEFORE calling any other procedures in the unit. ccDrvIoInit should only be called once, at the beginning of your program.

The following program fragment demonstrates a normal command sequence:

```
...
USES {CCLIB} CCDefn,
     {C2LIB} ccDrvio;

VAR  xcv: SndRcvStr;
     NetLoc: CDAddr;
     x:   INTEGER;

BEGIN
ccDrvIoInit;                        { initialize the unit }
InitSlot( NetLoc );                 { sets NetLoc to boot device }

xcv.sln := 10; xcv.rln := 530;
xcv.b[1] := 11;  xcv.b[2] := 1; { semaphore lock command }
xcv.c[3] := 'S'; xcv.c[4] := ' ';
...
xcv.c[10] := ' ';

CDSEND(NetLoc, xcv);
CDRECV(NetLoc, xcv);

IF xcv.b[1] < 0 THEN { report disk error } ELSE
IF xcv.b[2] = 0 THEN { semaphore successfully locked } ELSE
   BEGIN
   x := xcv.b[2];
   IF x < 0 THEN x := x+256;
   CASE x OF
      128:  { already locked }
      253:  { table full }
      254:  { error on table read-write }
      END;
   END;
```

The procedures CDSEND and CDRECV are found in the unit ccDrvio in the file C2LIB. This unit has several other procedures in it, so the unit is rather large. If space is a problem, you can interface directly to the SlotIO driver as described below.

Commands are sent using the UNITWRITE procedure. Results are received with the UNITREAD procedure. The parameters are described below:

```
UNITWRITE ( unitno,       { the SlotIO driver }
            buffer,       { the command to be sent }
            length,       { length of the command }
            0,            { not used }
            control );    { control contains the slot and
                          { server # where the command is
                          { to be sent; msb is server # and
                          { lsb is slot #.  server # is 0
                          { for slots 1 to 4 (local disk) }

UNITREAD  ( unitno,       { the SlotIO driver }
            buffer,       { where the results will be stored }
            length,       { maximum length to be received }
            0,            { not used }
            control );    { same as on UNITWRITE }
```

UNITWRITE and UNITREAD should always be used in pairs; i.e., a UNITWRITE should be followed immediately by a UNITREAD. The function IORESULT should be called following each call to UNITWRITE or UNITREAD to check for an error. The following errors may be returned:

Value	Meaning
0	no error
4	disk error (disk result > 7Fh)

The unit number to which the SlotIO driver is assigned may be obtained by calling the EXTERNAL procedure OSSltDv.

For instance, the following code fragment sends a semaphore lock command:

```
VAR c: PACKED ARRAY [1..530] OF CHAR; { the longest command
                                      { is 530 bytes }

FUNCTION OSSltDv: EXTERNAL;

BEGIN
...
c[1] := CHR(11);              { semaphore command }
c[2] := CHR(1);               { lock }
c[3] := 'S';                  { semaphore name }
...
```

```
    c[10] := ' ';
    UNITWRITE( OSSlotDv, c, 10, 0, $105); { send command to  }

    ior := IORESULT;
    IF ior = 0 THEN BEGIN
       UNITREAD( OSSlotDv, c, 530, 0, $105);   { get results }
       ior := IORESULT;
       END;
    IF ior=0 THEN {all ok} ELSE {report error};
    CASE ORD(c[2]) OF
        0:  { semaphore locked successfully }
      128:  { semaphore was already locked }
      253:  { semaphore table full }
      254:  { error reading-writing semaphore table }
      END;
    ...
```

MS-DOS 1.x, 2.x Constellation II:

For MS-DOS, direct communication with the Corvus drive is handled by the two procedures CDSEND and CDRECV. Any command described in the Chapter 1 may be sent to the Corvus drive using these routines.

The source and object files for the routines described here are available on diskette as part of the Software Developer's Kit for MS-DOS. See Appendix F for details. Appendix E contains a listing of the flat cable versions of the CDSEND and CDRECV routines.

The procedures CDSEND and CDRECV are written in machine language and are assembled using the Microsoft Assembler. Because there is no standard or dominant language for MS-DOS applications developers, we have chosen to give the examples here in the language used by Corvus for MS-DOS applications, MS Pascal. Unfortunately, each language uses a slightly different parameter passing mechanism. On the developer's diskette mentioned above, interfaces are provided for MS Pascal and compiled Basic. If you are using some other language, you will have to make the appropriate changes to the source for DRIVEC2.ASM and reassemble it.

The procedures CDSEND and CDRECV are contained in the module DRIVEC2.OBJ. The routines in this module must be initialized by calling the function INITIO BEFORE calling any other procedures in the module. INITIO should be called only once, at the beginning of your program.

CDSEND and CDRECV each have one parameter described by the following type declaration:

```
type Longstring= RECORD
   length:  INTEGER;
   CASE INTEGER OF
     { n should be equal to the length of the longest }
     { command you intend to send or receive          }
     1: (int: PACKED ARRAY [1..n] OF 0..255);
     2: (str: PACKED ARRAY [1..n] OF CHAR);
   END;
```

Calls to these procedures occur in pairs. That is, a call to CDSEND is followed immediately by a call to CDRECV. The same variable is normally used for both calls. The following program fragment demonstrates a normal command sequence:

```
...
PROCEDURE CDSEND(xcv:longstring);  EXTERN;
PROCEDURE CDRECV(xcv:longstring);  EXTERN;
FUNCTION  INITIO: INTEGER;         EXTERN;

VAR  xcv: longstring;

BEGIN

IF INITIO <> 0 THEN {error...};     { initialize the unit }

xcv.length := 10;
xcv.int[1] := 11;  xcv.int[2] := 1; { semaphore lock command }
xcv.str[3] := 'S';
xcv.str[4] := ' ';
...
xcv.str[10] := ' ';

CDSEND(xcv);
CDRECV(xcv);

IF xcv.int[1]>127 THEN { report disk error } ELSE
IF xcv.int[2]=0   THEN { semaphore successfully locked } ELSE
   BEGIN
   CASE xcv.int[2] OF
      128:  { already locked }
      253:  { table full }
      254:  { error on table read-write }
      END;
   END;
...
```

In a multiple server environment, the default server to be accessed is the boot server. If you wish to send a command to a server other than the boot server, you can so specify by calling the procedure SETSRVR. The declaration for this procedure is:

```
function SETSRVR( srvr: INTEGER ): INTEGER; EXTERNAL;
```

The following function call sets the server to server 3:

```
...
IF INITIO <> 0 THEN { error ... }
b :=  SETSRVR(3);
```

The function SETSRVR returns the boot server address, and ignores the parameter if it is greater than 255, or negative. Thus, you can also use this function to find out the boot server address:

```
...
IF INITIO <> 0 THEN { error... }
b := SETSRVR(-1);
{ now b contains the Omninet address of the boot server }
```

CP/M-80 and CP/M-86 Constellation II:

For CP/M-80 and CP/M-86 (TM), direct communication with the Corvus drive is handled by the two procedures SEND and RECV. Any command described in the Chapter 1 may be sent to the Corvus drive using these routines.

The source and object files for the routines described here are available on diskette as part of the Software Developer's Kit for Constellation II, CP/M-80 or CP/M-86. See Appendix F for details.

The procedures SEND and RECV are written in machine language and are assembled using the Digital Research assembler. Because there is no standard or dominant language for CP/M applications developers, we have chosen to give the examples here in the language used by Corvus for CP/M applications, Pascal MT+. Unfortunately, each language uses a slightly different parameter passing mechanism. On the developer's diskette mentioned above, an interface is provided for Pascal MT+. If you are using some other language, you will have to make the appropriate changes to the source for CPMIO.ASM or CPMIO86.A86 and reassemble it.

The procedures SEND and RECV are contained in the module CPMIO.ERL for CP/M-80 and in CPMIO86.R86 for CP/M-86. The routines in this module must be initialized by calling the function INITIO BEFORE calling any other procedures in the module. INITIO returns the address of the Corvus driver if it is successful, otherwise it returns 0. INITIO should be called only once, at the beginning of your program.

SEND and RECV each have one parameter described by the following type declaration:

```
type Longstring= RECORD
   length:  INTEGER;
   CASE INTEGER OF
```

```
    { n should be equal to the length of the longest }
    { command you intend to send or receive          }
    1: (int: PACKED ARRAY [1..n] OF 0..255);
    2: (str: PACKED ARRAY [1..n] OF CHAR);
  END;
```

Calls to these procedures occur in pairs. That is, a call to SEND is followed immediately by a call to RECV. The same variable is normally used for both calls. The following program fragment demonstrates a normal command sequence:

```
...
EXTERNAL PROCEDURE SEND(xcv:longstring);
EXTERNAL PROCEDURE CDRECV(xcv:longstring);
EXTERNAL FUNCTION  INITIO: INTEGER;

VAR  xcv: longstring;

BEGIN

IF INITIO = 0 THEN (error...);      { initialize the unit }

xcv.length := 10;
xcv.int[1] := 11;  xcv.int[2] := 1; { semaphore lock command }
xcv.str[3] := 'S';
xcv.str[4] := ' ';
...
xcv.str[10] := ' ';

SEND(xcv);
RECV(xcv);

IF xcv.int[1]>127 THEN { report disk error } ELSE
IF xcv.int[2]=0   THEN { semaphore successfully locked } ELSE
   BEGIN
   CASE xcv.int[2] OF
      128:  { already locked }
      253:  { table full }
      254:  { error on table read-write }
      END;
   END;
...
```

In a multiple server environment, the default server to be accessed is the boot server. If you wish to send a command to a server other than the boot server, you can so specify by calling the procedure SETSRVR. The declaration for this procedure is:

```
EXTERNAL function SETSRVR( srvr: INTEGER ): INTEGER;
```

The following function call sets the server to server 3:

```
...
IF INITIO = 0 THEN { error ... }
b :=  SETSRVR(3);
```

The function SETSRVR returns the boot server address and ignores the parameter, if the parameter is greater than 255, or negative. Thus, you can also use this function to find out the boot server address:

```
...
IF INITIO = 0 THEN { error... }
b := SETSRVR(-1);
{ now b contains the Omninet address of the boot server }
```

Apple DOS Constellation II:

Please read the section on Apple DOS Constellation I first. Constellation II is not supported on multiplexer networks. If you are using an Omninet network, you should assemble and use the code given below in place of OMNIBCI.OBJ, because the transporter RAM code is different for Constellation II than it was for Constellation I.

For Apple Constellation II, direct communication with the Corvus drive is handled by calling an entry point in the Corvus driver. The Corvus driver must have been previously loaded into the RAM on the transporter card; it is loaded by the boot process.

The driver is called by activating the slot containing the card, and then executing a JSR to location C80Bh. The next 8 bytes following the JSR instruction contain the parameters to the driver:

Bytes	Meaning
0 and 1	Address of command buffer.
2 and 3	Length of command.
4 and 5	Address of result buffer.
6 and 7	Maximum length of result.

Here is a listing of OMNIBCI.OBJ for Constellation II:

```
        .ABSOLUTE
        .PROC OMNIBCI

LEN     .EQU 0300
BUF     .EQU 0302

START   .ORG 8A00

        LDA  LEN            ; move command length
        STA  CmdLen
```

```
        LDA   LEN+1
        STA   CmdLen+1
        LDA   BUF             ; move command address
        STA   CmdBuf
        STA   RsltBuf         ; make result address same as command
        LDA   BUF+1           ;    address
        STA   CmdBuf+1
        STA   RsltBuf+1
        LDY   #28.            ; make result length = 530
        STY   RsltLen
        LDY   #2
        STY   RsltLen+1

        JSR   GoRAM           ; RAM code will return to next instruction

        LDA   RsltLen         ; return result length
        STA   LEN
        LDA   RsltLen+1
        STA   LEN+1
        RTS                   ; return to caller

GoRAM   BIT   0CFFF           ; enable Omninet RAM
        BIT   0C600           ; assumes slot 6
        JSR   0C80B           ; no return necessary

CmdBuf .WORD  0               ; address of command
CmdLen .WORD  0               ; length of command
RsltBuf.WORD  0               ; address of result
RsltLen.WORD  0               ; maximum length of result

        .END
```

If you use this version of OMNIBCI.OBJ, your programs that were coded using the OMNIBCI.OBJ provided by Corvus for Constellation I need not be modified for Constellation II.

Version IV p-system and Apple Pascal Constellation II:

Direct communication with the Corvus drive is handled by the two procedures CDSEND and CDRECV. Any command described in Chapter 1 may be sent to the Corvus drive using these routines. These procedures are contained in the file CORVUS.LIBRARY, which is part of the Software Developer's Kit available for Version IV p-system and Apple Pascal 1.2. See Appendix F for details.

CDSEND and CDRECV are contained in unit UCDRVIO.

CDSEND and CDRECV each have two parameters described by the following type declarations (these declarations appear in the interface section of unit UCDrvio):

```
const SndRcvMax = 530;
```

```
type CDaddr   = RECORD
   SlotNo:  byte;       { slot number }
   Kind:    SlotTypes;  { OmninetDisk or LocalDisk (defined in CCDefn)
   NetNo:   byte;       { unused }
   Stationno: byte;     { Omninet server address }
   Driveno: byte;       { drive number }
   BlkNo:   LONGINT;    { block number }

type SndRcvStr= RECORD
   sln:     INTEGER;    { length of command to be sent }
   rln:     INTEGER;    { maximum number of bytes to be returned }
   CASE INTEGER OF
     2: (c:   PACKED ARRAY [1..SndRcvMax] OF CHAR);
     1: (b:   PACKED ARRAY [1..SndRcvMax] OF byte);
   END;
```

Calls to these procedures occur in pairs. That is, a call to CDSEND is followed immediately by a call to CDRECV. The same variables are normally used for both calls.

The unit UCDRVIO must be initialized by calling the procedure ccDrvIoInit BEFORE calling any other procedures in the unit. ccDrvIoInit should only be called once, at the beginning of your program.

The following program fragment demonstrates a normal command sequence:

```
...
USES {CORVUS.LIBRARY} UCDefn, UCDRVIO;

VAR  xcv: SndRcvStr;
     NetLoc: CDAddr;
     x:   INTEGER;

BEGIN
ccDrvIoInit;                      { initialize the unit }
InitSlot( NetLoc );               { sets NetLoc to boot device }

xcv.sln := 10; xcv.rln := 530;
xcv.b[1] := 11;  xcv.b[2] := 1; { semaphore lock command }
xcv.c[3] := 'S'; xcv.c[4] := ' ';
...
xcv.c[10] := ' ';

CDSEND(NetLoc, xcv);
CDRECV(NetLoc, xcv);

IF xcv.b[1] > 127 THEN { report disk error } ELSE
IF xcv.b[2] = 0 THEN { semaphore successfully locked } ELSE
   BEGIN
   x := xcv.b[2];
```

```
        CASE x OF
           128:  { already locked }
           253:  { table full }
           254:  { error on table read-write }
           END;
        END;
     ...
```

The procedures CDSEND and CDRECV are found in the unit UCDrvio in the file CORVUS.LIBRARY. This unit has several other procedures in it, so the unit is rather large. If space is a problem, you can interface directly to the machine language routines contained in the module DRVSTF.CODE. The routines are:

```
PROCEDURE drvSend(VAR s:sndRcvStr); EXTERNAL
PROCEDURE drvRecv(VAR s:sndRcvStr); EXTERNAL
    Uses PASCAL global variable DISK_SERVER

FUNCTION OSactSlt:INTEGER; EXTERNAL
    Returns 1 if we have booted up under CONSTELLATION II,
    0 if we have not.

FUNCTION OSSltType(slot : INTEGER) : INTEGER;  EXTERNAL;
    For valid slots, return the interface card type,
    1=flat cable 2=Omninet; for all other slots
    returns 0=no disk

FUNCTION OSactSrv : INTEGER;
    Return the active disk server. This procedure assumes
    that the driver is attached and we have booted up under
    CONSTELLATION II. No checking is done

FUNCTION XPORTER_OK : BOOLEAN;
    Returns true if transporter is ok, false if transporter
    with duplicate address is on the network. Returns true
    if flatCable interface is present.

FUNCTION FIND_ANY_SERVER(VAR server : INTEGER): BOOLEAN;
    Returns true if any disk server is found on the network,
    and sets the variable server to the address of the disk
    server.  Returns false if no disk server replys.
    Returns true with a server of zero if the interface card
    is flat cable.
```

Commands are sent using the drvSend procedure. Results are received with the drvRecv procedure.

Two global variables must also be declared: **active_slot** and **disk_server**. These must be set prior to calling **drv_send**.

For instance, the following code fragment sends a semaphore lock command:

```
VAR active_slot:  INTEGER;
    disk_server:  INTEGER;
    omni_error:   INTEGER;

    xcv: SndRcvStr;

BEGIN
active_slot := OSactSlt;  Disk_server := OSActSrv;
...
xcv.sln := 10; xcv.rln := 530;
xcv.b[1] := 11;  xcv.b[2] := 1; { semaphore lock command }
xcv.c[3] := 'S'; xcv.c[4] := ' ';
...
xcv.c[10] := ' ';

drv_send(xcv);
drv_recv(xcv);

IF xcv.b[1] > 127 THEN { report disk error } ELSE
IF xcv.b[2] = 0 THEN { semaphore successfully locked } ELSE
   BEGIN
   x := xcv.b[2];
   CASE x OF
      128:  { already locked }
      253:  { table full }
      254:  { error on table read-write }
      END;
   END;
...
```

Apple Pascal Constellation I:

In Pascal, direct communication with the Corvus drive is handled by the two procedures ~~-~~CDSEND and CDRECV. Any command described in Chapter 1 may be sent to the Corvus drive using these routines.

These procedures are contained in the unit Driveio of CORVUS.LIBRARY. This unit must be initialized by calling the procedure Driveioinit BEFORE calling any other procedures in the unit.
Driveioinit should only be called once, at the beginning of your program.

CDSEND and CDRECV each have one parameter described by the following type declaration (which appears in the interface section of Driveio):

```
type LONGSTR= RECORD
   length:  INTEGER;
   CASE INTEGER OF
     { n should be equal to the length of the longest }
```

```
        { command you intend to send or receive         }
        1: (int: PACKED ARRAY [1..n] OF 0..255);
        2: (byt: PACKED ARRAY [1..n] OF CHAR);
      END;
```

Calls to these procedures occur in pairs. That is, a call to CDSEND is followed immediately by a call to CDRECV. The same variable is normally used for both calls. The following program fragment demonstrates a normal command sequence:

```
...
USES Driveio;

VAR  xcv: LONGSTR;

BEGIN

Driveioinit;                         { initialize the unit }

xcv.length := 10;
xcv.int[1] := 11;  xcv.int[2] := 1; { semaphore lock command }
xcv.byt[3] := 'S';
xcv.byt[4] := ' ';
...
xcv.byt[10] := ' ';

CDSEND(xcv);
CDRECV(xcv);

IF xcv.int[1]>127 THEN { report disk error } ELSE
IF xcv.int[2]=0   THEN { semaphore successfully locked } ELSE
   BEGIN
   CASE xcv.int[2] OF
      128:  { already locked }
      253:  { table full }
      254:  { error on table read-write }
      END;
   END;
...
```

The procedures CDSEND and CDRECV are found in the unit DRIVEIO in the file CORVUS.LIBRARY. These procedures are independent of whether you are using flat cable or Omninet. The price you pay for this independence is that the unit DRIVEIO is fairly large. You can interface directly to the assembly language drivers for flat cable or Omninet with the routines in the unit OMNISEND, also in the file CORVUS.LIBRARY. The interface to these assembly language routines is described next.

Use drv_send and drv_recv for flat cable interface. **Active_slot** must be a global variable.

Use omni_send and omni_recv for Omninet interface. Prior to the first use of these routines in a program, you should use the code shown below to get the disk server address, unless you make the assumption that the disk server has a fixed address. **Disk_server** and **active_slot** must be global variables.

In either case, the Corvus interface card may be used in any slot. The variable active_slot is set to the slot number that the card is plugged into. But remember that the interface card must be in slot 6 for normal operation.

```
CONST
  longstr_max = 1030;
  broadcast_add = 255;

TYPE
  byte = 0..255;
  LONGSTR= RECORD
    length:  INTEGER;
    CASE INTEGER OF
      { n should be equal to the length of the longest }
      { command you intend to send or receive          }
      1: (int: PACKED ARRAY [1..n] OF byte);
      2: (byt: PACKED ARRAY [1..n] OF CHAR);
    END;

  valid_slot = 1..7;

VAR
  active_slot : valid_slot; (* used by assembler routines to
                               determine io location *)
  disk_server : byte;        (* used by assembler routines *)
  omni_error  : integer;     (* used by asm - returns timeout status *)


PROCEDURE drv_send(VAR st : longstr); EXTERNAL;
PROCEDURE drv_recv(VAR st : longstr); EXTERNAL;
PROCEDURE omni_send(VAR st : longstr); EXTERNAL;
PROCEDURE omni_recv(VAR st); EXTERNAL;
(* did not specify type so init portion could send a dummy *)
```

The following initialization is required for omni_send and omni_recv:

```
   disk_server := broadcast_add;
   omnirecv(dummy); (* looks for disk server *)
   IF disk_server = broadcast_add THEN  (* omnirecv sets disk_server *)
      error;
```

Apple DOS Constellation I:

Corvus provides two assembly language procedures (BCI.OBJ and OMNIBCI.OBJ) for sending arbitrary disk commands. BCI.OBJ is for multiplexer networks, and OMNIBCI.OBJ is for Omninet networks.

Each routine is a binary file which must be BLOADed into memory before being called. BCI.OBJ must be loaded at location 300h, while OMNIBCI.OBJ must be loaded at location 8A00h. Neither routine is relocatable. BCI.OBJ ends at location 386h, while OMNIBCI.OBJ ends at location 9044h. OMNIBCI.OBJ is much longer because it includes buffer space for Omninet messages.

A drive command is poked into memory, and the address and length of the command are passed to BCI (or OMNIBCI) by poking the address into location 302h and 303h, and poking the length of the command into locations 300h and 301h. BCI (or OMNIBCI) is then CALLed. Upon return, the length of the result can be peeked from location 300h and 301h, and the result itself has been written into the space pointed-to by the address parameter.

See the DIAGNOSTIC program, lines 10000-10007 for an example of how to load BCI (or OMNIBCI). See lines 15000-15110 for an example of how to call BCI (or OMNIBCI).

BCI does not use the ROM on the Corvus interface card. OMNIBCI does use the RAM on the transporter card. This RAM is loaded from a reserved area on the Corvus drive at boot time. If you want to use OMNIBCI without booting from the Corvus drive, you must execute the code that loads the RAM. See the BSYSGEN program, lines 20000-20060 for an example of how to initialize OMNIBCI.

A listing of BCI.OBJ is included in appendix E.

CP/M 80 Constellation I:

You may order the Software Developer's Kit for your particular machine for examples of how to send commands using the flat cable interface. Version available are listed in Appendix F.

This page intentionally left blank.

SEMAPHORES | 5

This chapter gives examples of how the semaphores feature of the Corvus mass storage systems may be used.

Semaphores can be used to control access to any shared resource on the network. Most often, semaphores are used to coordinate access to shared files. You should understand that semaphores merely provide the capability to access shared files; it is you who must ensure that your programs use this capability.

Programs written for single-user access may not be used to access shared files; they must be modified to include semaphore calls.

User libraries that implement semaphore calls are supplied with most of the versions of Corvus utilities. A typical interface consists of two function calls, each with one parameter specifying the name of the semaphore to be accessed:

```
function LOCK ( SEMA4: string ): integer;

function UNLOCK ( SEMA4: string ): integer;
```

Each function returns a value which indicates the result of the operation. The values are as follows:

0	Semaphore was not previously locked. For LOCK, this means that the semaphore has now been locked successfully.
128	Semaphore was previously locked. For LOCK, this means that the semaphore could not be locked by this call. For UNLOCK, this means that the semaphore is now unlocked.
< 0	Some error occurred, and the semaphore could not be locked. Specifically, the values returned are
-253	Semaphore table is full.
-254	Error reading/writing semaphore table.
-255	Unknown error.

Thus, a successful LOCK call returns a value of 0. A successful UNLOCK call returns 0 or 128.

As mentioned above, semaphores can be used to control access to any shared resource on the network. Let's look in detail at two common uses for semaphores: shared volumes and shared files.

Volume sharing implies that several users will be modifying different files in the same volume. To coordinate such access, some sort of volume locking scheme must be used. File sharing implies that several users will be modifying a particular file. This access requires a file locking scheme.

VOLUME SHARING

The problems associated with volume sharing include directory update and dynamic file allocation. Both of these problems can be solved by the volume locking scheme described below. First, let's look at what happens if you try to do volume sharing without some sort of locking scheme.

Most systems keep a copy of the directory in memory. Whenever a new file is opened, an entry is made in the memory copy of the directory, but this copy is not necessarily written to disk right away. Thus, if two users open two different files at approximately the same time, the memory copies of the directory will differ. Eventually, both copies will be written back to disk, and one user will lose the file just opened.

Systems which use dynamic file allocation, such as MS-DOS and CP/M, keep a memory image of the disk space allocated. Whenever a new file is opened, or a new record is written past the current end of file, the file system searches its file allocation table for free space on the disk. Enough free space is allocated to the file to contain up to and including the new record, and a new end of file mark is written. The file allocation table is written back to the disk only when absolutely necessary, in order to minimize disk I/O.

Let's look at what happens when two users are creating files on the same volume at the same time. Each user has a current copy of the file allocation table in memory; the operating system searches the memory copy of the file allocation table for free space, and allocates the same disk blocks to two different files. Everytime one user updates the data in that disk block, the data for the other user is destroyed. This can result in many confusing error messages and incomprehensible data.

Many application writers, for this reason, preallocate any files their application requires. This operation consists of opening a file, writing to the last record, and then flushing the allocation map. Then the application does not have to worry about further allocation, until the file fills up. Most data bases are preallocated anyway, as this makes it easier for the application to manage the data base.

VOLUME LOCKING

Unlike some other network systems, Corvus software does not define a volume type of shared access. Instead, Corvus software defines volume access in terms of read-write access or read-only access. If more than one user has read-write access to the same volume, then that volume is a shared volume, and access to it must be protected by using semaphores.

When two users wish to access the same volume, they must coordinate that access in some way. One way to do this is with volume locking. In the scheme described here, it is assumed that each user has the volume in question mounted with read-only access.

Users must indicate when they are ready to write to the volume by executing a LOCK program, and specifying the name of the volume to be locked. The LOCK program will ensure that no other user currently has write access to the volume, and then grant the user write access.

How does the program know if any user currently has write access to the volume in question? This example assumes that if a certain file, called LOCKED, exists in the volume, then the volume is currently locked by some user. Furthermore, the name of the user who locked the volume is contained in the file LOCKED.

The steps the LOCK program must take are listed below:

1) Try to open the file LOCKED. If found, report that the volume is currently locked, and exit.

2) Change the user's access to read-write. This change is done in memory, so that it is temporary.

3) Create a file called LOCKED in the volume, and write the user's name into it.

Thus, if a user executes the LOCK program after the volume is locked, the user receives an error message saying that the volume is already locked. Let's look at what happens, however, if the volume is not locked, and two users happen to execute the LOCK program at the same time.

```
User 1                      User 2
------                      ------
open file LOCKED            open file LOCKED

not found, so change        not found, so change
  access to read-write        access to read-write

create file LOCKED,         create file LOCKED,
  write user name             write user name
```

As you can see, both users think that the volume has been successfully locked, and both have write access to the volume. This is NOT supposed to happen. While the likelihood of two users executing the program at the same time is small, it still has to be prevented. The only way to prevent it is to use semaphores.

The reason that both users were able to lock the volume is that, on a Corvus network, computers have no way to do a read followed immediately by a write. The computer may send the write command immediately after the read, but some other computer may be serviced in between the two operations. The semaphore operation is the only way to do an indivisible write after read operation.

In our example, a semaphore called VOLLOCK is used to synchronize access between the two users. The steps the LOCK program must do are expanded to the following:

1) Lock the semaphore VOLLOCK. If it can't be locked, wait in a loop, and try again.

2) Try to open the file LOCKED. If found, report that the volume is currently locked, unlock the semaphore, and exit.

3) Change the user's access to read-write. This change is done in memory, so that it is temporary.

4) Create a file called LOCKED in the volume, and write the user's name into it. Flush file buffers and

5) Unlock the semaphore VOLLOCK.

Now let's look at what happens when two users execute the LOCK program at the same time.

User 1	User 2
Lock semaphore VOLLOCK	Lock semaphore VOLLOCK
Semaphore successfully locked.	Semaphore already locked, wait in loop.
Open file LOCKED	semaphore still locked...
Not found, so change access to read-write	semaphore still locked...
Create file LOCKED, write user name	semaphore still locked...
Unlock semaphore	Semaphore successfully locked.
	Open file LOCKED.
	Found, so cannot lock volume. Print message, unlock semaphore and exit.

As you can see, only one user is able to lock the volume at any one time.

There are still some problems with the algorithm given above. On file systems which do directory buffering, the program must force the directory to be flushed to the disk after creating the file. Some hints for this are given in the specific operating system sections below. Also, an UNLOCK program must be provided so that a user can release access to a volume. This program must perform the following steps:

1) Delete the file LOCKED.

2) Change the user's access to read only.

Again, in certain file systems, the directory must be flushed after deleting the file. In this case, no semaphore is locked, because, in order to delete the file, the user must already have write access to the volume.

Other problems include a user forgetting to unlock a volume before powering off. Now no one can write to the volume, since it is locked and no one has write access to it. This problem can be gotten round in part by making the LOCK program a little smarter: if the user executing the LOCK program has the same name as the user name in the file LOCKED, then grant the user read-write access.

Note that the same semaphore name, VOLLOCK, is used, regardless of which volume is being locked. Thus, if two users attempt to lock different volumes at the same time, one user finds that the semaphore is locked. This is generally not a problem, since the length of time that the semaphore is locked should be very short; the second user should notice only a slight delay before the program completes. Of course, the LOCK program could use the name of the volume to be locked as the semaphore name.

In fact, the LOCK program could be made much simpler if the following algorithm were used:

1) Lock a semaphore with the same name as the volume. If the semaphore cannot be locked, report error and exit.

2) Change user access to read-write.

The UNLOCK program has only 2 steps as well:

1) Change user access to read only.

2) Unlock the semaphore with the same name as the volume.

While this algorithm avoids the directory buffering problem mentioned above, there are two disadvantages to it:

1) There is no way to tell who has the volume locked.

2) Since the semaphore may be locked for an extended period of time, a network with many users could fill up the semaphore table.

FILE OR RECORD LOCKING

File or record locking is complicated by the file buffering schemes used by most operating systems.

Most file systems have one or more file buffers. These buffers are used to minimize disk overhead by keeping the most recently accessed file blocks in memory. When the operating system receives a file read or write call, it first checks its buffers to see if the specified file block is already in memory; if it is, then the I/O is done to the memory image, rather than to the disk. The buffer is flushed to the disk only when necessary, usually when the buffer must be used for some other I/O operation. Depending on the number and size of the buffers, it may be quite a while before a file write is actually transferred to the disk itself. Most operating systems provide a system call that forces all buffers to be flushed to the disk.

Thus a write to a file does not actually get recorded on the disk until some later time. In a network environment, this can mean disaster for shared data bases, where many users are attempting to read or write to a common file. Shared file applications must therefore be coded very carefully; you must completely understand the file buffering characteristics of the file system you are using. The following description of record locking assumes that you do understand your system's file buffering.

Basically, you must lock a semaphore on filling a file buffer, and unlock the semaphore after the buffer has been flushed. Thus the steps in updating a record are as follows:

1. Lock the semaphore.

2. Read the record (fill the file buffer)

3. Modify the data.

4. Flush the file buffer.

5. Unlock the semaphore.

The semaphore name associated with a given record must be specified by your program. Your program must ensure that each record that resides in the same disk block is assigned the same semaphore name. For example, let's assume that your application is called ZXY, and it deals with a file structure that has 32 records per disk block (that is, each file buffer can hold 32 of your application's records). A good algorithm for assigning semaphore names is shown below:

1. Compute record number DIV 32.

2. Embed this ASCII representation of this number in the string ZXY00000.

For record 50, your application should lock semaphore ZXY00001.
For record 600, your application should lock semaphore ZXY00018.

Using this algorithm, each record which falls within the same file buffer is assigned the same semaphore name. Let's look at what happens when two users execute the program at the same time:

User 1	User 2
Update record 50:	Update record 52:
Lock semaphore ZXY00001.	Lock semaphore ZXY00001.
Semaphore successfully locked.	Semaphore already locked, wait in loop...
Read record 50.	Semaphore still locked...
Make changes.	Semaphore still locked...
Flush file buffer to disk.	Semaphore still locked...
Unlock semaphore ZXY00001.	Semaphore successfully locked.
	Read record 52.
	Make changes.
	Flush file buffer to disk.
	Unlock semaphore ZXY00001.

Note that using this algorithm causes your program to use many more than the 32 semaphore names provided by Corvus semaphores. However, only a few semaphores will be locked at any one time, so chances are you will never fill up the semaphore table. If you are worried about this problem, you can set up your own semaphore table, with semaphore names as long as you wish and with as many semaphores as you wish. This table could reside in a file or in a reserved disk block. Access to this user semaphore table can be controlled with one Corvus semaphore in the following manner:

1. Lock the Corvus semaphore SEMTAB.
2. Search the user semaphore table for the specified semaphore name. If there, return the appropriate error. If not there, add the semaphore and return the appropriate return code.
3. Unlock the Corvus semaphore SEMTAB.

In the above discussion, we have tried to highlight some of the problems involved in resource sharing, and how these problems can be solved by proper use of semaphores. The next sections describe the library routines provided for each operating system supported by Corvus.

Corvus Concept Operating System:

Please refer to the Pascal Library User Guide (Corvus P/N 7100-04978). You need to look at Chapter 14, "Corvus Disk Interface Unit" (ccDRVIO), and Chapter 16, "Corvus Disk Semaphores Interface Unit" (ccSEMA4).

Note that the procedure CCSEMA4INIT must be called prior to calling any of the other procedures or functions in the ccSEMA4 unit. The parameter **NetLoc** specifies which server will be used for semaphore operations. Specifically, the following fields of **Netloc** must be defined before calling CCSEMA4INIT:

Netloc.slotno	slot number
Netloc.stationno	server number (ignored for MUX)
Netloc.Kind	either OmninetDisk or LocalDisk

Here is a portion of a LOCK program for Concept Pascal:

```
PROGRAM LOCK;
USES {CCLIB}  CCDEFN,
     {C2LIB}  CCDRVIO, CCSEMA4;

VAR s: Semkey;
    NetAddr: CDAddr;    { CDAddr is declared in ccDrvio }
    i, err:  INTEGER;

BEGIN
ccDrvioInit;           { initialize unit ccDRVIO }

Initslot(NetAddr);     { this procedure, from ccDrvio,
                       { initializes slotno, stationno, and kind
                       { fields to boot device.  Sets driveno
                       { to 1, all other fields to 0 }

ccSema4Init(NetAddr);   { initialize unit ccSEMA4 }

...                    { get volume name to be locked }

s := 'VOLLOCK';
i := 0;
REPEAT
  i := i+1;
  err := SemLock(s);
UNTIL (err <> SemWasSet)  { wait for semaphore to be not set }
   OR (i > 32000);        { or timeout }

IF err <> SemNotSet THEN ... { report error and exit program }

...                    { lock volume }
{ closing the file causes the directory on disk to be updated }
```

```
err := SemUnlock(s);      { don't forget to unlock semaphore }

END.
```

Version IV p-system and Apple Pascal Constellation II:

Look at the interface sections for the following units:

UCDEFN, UCDRVIO, and UCSEMA4.

These units are found in library CORVUS.LIBRARY.

Note that the procedure CCSEMA4INIT must be called prior to calling any of the other procedures or functions in the UCSEMA4 unit. The parameter **Netloc** specifies which server will be used for semaphore operations. Specifically, the following fields of **Netloc** must be defined before calling CCSEMA4INIT:

```
Netloc.slotno       slot number
Netloc.stationno    server number (ignored for MUX)
Netloc.Kind         either OmninetDisk or LocalDisk
```

Here is a portion of a LOCK program:

```
PROGRAM LOCK;
USES (CORVUS.LIBRARY)  UCDEFN, UCDRVIO, UCSEMA4;

VAR s: Semkey;
    NetAddr: CDAddr;   { CDAddr is declared in ccDrvio }
    i, err:  INTEGER;

BEGIN
ccDrvioInit;          { initialize unit ccDRVIO }

Initslot(NetAddr);    { this procedure, from ccDrvio,
                      { initializes slotno, stationno, and kind
                      { fields to boot device.  Sets driveno
                      { to 1, all other fields to 0 }

ccSema4Init(NetAddr);  { initialize unit ccSEMA4 }

...                   { get volume name to be locked }

s := 'VOLLOCK';
i := 0;
REPEAT
  i := i+1;
  err := SemLock(s);
UNTIL (err <> SemWasSet)  { wait for semaphore to be not set }
   OR (i > 5000);         { or timeout }

IF err <> SemNotSet THEN ... { report error and exit program }
```

```
...                   { lock volume }
{ closing the file causes the directory on disk to be updated }

err := SemUnlock(s);      { don't forget to unlock semaphore }

END.
```

MS-DOS 1.x and 2.x Constellation II:

The MS-DOS file system uses both file buffering and dynamic file allocation. Refer to the DOS manual for information on managing file buffers and file allocation tables.

The machine language interface described in Chapter 4 may be used to send semaphore commands. The Software Developer's Kit contains examples of using semaphores with MS Pascal and compiled Basic.

A new set of routines provides direct semaphore calls. These routines are written in machine language and are assembled using the Microsoft Assembler. Interfacing to these routines from a high level language may require changing the routines slightly. This change is required because there is no standard parameter passing mechanism in MS-DOS.

The routine declarations are as follows:

```
FUNCTION SemLock( VAR Name: STRING ): INTEGER; EXTERN;
FUNCTION SemUnLock( VAR Name: STRING): INTEGER; EXTERN;
FUNCTION SemStatus( VAR Name: STRING): INTEGER; EXTERN;
```

These routines are found in the file SEMAASM.OBJ. You must also use the INITIO and SETSRVR procedures from DRIVEC2.OBJ.

Here is a portion of a LOCK program:

```
PROGRAM Lock (INPUT,OUTPUT);

CONST  SemWasSet = 128;
       SemNotSet = 0;

VAR s: LSTRING(80);
    err, i: INTEGER;

FUNCTION SemLock( VAR Name: STRING ): INTEGER; EXTERN;
FUNCTION SemUnLock( VAR Name: STRING): INTEGER; EXTERN;
FUNCTION InitIO: INTEGER; EXTERN;

BEGIN
IF INITIO <> 0 THEN { error... }
```

```
...    { get volume name to be locked }

s := 'VOLLOCK';
i := 0;
REPEAT
  i := i+1;
  err := SemLock(s);
UNTIL (err <> SemWasSet)    { wait for semaphore to be not set }
  OR  (i > 32000 );         { or timeout }

IF err <> SemNotSet THEN ... { report error and exit program }

...                 { lock volume }
{ flush directory to disk }

err := SemUnlock(s);      { don't forget to unlock semaphore }

END.
```

CP/M-80 and CP/M-86 Constellation II:

The machine language interface described in Chapter 4 must be used to send semaphore commands. The Software Developer's Kit contains examples of using semaphores with Pascal MT+.

Apple Pascal Constellation I:

Look at the interface sections for the following units:

DRIVEIO and SEMA4S.

These units are found in library CORVUS.LIBRARY.

Note that the procedure SEMA4INIT must be called prior to calling any of the other procedures or functions in the SEMA4S unit. The parameter is a BOOLEAN which should be set to FALSE. A TRUE value results in some debugging statements being printed.

Here is a portion of a LOCK program:

```
PROGRAM LOCK;
USES {CORVUS.LIBRARY}  DRIVEIO, SEMA4S;

VAR s: Semkey;
    i, err:  INTEGER;

BEGIN
DriveioInit;          { initialize unit Driveio }

Sema4Init(FALSE);     { initialize unit SEMA4S }
```

```
...                     { get volume name to be locked }

s := 'VOLLOCK';
i := 0;
REPEAT
  i := i+1;
  err := SemLock(s);
UNTIL (err <> SemWasSet)  { wait for semaphore to be not set }
   OR (i > 5000);         { or timeout }

IF err <> SemNotSet THEN ... { report error and exit program }

...                     { lock volume }
{ closing the file causes the directory on disk to be updated }

err := SemUnlock(s);      { don't forget to unlock semaphore }

END.
```

If you have limited memory available, you may wish to write your own semaphore routines. See Chapter 4 for information on interfacing directly to unit DriveIO.

Refer to the Apple Pascal Operating System Reference manual for information on file buffering and allocation.

Apple DOS Constellation I/II:

Corvus provides two assembly language procedures (BCI.OBJ and OMNIBCI.OBJ) for sending arbitrary disk commands. BCI.OBJ is for multiplexer networks, and OMNIBCI.OBJ is for Omninet networks.

The program SHARE on the distribution floppy for Constellation I shows how to send semaphore commands using these routines.

Refer to the Apple DOS manual for information on file buffering and allocation.

This page intentionally left blank.

USING PIPES | 6

This chapter gives two examples of how the pipes features of the Corvus mass storage systems may be used. The first example is a spooling program; the second shows how messages can be exchanged using pipes. The features of the Corvus-supplied Spool program are also described.

User libraries that implement pipes calls are supplied with several of the versions of Corvus utilities. A typical interface consists of 9 functions. These are summarized below:

Function	Description
PipeStatus	Get status of pipes area
PipeOpRd	Open pipe for reading
PipeOpWr	Open pipe for writing
PipeRead	Read data from pipe
PipeWrite	Write data to pipe
PipeClRd	Close pipe for reading
PipeClWr	Close pipe for writing
PipePurge	Purge pipe
PipesInit	Initialize pipes area on disk

Sample declarations of each function are listed below.

The DrvBlk data type used in these declarations is

```
TYPE DrvBlk = PACKED ARRAY 0..511 OF 0..255;
```

The negative error codes referred to in the declarations are listed here:

Value	Meaning
-8	Tried to read an empty pipe
-9	Pipe not opened
-10	Tried to write to a full pipe
-11	Pipe open error
-12	Pipe does not exist
-13	No room to open new pipe
-14	Invalid pipes command
-15	Pipes area not initialized
< -127	Disk error

PipeStatus Function

PipesStatus uses the Pipe Status command to read the Pipe Name table and the Pipe Pointer table. The definition of the function is as follows:

```
FUNCTION PipeStatus( VAR Names, Ptrs: DrvBlk ): INTEGER;
```

Parameter	Data Type	Description
Names	DrvBlk	Pipe Name Table
Ptrs	DrvBlk	Pipe Pointer Table

This function returns 0 if ok; a negative result indicates a pipe error.

PipeOpRd function

PipeOpRd uses the Pipe Open for Read command to open a pipe for reading. The definition of this function is as follows:

```
FUNCTION PipeOpRd( PName: PNameStr ): INTEGER;
```

Parameter	Data Type	Description
PName	PNameStr	Name of pipe to open

This function returns the pipe number if the specified pipe exists, and can be opened. Otherwise, a negative error code is returned.

PipeOpWr function

PipeOpWr uses the Pipe Open for Write command to open a pipe for writing. The definition of this function is as follows:

```
FUNCTION PipeOpWr( PName: PNameStr ): INTEGER;
```

Parameter	Data Type	Description
PName	PNameStr	Name of pipe to open

This function returns the pipe number if the pipe was successfully opened. Otherwise, a negative error code is returned.

PipeRead function --

PipeRead uses the Pipe Read command to read a block of data from the specified pipe. The definition of this function is as follows:

```
FUNCTION PipeRead( PNum: INTEGER; VAR Info: Drvlk ): INTEGER;
```

Parameter	Data Type	Description
Pnum	INTEGER	Pipe number
Info	DrvBlk	Data read from pipe

This function returns the number of bytes read if the read is successful. Otherwise, a negative error code is returned. The number of bytes read should always be 512.

PipeWrite function ---

PipeWrite uses the Pipe Write command to write a block of data to the specified pipe. The definition of this function is as follows:

```
FUNCTION PipeWrite( PNum, Wlen: INTEGER;
                    VAR Info: Drvlk ): INTEGER;
```

Parameter	Data Type	Description
Pnum	INTEGER	Pipe number
Wlen	INTEGER	Number of bytes to write (=512)
Info	DrvBlk	Data to be written

This function returns the number of bytes written if the write is successful. Otherwise, a negative error code is returned. The number of bytes to write should always be 512.

PipeClRd function --

PipeClRd uses the Pipe Close command to close the pipe for reading. The definition of this function is as follows:

```
FUNCTION PipeClRd( PNum: INTEGER ): INTEGER;
```

Parameter	Data Type	Description
PNum	INTEGER	Pipe number

This function returns 0 if the pipe was successfully closed. Otherwise, a negative error code is returned. If the pipe is empty, it is deleted.

PipeClWr function --

PipeClWr uses the Pipe Close command to close the pipe for writing. The definition of this function is as follows:

```
FUNCTION PipeClWr( PNum: INTEGER ): INTEGER;
```

Parameter	Data Type	Description
PNum	INTEGER	Pipe number

This function returns 0 if the pipe was successfully closed. Otherwise, a negative error code is returned. Once a pipe has been closed for writing, no additional data can be written to it.

PipePurge function ---

PipePurge uses the Pipe Close command to purge the pipe. The definition of this function is as follows:

```
FUNCTION PipePurge( PNum: INTEGER ): INTEGER;
```

Parameter	Data Type	Description
PNum	INTEGER	Pipe number

This function returns 0 if the pipe was successfully purged. Otherwise, a negative error code is returned.

PipesInit function ---

PipesInit uses the Pipe Area Initialize command to initialize the pipes area. The definition of this function is as follows:

```
FUNCTION PipesInit( Baddr, Bsize: INTEGER ): INTEGER;
```

Parameter	Data Type	Description
Baddr	INTEGER	Pipes area starting block number
Bsize	INTEGER	Pipes area length, in blocks

This function returns 0 if the pipes area was successfully initialized. Otherwise, a negative error code is returned. You should use this function with caution, since calling this function overwrites any data located within the area specified. The pipes area must be allocated within the first 32k blocks of drive 1.

A SIMPLE SPOOLER

A spool program can be used to control access to a shared printer on a network. One computer is used as a despooler, and has the printer attached to it. It is running a despool program, which is looping, looking for pipes with the name PRINTER to open for read.

A second utility program, called the spooler, can be run on any other computer on the network. This program asks for the name of a file to be spooled, opens for write a pipe called PRINTER, copies the file to the pipe, and then closes the pipe.

```
       Despooler                          Spooler

  { look for a pipe to open }
  REPEAT                          Open file f ...
     p := PipeOpRd('PRINTER')     p2 := PipeOpWr('PRINTER');
  UNTIL p>0;                      IF p2 < 0 THEN { error };

                                  { copy file to pipe }
                                  REPEAT
                                     READBLOCK(f, buf);
                                     e := PipeWrite(p2, buf);
                                  UNTIL EOF(f) OR (e<0);

                                  e := PipeClWr(p2);
                                  Close file f...

  {Pipe 'PRINTER' opened.}

  { copy data from pipe to  }
  { printer                 }
  REPEAT
     e := PipeRead(p, buf);
     IF e > 0 THEN PRINT(buf);
  UNTIL e<0;

  e := PipeClRd(p);

  { the pipe has been purged }
```

Of course, the real versions of the DESPOOL and SPOOL programs will be much longer, as they must provide error handling and recovery, as well as some text processing. See the description of the Corvus spool program later in this chapter.

The pipes functions themselves handle the case where two users execute the SPOOL program at the same time. Each user is returned a unique pipe number from the PipeOpWr function, which is used in the calls to the other pipe functions. In fact, the reason pipes are implemented is to provide exactly this capability: two users can access the pipes area at the same time, and not worry about interfering with each other.

It is not possible to control the order in which pipes will be despooled. Both the PipeOpWr and the PipeOpRd functions always open the lowest numbered available pipe.

USING PIPES TO SEND MESSAGES

One of the electronic mail packages available for the Corvus network uses the pipes area for two functions: to send messages between two computers on the network, and to synchronize access to a shared volume. We will look at how the message passing is accomplished.

The Mail Monitor package from Software Connections consists of two programs: a Mail program which a user invokes in order to send or receive mail, and a PostOffice program which is always running on a dedicated computer. Several users can be running the Mail program at the same time.

Messages between the Mail programs and the PostOffice are sent via the pipes area. When the user is ready to receive mail, the Mail program opens and writes the user number into a pipe called MSG. The PostOffice sees the pipe, opens it, and reads the user number contained in it. The PostOffice checks if any mail is waiting for that user, and sends a message back by writing to a pipe called USERnn, where nn is the user number contained in the MSG pipe. The Mail program then opens the USERnn pipe to get the reply. This process is demonstrated by the following program fragments:

```
   Mail                            PostOffice

{ send message }                 { wait for messages }
p := PipeOpWr('MSG');            REPEAT
IF p<0 THEN {error}                p1 := PipeOpRd('MSG');
message := 'USER01';             UNTIL p1>0;
e := PipeWrite(p, 512, message);
IF e<0 THEN {error}
e := PipeClWr(p);

{ wait for reply }
REPEAT                           (Pipe 'MSG' opened.)
  p := PipeOpRd('USER01');
UNTIL p>0;                       { read message }
                                 e := PipeRead(p1, msg);
                                 e := PipeClRd(p1);
                                 { extract pname from        }
                                 { message, and build reply }
                                 p1 := PipeOpWr(pname);
                                 IF p1 < 0 THEN {error}
                                 e := PipeWrite(p1, 512, msg2);
                                 e := PipeClWr(p1);

(Pipe 'USER01' opened.)          { go back to initial loop to }
{ read reply }                   { look for more messages     }
e := PipeRead(p,msg);
e := PipeClRd(p);
```

Again, there is no code needed to handle the case when two users execute the Mail program at the same time. The pipes functions handle all sharing of the pipe area transparently.

THE CORVUS SPOOL PROGRAM

Corvus provides a spool program for most of the operating systems supported. Corvus defines the following format for each pipe:

Block 1: preamble block

Offset/Len	Type	Description
0 / 1	BYTE	Unused - use 0.
1 / 1	BYTE	Length of file name.
2 / 80	BSTR	File name.
82 / 1	BYTE	Length of message.
83 / 80	BSTR	Message.
163 / 1	BYTE	File type (30h=data, 31h=text).
164 / 348	ARRY	Unused - use 0's.

Blocks 2-n: text or data blocks. If file type is text (31h), then each block contains ASCII characters. End-of-line is indicated by the two byte sequence 0Dh, 0Ah (carriage return/line feed). The last block is padded with ASCII NUL characters (00h).

If file type is data (30h), then each block contains data, which is not looked at or changed by either the spool program or the despooler.

The spool program opens the specified pipe for writing, and creates and writes the preamble block. Then it reads from the text file, converting end-of-line sequences from whatever is used by the operating system to 0Dh, 0Ah. Most of the Corvus spool programs also convert a specified new page sequence to the ASCII form feed character (0Ch), and also chain text files as specified by the include sequence.

The despooling function is performed either by a computer running the despool program (or despool option of the Spool program), or by a Corvus Utility Server. In either case, the despool function is going to read pipes and write their contents to a printer. The despooler opens the pipe and reads the preamble block. It writes the file name and user message on a header page. If the preamble block indicates that the file is a data file, the despooler merely writes the entire contents of each pipe block to the printer (some versions will refuse to print a data file). If the preamble block indicates that the file is a text file, then the despooler must look at the contents of each pipe block. If line feeds are off, it looks for all 0Dh, 0Ah byte pairs, and either changes the 0Ah to a 00h or deletes the 0Ah byte. It also handles paging by counting all 0Dh, 0Ah sequences. If the count reaches the lines per page count specified, the despooler inserts a form feed (0Ch) character. The despooler is also looking for form feed characters embedded in the text, and resets to count to zero when one is found. Some despoolers also implement a TAB function.

The spool program can also be used to send a file to another user. One user can spool a file to an agreed upon pipe name, and another user can then despool from the specified pipe name into a file. Both text files and data files may be exchanged. This feature is especially useful for converting files from one file system format to another.

The pipe name used is usually the name of the receiving user. For example, a CP/M user can spool a file developed with WORDSTAR to a pipe called JOAN. MS-DOS user JOAN can then despool the file, and modify it using EASYWRITER.

Corvus Concept Operating System:

Please refer to the Pascal Library User Guide (7100-04978). You should look at Chapter 14, "Corvus Disk Interface Unit" (ccDRVIO), and Chapter 15, Corvus "Disk Pipes Interface Unit" (ccPIPES).

Note that procedure CCPIPEINIT must be called prior to calling any of the other procedures or functions in the ccPIPES unit. The parameter **Netloc** specifies which server will be used for pipe operations. Specifically, the following fields of **Netloc** must be defined before calling CCPIPEINIT:

Netloc.slotno	slot number
Netloc.stationno	server number (ignored for MUX)
Netloc.Kind	either OmninetDisk or LocalDisk

Here is a portion of a SPOOL program for Concept Pascal:

```
PROGRAM SPOOL;
USES {CCLIB} CCDEFN,
     {C2LIB} CCDRVIO, CCPIPES;

VAR pname: PNameStr;
    pno:   INTEGER;
    err:   INTEGER;  {error code}
    NetAddr: CDAddr;
    f:     FILE;
    n:     INTEGER;
    buf:   DrvBlk;

BEGIN

ccDrvioInit;        { initialize unit ccDRVIO }

Initslot(NetAddr); { this procedure, from ccDrvio,
```

```
                    { initializes slotno, stationno, and kind
                    { fields to boot device.  Set driveno to
                    { 1, all other fields to 0 }

ccPipeInit(NetAddr);  { initialize unit ccPipes }

{ get file name and open it... }

pname := 'PRINTER';        { open pipe for writing }
pno := PipeOpWr( pname );
IF pno < 0 THEN { report error and exit... };

WHILE NOT EOF(f) DO BEGIN
   n := BLOCKREAD( f, 1, buf );
   err := PipeWrite( pno, 512, buf );
   IF err < 0 THEN { report error, purge pipe, and exit... };
   END;

err := PipeClWr(pno);

{ close file... }

END.
```

Version IV p-system and Apple Pascal Constellation II:

Look at the interface secitons for the following units:

UCDEFN, UCDRVIO, and UCPIPES

These units are found in library CORVUS.LIBRARY, which is included in the Software Developer's Kit.

Note that the procedure CCPIPEINIT must be called prior to calling any of the other procedures or functions in the ccPIPES unit. The parameter **Netloc** specifies which server will be used for pipe operations. Specifically, the following fields of **Netloc** must be defined before calling CCPIPEINIT:

Netloc.slotno	slot number
Netloc.stationno	server number (ignored for MUX)
Netloc.Kind	either OmninetDisk or LocalDisk

Here is a portion of a SPOOL program for Concept Pascal:

```
PROGRAM SPOOL;
USES {CORVUS.LIBRARY} UCDEFN, UCDRVIO, UCPIPES;

VAR pname: PNameStr;
    pno:   INTEGER;
    err:   INTEGER;  {error code}
    NetAddr: CDAddr;
```

```
         f:      FILE;
         n:      INTEGER;
         buf:    DrvBlk;

     BEGIN

     ccDrvioInit;          { initialize unit ccDRVIO }

     Initslot(NetAddr); { this procedure, from ccDrvio,
                        { initializes slotno, stationno, and kind
                        { fields to boot device.  Set driveno to
                        { 1, all other fields to 0 }

     ccPipeInit(NetAddr);  { initialize unit ccPipes }

     { get file name and open it... }

     pname := 'PRINTER';        { open pipe for writing }
     pno := PipeOpWr( pname );
     IF pno < 0 THEN { report error and exit... };

     WHILE NOT EOF(f) DO BEGIN
        n := BLOCKREAD( f, 1, buf );
        err := PipeWrite( pno, 512, buf );
        IF err < 0 THEN { report error, purge pipe, and exit... };
        END;

     err := PipeClWr(pno);

     { close file... }

     END.
```

MS-DOS 1.x and 2.x Constellation II:

The machine language interface described in Chapter 4 must be used to send pipes commands. The Software Developer's Kit contains examples of using pipes with MS Pascal.

CP/M 86 and CP/M 80 Constellation II:

The machine language interface described in Chapter 4 must be used to send pipes commands. The Software Developer's Kit contains examples of using pipes with Pascal MT+.

Apple Pascal Constellation I:

Look at the interface sections for the following units:

DRIVEIO and PIPES.

These units are found in library CORVUS.LIBRARY, which is contained on the standard distribution diskettes.

Note that the procedure PIPESINIT must be called prior to calling any of the other procedures or functions in the PIPES unit. The parameter should be set to FALSE.

Here is a portion of a SPOOL program for Apple Pascal:

```
PROGRAM SPOOL;
USES {CORVUS.LIBRARY} DRIVEIO, PIPES;

VAR pname: PNameStr;
    pno:   INTEGER;
    err:   INTEGER;  {error code}
    f:     FILE;
    n:     INTEGER;
    buf:   BLOCK;

BEGIN

DriveIoInit;       { initialize unit DriveIO }

PipesInit(FALSE);  { initialize unit Pipes }

{ get file name and open it... }

pname := 'PRINTER';        { open pipe for writing }
pno := PipeOpWr( pname );
IF pno < 0 THEN { report error and exit... };

WHILE NOT EOF(f) DO BEGIN
   n := BLOCKREAD( f, 1, buf );
   err := PipeWrite( pno, 512, buf );
   IF err < 0 THEN { report error, purge pipe, and exit... };
   END;

err := PipeClWr(pno);

{ close file... }

END.
```

Apple DOS Constellation I/II:

Corvus provides two assembly language procedures (BCI.OBJ and OMNIBCI.OBJ) for sending arbitrary disk commands. BCI.OBJ is for MUX networks, and OMNIBCI.OBJ is for OmniNet networks. See Chapter 4 for information on these procedures.

The program SPOOL on the distribution floppy for Constellation I shows how to send pipes commands using these routines.

DEVICE SPECIFIC INFORMATION | A

This appendix discusses the unique characteristics of each mass storage device.

The following devices are described:

Rev B/H drive
OmniDrive
The Bank

For each device, the following information is provided:

Hardware description
Firmware and PROM code interaction
Firmware layout
Device parameters
Front panel LED's
DIP switch settings

REV B/H DRIVES

The Rev B/H drives may be used stand-alone, in a Constellation network attached to a Corvus multiplexer, or in an Omninet network attached to a Corvus disk server.

Up to four drives may be daisy-chained. The controller on drive one handles all commands except those with a drive number specifying an add-on drive. For add-on drives to work, drive one must know how many drives are daisy-chained to it. Drive one gets this information as part of its power-up procedure. Thus the add-on drives must be powered-on when drive one is reset. The drive number is set with a DIP switch; the DIP switch settings are described later in this section.

Rev B/H Hardware Description

This section attempts to identify major pieces of the hardware. It does not try to explain how it works. Refer to the hardware specification for more details.

The Rev B/H Corvus drives consist of an IMI Winchester hard disk, two or three printed circuit boards (depending on model), and a power supply.

The disk controller consists of a Z80 microprocessor, 4k bytes of EPROM, and 5k bytes of RAM. Communication with the outside world is handled through two input/output ports: one connected to a bidirectional data bus, and the other providing control signals. These signals are available on the 34-pin Corvus-IMI bus at the back of the drive. The signals on this bus are further described at the end of this section.

Rev B/H Firmware And Prom Code

Conceptually, firmware is the code running in the controller. As described in the hardware requirements, Rev B/H code is resident both in PROM and RAM. Corvus has a convention that designates the code in PROM as PROM code and that in RAM as firmware. This document follows that convention.

Part of the controller code is in the 4k PROM. Because of the limited controller RAM, the firmware consists of several segments which are overlayed as needed. The main part of the firmware, the dispatcher, is 1k bytes long and is the command dispatcher. It intercepts the command string sent from the host, decodes it, then activates the appropriate routines in the PROM or overlays the appropriate firmware into the RAM.

The firmware code occupies several blocks in an area called the firmware area. The firmware area occupies the first two cylinders of the Rev B/H drive. The first cylinder contains the firmware, the second one is a duplicate. Besides the firmware code, the firmware area contains other information such as the track sparing information, the drive parameters, etc. Refer to the next section for the layout of this area.

At power on, the PROM code initializes itself and then examines the front panel switches. If all switches are in the normal position, the controller reads in the boot block (block 0 of the firmware). The boot block performs some initialization, then loads the dispatcher into RAM and transfers control to it. If the firmware is bad, the drive will not come ready.

If, on power on, the PROM code finds that the Format switch is on, it utilizes the command dispatcher in PROM. The capability of this dispatcher is quite limited, however, as it allows the host only the functions such as format, verify, and read-write to the firmware area. If, on power on, the PROM code finds that the LSI-11 switch is on, the LSI code is loaded from the firmware area into RAM.

Rev B/H Firmware Layout

The first two cylinders on all drives are allocated as the firmware area, the second cylinder being a backup copy of the first. There are no spared tracks allowed in this region; all blocks must be good. The usage for the blocks within a cylinder is shown below.

Block	Len	Description
0	1	Boot Block
1	1	Disk parameter block (see below)
2	1	Diagnostic block (prep code)
3	1	Constellation parameter block (see below)
4	2	Dispatcher code
6	2	Pipes and semaphores code. The semaphore table is contained in block 7, bytes 1 - 256.
8	10	Mirror controller code
18	2	LSI-11 controller code
20	2	Pipes controller code
22	3	Reserved for future use
25	8	Boot blocks 0-7. Apple II uses 0-3, Concept uses 4-7.
33	4	Active user table
37	3	Reserved

Block 1, the disk parameter block, contains the following information:

Byte	Len	Description
0	16	Spared track table (Rev B drives) - 2 bytes per spared track (lsb,msb). End of table is FFFFh.
16	1	Interleave factor
17	1	Reserved
18	14	Virtual drive table -- 2 bytes/entry (lsb,msb). Unused entries are FFFFh.
32	8	LSI-11 Virtual drive table
40	8	LSI-11 spared track table
48	432	Reserved
480	32	Spared track table (Rev H drives) 2 bytes per spared track (lsb,msb). End of table is FFFFh. Bytes 480-493 must match bytes 0 to 13 (see below)

There are two spared track tables for Rev B/H. The first 7 entries in the second table should match the 7 entries in the first table. Rev B drives can have a maximum of 7 spared tracks; Rev H drives can have a maximum of 31 spared tracks.

Block 2 is the diagnostic, or prep, block. It contains the code necessary to perform the prep mode functions. This code is put in the firmware area for archival purposes only. The host uses a diag file separate from the firmware area.

Block 3 is the Constellation parameter block. Its format is shown below:

Byte	Len	Description
0	12	Multiplexer slot and polling parameters
12	2	Block address of Pipe Name Table (lsb,msb) (start of pipes area)
14	2	Block address of Pipe Pointer Table (lsb,msb)
16	2	Number of blocks in pipes area (lsb,msb)
18	470	Reserved
488	12	Reserved for software protection
500	12	Reserved for serial number

Rev B Parameters

	Model 6 Mb	Model 11 Mb	Model 20 Mb
Sectors per track	20	20	20
Surfaces (heads)	4	3	5
Cylinders	144	358	388
Total tracks per drive	576	1074	1940
Reserved for spares	7	7	7
Reserved for firmware	8	6	10
Usable tracks per drive	561	1061	1923
Blocks per drive	11220	21220	38460

Rev B Front Panel LED's And Switches

The front panel of the Rev B/H drive has three (3) LED's: a FAULT LED, a BUSY LED and a READY LED. During power on , the FAULT LED and the READY LED should be on, and the BUSY LED flashing, until the end of the initialization. When the initialization is done, the following light conditions may occur during drive operations:

FLT LED	BSY LED	RDY LED	Condition
off	on	off	Firmware not installed or or corrupted
off	off	on	Ready
off	on	off	In prep mode
on	flash 1/4 sec	off	Operation error

When the drive is put in prep mode to be formatted or to have firmware updated, the FLT and RDY LED are turned off and the BSY LED turned on. You must be careful when this condition occurs as the disk can be reformatted and all data can be lost.

There are four toggle switches located beneath the front panel LED's. These are, from left to right, (1) LSI-11 switch, (2) MUX switch, (3) format switch, (4) reset switch. The normal position for each switch is to the left.

Rev B DIP Switches

There is an 8 position DIP switch accessible through the trap door located on the bottom of the drive case. This switch is used to set the drive number for daisy-chained drives.

Switch setting

Drive number	1	2	3	4	5	6	7	8
1	X	X	O	-	-	-	-	-
2	X	O	X	-	-	-	-	-
3	X	O	O	-	-	-	-	-
4	O	X	X	-	-	-	-	-
5	O	X	O	-	-	-	-	-
6	O	O	X	-	-	-	-	-
7	O	O	O	-	-	-	-	-

X = CLOSED; O = OPEN

The DIP switch pressed in on the side marked OPEN is considered OPEN.

Rev H Parameters

	Model 6 Mb	Model 11 Mb	Model 20 Mb
Sectors per track	20	20	20
Surfaces (heads)	2	4	6
Cylinders	306	306	306
Total tracks per drive	612	1224	1836
Reserved for spares	31	31	31
Reserved for firmware	4	8	12
Usable tracks per drive	577	1185	1793
Blocks per drive	11540	23700	35860

Rev H Front Panel LED's And Switches

Same as Rev B.

Rev H DIP Switches

There is an 8 position DIP switch located on the controller PC board. This switch is used to set the drive number for daisy-chained drives. To access this switch, you must remove the top drive cover; the board is mounted on the inside of the drive cover.

Drive number	Switch setting 1	2	3	4	5	6	7	8
1	X	-	-	X	-	-	-	-
2	X	-	-	O	-	-	-	-
3	O	-	-	X	-	-	-	-
4	O	-	-	O	-	-	-	-

X = CLOSED; O = OPEN

The DIP switch pressed in on the side marked OPEN is considered OPEN.

There is also a 4 position DIP switch located on the back panel of the drive. This switch is used to specify whether an internal Corvus MIRROR card is present in the drive.

Meaning	Switch setting 1	2	3	4
No MIRROR/external MIRROR	X	X	X	X
PAL/SECAM MIRROR	X	O	O	O
NTSC MIRROR	O	O	O	O

X = CLOSED; O = OPEN

The DIP switch pressed in on the side marked OPEN is considered OPEN.

Disk Flat Cable Interface

All cable assignments are TTL.

Cable wire assignments:

NAME	ORIGINATOR	FLAT CABLE WIRE
Data Bit 0	bi-directitonal	25
Data Bit 1	bi-directitonal	26
Data Bit 2	bi-directitonal	23
Data Bit 3	bi-directitonal	24
Data Bit 4	bi-directitonal	21
Data Bit 5	bi-directitonal	22
Data Bit 6	bi-directitonal	19
Data Bit 7	bi-directitonal	20
DIRC (bus dir)	drive	9
READY	drive	27
-STROBE	computer	29
-RESET	drive	31
+5 volts	drive	3,4,34
Ground	drive	6,8,10,17,28,30,32
Alternate select	drive	11
Reserved	computer	5
Unused	----	1,2,7,12-16,18,33

Cable timing

General case:

Command initiation and computer to drive data transfer.

```
READY   ----------------+        +-------------------+        +-
                        |        |                   |        |
                        +--------+                   +--------+

                   500 nsec.
                   <>
-STROBE ----------+  +-------------------------+  +-------------
                  |  |                         |  |
                  +--+                         +--+

                       > 50 nsec.
                      <========>
                /------------                /------------
DATA  ---------<              >-------------<              >----
                ------------/                ------------/

     -----------------------------------------------------------

DIRC
```

The drive indicates its readiness to accept a command by raising the READY line. The computer then puts a command byte to the data lines and pulses -STROBE (the command byte is to be latched by the drive on the rising edge of -STROBE). Upon seeing the -STROBE pulse, the drive drops the READY line as an acknowledgement to the computer. When ready for the next command byte the drive again raises the READY line.

The drive takes each command byte as it needs it. If it is expecting another command byte, and one is not there, the drive will timeout after approximately 4 seconds. The drive flushes the current command, and waits for a new command to start.

At the end of the command sequence, the drive keeps the READY line low until the desired operation has been performed. Upon completion of the operation, the drive lowers the DIRC line and raises the READY line, allowing the computer to read data and status information. Note that all commands consist of a write phase, during which command and data information is sent to the drive, followed by a read phase, during which status and data information is received from the drive.

Drive to computer data transfer:

```
            +---------+           +--------+                    +-
            |         |           |        |                    |
READY   ----+         +-----------+        +------------//------+

-STROBE -------+  +-----------------+  +----------------//------
               |  |                 |  |
               +--+                 +--+

            /------------        /------------
DATA  -----<              >-----<              >-------//----
            ------------/        ------------/

DIRC ----+                                                  +----
         |                                                  |
         +----------------------------------------------//---+
```

The drive starts a computer read sequence by lowering the DIRC line. The drive then puts a byte to the data lines and raises the ready line. The computer then pulses the -STROBE line, capturing the data on the rising edge. The drive then lowers the READY line until the next data byte is ready to send. After the last byte is transferred, the drive raises the DIRC line prior to raising the READY line.

Special conditions:

There are two special conditions which deviate from the general cable timing information presented and must be accounted for by the computer-disk controller or by the computer-disk handler.

Case 1 -- READY line glitch after the last byte of command.

After the last command byte is received by the drive, the READY line goes high (for 20 uSEC. or less). Since this occurs prior to the completion of the command operation, it must be ignored. Since the glitch occurs while the DIRC line is high, it is easy to detect either in hardware, by gating, or in software, by the procedure shown below in pseudo-code.

```
        REPEAT UNTIL (DIRC = LOW) AND (READY = HIGH );
```

Case 2 -- DIRC line glitches after last byte of Mirror command.

After the last command byte of a Mirror command is received, the DIRC line repeatedly alternates between high and low, while the drive talks to the Mirror. Since these changes occur while the READY line is low, they are easy to detect either in hardware, by gating, or in software, by the procedure shown below in pseudo-code.

REPEAT UNTIL (READY = HIGH) AND (DIRC = LOW);

Note that the two glitch cases are resolved with a single fix.

Cable Connector Description

A 17 x 2 female connector is attached to the cable. The red stripe on cable is pin 1.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 1| 3| 5| 7| 9|11|13|15|17|19|21|23|25|27|29|31|33|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
| 2| 4| 6| 8|10|12|14|16|18|20|22|24|26|28|30|32|34|
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Pin 1 is normally designated by a square pad on the circuit side of the interface card.

OMNIDRIVE

The OmniDrive is a Winchester hard disk device with a built-in Omninet disk server interface. Functionally, it resembles a Rev B/H drive connected to a disk server. The OmniDrive is designed such that it is compatible with the old disk server and disk drive combination to minimize software impact. However, some changes are warranted due to hardware constraints and systems requirements. Also, certain features are intended as upgrades to the feature set. All the changes from Rev B/H controllers are documented in Appendix C.

The OmniDrive is a self-contained box with a controller and disk server on the same PCB. It does not support a flat cable interface and has no daisy chain capability. To expand the capacity of the network, more OmniDrives can be attached to the Omninet cable, effectively forming a multiple server network.

OmniDrive Hardware Description

This section attempts to identify major pieces of the hardware. It does not try to explain how it works. Refer to the hardware specification for more details.

The OmniDrive controller consists of three main sections: a transporter, a disk server and a disk controller. The transporter section communicates to the Omninet. It mainly consists of three chips: a 6801 processor, an ADLC and a custom gate array. The disk server section adds one RAM to buffer data in and out of the network. It also has some firmware code that understands Constellation protocols. The disk controller utilizes a hard disk controller chip (WD1010) and the 6801 is used as the processor.

The EPROM requirements are:
8k bytes - 2k disk server, 6k disk controller
(socket can also accommodate 16k bytes PROM; the extra PROM space is used if more code is needed)

There are four RAM sockets on the controller: two designated as share RAMs and two as scratch RAMs. The share RAMs can be accessed by the Omninet gate array chip, thus they can be DMAed from and to the network. The 6801 processor can also read-write to these share RAMs. The two scratch RAMs, however, can only be accessed by the processor (6801). Each RAM socket can take a 2k by 8 static RAM chip.

The shared RAMs are utilized as follows:

2k bytes	- disk server buffer
2k bytes	- read-write buffer to 1010

The scratch RAMs are utilized as follows:

1k bytes	- disk server scratch RAM
1k bytes	- disk controller scratch RAM and semaphore table
1k bytes	- pipes table
1k bytes	- downloaded controller code

OmniDrive Firmware And Prom Code

Conceptually, firmware is the code running in the controller. As described in the hardware requirements, OmniDrive code is resident both in PROM and RAM. Corvus has a convention that designates the code in PROM as PROM code and that in RAM as firmware. This document follows that convention.

Most of the controller code is in the 8k PROM. It handles the disk server function as well as the actual disk controller function. The firmware code, 1k bytes long, is essentially a command dispatcher. It intercepts the command string sent from hosts, decodes it, then activates the appropriate routines in the PROM.

The firmware code occupies two blocks in an area called the firmware area. The firmware area occupies the first four tracks of the OmniDrive. The first two tracks contain the firmware, the last two are duplicates. Beside the firmware code, the firmware area contains other information such as the track sparing information, the drive information, the pipes table, etc. Refer to the next section for the layout of this area.

At power on, the two dispatcher blocks are loaded from the media to RAM. This RAM code now functions as the command dispatcher. If the firmware does not exist on the disk, the controller switches to a special command dispatcher entirely resident in PROM. The capability of this dispatcher is quite limited, however, as it allows the host only the functions such as format, verify, and read-write to the firmware area.

OmniDrive Firmware Layout

In the OmniDrive, the first four tracks of the drive are reserved for the Corvus firmware. The firmware is 36 blocks long (block number 0-35) and thus occupies 2 tracks. The firmware is duplicated for safety in the next two tracks.

The following is the layout of the firmware area:

Block	Len	Description
0	1	Spared track table (see below)
1	1	Disk parameter block (see below)
2	1	Diagnostic block (prep block)
3	1	Constellation parameters (see below)
4	2	Reserved
6	2	Dispatcher code
8	1	Pipe Name table
9	11	Reserved
20	1	Pipe Pointer table
21	3	Reserved
24	8	Boot blocks 0-7. Apple II uses blocks 0-3, Concept uses blocks 4-7
32	4	Active user table

Block 0 is the spared track table. The table has the following format:

Byte	Len	Description
0	2	First spared track (msb,lsb)
2	2	Second spared track (msb,lsb)
...		...

The end of the table is indicated by an entry of FFFFh. The number of spared tracks reserved is different for various drive models. The maximum number of spared tracks for a drive is in ROM, and can be obtained by the Get Drive Parameters command. The maximum number of spared tracks supported by the controller is 64.

Block 1 is the disk parameter block. It contains the following information:

Byte	Len	Description
0	16	Reserved
16	1	Interleave factor
17	31	Reserved
48	2	Starting block address of pipes area (lsb,msb)
50	2	Number of blocks in pipes area (lsb,msb)
52	1	Write-verify flag
53	195	Reserved
248	8	Format password
256	256	Reserved

Block 2 is the diagnostic, or prep, block. It contains the code necessary to perform the prep mode functions. This code is put in the firmware area for archival purposes only. The host uses a diag file separate from the firmware area.

Block 3 is the Constellation block. It currently contains the following information:

Byte	Len	Description
0	488	Reserved.
488	12	Reserved for software protection.
500	12	Reserved for serial number.

OmniDrive Parameters (1-Feb-84)

	Heads	Cyls	Max Spared Tracks	Capacity	Precom Cyl	Land Cyl
IMI 5006H	2	306	12	10728	256	329
IMI 5012H	4	306	20	21600	256	329
IMI 5018H	6	306	28	32472	256	329
Rodime 201	2	306	12	10728	0	319
Rodime 202	4	306	20	21600	0	319
Rodime 203	6	306	28	32472	0	319
Rodime 204	8	306	36	43344	0	319
Dansei RD4064	2	306	12	10728	128	337
Dansei RD4127	4	306	20	21600	128	337
Dansei RD4191	6	306	28	32472	128	337
Dansei RD4255	8	306	36	43344	128	337
Ampex 7	2	306	12	10728	128	319
Ampex 13	4	306	20	21600	128	319
Ampex 20	6	306	28	32472	128	319
Ampex 27	8	306	36	43344	128	319
Microp 1304	6	823	40	88092	400	N/A
Vertex 150	5	987	40	88038	N/A	N/A
Rodime RO204E	8	618	40	88200	0	640
Maxtor XT1065	7	918	46	114768	N/A	N/A
Maxtor XT1105	1	918	70	180432	N/A	N/A
Maxtor XT1140	15	918	94	246096	N/A	N/A
Miniscr 2006	2	306	12	10728	0	336
Miniscr 2012	4	306	20	21600	0	336
Miniscr 4020	4	459	28	32472	0	522

OmniDrive Front Panel LED's

The front panel of the OmniDrive has three LED's: a FAULT LED, a BUSY LED and a READY LED. During power on , the BUSY LED should be on until the end of the initialization. When the initialization is done, the following light condition might occur:

FLT LED	BSY LED	RDY LED	Condition
on	on	off	Firmware not installed or corrupted
on	on	on	Same address as another node on network
off	off	on	Ready
on	off	on	In prep mode
flash 1/4 sec	off	off	Wrong transporter version
each light flash 1/4 sec			RAM error
quick flash	off	off	Operation error

When the drive is put in prep mode to be formatted or to have firmware updated, the FLT and RDY LED are turned on and the BSY LED turned off. You must be careful when this condition occurs as the disk can be reformatted and all data lost.

The Drive Tables of the Shadow Drive in the Second Preliminary Shadow Drive Prom are as follows:

ODC230 on

Switch U22	HDA	Model	Cyl	Hds	Spare Blks	Total Blks	RWC	PreComp	Park	JP E3	JP E4
0	IMI	5006H	306	2	12	10728	-	256	329	-	A-B
1	IMI	5012H	306	4	20	21600	-	256	329	-	A-B
2	IMI	5018H	306	6	28	32472	-	256	329	-	A-B
3	Rodi	201	306	2	12	10728	132	0	319	A-B	B-C
4	Rodi	202	306	4	20	21600	132	0	319	A-B	B-C
5	Rodi	204	306	8	36	43344	132	0	319	A-B	B-C
6	Rodi	204E	620	8	56	88200	-	0	639	-	B-C
7	Mini	3212	612	2	20	21600	-	128	656	-	A-B
8	Mini	3425	612	4	36	43344	-	128	656	-	A-B
9	Mini	6053	992	5	56	88200	-	512	auto	-	A-B
10	Den	4064	306	2	12	10728	-	132	337	-	A-B
11	Den	4127	306	4	20	21600	-	132	337	-	A-B
12	Den	4255	306	8	36	43344	-	132	337	-	A-B
13	CMI	5206	306	2	12	10728	-	132	329	-	A-B
14	CMI	5412	306	4	20	21600	-	132	329	-	A-B
15	Bull	530	987	3	20	52866	-	none	auto	-	A-B
16	Maxt	1140	918	15	128	245484	-	none	auto	B-C	A-B

New Additions to the Drive Tables for OmniDrive Prom '18H' the Shadow Drive.

Switch U22	HDA	Model	Cyl	Hds	Spare Blks	Total Blks	RWC	PreComp	Park	JP E3	JP E4
17	Sea	ST212	306	4	20	21600	-	128	305	-	A-B
18	Sea	ST225	612	4	36	43344	-	256	633	-	A-B
19	Rodi	352	306	4	20	21600	132	0	319	A-B	B-C
20	Rodi	202E	620	8	68	43344	-	0	639	A-B	B-C
21	Sea	ST4026	615	4	36	43344	-	300	auto	-	A-B
? 22	Sea	ST4038	733	5	40	65178	-	556	auto	-	A-B
? 23	Sea	ST4051	977	5	56	~~88650~~ 66650	-	556	auto	-	A-B
24	Mini	8425	612	4	36	43344	-	0	663	-	A-B
25	Mini	6032	1024	3	30	~~54694~~ 54654	-	512	auto	-	A-B
26	Mini	6085	1024	8	85	~~88200~~ 145854	-	512	auto	-	A-B
27	Maxt	2085	1224	7	90	152532	-	none	auto	B-C	A-B
28	Maxt	2140	1224	11	128	239976	-	none	auto	B-C	A-B
29	Maxt	2190	1224	15	128	328104	-	none	auto	B-C	A-B
30	Micr	1304	823 810	6	40	88092 88848	-	400	auto	-	A-B
31	Micr	1325	1024	8	85	145854	-	400	auto	-	A-B
32	Bull	550	987	5	31	88200	-	none	auto	-	A-B
33	Bull	570	987	7	40	123570	-	none	auto	-	A-B
34	Bull	585	1166	7	85	145314	-	none	auto	-	A-B
35	Hit	511-3	699	5	30	62298	-	256	auto	-	A-B
36	Hit	511-5	699	7	51	86958 87084	-	256	auto	-	A-B
37	Hit	511-8	699	9	85	111636	-	256	auto	-	A-B
38	Tand	755	977	5	50	85958 86958	-	0	auto	-	A-B
39	Tand	362	612	4	36	43344	-	0	auto	-	A-B

? denotes a questionable entry for the Precomp cylinder.

OmniDrive DIP Switches

One of the design objectives for the OmniDrive controller is to have a standard disk interface so that it can communicate with drive mechanisms from various manufacturers. (ST-412 is the de-facto standard for 5 1/4" disk drive).

The ST-412 standard only specifies electrical interface requirements, but drives have different disk parameters (number of heads, number of cylinders, landing track, etc). The OmniDrive controller has an 8 position DIP switch which is used to select the drive mechanism type. The tables of the drive parameters are built into the PROM. The DIP switch selection forces the controller at power-on time to load the appropriate table entry into RAM, which the controller then uses as the set of parameters.

The DIP switch settings for PROM version ODB 0.9 are listed below.

Drive type	Switch setting 1	2	3	4	5	6	7	8
IMI 5006H	X	X	X	X	X	X	X	X
IMI 5012H	O	X	X	X	X	X	X	X
IMI 5018H	X	O	X	X	X	X	X	X
Rodime 201	O	O	X	X	X	X	X	X
Rodime 202	X	X	O	X	X	X	X	X
Rodime 203	O	X	O	X	X	X	X	X
Rodime 204	X	O	O	X	X	X	X	X
Dansei RD4064	O	O	O	X	X	X	X	X
Dansei RD4127	X	X	X	O	X	X	X	X
Dansei RD4191	O	X	X	O	X	X	X	X
Dansei RD4255	X	O	X	O	X	X	X	X

X = CLOSED; O = OPEN

The DIP switch pressed in on the side marked OPEN is considered OPEN.

Switch setting

Drive type	1	2	3	4	5	6	7	8
Ampex 7	O	O	X	O	X	X	X	X
Ampex 13	X	X	O	O	X	X	X	X
Ampex 20	O	X	O	O	X	X	X	X
Ampex 27	X	O	O	O	X	X	X	X
Micropolis 1304	O	O	O	O	X	X	X	X
Vertex 150	X	X	X	X	O	X	X	X
Rodime RO204E	O	X	X	X	O	X	X	X
Maxtor XT1065	X	O	X	X	O	X	X	X
Maxtor XT1105	O	O	X	X	O	X	X	X
Maxtor XT1140	X	X	O	X	O	X	X	X
Miniscribe 2006	O	X	O	X	O	X	X	X
Miniscribe 2012	X	O	O	X	O	X	X	X
Miniscribe 4020	O	O	O	X	O	X	X	X

X = CLOSED; O = OPEN

The DIP switch pressed in on the side marked OPEN is considered OPEN.

THE BANK

The Bank is a random access tape device designed to be a back up and on-line device in an Omninet network. The product consists of a tape transport (LM 101) and a Bank controller. The device has a built-in Omninet interface and is a server on the network. It supports all the standard Corvus disk commands.

The tape is a continuous loop with a loop time of 20 seconds for a 200MB tape and 10 seconds for a 100MB tape. The long tape has 103 meters of media and the short one 53 meters. The tape spins at a speed of 5.5 meters/sec. There are 101 tracks on the tape. Track 0 is designated as the landing track. Track 1 is used as the firmware track. Tracks 2-100 are the user tracks.

Each track is internally divided into sections, called heads. Each section is analogous to a track on a Winchester. A section contains 256 sectors, 1024 bytes each. A 200MB tape has eight sections, while a 100MB tape has four sections. A 200MB tape therefore has 2048 sectors per track; four sectors are reserved for sparing bad ones, so there are 2044 user sectors per track. For a 100MB tape, there are 1024 sectors per track, with four used for sparing, leaving 1020 user sectors per track.

The Bank Hardware Description

This section attempts to identify major pieces of the hardware. It does not try to explain how it works. Refer to the hardware specification for more details.

The Bank controller consists of three main sections: a transporter, a disk server and a tape controller. The transporter section communicates to the Omninet. It mainly consists of 3 chips: a 6801 processor, an ADLC and a custom gate array. The disk server section adds one RAM to buffer data in and out of the net. It also has some firmware code that understands Constellation protocols. The tape controller utilizes a hard disk controller chip (WD1010) and the 6801 is used as the processor.

The EPROM requirements are:
8k bytes - 2k disk server, 6k disk controller

There are 5 RAM sockets on the controller: 2 designated as share RAMs and 3 as scratch RAMs. The share RAMs can be accessed by the Omninet gate array chip, thus they can be DMAed from or to the network. The 6801 processor can also read-write to these share RAMs. The three scratch RAMs, however, can only be accessed by the processor (6801). Each RAM socket can take a 2k by 8 static RAM chip.

The shared RAMs are utilized as follows:

2k bytes	- disk server buffer
2k bytes	- read-write buffer to 1010

The scratch RAMs are utilized as follows:

1k bytes	- disk server scratch RAM
1k bytes	- disk controller scratch RAM and semaphore table
1k bytes	- pipes table
3k bytes	- downloaded controller code

The Bank Firmware And Prom Code

Conceptually, firmware is the code running in the controller. As described in the hardware requirements, Bank code is resident both in PROM and RAM. Corvus has a convention that designates the code in PROM as PROM code and that in RAM as firmware. This document follows that convention.

Most of the controller code is in the 8k PROM. It handles the disk server function as well as the actual tape controller function. The firmware code, 3k bytes long, is essentially a command dispatcher, but also contains the pipes and semaphore code. The command dispatcher intercepts the command string sent from a host, decodes it, then activates the appropriate routines in the PROM. The pipes and semaphore code perform the functions their names imply.

The firmware occupies the first 38 blocks of track 1. The first block is the boot block which contains the parameters for that tape. This block is duplicated in the next two blocks for reliability. The dispatcher code occupies two blocks in the firmware. The pipe and semaphore code occupies four blocks. Besides this code, the firmware area contains other information such as the track sparing information, the pipes table, etc. Refer to the next section for the layout of this area.

At power on, the dispatcher and the pipes and semaphore code are loaded from the media to RAM. If the firmware does not exist on the tape, the controller switches to a special command dispatcher entirely resident in PROM. The capability of this dispatcher is quite limited, however, as it allows the host only the functions such as format, verify, read-write to the firmware area.

The Bank Firmware Layout

In each Bank Tape, there is a non-user accessible area where the Corvus firmware is located. The firmware is 36 blocks long (block number 0-35) and occupies 38 sectors in track 1 of the tape. Each sector is 1024 bytes long, but the firmware only utilizes the first 512 bytes of each sector. The first firmware block, the boot block, contains vital information about the tape and is triplicated.

The following is the layout of the firmware area:

Block	Len	Description
0	1	Boot block, tape parameters, start of spare sector table (see below)
1	1	Contains the rest of the spare sector table
2	1	Format results (see below)
3	1	Constellation block (see below)
4	2	Reserved
6	2	Dispatcher
8	1	Pipe name table
9	3	Diag blocks 0, 1, 2
12	4	Pipes and semaphore code
16	4	Reserved
20	1	Pipe pointer table
21	3	Reserved
24	8	Boot blocks 0-7. Apple II uses 0-3, Concept uses 4-7
32	4	Active User table

Block 0 contains tape information and sector sparing of the first 40 tracks in the following format:

Byte	Len	Description
0	2	Boot hello message (5AA5h)
2	12	Bad track bit map (first byte corresponds to tracks 0-7, arranged MSB: T0, T1, ... T7 :LSB)
15	1	Interleave factor (1 to 31, odd)
16	1	Number of heads on this tape (4 or 8)
17	1	Number of sectors per section (0 = 256 sectors)
18	2	Number of sectors per track (1024 or 2048 - msb,lsb)
20	2	Number of user sectors per track (1020 or 2044 - msb,lsb)
22	3	Total user sectors (101376 or 202356 - msb..lsb)
25	3	Tape index counter (msb,lsb)
28	2	Number of motor start-stop (msb,lsb)
30	12	Reserved
52	2	Pipe area starting block number (lsb,msb)
54	2	Pipe area size (length in blocks) (lsb,msb)
56	1	Tape type (bit 0 set - fast tracks on; bits 1-7 reserved)
57	8	Tape name in ASCII
65	8	Tape password in ASCII
73	2	Format date in ASCII
75	32	Tape comment in ASCII
107	85	Reserved
192	320	Track 0 to track 39 bad sector table

Each track has eight bytes reserved in the bad sector table for four entries (an entry is two bytes). The first byte of the entry is the head of the bad sector; the second byte is the sector number. The entries within a track are sorted in order (low to high). The unused entries are filled with 0FFFFH.

Block 1 contains the rest of the spare sector table:

Byte	Len	Description
0	488	Track 40 to track 100 bad sector table
488	24	Reserved.

Block 2 contains the result of the last tape format. The layout of this data is shown:

Byte	Len	Description
0	1	Result code
1	1	Bad track count
2	510	Bad track list, each entry two bytes (lsb,msb)

Block 3 is the Constellation block. It currently contains the following information:

Byte	Len	Description
0	488	Reserved.
488	12	Reserved for software protection
500	12	Reserved for serial number

Blocks 9, 10, 11 are the diag blocks. They contains code to format, verify, and read-write firmware area. This code is put in the firmware area for archival only. The host uses a diag block file that is separate from the firmware file.

The Bank Parameters

	100MB tape	200MB tape
Number of tracks per tape	101	101
Number of sections per track	4	8
Number of sectors per section	256	256
Number of sectors per track	1024	2048
Number of bytes per sector	1024	1024
Number of spare sectors per track	4	4
Number of user sectors per track	1020	2044
Landing track number	0	0
Firmware track number	1	1
Number of user data tracks	99	99
Loop time	9.4 sec	18.8 sec
Tape life	500 hours	500 hours
Number of start-stops	2000	2000

The Bank Front Panel LED'S

The front panel of The Bank has three LED's: a FAULT LED, a BUSY LED and a READY LED. During power on , the BUSY LED should be on until the end of the initialization. When the initialization is done, the following light condition might occur:

FLT LED	BSY LED	RDY LED	Condition
on	off	off	Fatal hardware error
on	on	off	Firmware not installed or corrupted
on	on	on	Same address as another host on network
off	off	on	Ready, tape is OK
flash 1/4 sec	off	off	Wrong transporter version
flash each light 1/4 sec			RAM error
quick flash	off	off	Operation error

When The Bank is put in prep mode to be formatted or to have firmware updated, the FLT and RDY LED are turned on and the BSY LED turned off. You must be careful when this condition occurs as the tape can be reformatted and all data lost. The following lights could happen in prep mode:

FLT LED	BSY LED	RDY LED	Condition
on	off	on	Bank in prep mode
on	on	on	Bank is formatting
off	on	on	Bank is filling during format
off	on	off	Bank is verifying during format
off	on	off	Bank is executing cmnds in prep

This page intentionally left blank.

CONSTELLATION DEVICE TYPES

Specific types are indented below their generic type.

Value	Meaning
01	Generic disk device, booting; Corvus disk server
02	Generic Print Server
03	Reserved
04	Mirror Server
05	Bank
06	Omnidrive (generic type = 01)
07-0Fh	Reserved.
10h	Generic disk device, non-booting
11h-1Fh	Reserved for future mass storage devices.
20h-3Fh	Workstations. Workstations are Constellation Boot number plus 20.
20h	Generic Workstation Device Type
21h	Apple II
25h	Corvus Concept
29h	IBM/PC or IBM/XT
2Ah	Xerox 820
2Bh	Zenith H89
2Ch	NEC PC8000
2Dh	Commodore PET
2Eh	Atari 800
2Fh	TRS-80 Model I
30h	TRS-80 Model II
31h	LSI-11
33h	Apple ///
34h	DEC Rainbow
35h	TI Professional
36h	Zenith Z-100
37h	Corvus Concept Plus

38h	Corvus Companion
39h	Apple MacIntosh
3Ah	Sony SMC-7086

40h-5Fh Reserved for future workstations.

60h-7Fh Operating system types. Operating system types are Constellation operating system number plus 60h.

61h	Apple Pascal
62h	Apple DOS 3.3
63h	UCSD Pascal version 2.x
64h	MS-DOS 1.x
65h	Apple SOS
66h	Apple Pascal Runtime
67h	CP/M 80
68h	RT-11
69h	RSX-11
6Ah	PET DOS
6Bh	NEWDOS (TRS-80 Mod I/III)
6Ch	NEWDOS-80 (TRS-80 Mod I/III)
6Dh	Atari DOS 2.0
6Eh	UNIX System 3
6Fh	CP/M 86
70h	CCOS (Corvus Concept)
71h	Constellation II Pascal IV.x
72h	CP/M 68
73h	NCI p-system
74h	Softech p-system IV.1
75h	Apple ProDOS
76h	Apple MacIntosh
77h	UNIX System 5
78h	Apple II CP/M

80n-8Fh Gateways

80h	Generic gateway
81h	SNA gateway

90h-9Fh Reserved.

A0h-A8h Z80 based utility servers

A0h	Generic Utility Server II server
A1h	Enhanced print service
A2h	Simple pipes bridge

A9h-AFh Reserved for future servers

B0h-FEh Reserved for future use

FFh Any device.

CONSTELLATION BOOT NUMBER ASSIGNMENTS

Boot number	Computer type
0, 1, 2, 3	Apple II
4, 5, 6, 7	Concept
9	IBM
10	Xerox 820
11	Zenith H89
12	NEC PC8000
13	Pet
14	Atari 800
15	TRS-80 MOD I
16	TRS-80 MOD III
17	LSI-11
18	Printer server
19	Apple ///
20	DEC Rainbow
21	TI Pro
22	Z-100
23	Concept2
24	Companion
25	MacIntosh
26	Sony SMC-7086

SUMMARY OF DISK COMMANDS IN NUMERICAL ORDER

Command	Code:Modifier	Number of Data Bytes Sent	Number of Data Bytes Received
Read Sector (256 bytes)	02h	4	257
Write Sector (256 bytes)	03h	260	1
Semaphore Lock	0Bh:01h	10	12
Semaphore Unlock	0Bh:11h	10	12
Get Drive Parameters	10h	2	129
Prep Mode Select	11h	514	1
Park heads (Rev H)	11h	514	1
Read Sector (128 bytes)	12h	4	129
Write Sector (128 bytes)	13h	132	1
Boot	14h	2	513
Record Write	16h	2	1
Semaphore Initialize	1Ah:10h	5	1
Pipe Read	1Ah:20h	5	516
Pipe Write	1Ah:21h	x+5	12
Pipe Close	1Ah:40h	5	12
Pipe Status 1	1Ah:41h	5	513
Pipe Status 2	1Ah:41h	5	513
Pipe Status 0	1Ah:41h	5	1025
Semaphore Status	1Ah:41h	5	257
Pipe Open Write	1Bh:80h	10	12
Pipe Area Initialize	1Bh:A0h	10	12
Pipe Open Read	1Bh:C0h	10	12
Read Sector (256 bytes)	22h	4	257
Write Sector (256 bytes)	23h	260	1
Read Sector (512 bytes)	32h	4	513
Write Sector (512 bytes)	33h	516	1
AddActive	34h:03h	18	2
DeleteActiveUsr (Rev B/H)	34h:00h	18	2
DeleteActiveUsr (Omnidrive)	34h:01h	18	2
DeleteActiveNumber(Omnidrive)	34h:00h	18	2
FindActive	34h:05h	18	17
Read Sector (1024 bytes)(Bank)	42h	4	1025
Write Sector (1024 bytes)(Bank)	43h	1028	1
Read Boot Block	44h	3	513
Park heads (Omnidrive)	80h	1	1
WriteTempBlock	B4h	514	1
ReadTempBlock	C4h	2	513
Echo (Omnidrive/Bank)	F4h	513	513

RETURN CODES FOR REV B/H DRIVES

The disk return code is a byte. The bits are interpreted as shown below:

Bit #	Meaning
bits 4-0	Error code (see below).
bit 5	1=recoverable error.
bit 6	1=verify error.
bit 7	1=hard error.

Error code	Meaning
0 00h	Header fault.
1 01h	Seek timeout.
2 02h	Seek fault.
3 03h	Seek error.
4 04h	Header CRC error.
5 05h	Rezero fault.
6 06h	Rezero timeout.
7 07h	Drive not online.
8 08h	Write fault.
9 09h	Unused.
10 0Ah	Read data fault.
11 0Bh	Data CRC error.
12 0Ch	Sector locate error.
13 0Dh	Write protected.
14 0Eh	Illegal sector address.
15 0Fh	Illegal command op code.
16 10h	Drive not acknowledged.
17 11h	Acknowledge stuck active.
18 12h	Timeout.
19 13h	Fault.
20 14h	CRC.
21 15h	Seek.
22 16h	Verification.
23 17h	Drive speed error.
24 18h	Drive illegal address error.
25 19h	Drive r/w fault error.
26 1Ah	Drive servo error.
27 1Bh	Drive guard band.
28 1Ch	Drive PLO error.
29 1Dh	Drive r/w unsafe.

The error codes on the previous page have significance only if one or more of bits 5, 6, or 7 are also on. The table below allows you to easily convert the disk result code into an error code. Bits 5 and 6, or both, are set whenever a soft error occurs. For a hard error, bit 7 is always set, and bits 5 and 6 may be set. For example, if the disk return code is 87h, then there is a hard error, and the error code is 07h, Drive not online.

	Soft error		Hard error		
Error code	bit 5	bit 6	bit 7	bit 5,7	bit 6,7
0 00h	32 20h	64 40h	128 80h	160 A0h	192 C0h
1 01h	33 21h	65 41h	129 81h	161 A1h	193 C1h
2 02h	34 22h	66 42h	130 82h	162 A2h	194 C2h
3 03h	35 23h	67 43h	131 83h	163 A3h	195 C3h
4 04h	36 24h	68 44h	132 84h	164 A4h	196 C4h
5 05h	37 25h	69 45h	133 85h	165 A5h	197 C5h
6 06h	38 26h	70 46h	134 86h	166 A6h	198 C6h
7 07h	39 27h	71 47h	135 87h	167 A7h	199 C7h
8 08h	40 28h	72 48h	136 88h	168 A8h	200 C8h
9 09h	41 29h	73 49h	137 89h	169 A9h	201 C9h
10 0Ah	42 2Ah	74 4Ah	138 8Ah	170 AAh	202 CAh
11 0Bh	43 2Bh	75 4Bh	139 8Bh	171 ABh	203 CBh
12 0Ch	44 2Ch	76 4Ch	140 8Ch	172 ACh	204 CCh
13 0Dh	45 2Dh	77 4Dh	141 8Dh	173 ADh	205 CDh
14 0Eh	46 2Eh	78 4Eh	142 8Eh	174 AEh	206 CEh
15 0Fh	47 2Fh	79 4Fh	143 8Fh	175 AFh	207 CFh
16 10h	48 30h	80 50h	144 90h	176 B0h	208 D0h
17 11h	49 31h	81 51h	145 91h	177 B1h	209 D1h
18 12h	50 32h	82 52h	146 92h	178 B2h	210 D2h
19 13h	51 33h	83 53h	147 93h	179 B3h	211 D3h
20 14h	52 34h	84 54h	148 94h	180 B4h	212 D4h
21 15h	53 35h	85 55h	149 95h	181 B5h	213 D5h
22 16h	54 36h	86 56h	150 96h	182 B6h	214 D6h
23 17h	55 37h	87 57h	151 97h	183 B7h	215 D7h
24 18h	56 38h	88 58h	152 98h	184 B8h	216 D8h
25 19h	57 39h	89 59h	153 99h	185 B9h	217 D9h
26 1Ah	58 3Ah	90 5Ah	154 9Ah	186 BAh	218 DAh
27 1Bh	59 3Bh	91 5Bh	155 9Bh	187 BBh	219 DBh
28 1Ch	60 3Ch	92 5Ch	156 9Ch	188 BCh	220 DCh
29 1Dh	61 3Dh	93 5Dh	157 9Dh	189 BDh	221 DDh

RETURN CODES FOR OmniDrive/BANK

Value		Meaning
0	0h	No error.
131	83h	Seek error.
36	24h	Soft sector header error.
132	84h	Hard sector header error.
135	87h	Drive not ready.
136	88h	Write fault.
43	2Bh	Soft CRC error (data).
139	8Bh	Hard CRC error (data).
142	8Eh	Illegal sector address.
143	8Fh	Illegal opcode.
157	9Dh	Format firmware track failure.
158	9Eh	No tape inserted.
159	9Fh	Cannot read boot block.

ACTIVE USER TABLE ERRORS

Value	Meaning
0	No error.
1	No room in active user table.
2	Duplicate name in active user table.
3	User not found in active user table.

BOOT COMMAND ERRORS

Value	Meaning
4	Drive is not initialized (Const II).

PIPE STATES

bit #	Meaning
bit 7	1=contains data / 0=empty
bit 1	1=open for read
bit 0	1=open for write

PIPE ERRORS

Value	Meaning
0 00h	No error.
8 08h	Tried to read an empty pipe.
9 09h	Pipe not open for read or write.
10 0Ah	Tried to write to a full pipe.
11 0Bh	Tried to open an open pipe.
12 0Ch	Pipe does not exist.
13 0Dh	Pipe buffer full.
14 0Eh	Illegal pipe command.
15 0Fh	Pipes area not initialized.

SEMAPHORE STATES

Value	Meaning
0 00h	Semaphore not set.
128 80h	Semaphore set.

SEMAPHORE ERRORS

Value	Meaning
0 00h	No error.
253 FDh	Semaphore table full.
254 FEh	Semaphore table read-write error.
255 FFh	Unknown error.

TRANSPORTER RESULT CODES

Value		Meaning
0	00h	No error.
<64	<40h	Node identification number resulting from an Initialize or Who Am I command.
<128	<80h	Transmit retry count.
128	80h	Transmit failure (retry count exceeded).
129	81h	Transmitted messages user data portion was too long for the receiver's buffer.
130	82h	Message was sent to an uninitialized socket.
131	83h	Transmitted message control portion length did not equal receive socket's control buffer length.
132	84h	Invalid socket number in command vector (must be 80h, 90h, A0h, or B0h).
133	85h	Receive socket in user.
134	86h	Invalid destination node number in command vector. (must be 00-3Fh or FFh).
192	C0h	Received an ACK for an Echo command.
254	FEh	Socket set up successfully.

Transporter command summary

Send message

Command vector

Byte	Contents
0	Command code = 40h
1	Result record address
4	Destination socket
5	Data address
8	Data length
10	User control length
11	Destination host

Result record

Byte	Contents
0	Return code
1	Unused
4	User control info

Setup receive

Command vector

Byte	Contents
0	Command code = F0h
1	Result record address
4	Socket number
5	Data address
8	Data length
10	User control length

Result record

Byte	Contents
0	Return code
1	Source host
2	Unused
4	User control info

End receive

Command vector

Byte	Contents
0	Command code = 10h
1	Result record address
4	Socket number

Result record

Byte	Contents
0	Return code

Initialize

Command vector

Byte	Contents
0	Command code = 20h
1	Result record address

Result record

Byte	Contents
0	Return code

Who am I

Command vector

Byte	Contents
0	Command code = 01h
1	Result record address

Result record

Byte	Contents
0	Return code

Echo

Command vector

Byte	Contents
0	Command code = 02h
1	Result record address
4	Destination node

Result record

Byte	Contents
0	Return code

OMNIDRIVE AND REV B/H DRIVES | C

This appendix describes the differences between the OmniDrive and the Rev B/H drives:

Physical Characteristics:

The OmniDrive has 18 sectors per track while Rev B/H drives have 20 sectors per track.

Firmware Layout:

The OmniDrive firmware area is arranged differently from that of Rev B/H. Refer to Appendix A for details; the differences are summarized below:

The firmware block number ranges from 0 to 35 for OmniDrive. Rev B/H drives use physical head/sector number.

The sparing information for the OmniDrive is recorded in block 0 of the firmware. The Rev B/H drive records information in block 1. OmniDrive allows variable number of spare tracks for different drives.

Prep Mode:

In Prep mode, the OmniDrive turns on FAULT and READY LEDs; the Rev B/H turns on BUSY LED.

OmniDrive can accept up to four prep blocks. Rev B/H accepts only one.

OmniDrive formats with a FFH pattern. A specific fill command has to be sent to have a different pattern written.

Read-Write:

Read after write is an option selectable in the diagnostic program.

Sector addressing scheme has been changed to support 24-bit address.

Parking:

OmniDrive implements parking as a firmware command (80h). Rev B/H requires a special prep block.

Omninet Device Type:

The OmniDrive has a new Omninet device type (device 6). This device type is returned to a Who Are You command.

Constellation Support:

A new DeleteActiveNumber command is provided to delete all active users with the same host number. This command is currently not supported in Rev B/H drives.

OmniDrive does not have Constellation parameters to support the multiplexer.

Virtual drives are not supported. To replace the virtual table, a new sector address scheme is implemented (24 bit address).

The OmniDrive supports the new Constellation Disk Server Protocol as well as the existing version. Refer to Chapter 2 for details.

Pipes And Sempahores:

Pipes tables (pointer and name) are located in the firmware area of OmniDrive. Rev B/H pipes tables are stored in the pipes area.

Pipe tables are resident in RAM at all time. They are written to the disk when a pipe is closed after write or when the drive is put in prep mode.

Pipe read-write only works with 512 bytes of data even though the interface stays the same.

Wild card character (NUL) is supported in semaphore and pipe operations.

OmniDrive semaphore table is not saved. It is resident in RAM all the time. It is destroyed when the drive is powered off.

THE APPLE II TRANSPORTER

The Apple II communicates with its transporter by first formatting a command vector and then sending the command vector address to the transporter through the use of one control register. This control register is referred to as the Command Address Register (CAR). When the command is completed, a return code is placed in the result record. The address of the result record is specified in the command vector.

The CAR is an 8-bit register. Its address is determined by the slot in which the transporter is installed as shown in the chart below. Apple II I/IO space is memory mapped so the addresses below are normal memory addresses and not I/O addresses.

SLOT NUMBER	CAR ADDRESS Hexadecimal	Decimal	Decimal
1	C090	49296	-16240
2	C0A0	49312	-16224
3	C0B0	49328	-16208
4	C0C0	49344	-16192
5	C0D0	49360	-16176
6	C0E0	49376	-16160
7	C0F0	49392	-16144

When set, this bit indicates that the transporter is ready to receive the next address byte of the three byte command vector address. To issue a command to the transporter, this address must be given to the transporter one byte at a time. Every time an address byte is placed into the CAR, the RDY bit of the CAR goes low and the next byte cannot be sent until the RDY bit returns high again.

The three byte address is sent with the most significant byte first. For the Apple II the first byte is always zero since the Apple II address space only requires two address bytes.

Software Notes

While the transporter is receiving a packet from the network it cannot process a byte moved into the CAR, so the RDY bit of the CAR remains low until the transporter can process the next byte. This leads to a situation where a software I/O driver may have to wait up to several milliseconds before the RDY goes high again.

Since the Apple II processor does not support interrupts, the communication program should periodically check the return code for a change in value. As it is conceivable, though highly improbable, that the transporter could be modifying the return code at the same moment as the processor is viewing it, the processor should check the code a second time after detecting a change. This will insure that the processor sees the correct code value rather than a mid-change garble.

Until the command has completed as indicated by the return code, no additional data should be placed into the CAR by the sending computer. This is because the transporter will only process one command at a time.

The Apple II transporter is unbuffered. Data transfers with host memory take place through DMA and do not disturb the processor. There is no DMA overrun detection circuitry on board the Apple II transporter card because host memory is sufficiently fast that it is not needed.

An onboard boot ROM is provided with the Apple II transporter.

THE CONCEPT TRANSPORTER

The Concept transporter is a normal DMA transporter which supports interrupts. Interrupts arrive at priority three. After an interrupt arrives, the host must reset the interrupt mechanism before another interrupt can happen. Interrupts are reset when the processor performs a write operation to any address between 030FC1 and 030FDF. The contents of the write are unimportant.

A potential problem exists when several transporter operations are pending concurrently. If two commands complete within a short time of each other it is possible that the processor will not have a chance to reset the interrupt mechanism between the two command completion interrupt. To avoid this eventuality, the processor should check the values of all the outstanding return codes before returning from the interrupt subroutine. If any of these return codes indicates that the associated command has also completed, the processor can then take appropriate action.

Concept I/O space is memory mapped so all I/O addresses are simply standard memory addresses. This includes those given above for interrupt resets.

To issue a command to the transporter, the processor must write the command vector address, byte to byte, to any address between 030FA1 and 030FBF. Between each byte write, the processor must check the transporter READY bit. This is bit 0 of address 030F7F. Bit 0 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

A boot ROM is included on board the Concept transporter.

THE IBM PC TRANSPORTER

The IBM PC transporter is a buffered transporter which does not support interrupts. There is a boot ROM on board which extends from host CPU address DF000 to address E0000. The ROM utilizes the first 1024 bytes of the 4K buffer RAM and must have exclusive use of this area. The host should not place command vectors or other command information in this section of this buffer.

All processor read and write operations from and to the PC transporter take place through the I/O ports. The following is a list of the possible processor actions and the I/O ports to which they should be directed.

Operation	I/O Port
Read Transporter Status Byte	0248
Read RAM	0249
Read RAM; then Increase the Counter by 1	024B
Write to the CAR	0249
Write the Counter High Byte	0248
Write the Counter Low Byte	024A
Write to RAM; then Increase the Counter by 1	024B

All read and write operations directed at the RAM occur at the address to which the counter is currently pointing. The counter is subsequently increased only for those commands which specify a post-increment.

Bit 7 of the transporter status byte is the READY signal.

Bit 7 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

Rom Services

There are four separately executable routines contained within the IBM transporter onboard ROM. Each routine is initiated by a standard 8086 intersegment long CALL to one of the four ROM entry points. The four routines and their entry points are as follows:

```
COLDSTART - DF000
WARMSTART - DF003
I/O       - DF006
DUMMYRET  - DF009
```

The COLDSTART routine initializes the transporter card, locates a disk server on the network, loads the Constellation II boot program from the disk and transfers control to that program.

The WARMSTART routine initializes the transporter card.

The I/O routine performs one of a number of services depending on the contents of the AH register at the time the routine is entered. The I/O services are discussed in detail below.

The DUMMYRET routine performs a dummy interrupt return.

I/O Services

There are six I/O services. The contents of the 8086 AH register at the time of entry to the I/O routine determines which service is performed. However, before any I/O service is requested, the host must call the WARMSTART routine. The I/O services will not function until the WARMSTART routine has been executed. COLDSTART calls WARMSTART, though, so the host need not make a separate WARMSTART call if the host used the ROM to boot. Each I/O service is described below.

- Identify Interface: (AH) = 00

  Contents of 8086 registers on entry:

  (AH) = 00

  Contents of 8086 registers on exit:

(AL) = 00
(AH) = OMNINET node number of the transporter (if the node number is unique)
= FF (if a second transporter exists on the network with the same node number)

- Transmit Data to the Drive and Accept a Response: (AH) = 01

  Contents of 8086 registers on entry:

(AH)	= 01
DS: (SI)	= address of data to send to drive
ES: (DI)	= address of buffer to receive data from drive.
(CX)	= number of data bytes to send to the drive (maximum = 530)
(DX)	= number of bytes expected back from the drive excluding the return code (maximum = 530)
(AL)	= network address of disk server
(BL)	= number of timer units to wait for a reply from the disk server. 00 = do not abort; wait forever. (a timer is approximately .86 seconds)
(BH)	= number of transmit tries. 00 = 255 tries. FF = try until successful. Should be greater than 0.

  Contents of 8086 registers on exit:

  (AL) = return code from the drive. FF = aborted.
  (CX) = number of bytes received from the drive including the return code.

- Transmit Data to a Network Server: (AH) = 02

  Contents of 8086 registers on entry:

(AH)	= 02
ES: (DI)	= address of data to transmit
(CX)	= number of data bytes to transmit
(AL)	= network address of server. FF = broadcast to all servers.
(BH)	= number of transmit tries. For broadcasts, (BH) = number of times to transmit the data.

  Contents of 8086 registers on exit:

```
    (AL) = 00 (transmit successful)
         = FF (transmit aborted)
```

o Transmit Data to a Network Server and Accept a Response:
(AH) = 03

Contents of 8086 registers on entry:

```
    (AH)      = 03
    DS: (SI)  = address of data to transmit
    ES: (DI)  = address of buffer to receive data from
                server.
    (CX)      = number of data bytes to transmit
                (maximum = 530)
    (DX)      = number of data bytes expected from the
                server (maximum = 530)
    (AL)      = network address of server.  FF = broadcast
                to all servers.
    (BL)      = number of timer units to wait for a reply
                from the server.  00 = do not abort; wait
                forever.  (a timer unit is approximately
                .86 seconds)
    (BH)      = number of transmit tries.  00 = 255 tries.
                Should be greater than 0.
```

Contents of 8086 registers on exit:

```
    (AL) = 00 (transmit successful)
         = FF (transmit aborted)
```

o Find any Disk Server on the Network: (AH) = 04

Contents of 8086 registers on entry:

```
    (AH) = 04
    (BL) = number of timer units to wait for a reply from
           a disk server.  00 = do not abort; wait forever.
           (a timer unit is approximately .86 seconds)
    (BH) = number of tries.  00 = 255 tries.  Should be
           greater than 0.
```

Contents of 8086 registers on exit:

```
    (AL) = 00 (operation successful)
         = FF (operation unsuccessful)
    (AH) = network address of the disk server that
           responded.
```

o Send a Write Command to the Drive: (AH) = 05

Contents of 8086 registers on entry:

```
(AH)      = 05
DS: (SI)  = address of command block to send to drive
ES: (DI)  = address of data to send to drive
(CX)      = length of command block in bytes
            (normally 4)
(DX)      = number of data bytes to send to the drive
            (normally 512)
(AL)      = network address of disk server
(BL)      = number of timer units to wait for a reply
            from the disk server.  00 = do not abort;
            wait forever.  (a timer unit is
            approximately .86 seconds)
(BH)      = number of transmit tries.  00 = 255 tries.
            Should be greater than 0.
```

Contents of 8086 registers on exit:

```
(AL) = return code from the drive.  FF = aborted.
(CX) = number of bytes received from the drive
       including the return code.
```

THE NC-TRANSPORTER

The NC-Transporter is a buffered transporter which functions with both the 8001 and 8801 NEC microcomputers. When used with an 8001 it should be plugged into an 8031 expansion box.

The NC-Transporter has a 2K boot ROM on board which occupies addresses 00000h to 03FFFh and kills the microcomputer internal ROM when enabled. The ROM can be software enabled by setting bit 5 of I/O port 97 high, or, at reset time, by selecting the auto boot option with the jumpers. For more information see the NC-Transporter Installation Guide.

The NC-Transporter also supports interrupts. The interrupt level can be selected using the transporter jumpers. Information on the jumpers is available in the NC-Transporter Installation Guide. To enable the interrupt facility, the processor must set bit 4 of I/O port 97 high. The processor can check interrupt status by examining bit 4 of I/O port 97.

As the NC-Transporter is buffered there is no need for DMA overrun detection circuitry.

All processor read and write operations from and to the H-89 transporter take place through the I/O ports. The following is a list of the possible processor actions and the I/O ports to which they should be directed.

Operation	I/O Port
Read transporter Status Byte	97
Read RAM	96
Read RAM; then Increase the Counter by 1	94
Write to the CAR	96
Write the Counter High Nibble, the Interrupt Enable Bit, and the Boot ROM Enable Bit	97
Write the Counter Low Byte	95
Write to RAM; then Increase by Counter by 1	94

All read and write operations directed at the RAM occur at the address to which the counter is currently pointing. The counter is subsequently increased only for those commands which specify a post-increment.
As is clear from the list above, port 97 is used for a number of different operatons. A clearer understanding of the structure of port 97 may be gained from the diagram below.

PORT 97

```
+------+------+------+------+------+------+------+------+
|      |      |      |      |      |      |      |      |
|READY |      | ROM  | INT  |      High Nibble of Address|
| BIT  |      |ENABLE|ENABLE|      |Counter      |      |
|      |      |      |      |      |      |      |      |
+------+------+------+------+------+------+------+------+
 Bit 7  Bit 6  Bit 5  Bit 4  Bit 3  Bit 2  Bit 1  Bit 0
```

As shown above, bit 7 of the transporter status byte (port 97) is the READY signal. When port 97 is read, bit 7 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR. When writing to port 97, the value to which bit 7 is set is unimportant.

THE VT-180 TRANSPORTER

The VT-180 transporter is a normal DMA transporter which supports interrupts. Interrupts may be enabled by setting bit 0 of address EE high. The CPU must also be running in interrupt mode 0. After an interrupt occurs, the RESTART 8 command should be issued to the CPU.

The command address register on the VT-180 card lies at I/O address EF. Command vector address bytes must be written to this address.

The status port of the VT-180 card lies at I/O address EE. Bit 7 of this byte is the transporter READY line. Bit 7 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

THE SONY TRANSPORTER

The Sony transporter is a buffered transporter which has an interrupt status bit that can be checked when line time 60 HZ interrupt occurs (or other interrupts).

All processor read and write operations from and to the Sony transporter take place through the I/O port 4CH-4FH. The following is a list of the possible processor actions and the I/O ports to which they should be directed.

Operation	I/O Port HEX
Read transporter status byte	4F
Read RAM	4E
Read clears interrupt status bit	4D
Read buffer RAM; then Increment the Counter	4C
Write to the command address register (CAR)	4E
Write the counter high byte	4F
Write the counter low byte	4D
Write the RAM; then Increment the Counter	4C

All read and write operations directed at the RAM occur at the address to which the counter is currently pointing. The counter is subsequently incremented only for those commands which specify a post-increment.

Bit 7 of the transporter status byte is the READY signal. Bit 7 high indicates that the transporter is ready to accept the next byte of the command vector address into CAR.

Bit 4 of the transporter status byte is the INT STATUS (interrupt status) and is set upon completion of each transporter command. This bit is cleared by reading port 4D.

THE UNIVERSAL BUFFERED TRANSPORTER

The Universal Buffered Transporter (UBT) is a basic buffered transporter upon which many buffered transporters for specific microcomputers are built.

There is no DMA overrun detection circuitry on board the UBT and in fact no overrun detection circuitry on any buffered transporter. Buffered transporters perform their DMA operations on the buffer RAM which by design is sufficiently fast that no overruns can occur.

The UBT does not support interrupts and there is no boot ROM on board.

All processor read and write operations from and to the UBT take place through I/O ports. The two least significant port address bits for each operation are determined by the UBT. The upper 6 port address bits are defined by host-dependent circuitry.

The following is a list of the possible processor actions and the I/O ports to which they should be directed. In the table, "n" represents the upper six bits of the port address.

Operation	I/O Port
Read Transporter Status Byte	n3
Read RAM	n2
Read RAM; then Increase the Counter by 1	n0
Write to the CAR	n2
Write the Counter High Byte	n3
Write the Counter Low Byte	n1
Write to RAM; then Increase the Counter by 1	n0

All read and write operations directed at the RAM occur at the address to which the counter is currently pointing. The counter is subsequently increased only for those commands which specify a post-increment.

Bit 7 of the transporter status byte is the READY signal. Bit 7 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

THE Z-80 ENGINEERING TRANSPORTER

The Z-80 transporter is a normal DMA transporter which does not support interrupts. There is no boot ROM on board the Z-80 transporter but there is limited DMA overrun detection circuitry.

The command address register on the Z-80 card lies at I/O address F8. Command vector address bytes must be written to this address.

The status port of the Z-80 card lies at I/O address F9. Bit 4 of this byte is the transporter READY line. Bit 4 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

THE IBM PC-JR. TRANSPORTER

The IBM PC-Jr. transporter is a buffered transporter which does not support interrupts. There is a boot ROM on board which extends from host CPU address DF000 to address E0000. The ROM utilizes the first 1024 bytes of the 4K buffer RAM and must have exclusive use of this area. The host should not place command vectors or other command information in this section of this buffer.

All processor read and write operations from and to the PC-Jr. transporter take place through the I/O ports. The following is a list of the possible processor actions and the I/O ports to which they should be directed.

Operation	I/O Port HEX
Read Transporter Status Byte	3F8
Read RAM	3F9
Read RAM; then Increase the Counter by 1	3FB
Write to the CAR	3F9
Write the Counter High Byte	3F8
Write the Counter Low Byte	3FA
Write to RAM; then Increase the Counter by 1	3FB

All read and write operations directed at the RAM occur at the address to which the counter is currently pointing. The counter is subsequently increased only for those commands which specify a post-increment.

Bit 7 of the transporter status byte is the READY signal. Bit 7 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

THE Z-100 TRANSPORTER

The Z-100 transporter is a normal DMA transporter which supports interrupts. It is possible, using the jumpers and exposed pins on the Z-100 card, to select the level at which interrupts will arrive. For details see the Z-100 Installation and Programming Guide.

Once an interrupt arrives it is necessary for the interrupt handler software to reset the interrupt mechanism before returning control to the interrupted program. The interrupt is reset by writing to the Reset Interrupt Register at I/O port FB. The contents of the write are unimportant. If the interrupt is not reset, it will be impossible for the transporter to interrupt the processor again.

The Z-100 transporter has the facility to support an onboard boot ROM at IC location 7, but Corvus Systems does not supply a ROM. A user installed ROM is addressed using the phantom scheme to overlay an area of memory. The user selects this address space by using jumpers E2 through E5. The chart below shows how to select the phantom address.

Memory Address Bit	15	14	13	12	11-0
Jumper	E3	E4	E2	E5	No Jumper
Default (0XXX)	0	0	0	0	X

jumper A-B = bit low   jumper A-C = bit high

For more information on how to access an onboard ROM, see the Z-100 Installation and Programming Guide.

The command address register on the Z-100 card lies at I/O address 5A. Command vector address bytes must be written to this address.

The status port of the Z-100 card also lies at I/O address 5A. The reason that this does not create confusion is that the host only writes to the CAR and only reads from the status port. The read/write line from the CPU determines which register is attached to the data lines.

Bit 0 of the status byte is the transporter READY line. Bit 0 high indicates that the transporter is ready to accept the next byte of the command vector address into the CAR.

The Z-100 transporter card includes DMA overrun detection circuitry.

THE RAINBOW TRANSPORTER

The Rainbow Transporter is of unbuffered type with no underrun/overrun support. This means that most communication with the transporter is done via DMA. A DMA cycle is guaranteed to start within 3.5 microseconds from a request so over/under run will never happen. The host passes command addresses, controls interrupts and RESET with the help of 2 I/O registers (address 22H-23H).

	bit 1		bit 1	bit 0
22H	RDY		RESET	IE
23H	CAR			

RDY - Transporter ready to accept one byte of a command address. (For restrictions see the Omninet Technical Reference Manual). Read only bit. Write operation does not have any effect on this bit.

IE - Interrupt enable. When set (=1) the transporter will interrupt the host as described in the Omninet Technical Reference Manual. It is cleared by reading in the CAR register. This bit is cleared on power up.

CAR - Command address register. For each Omninet command a three byte address is passed in this register (MSB first). Reading this register will clear interrupt requests.

RESET - When set, the RESET line to the generic transporter is held low and pending interrupts are cleared. This bit is cleared on power up. Interrupts must not be enabled until 50 micorseconds after reset cycle has been completed.

Interrupts

The Transporter supports the DMA Controller Interrupt normally used by the extended communication option. The interrupt is of type 23H and uses interrupt vector 3CH. An interrupt request is cleared by reading the CAR (address 23H).

LSI-11 TRANSPORTER

Jumpers And Switches

The LSI-11 OMNINET interface board, called a transporter, contains jumpers to select the LSI-11 control and status register (CSR) address, the interrupt vector address, and interrupt priority. There is also a jumper to enable/disable the bootstrap.

The transporter contains a Dip switch with eight microswitches. Microswitches 1-6 are used to set the unique OMNINET device address.

Microswitch number 7 is used to set a bias offset on the OMNINET cable to reduce the effect of noise on the line when it is idle. Exactly one device on the network should have this switch set on.

Microswitch number 8 is reserved for network termination. Normally, switch 8 is off for all transporters because terminators are physically installed at both ends of the network.

Bootstrap

The transporter board has a 256 word bootstrap area with a starting address of 773000. The bootstrap sockets accept two 256 x 8 proms compatible with MMI 6309-1J or TI 74S471. Location U23 contains the low order bytes and location U16 contains the high order bytes of the bootstrap code. When shipped, the bootstrap is enabled and contains the boot code for a DEC RL01 disk drive or the Corvus RL01 compatible disk system. The bootstrap can be disabled by removing the jumper between pins J8 and J13.

Device Address

The transporter hardware has support for a 20-bit address; However, an 18-bit address is normally used with Q-bus devices. The transporter contains jumpers to select bit 3 to bit 12 of the device CSR address. Pins used to set the CSR address are J1-J6 and J9-J12. Pin J7 is used as a ground. A jumper installed from an address pin to the ground pin results in a zero for that bit of the device address. Since there is a single ground pin, the jumpers are installed in a daisy-chained fashion. The CSR device address is preset to 766000 as shown in the chart that follows:

Bit	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01
Pin	1	1	1	1	1	J1	J2	J3	J4	J5	J6	J9	J10	J11	J12	0	0
766000	1	1	1	1	1	0	1	1	0	0	0	0	0	0	0	0	0

Bit 17-13 are implied ones and bits 2-0 are implied zeroes. To create the preset device address of 766000, pins J1 and J4-J12 must be jumped to the ground pin J7. This can be performed with the following jumpers: J1-J4, J4-J5, J5-J6, J6-J7, J9-J10, J10-J11, J11-J12, and J12-J7.

Programming Guide

Chapter three of the Omninet Local Area Network General Technical Information Guide describes the commands that can be used with the transporter. The LSI-11 communicates with the transporter by first formatting a command control block and then sending the command control block address to the transporter through the use of two control registers. When the command is completed, a return code is placed in the result record address as specified in the command control block. An interrupt is generated when the operation is completed. For a detailed description of commands, control block formats, and return codes see Chapter three

CSR - Control And Status Register

The Control and Status Register (CSR) is a 16-bit register with a standard address of 766000. All bits can be read or written.

Bit 0-6 Not used

Bit 7 Interrupt Enable (IE)

This bit is set to 1 upon power up and hardware reset. If this bit is cleared, the transporter cannot interrupt the processor.

Bit 8-14 Not used

Bit 15 Transporter Ready (RDY)

When set, this bit indicates the transporter is ready to receive the next address byte of the three byte command control block address. This bit is cleared when a byte is moved into the Command Address Register (CAR).

CAR - Command Address Register

The Command Address Register (CAR) is a 16-bit write-only register with a standard address of 7660002.

Bit 0-7 Command Address Byte

To issue a command to the transporter, the three byte address of the command control block must be given to the transporter one byte at a time. Every time an address byte is placed into the CAR, the RDY bit of the CSR goes low and the next byte cannot be sent until the RDY bit returns high again. The three byte address is sent with the most significant byte first.

Bit 8-15 Not used

Software Notes

While the transporter is receiving a packet from the network, it will not process a byte moved into the CAR so the RDY bit of the CSR remains low until the transporter can process the next byte. This leads to a situation where a software I/O driver may have to wait up to several milliseconds before the RDY goes high again. Since the transporter processes one command at a time, the computer should not place any additonal data into the CAR after it has issued a command, until the command has completed as indicated by the command return code.

Interrupts

An operation complete interrupt is generated after the completion of each command issued to the transporter. Before the interrupt is generated, a return code is placed in the address specified as the result record address in the command control block. Two interrupts are generated for a valid setup receive command. The first interrupt indicates the command was accepted and the socket is setup to receive a message. The second interrupt occurs when a message is received. The program should initialize the return code byte in the result record to hex $FF before the command code block is sent to the transporter. When a transporter interrupt occurs, the program must check the return code value of each active transporter command to determine which operation has just completed.

Byte Order

All OMNINET addresses and lengths must be specified with the most significant byte first and the least significant byte last. Additionally, some addresses and lengths are not on word boundaries.

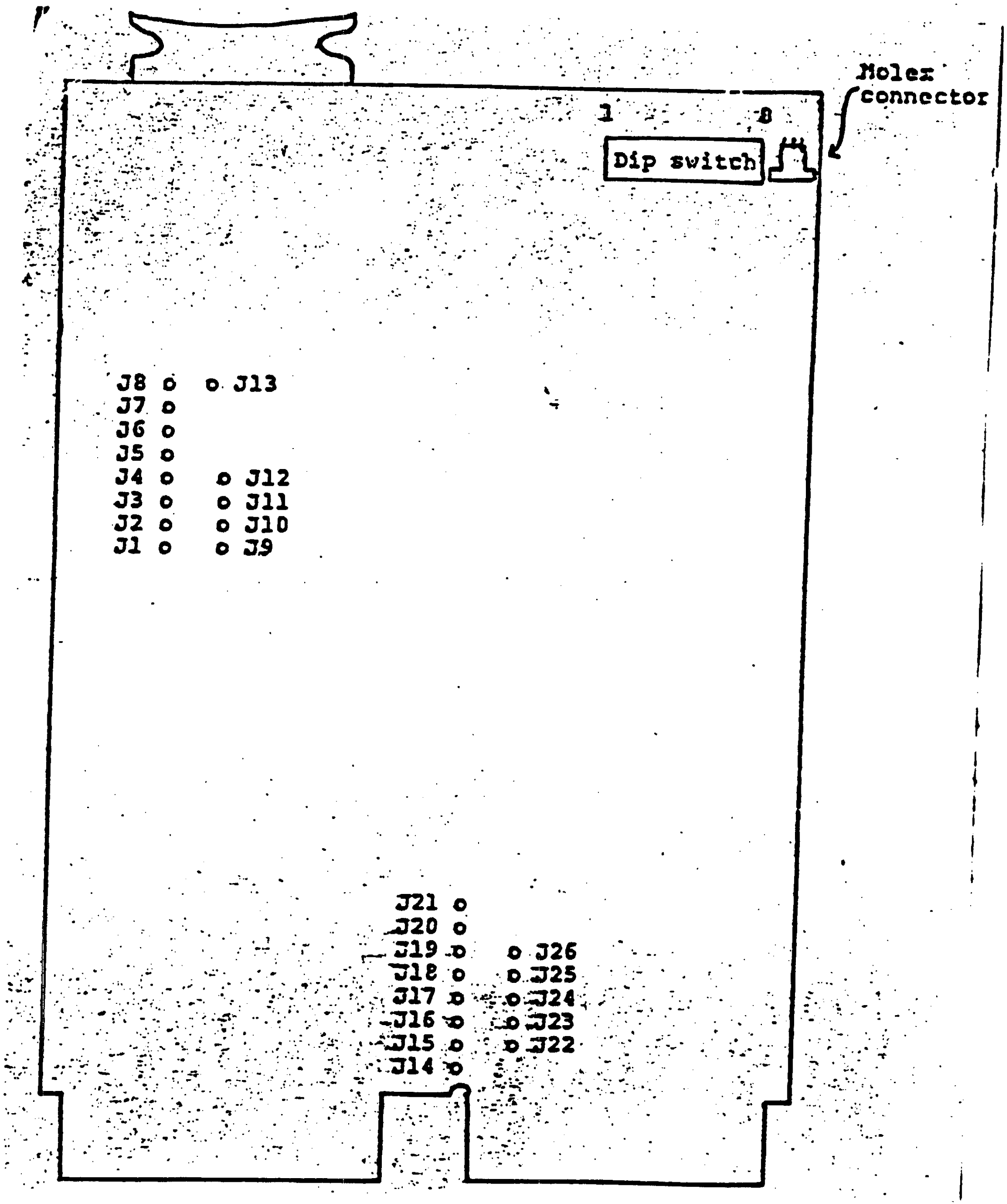

Figure D.1: LSI-11 Transporter Board Jumper Locations

This page intentionally left blank.

CORVUS FLAT CABLE INTERFACE CARDS | E

This appendix describes the flat cable interface provided by Corvus. It contains a table describing the flat cable interface cards, and gives listings of sample interface routines.

The table on the next page describes the flat cable interface cards provided by Corvus or other developers. See Appendix A for a description of the flat cable signal assignments, including READY and DIRC.

For each interface card, the table contains the following information:

1. The processor type (Z80, 8080, 6502, 8088, 8086).
2. Whether the I/O is memory mapped or through I/O ports.
3. The data port or memory address.
4. The status port or memory address.
5. Which bit (bit 7 is msb) of the status port is the READY line, and the value for READY.
6. Which bit of the status port is the DIRC line, and the value for Host-to-Drive.
7. Additional notes are given below.

Notes:

(1) Card contains space for a 2k PROM; card must be in slot 6
(2) Must output 1 to bit 6 of port 0ECh first
(3) Same card as TRS-80 I, except jumpered.
(4) Contains space for a PROM;
bit 2 - auto boot switch, bit 7 - power on
(5) Complex strobe.
(6) Complex bus direction control
(7) Card contains space for a 4k PROM
(8) Interface is through game ports 3 and 4.
(9) Not a Corvus product. The Alspa card was developed by Alspa; the LNW80 card was developed by an independent developer; the Magnolia Z-89 was developed by Magnolia Microsystems.

Computer	1. Processor Type	2. I/O Type	3. Data Port Address Hex/Dec	4. Status Port Address Hex/Dec	5. Ready Status bit #/value	6. H-t-D Status bit#/value	7. Notes
Alspa	Z80	I/O	D0h/208	D2h/210	0/1	1/1	(5,9)
Altos	Z80	I/O	81h/129	80h/128	0/0	1/0	
Atari 400/800	6502						(8)
Apple II	6502	Mem	C0E0h/ 49376	C0E1h/ 49377	7/0	6/0	(1)
DEC Rainbow	8088	I/O	20h/ 32	21h/ 33	0/0	1/1	
DEC Robin	Z80	I/O	DEh/222	DFh/223	0/0	1/1	
IBM PC	8088	I/O	2EEh/750	2EFh/751	0/0	1/1	(7)
LNW80	Z80	Mem	F781h/	F780h/	0/0	1/0	(3,9)
Magnolia Z-89	Z80	I/O	59h/ 89	58h/ 88	0/0	1/0	(4,9)
NEC	Z80	I/O	81h/129	80h/128	0/0	1/0	
Osborne O-1	Z80	Mem	(5)	(5)	6/0	7/1	(5)
S-100, Z80 ripoff	8080, Z80	I/O	DEh/222	DFh/223	0/0	1/1	
Sony SMC-70	Z80	I/O	48h/ 72	49h/ 73	0/0	1/1	
SuperBrain	Z80	I/O	81h/129	80h/128	0/0	1/0	
TRS-80 I	Z80	Mem	3781h/ 14209	3780h/ 14208	0/0	1/0	
TRS-80 II	Z80	I/O	DEh/222	DFh/223	0/0	1/1	
TRS-80 III	Z80	I/O	DEh/222	DFh/223	0/0	1/1	(2)
Xerox 820	Z80	I/O	08h/8	09h/9	0/0	1/1	(6)
Zenith H-89	Z80	I/O	7Ah/122	7Bh/123	0/0	1/1	
Zenith Z-90, Zenith Z-100	Z80 8085	I/O	7Eh/126	7Fh/127	0/0	1/1	

SAMPLE INTERFACE ROUTINE FOR 6502

```
; This section describes the source for the machine language program known
; as BCI.  BCI stands for Basic Corvus Interface; this program is used by
; the various Basic utilities to communicate with the Corvus drive.  The
; function of this program is to send one command to the Corvus interface,
; and then wait for a reply.  The parameters to BCI are used both as input
; (i.e., the length and command are passed in), and output (i.e., the lengt
; and result bytes of the reply are passed back in the input locations).
;
; Parameters to BCI are:
;    Length of command - this parameter is a word, and is passed
;        in locations 300,301 (hex; least significant byte first).
;        Length must always be greater than 0.
;    Address of buffer containing command - this parameter is a word,
;        and is passed in locations 302,303 (hex; least significant byte
;        is first).
; Entry point to BCI is 304 (hex).
; BCI is NOT relocatable; it loads at 300 (hex).
; Uses the DMA buffer address location at 48,49 (hex)
; Assumes that the CORVUS card is in slot 6.

        .ABSOLUTE
        .TITLE "BCI  Copyright 1981, All rights reserved, Corvus Systems, I
        .PROC BCI

LEN     .EQU   0300     ; length of command
BUF     .EQU   0302     ; address of data buffer containing command

RENBL   .EQU   0C0E2    ; read strobe
STATUS  .EQU   0C0E1    ; status byte
DATA    .EQU   0C0E0    ; input/output line
DMABUF  .EQU   48       ; DMA buffer location


START   .ORG   0304
        LDA RENBL       ; enable read strobe

; initialize byte count, DMA index

        LDA BUF
        STA DMABUF
        LDA BUF+1
        STA DMABUF+1
        LDY #0

; send command to drive

        LDX LEN
        BNE STEST1
OUTL    DEC LEN+1       ; count down upper byte of length
STEST1  BIT STATUS      ; wait for drive to be ready
        BMI STEST1
```

```
        LDA @DMABUF,Y    ; send byte to drive
        STA DATA
        INY              ; get next byte
        BNE NEXT1        ; check for 256 byte rollover
        INC DMABUF+1
NEXT1   DEX
        BNE STEST1
        LDA LEN+1
        BNE OUTL

; done with sending command, now wait for line to turn around

TEST2   BIT STATUS       ; read status bit
        BVC TEST2        ; wait for bus to turn around
        BMI TEST2        ; wait for "ready" bit

        LDY #10          ; delay loop to avoid "ready" glitch
LOOP1   DEY
        BNE LOOP1

        BIT STATUS       ; check it again, just to be sure
        BVC TEST2
        BMI TEST2

; now receive the result

        LDA #0           ; initialize returned byte count
        STA LEN
        STA LEN+1
        LDA BUF          ; reset DMA address
        STA DMABUF
        LDA BUF+1
        STA DMABUF+1

STEST3  BIT STATUS
        BVC DONE         ; exit if "host to drive"
        BMI STEST3

        LDA DATA         ; read byte from controller
        STA @DMABUF,Y    ; save in memory buffer
        INY
        BNE STEST3       ; check for 256 byte rollover
        INC DMABUF+1
        BNE STEST3       ; keep looping until exit

; compute address of end of received data+1, then subtract starting address
; to get total number of bytes received

DONE    TYA
        CLC
        ADC DMABUF
        PHA
        LDA DMABUF+1
```

```
ADC #0
STA DMABUF+1
PLA
SEC
SBC BUF
STA LEN
LDA DMABUF+1
SBC BUF+1
STA LEN+1
RTS
.END
```

SAMPLE INTERFACE ROUTINE 8080/Z80

```
; --- UTILITY DRIVER FOR CORVUS CP/M PROGRAMS WITH PASCAL MT+ ---
;               using the FLAT CABLE interface cards
;
;                  ( MICROSOFT M80 ASSEMBLER FORMAT )
;
;                          BY KO & BRK
;
;
; THIS UNIT IMPLEMENTS  3  SUPPORT PROCEDURES AND FUNCTIONS
; FOR PASCAL MT+ :
;
;  INITIO        - init corvus drivers and return "bios" pointer
;  SEND          - send data to corvus drive
;  RECV          - receive data from corvus driver
;
;       THESE ARE EXPLAINED BELOW:
;
;               function INITIO
;
;       Calling the function does some initialization of the
;       driver.  This function MUST be called once and only once
;       before any use of the SEND or RECV routines is attempted.
;
;       FOR  SEND  AND  RECV  THE CALLING PROCEDURE IN PASCAL IS:
;
;               SEND (VAR st : LONGSTRING )
;
;       The first two bytes of the string are the length
;       of the string to be sent or the length of the
;       string received.  Typically one first uses  SEND
;       to send a string to the drive then follows this with a  RECV
;       command to get back any returned data from the CORVUS drive.
;
;
;       NOTE: These drivers are not necessarily implemented in the
;             fastest or most direct way.  The MT+ programs are
;             so slow that speed here is not the overriding concern.
;
; ================================================================
;
        public INITIO, RECV, SEND
        .8080           ;8080 opcodes
;
;
;
; ---- SYSTEM TYPE DESIGNATORS ---
;
H89     EQU     1       ; ZENITH H89 SYSTEM
HRD     EQU     2       ; H89 R&D VERSION
SB      EQU     3       ; SUPERBRAIN SYSTEM
ALTOS   EQU     3       ; ALTOS SYSTEM DESIGNATOR
```

```
S100     EQU     4          ; S100 SYSTEM DESIGNATOR
TRS2     EQU     4          ; TRS-80 MODEL II DESIGNATOR
APPLE    EQU     5          ; APPLE CPM DESIGNATOR
XRX      EQU     6          ; XEROX CPM DESIGNATOR
ALSPA    EQU     7          ; ALSPA CPM DESIGNATOR
MAGNOLIA EQU     8          ; MAGNOLIA Z-89 DESIGNATOR
OS1      EQU     9          ; OSBORNE O-1 DESIGNATOR
OSX1     EQU     10         ; OLD EXPERIMENTAL OSBORNE VERSION
SNY70    EQU     11         ; SONY SMC-70 DESIGNATOR
ZS10     EQU     12         ; OLD Z100 WITH S-100 PORTS
ZS1      EQU     13         ; ZENITH Z-100 DESIGNATOR
LNW80    EQU     14         ; LNW80 II DESIGNATOR
;
;
; === SPECIFY SYSTEM TYPE HERE USING ABOVE DESIGNATORS ===
;
sys      EQU     ZS1        ; Designates sys
;
;
; --- SETUP EQUATES BASED UPON ABOVE DESIGNATOR CHOICE ---
;
OS1T     EQU     sys EQ OS1 OR sys EQ OSX1        ; true if OSBORNE
;
ZS1T     EQU     sys EQ ZS10 OR sys EQ ZS1        ; true if Z-100
;
         if sys EQ H89      ; IF SYSTEM IS H89 THEN
DATA     EQU     07AH       ; Controller data I/O port
STAT     EQU     07BH       ; Controller status port
HTDRDY   EQU     2          ; Host-To-Drive , Drive Ready status
DTHRDY   EQU     0          ; Drive-to-Host , Drive Ready status
         endif
;
         if sys EQ HRD      ; for H89 R&D INTERFACE
DATA     EQU     0D1H       ;
STAT     EQU     0D0H       ;
HTDRDY   EQU     0          ;
DTHRDY   EQU     2          ;
         endif
;
        if sys   EQ MAGNOLIA ; IF SYSTEM IS MAGNOLIA Z-89
DATA     EQU     059H       ; DATA INPUT PORT
STAT     EQU     058H       ; STATUS INPUT PORT
HTDRDY   EQU     0          ;
DTHRDY   EQU     2          ;
         endif
;
         if sys EQ SB OR sys EQ ALTOS ; IF SYSTEM IS SUPERBRAIN OR ALTOS
DATA     EQU     081H       ; DATA INPUT PORT
STAT     EQU     080H       ; STATUS INPUT PORT
HTDRDY   EQU     0          ;
DTHRDY   EQU     2          ;
         endif
;
```

```
         if sys EQ S100 ; for S100 type syss
STAT     EQU     0DFH     ;
DATA     EQU     0DEH     ;
HTDRDY   EQU     2        ;
DTHRDY   EQU     0        ;
         endif
;
         if sys EQ ALSPA; IF SYSTEM IS ALSPA THEN
DATA     EQU     0D0H     ;
STAT     EQU     0D2H     ;
HTDRDY   EQU     3        ; Host-To-Drive , Drive Ready status
DTHRDY   EQU     1        ;
;
COMDD    EQU     0D3H     ; COMMAND PORT
MODI     EQU     93H
MODO     EQU     83H
         endif
;
         if sys EQ XRX    ; XEROX 820 equates
DATA     EQU     08H      ;
PDATA    EQU     09H      ; Control of data port
STAT     EQU     0AH      ; status port
PSTAT    EQU     0BH      ; Control of status port
;
HTDRDY   EQU     02H      ; Host-To-Drive & Drive Ready status
DTHRDY   EQU      0H      ; Drive-To-Host & Drive Ready status
;
OTMODE   EQU     0FH      ; PIO output mode
INMODE   EQU     4FH      ; PIP input mode
CTLMODE  EQU     0CFH     ; PIO bit control mode
CTLMASK  EQU     0FH      ; mask for PIO when in CTLMODE
NOINT    EQU     7H       ; disable PIO interupts
;
OTDIS    EQU     30H      ; interface output mode, strobes disabled
INDIS    EQU     10H      ; interface input mode,  strobes disabled
OTEN     EQU     20H      ; interface output mode, strobes enabled
INEN     EQU      0H      ; interface input mode,  strobes enabled
         endif
;
         if sys NE APPLE AND NOT OS1T
DRDY     EQU     1        ; MASK FOR DRIVE READY BIT
DIFAC    EQU     2        ; MASK FOR DRIVE ACTIVE BIT
         endif
;
         if sys EQ APPLE ;Apple CP/M equates (Corvus card in slot #6)
DATA     EQU     0E0E0H   ;I/O data pointer
STAT     EQU     0E0E1H   ; I/O status pointer
DRDY     EQU     080H     ;Status -- Data ready flag
DIFAC    EQU     040H     ;Status -- Active flag
HTDRDY   EQU     0        ;Host-To-Drive ReaDY status
DTHRDY   EQU     040H     ;Drive-To-Host ReaDY status
;
Z$PU     EQU     0F3DEh   ;Pointer to SoftCard
```

```
A$VEC   EQU     0F3D0h  ;Pointer to 6502 subroutine address
A$ACC   EQU     0F045h  ;Pointer to 6502 A register
CWRIT6  EQU     0FBAh   ;6502 write data byte subr address
CREAD6  EQU     0FC9h   ;6502 read data byte subr address
         endif
;
         if sys EQ OS1 OR sys EQ OSX1    ; if OSBORNE O-1
;
ADATA   EQU     2900H    ; PORT A DATA/DIRECTION CONTROL
CTLA    EQU     2901H    ; PORT A REGISTER SELECT
BDATA   EQU     2902H    ; PORT B DATA/DIRECTION CONTROL
CTLB    EQU     2903H    ; PORT B REGISTER SELECT
;
STAT    EQU     BDATA    ; STATUS I/O PORT
DATA    EQU     ADATA    ; DATA I/O PORT
        endif
;
         if sys EQ OS1   ; standard Corvus OSBORNE version
HTDRDY  EQU     80H      ; Host-to-drive, ready status
DTHRDY  EQU     0        ; Drive-to-host
DRDY    EQU     40H      ; MASK FOR DRIVE READY BIT
DIFAC   EQU     80H      ; MASK FOR DRIVE ACTIVE BIT
         endif
;
         if sys EQ OSX1  ; old Corvus experimental Osborne version
HTDRDY  EQU     00H      ; Host-to-drive, ready status
DTHRDY  EQU     40H      ; Drive-to-host
DRDY    EQU     80H      ; MASK FOR DRIVE READY BIT
DIFAC   EQU     40H      ; MASK FOR DRIVE ACTIVE BIT
         endif
;
         if sys EQ SNY70 ; for SONY SMC-70
STAT    EQU     049H     ;
DATA    EQU     048H     ;
HTDRDY  EQU     2        ;
DTHRDY  EQU     0        ;
         endif
;
         if sys EQ ZS10  ; for ZENITH  Z-100 with S-100 ports
STAT    EQU     0DFH     ;   ( this only worked on OLD Z-100's )
DATA    EQU     0DEH     ;
HTDRDY  EQU     2        ;
DTHRDY  EQU     0        ;
         endif
;
         if sys EQ ZS1   ; for ZENITH  Z-100 ( std Corvus release )
STAT    EQU     07FH     ;
DATA    EQU     07EH     ;
HTDRDY  EQU     2        ;
DTHRDY  EQU     0        ;
         endif
;
         if sys EQ LNW80 ; for LNW80 ( using TRS-80 model 1 interface )
```

```
STAT    EQU     0F780H  ; MEMORY MAPPED PORT ADDRESS
DATA    EQU     0F781H  ;
HTDRDY  EQU     0       ;
DTHRDY  EQU     2       ;
          endif
;
;
; ====================================================================
;
; --- DEFINE MACROS FOR BASIC CORVUS OPERATIONS ---
;
;
;       INSTAT -- Get disk controller status subroutine
;
INSTAT  MACRO                   ; macro to choose how to get status
        if sys EQ APPLE OR sys EQ LNW80
        LDA     STAT            ; Get status if memory mapped I/O
        else
                if OS1T
        CALL    OSTAT           ; Get status if Osborn
                else
        IN      STAT            ; Get status if port I/O
                endif
         endif
        ENDM                    ;Return
;
;
;       TSTIN -- Set Z-flag if status = "Drive-To-Host", "Drive Ready"
;
TSTIN   MACRO                   ; macro for testing input status
         if sys EQ XRX OR OS1T  ; if system is XEROX

        ; or OSBORNE
        CALL    SETIN           ;  set port direction
         endif
        INSTAT                  ; get status
        ANI     DIFAC OR DRDY   ; mask status bits
        CPI     DTHRDY          ; set Z-flag if status is right
        ENDM                    ;
;
;
;       TSTOT -- Set Z-flag if status = "Host-To-Drive", "Drive Ready"
;
TSTOT   MACRO                   ; macro to test output status
         if sys EQ XRX OR OS1T  ; if XEROX or OSBORNE  system
        CALL    SETOT           ;   set port direction
         endif
        INSTAT                  ;get status
        ANI     DIFAC OR DRDY   ; mask status bits
        CPI     HTDRDY          ; set Z-flag if status is right
        ENDM                    ;
;
;
```

```
;       INDATA -- Get disk controller data subroutine
;        ( get a single byte back from controller )
;
INDATA  MACRO                      ; macro to chose how to get data
         if sys EQ APPLE
        PUSH    H                  ;Save (H,L)
        LXI     H,CREAD6
  ;Get 6502 read subr address
        CALL    X6502              ;Read data byte (6502)
        POP     H                  ;Restore (H,L)
        LDA     A$ACC              ;Get data byte
         else
            if sys EQ ALSPA OR OS1T
        CALL    INACV
            else
                if sys EQ LNW80
        LDA     DATA
                else
        IN      DATA
                endif
            endif
         endif
        ENDM                      ;Return
;
;
;       OTDATA -- Put disk controller data subroutine
;          ( output a single byte passes in Acc )
;
OTDATA  MACRO                      ; macro to chose how to output data
         if sys EQ APPLE
        STA     A$ACC              ;Put data byte
        PUSH    H                  ;Save (H,L)
        LXI     H,CWRIT6           ;Get 6502 write subr address
        CALL    X6502              ;Write data byte (6502)
        POP     H                  ;Restore (H,L)
         else
          if sys EQ ALSPA OR OS1T
        CALL    OUTACV
          else

     if sys EQ LNW80
        STA     DATA
                else
        OUT     DATA
                endif
          endif
         endif
        ENDM                      ;Return

; ===================================================================
;
; --- DUMMY INITIALIZATION ENTRY IF NOT OSBORNE ---
;
```

```
        if NOT OS1T     ; if not osborne
INITIO: RET             ; DUMMY
        endif
;
;
; --- RECEIVE BLOCK INTO BUFFER ---
;
RECV:   POP     H       ; get return address
        XTHL            ; put return addr back, get buf address
        PUSH    H       ; save buf address
        INX     H       ; point past length field in buf
        INX     H       ;
        CALL    TURN    ; WAIT FOR BUSS TO TURN AROUND
        MVI     B, 15   ;
        CALL    DELAY   ;
        CALL    TURN    ; SECOND try to avoid glitches ( mainly for mirror
        MVI     B, 15   ;
        CALL    DELAY   ;
        CALL    GTBLK   ; get block of bytes and put count on stack
        POP     B       ; get count
        POP     H       ; get buf address
        MOV     M,C     ; put lower byte of len field
        INX     H       ;
        MOV     M,B     ; put upper byte of len field
        RET             ;
;
; --- SEND block from buffer ---
;
SEND:   POP     H       ; get ret addr
        XTHL            ; put ret addr back, get buf address
        MOV     C,M     ; BC is WTBLK length counter
        INX     H       ; load with len field of buffer
        MOV     B,M     ;
        INX     H       ; HL points to bytes to send
        CALL    WTBLK   ; write bytes to drive
        RET             ;
;
;
        if      ZS1T    ; IF Z100
; --- SPECIAL WTBLK ENTRY FOR Z-100 ---
;
WTBLK:  MOV     A,M     ; GET BYTE FROM MEMORY
        CALL    WAITOX  ; SEND FIRST BYTE TO DRIVE
        JMP     WTBLK1  ; ENTER STANDARD LOOP
        else
;
; --- WRITE A BLOCK OF DATA TO THE DISC ---
;
WTBLK:
        endif
;
WTBLKL: MOV     A,M     ; GET BYTE FROM MEMORY
        CALL    WAITO   ; output byte
```

```
WTBLK1: INX     H
        DCX     B
        MOV     A,B
        ORA     C
        JNZ     WTBLKL  ; LOOP UNTIL DONE
        RET
;
; --- GET A BLOCK OF UNDETERMINED LENGTH BACK FROM DISC ---
;
GTBLK:   if sys EQ XRX OR OS1T   ; if XEROX OR OSBORNE system
        CALL    SETIN            ;   set port direction
         endif
        LXI     D,0              ; set counter
GTB1:   INSTAT                   ; GET STATUS
        MOV     C,A              ; SAVE IT
        ANI     DRDY             ; TEST IF READY
        CPI     HTDRDY AND DTHRDY
        JNZ     GTB1             ; LOOP UNTIL READY
;
        MOV     A,C
        ANI     DIFAC OR DRDY    ; mask status bits
        CPI     HTDRDY           ; if "Host-To-Drive" & Ready
        JZ      GTB2             ;  then jump out of loop
;
        INDATA                   ; GET DATA BYTE
        MOV     M,A              ; SAVE IT
        INX     H
        INX     D
        JMP     GTB1
;
GTB2:   XCHG                     ; GET COUNT IN (H,L)
        XTHL                     ; SAVE IT
        PCHL                     ; R
TURN
        page
;
;
TURN:   TSTIN                    ; Set Z-flag if "Drive-To-Host" & Ready
        JNZ     TURN             ; loop if not
        RET                      ;
;
DELAY:  DCR     B
        JNZ     DELAY
        RET
;
WAITI:  TSTIN                    ; Set Z-flag if DTHRDY
        JNZ     WAITI            ; LOOP UNTIL READY
;
        INDATA                   ; READ BYTE FROM DISC
        RET
;
         if OS1T                 ; SPECIAL OSBORNE VERSION
WAITO:  PUSH    PSW
```

```
        CALL    SETOT              ; setup for output
WAITO1: EI                         ; enable ints
        NOP
        CALL    OSTATX             ; READ STATUS PORT
        ANI     DIFAC OR DRDY      ; MASK STATUS BYTE
        CPI     HTDRDY
        JNZ     WAITO1
;
        POP     PSW
        CALL    OUTACV             ; OUTPUT DATA
        RET
         endif
;
         if sys NE SNY70 AND NOT ZS1T AND NOT OS1T AND sys NE LNW80
WAITO:  PUSH    PSW                ; SAVE COMMAND
WAITO1: TSTOT                      ; Set Z-flag if HTDRDY
        JNZ
WAITO1          ; LOOP UNTIL READY
;
        POP     PSW
        OTDATA          ; WRITE BYTE TO DISC
        RET
         endif
;
         if sys EQ SNY70 OR ZS1T OR sys EQ LNW80
WAITO:  PUSH    PSW                ; SAVE COMMAND
WAITO1: EI                         ; ENABLE INTERRUPTS FOR A SHORT TIME
        NOP
        NOP
        DI                         ; DISABLE INTERRUPTS FOR TEST
        TSTOT                      ; Set Z-flag if HTDRDY
        JNZ     WAITO1             ; LOOP UNTIL READY
;
        POP     PSW
        OTDATA                     ; WRITE BYTE TO DISC
        EI                         ; RE-ENABLE INTERRUPTS
        RET
         endif
;
;
; --- SPECIAL ROUTINE TO SEND FIRST BYTE FOR Z-100 ---
;
         if ZS1T                   ; IF Z-100
WAITOX: PUSH    PSW                ; SAVE COMMAND
WAITOX1: MVI    A,1
        OUT     OFEH               ; ENABLE 8088 INTERRUPTS
;
        EI                         ; ENABLE INTERRUPTS FOR A SHORT TIME
        NOP
        NOP
        DI                         ; DISABLE INTERRUPTS FOR TEST
;
        XRA     A
```

```
        OUT     0FEH                ; DISABLE 8088 INTERRUPTS
;
        TSTOT                       ; Set Z-flag if HTDRDY
        JNZ     WAITOX1             ; LOOP UNTIL READY
;
        POP     PSW
        OTDATA                      ; WRITE BYTE TO DISC
;
        MVI     A,1
        OUT     0FEH                ; ENABLE 8088 INTERRUPTS
;
        EI                          ; RE-ENABLE INTERRUPTS
        RET
         endif
;
;
; --- SPECIAL APPLE SUPPORT ROUTINES ---
;
         if sys EQ APPLE
;       APPLE only
;       X6502 -- Call 6502 subroutine
;
X6502:  SHLD    A$VEC               ;Save 6502 subroutine address
        LHLD    Z$PU                ;Get pointer to Z80 card
        SHLD    X651+1              ;Save for 6502 call
X651:   STA     0                   ;Execute 6502 subroutine
        RET                         ;Return
         endif
;
;
; --- SPECIAL ZEROX 820 SUPPORT ROUTINES ---
;
         if sys EQ XRX
;       XEROX only
;
;       SETOT -- Set the port direction to out
;
SETOT:  LDA     DIRCTN              ; Get the direction of previous i/o
        CPI     HTDRDY              ; Was it "Host-To-Drive"
        RZ                          ; return if it was
        MVI     A, HTDRDY           ; get Host-To-Drive status
        STA     DIRCTN              ; put it in i/o direction indicator
;
        MVI     A,OTMODE            ; program data channel to output mode
        OUT     PDATA               ;
;
        MVI     A,NOINT             ; no interrupts on data channel
        OUT     PDATA               ;
;
        MVI     A,OTDIS             ; disable control channel
        OUT     STAT                ;
;
        MVI     A,CTLMODE           ; bit control mode on Status channel
```

```
        OUT     PSTAT           ;
;
        MVI     A,CTLMASK       ; hi nibble out, lo nibble in
        OUT     PSTAT           ;
;
        MVI     A,OTEN          ; enable control channel
        OUT     STAT            ;
        RET                     ;
;
;
;       SETIN -- Set port direction to in
;
SETIN:  LDA     DIRCTN          ; get direction of last i/

        CPI     DTHRDY          ; test if it was "Drive-To-Host"
        RZ                      ; return if it was
        MVI     A, DTHRDY       ; get Drive-To-Host status
        STA     DIRCTN          ; put it into i/o direction indicator
;
        MVI     A,OTDIS         ;
        OUT     STAT            ; disable control channel
;
        MVI     A,INMODE        ;
        OUT     PDATA           ; program data channel to input mode
;
        MVI     A,NOINT         ;
        OUT     PDATA           ; no interrupts on data channel
;
        MVI     A,INEN          ; enabel control channel
        OUT     STAT            ; enable control channel
        RET                     ;
;
DIRCTN: DB      0FFH            ; initialized to illegal value
         endif
;
;
; ---- STROBE ROUTINES FOR ALSPA ---
;
         if sys EQ ALSPA
;
OUTACV: PUSH    PSW
        MVI     A,MODO  ; EXCHANGE MODES
        OUT     COMDD
        POP     PSW
        OUT     DATA    ; PUT DATA ON BUS
        MVI     A,09H
        OUT     COMDD   ; TOGGLE STROBE DOWN
        DCR     A
        OUT     COMDD   ; TOGGLE STROBE UP
        RET
;
;
INACV:  MVI     A,MODI  ; EXCHANGE MODES
```

```
        OUT     COMDD
        MVI     A,09H
        OUT     COMDD     ; TOGGLE STROBE DOWN
        IN      DATA      ; READ DATA FROM BUS
        PUSH    PSW       ; SAVE IT
        MVI     A,08H
        OUT     COMDD     ; TOGGLE STROBE UP
        POP     PSW
        RET
;
         endif
;
;
; --- SPECIAL OSBORNE O-1  SUPPORT ROUTINES ---
;
         if  OS1T         ; IF OSBORNE O-1
;*********************************************************
;*                                                       *
;*      THESE ROUTINES MUST BE ABOVE 4000H IN THE        *
;*              PROGRAM THEY ARE USED IN                 *
;*                                                       *
;*********************************************************
;
; --- INITIALIZE DRIVER I/O ROUTINES AND HARDWARE ---
;
INITIO: CALL    LTEST     ; TEST IF CODE IS ABOVE 4000H
                          ; IF IT RET
RNS, DO INIT OF OSBORNE
        JMP     OSINIT
;
; --- READ STATUS BYTE FROM CORVUS ---
;
OSTAT:  DI
        OUT     0         ; FLIP IN I/O PAGE
        LDA     STAT      ; READ MEMORY MAPPED STATUS PORT
        OUT     1         ; FLIP IN STANDARD PAGE
        EI
        RET
;
OSTATX: DI
        OUT     0         ; FLIP IN I/O PAGE
        LDA     STAT      ; READ MEMORY MAPPED STATUS PORT
        OUT     1         ; FLIP IN STANDARD PAGE
        RET               ; THIS VERSION LEAVES INTS. DISABLED
         endif
;
         if sys EQ OS1
; --- SETUP OSBORNE PIO AND CORVUS BOARD ---
;
OSINIT: DI
        OUT     0         ; SWITCH TO ALTERNATE PAGE
        MVI     A,30H
        STA     CTLA      ; PORT A   DIRECTION PROGRAMMING
```

```
        XRA     A
        STA     ADATA   ; SET PIO FOR INPUT
        MVI     A,34H
        STA     CTLA    ; PORT A  R/W, DISABLE CORVUS DRIVERS
        MVI     A,38H
        STA     CTLB    ; PORT B  DIRECTION PROGRAMMING
        MVI     A,3FH
        STA     BDATA   ; SET ALL BUT STATUS BITS FOR OUTPUT
        MVI     A,3CH
        STA     CTLB    ; PORT B  R/W, CORVUS I/O TO INPUT
        MVI     A,2BH
        STA     BDATA   ; STROBES HIGH, IEEE DRIVERS TO INPUT
        MVI     A,3CH
        STA     CTLA    ; PORT A  R/W, ENABLE DRIVERS
        OUT     1       ; BACK TO NORMAL PAGE
        EI
        RET
;
; --- SETUP DRIVERS FOR DATA INPUT ---
;
SETIN:  LDA     DIRCTN          ; get direction of last i/o
        CPI     DTHRDY          ; test if it was "Drive-To-Host"
        RZ                      ; return if it was
        MVI     A, DTHRDY       ; get Drive-To-Host status
        STA     DIRCTN          ; put it into i/o direction indicator
        DI
        OUT     0
        MVI     A,30H   ; PORT A  DIRECTION PROGRAMMING
        STA     CTLA
        XRA     A
        STA     ADATA   ; SET PIO FOR INPUT
        MVI     A,3CH
        STA     CTLB    ; SET PORT B  FOR R/W, CORVUS DRIVER TO INPUT
        MVI     A,2BH
        STA     BDATA   ; SET IEEE DRIVERS TO INPUT, STROBE HIGH
        MVI     A,3CH
        STA     CTLA    ; SET PORT A  BACK TO R/W, ENABLE CORVUS DRIVER
        OUT     1
        EI
        RET
;
; --- SETUP DRIVERS FOR DATA OUTPUT ---
;
SETOT:  LDA     DIRCTN          ; Get the direction of previous i/o
        CPI     HTDRDY          ; Was it "Host-To-Drive"
        RZ                      ; return if it was
        MVI     A, HTDRDY       ; get Host-To-Drive status
        STA     DIRCTN          ; put it in i/o direction indicator
        DI
        OUT     0
        MVI     A,34H
        STA     CTLB    ; CORVUS DRIVER TO OUTPUT
        MVI     A,2AH
```

```
        STA     BDATA    ; SET IEEE DRIVERS TO OUTPUT
        MVI     A,30H
        STA     CTLA     ; SELECT PORT A DIRECTION PROGRAMMING
        MVI     A,0FFH
        STA     ADATA    ; SET PIO FOR OUTPUT
        MVI     A,3CH
        STA     CTLA     ; PORT A  R/W, ENABLE CORVUS DRIVERS
        OUT     1
        EI
        RET
;
; --- INPUT DATA BYTE FROM CORVUS CONTROLLER ---
;
INACV:  PUSH    B
        DI
        OUT     0
        MVI     A,0BH
        STA     BDATA    ; TOGGLE STROBE LOW
        LDA     DATA     ; GET DATA
        CMA              ; COMPENSATE FOR IEEE INVERTER
        MOV     C,A      ; SAVE IT
        MVI     A,2BH
        STA     BDATA    ; TOGGLE STROBE HIGH
        MOV     A,C
        OUT     1
        EI
        POP     B
        RET
;
; --- OUTPUT DATA BYTE TO CORVUS CONTROLLER ---
;
OUTACV: PUSH    PSW
        DI
        OUT     0
        CMA              ; COMPENSATE FOR IEEE INVERTER
        STA     DATA     ; PUT IN PIO REGISTER
        MVI     A,0AH
        STA     BDATA    ; TOGGLE STROBE LOW
        MVI     A,2AH
        STA     BDATA    ; TOGGLE STROBE HIGH
        OUT     1
        EI
        POP     PSW
        RET
         endif
;
;
; --- SPECIAL OSBORNE O-1 ROUTINES ( old scramble wire interface ) ---
;
         if sys EQ OSX1 ; if old experimental interface
; --- SETUP OSBORNE PIO AND CORVUS BOARD ---
;
OSINIT: DI
```

```
        OUT     0       ; SWITCH TO ALTERNATE PAGE
        MVI     A,4
        STA     CTLB    ; SET PORT B TO R/W
        MVI     A,3
        STA     BDATA   ; SET DRIVER AND STROBE LINES
        XRA     A
        STA     CTLB    ; PORT B DIRECTION SETUP
        MVI     A,27H
        STA     BDATA   ; SET DIRECTIONS
        MVI     A,4
        STA     CTLB    ; SET PORT B BACK TO R/W DATA
        OUT     1       ; BACK TO NORMAL PAGE
        EI
        RET
;
; --- SETUP DRIVERS FOR DATA INPUT ---
;
SETIN:  LDA     DIRCTN          ; get direction of last i/o
        CPI     DTHRDY          ; test if it was "Drive-To-Host"
        RZ                      ; return if it was
        MVI     A, DTHRDY       ; get Drive-To-Host status
        STA     DIRCTN          ; put it into i/o direction indicator
        DI
        OUT     0
        XRA     A
        STA     CTLA    ; SELECT PORT A  DIRECTION REGISTER
        STA     ADATA   ; SET ALL BITS TO INPUT
        MVI     A,4
        STA     CTLA    ; SET PORT A  TO R/W DATA
        MVI     A,3
        STA     BDATA   ; SET PORT A  DRIVERS FOR INPUT
        OUT     1
        EI
        RET
;
; --- SETUP DRIVERS FOR DATA OUTPUT ---
;
SETOT:  LDA     DIRCTN          ; Get the direction of previous i/o
        CPI     HTDRDY          ; Was it "Host-To-Drive"
        RZ                      ; return if it was
        MVI     A, HTDRDY       ; get Host-To-Drive status
        STA     DIRCTN          ; put it in i/o direction indicator
        DI
        OUT     0
        XRA     A
        STA     CTLA    ; DIRECTION SETUP OF PORT A
        MVI     A,0FFH
        STA     ADATA   ; SET ALL BITS TO OUTPUT
        MVI     A,4
        STA     CTLA    ; SET PORT A  FOR R/W DATA
        MVI     A,2
        STA     BDATA   ; SET DRIVERS FOR OUTPUT
        OUT     1
```

```
        EI
        RET
;
; --- INPUT DATA BYTE FROM CORVUS CONTROLLER ---
;
INACV:  PUSH    B
        DI
        OUT     0         ; FLIP IN I/O PAGE
        MVI     A,23H
        STA     BDATA     ; TOGGLE STROBE LOW
        LDA     DATA      ; GET DATA
        CMA               ; COMPENSATE FOR IEEE INVERTER
        MOV     C,A       ; SAVE IT
        MVI     A,03H
        STA     BDATA     ; TOGGLE STROBE HIGH
        MOV     A,C
        OUT     1         ; FLIP IN STANDARD PAGE
        EI
        POP     B
        RET
;
; --- OUTPUT DATA BYTE TO CORVUS CONTROLLER ---
;
OUTACV: PUSH    PSW
        DI
        OUT     0         ; FLIP IN I/O PAGE
        CMA               ; COMPENSATE FOR IEEE INVERTER
        STA     DATA      ; PUT IN PIO REGISTER
        MVI     A,22H
        STA     BDATA     ; TOGGLE STROBE LOW
        MVI     A,02H
        STA     BDATA     ; TOGGLE STROBE HIGH
        OUT     1         ; FLIP IN STANDARD PAGE
        EI
        POP     PSW
        RET
         endif
;
         if OS1T
DIRCTN: DB      0FFH      ; PORT DIRECTION FLAG ( INIT TO ILLEGAL VALUE )
;
; --- TEST IF CODE IS ABOVE 4000H AND EXIT WITH ERROR MESSAGE ---
;
LTEST:  POP     H         ; GET RETURN ADDRESS OFF STACK
        PUSH    H
        MOV     A,H       ; GET HIGH ADDRESS BYTE
        CPI     40H       ; IS IS ABOVE 4000H?
        RNC               ; YES, SO RETURN
        LXI     D,EMSG    ; POINT TO ERROR MESSAGE
        MVI     C,9       ; CP/M LIST STRING COMMAND
        CALL    5         ; DO IT
        JMP     0         ; EXIT PROGRAM
;
```

```
EMSG:   DB      0DH,0AH,0DH,0AH
        DB      07,' ** OSBORNE DRIVERS ARE BELOW 4000H **',0DH,0AH,'$'
;
         endif
;
;
        END
```

SAMPLE INTERFACE ROUTINE FOR 8086/8088

```
TITLE  DRIVEIO
;
; --- CORVUS/IBM DRIVE INTERFACE UNIT FOR MICROSOFT ----
;                    PASCAL AND BASIC
;
;                 VERSION 1.2  BY  BRK
;            (MICROSOFT ASSEMBLER VERSION )
;
;        THIS UNIT IMPLEMENTS  5  PROCEDURES:
;
;        INITIO
;        CDRECV = DRVRECV
;        CDSEND = DRVSEND
;
;
;    NOTE: THIS INTERFACE UNIT NOW SUPPORTS BOTH PASCAL AND BASIC
;          BUT IT MUST BE RE-ASSEMBLED WITH THE APPROPRIATE SETTING
;          OF THE  "LTYPE"  EQUATE TO DO THIS FOR EACH LANGUAGE.
;
;
;
;        THE CALLING PROCEDURE IN PASCAL IS :
;
;                CDSEND (VAR st : longstring )
;
;        THE FIRST TWO BYTES OF THE STRING ARE THE LENGTH
;        OF THE STRING TO BE SENT OR THE LENGTH OF THE
;        STRING RECEIVED.
;
;                function INITIO : INTEGER
;
;        THE FUNCTION RETURNS A VALUE TO INDICATE THE STATUS OF
;        THE INITIALIZATION OPERATION.  A VALUE OF ZERO INDICATES
;        THAT THE INITIALIZATION WAS SUCCESSFUL.  A NON-ZERO VALUE
;        INDICATES THE I/O WAS NOT SETUP AND THE CALLING PROGRAM
;        SHOULD NOT ATTEMPT TO USE THE CORVUS DRIVERS.
;
;
;
;
;        THE CALLING PROCEDURE BASIC IS :
;
;                CALL CDSEND (B$ )
;
;        THE FIRST TWO BYTES OF THE STRING ARE THE LENGTH
;        OF THE STRING TO BE SENT OR THE LENGTH OF THE
;        STRING RECEIVED ( I.E. LEFT$(B$,2) ).
;
;                CALL INITIO (A%)
;
;        THE FUNCTION RETURNS A VALUE TO INDICATE THE STATUS OF
```

```
;       THE INITIALIZATION OPERATION.  A VALUE OF ZERO INDICATES
;       THAT THE INITIALIZATION WAS SUCCESSFUL.  A NON-ZERO VALUE
;       INDICATES THE I/O WAS NOT SETUP AND THE CALLING PROGRAM
;       SHOULD NOT ATTEMPT TO USE THE CORVUS DRIVERS.
;
;=============================================================
;                       REVISION HISTORY
;
; FIRST VERSION : 10-05-82  BY BRK
;               : 11-01-82  improved turn around delay for mirror
;               : 05-16-83  merged Pascal and Basic versions
;
;=============================================================
;
TRUE    EQU     0FFFFH
FALSE   EQU     0
;
PASCAL  EQU     1       ; LANGUAGE TYPE DESCRIPTOR
BASIC   EQU     2       ; LANGUAGE TYPE DESCRIPTOR
;
;
LTYPE   EQU     PASCAL  ; SET TO LANGUAGE TYPE TO BE USED WITH
;
REVB    EQU     0       ; 0 IF REVA OR REVB DRIVE, 1 IF REVB DRIVE ONLY
;
;
; ----- CORVUS EQUATES FOR IBM PC -----
;
DATA    EQU     2EEH    ; DISC I/O PORT #
STAT    EQU     2EFH    ; DISC STATUS PORT
DRDY    EQU     1       ; MASK FOR DRIVE READY BIT
DIFAC   EQU     2       ; MASK FOR BUS DIRECTION BIT
;
;
PGSEG   SEGMENT 'CODE'
        ASSUME  CS:PGSEG
;
;
        IF      LTYPE EQ PASCAL
        DB      'CORVUS/IBM PC FLAT CABLE PASCAL DRIVER AS OF 05-16-83'
        ENDIF
;
        IF      LTYPE EQ BASIC
        DB      'CORVUS/IBM PC FLAT CABLE BASIC  DRIVER AS OF 05-16-83'
        ENDIF
;
;
; --- INITIALIZE CORVUS I/O DRIVERS ---
;
;       THIS ROUTINE MUST BE CALLED
;       ONCE TO SETUP THE DRIVERS BEFORE
;       THEY ARE USED. IF THE ROUTINE DOES
;       ANYTHING THAT CAN ONLY BE DONE ONCE,
```

```
;         IT MUST DISABLE THIS SECTION SO THAT
;         AND ACCIDENTAL SECOND CALL WILL NOT
;         LOCK UP THE HARDWARE.
;
          PUBLIC INITIO
;
INITIO    PROC      FAR
;
          IF        LTYPE EQ PASCAL
          MOV       AX,0               ; RETURN A ZERO
          RET
          ENDIF
;
          IF        LTYPE EQ BASIC
          PUSH      BP
          MOV       BP,SP
          MOV       BX,6 [BP]          ; GET POINTER TO DATA "INTEGER"
          MOV       word ptr [BX],0    ; RETURN A ZERO
          POP       BP
          RET       2
          ENDIF
;
INITIO    ENDP
;
;
; --- RECEIVE A STRING OF BYTES FROM THE DRIVE ---
;
          PUBLIC    CDRECV, DRVRECV
;
CDRECV    PROC      FAR
DRVRECV:
          PUSH      BP                 ; SAVE FRAME POINTER
          MOV       BP,SP              ; SET NEW ONE
;
          IF        LTYPE EQ PASCAL
          MOV       DI,6 [BP]          ; GET ADDRESS OF STRING TO SAVE DATA IN
          ENDIF
;
          IF        LTYPE EQ BASIC
          MOV       BX,6 [BP]          ; GET ADDRESS OF STRING DESCRIPTOR
          INC       BX
          INC       BX                 ; POINT TO STRING POINTER
          MOV       DI,[BX]            ; GET ADDRESS OF STRING TO SAVE DATA IN
          ENDIF
;
          PUSH      ES
          PUSH      DI                 ; SAVE POINTER TO 'LENGTH'
          INC       DI                 ; POINT TO START OF DATA AREA
          INC       DI
;
          MOV       AX,DS
          MOV       ES,AX              ; SET SEGMENT # FOR SAVING DATA
          CLD                          ; SET TO AUTO-INCREMENT
```

```
;
        MOV     DX,STAT             ; POINT TO STATUS PORT
;
;  --- FANCY "MIRROR" COMPATIBLE TURN ROUTINE ---
;
TURN:   IN      AL,DX               ; GET STATUS BYTE
        TEST    AL,DIFAC            ; LOOK AT BUSS DIRECTION
        JNE     TURN                ; WAIT FOR "DRIVE TO HOST"
        TEST    AL,DRDY             ; LOOK AT "READY STATUS"
        JNE     TURN                ; IF NOT READY, KEEP LOOPING
;
        CALL    SDELAY              ; WAIT A MOMENT
;
        IN      AL,DX               ; GET STATUS AGAIN
        TEST    AL,DIFAC
        JNE     TURN                ; WAIT FOR "DRIVE TO HOST"
        TEST    AL,DRDY             ; LOOK AT "READY STATUS"
        JNE     TURN                ; WAIT FOR "READY
;
        CALL    SDELAY
;
        MOV     CX,0                ; INIT LENGTH COUNT
;
RLP:    IN      AL,DX               ; GET STATUS BYTE
        TEST    AL,DRDY
        JNE     RLP                 ; LOOP UNTIL READY
;
        IN      AL,DX               ; GET STATUS BYTE
        TEST    AL,DIFAC            ; TEST BUS DIRECTION
        JNE     RLPE                ; IF "HOST TO DRIVE", EXIT
;
        TEST    AL,DRDY             ; TEST FOR 'READY'
        JNZ     RLP                 ; DOUBLE CHECK THAT IT IS READY
;
        DEC     DX                  ; POINT TO DATA PORT
        IN      AL,DX               ; GET DATA BYTE
        INC     DX                  ; POINT BACK TO STATUS PORT
        STOSB                       ; STORE DATA BYTE IN DATA STRING
        INC     CX                  ; INCREMENT LENGTH COUNTER
        JMP     RLP                 ; LOOP UNTIL DONE
;
RLPE:   POP     DI                  ; GET POINTER BACK TO LENGTH
        POP     ES
        MOV     [DI],CX             ; SET LENGTH OF RETURNED STRING
        POP     BP                  ; GET FRAME POINTER BACK
        RET     2                   ; CLEAR RETURN STACK
CDRECV  ENDP
;
; --- SEND STRING OF BYTES TO DRIVE ---
;
        PUBLIC  CDSEND, DRVSEND
;
CDSEND  PROC    FAR
```

```
DRVSEND:
        PUSH    BP              ; SAVE FRAME POINTER
        MOV     BP,SP           ; SET NEW ONE
;
        IF      LTYPE EQ PASCAL
        MOV     SI,6 [BP]       ; GET ADDRESS OF STRING TO SEND
        ENDIF
;
        IF      LTYPE EQ BASIC
        MOV     BX,6 [BP]       ; GET ADDRESS OF STRING DESCRIPTOR
        INC     BX
        INC     BX              ; POINT TO STRING POINTER
        MOV     SI,[BX]         ; GET ADDRESS OF STRING TO SAVE DATA IN
        ENDIF
;
        MOV     CX,[SI]         ; GET STRING LENGTH
        JCXZ    ENDSND          ; IF NULL STRING, JUST RETURN
;
        INC     SI              ; POINT TO START OF DATA TO SEND
        INC     SI
        CLD                     ; SET TO AUTO-INCREMENT
;
        LODSB                   ; GET FIRST BYTE OF DATA
        CALL    WAITO           ; SEND FIRST BYTE USING INTERRUPT TEST
;
        INC     DX              ; POINT TO STATUS PORT
        JMP     WLP1            ; ENTER COUNTING LOOP
;
WLP:    IN      AL,DX           ; READ STATUS BYTE
        TEST    AL,DRDY         ; IS DRIVE READY FOR NEXT ACTION?
        JNZ     WLP             ; NO, SO KEEP LOOPING
        DEC     DX              ; POINT TO DATA PORT
WLPB:   LODSB                   ; YES, GET DATA BYTE FROM 'DMA' LOCATION
;
        IF      REVB-1          ; FOR REV A OR REV B DRIVES
        OUT     DX,AL           ; SEND DATA BYTE TO DISC
        INC     DX              ; POINT BACK TO STATUS PORT
WLP1:   LOOP    WLP             ; LOOP UNTIL TRANSFER IS COMPLETE
        ENDIF
;
        IF      REVB            ; FOR REV B DRIVES ONLY
        OUT     DX,AL           ; SEND DATA BACK TO STATUS PORT
WLP1:   LOOP    WLPB            ; LOOP WITHOUT STATUS TEST
        ENDIF
;
ENDSND: POP     BP              ; GET FRAME POINTER BACK
        RET     2               ; CLEAR RETURN STACK
CDSEND  ENDP
;
;
; --- SHORT DELAY ROUTINE ---
;
SDELAY  PROC    NEAR
```

```
        MOV     CL,30           ; SETUP FOR SHORT DELAY
DELAY:  DEC     CL              ; LOOP UNTIL DONE
        JNZ     DELAY           ; DELAY TO AVOID BUS TURN AROUND GLITCHES
        RET
SDELAY  ENDP
;
; --- WAIT AND OUTPUT BYTE TO CONTROLLER ---
;       INTERRUPTS ARE SWITCHED HERE
;          TO AVOID PROBLEMS WITH
;              CONSTELLATION
;
WAITO   PROC    NEAR
        PUSH    AX              ; SAVE DATA BYTE
WAITO1: STI                     ; ALLOW INTERRUPTS
        MOV     DX,STAT         ; POINT TO STATUS PORT
        NOP                     ; ADDITIONAL DELAY FOR INTERRUPT
        CLI                     ; DISABLE INTERRUPTS
        IN      AL,DX           ; GET STATUS BYTE
        TEST    AL,DRDY         ; IS DRIVE READY?
        JNZ     WAITO1          ; NO, SO LOOP
        POP     AX              ; GET DATA BACK
        DEC     DX              ; POINT TO DATA PORT
        OUT     DX,AL           ; OUTPUT BYTE
        STI                     ; ALLOW INTERRUPTS
        RET
WAITO   ENDP
;
PGSEG   ENDS
;
        END
```

ENTRY POINTS FOR APPLE II ROM

The routines in the Apple II flat cable ROM assume that the card is in slot 6. (See Constellation Software General Technical Information manual for more information.)

Address	Function
C600h	Boot
C6CFh	RWTS
C68Dh	Save warm boot image
C815h	Read Corvus sector (256-byte read)
C818h	Write Corvus sector (256-byte write)

The following bytes identify the Corvus flat cable interface card:

Address	Contents
C600h	A9h
C601h	20h
C602h	A9h
C603h	00h
C604h	A9h
C605h	03h
C606h	A9h
C607h	3Ch

ENTRY POINTS FOR IBM-PC/TI ROM

Entry points are the same as those described for the Omninet ROM.

This page intentionally left blank.

SOFTWARE DEVELOPER'S INFORMATION | **F**

MSDOS

A Software Developer's diskette is available from Corvus customer service. It contains the following files:

SEMA4.BAS	An example program, written in Basic, which shows how to send disk commands. It uses the semaphore commands for the example. This program is meant to be compiled with the Microsoft BASIC compiler. It will NOT work with the Basic interpreter.
SEMA4.PAS SEMA4.EXE	An example program, written in Microsoft Pascal, showing how to send disk commands. It uses the semaphore commands for the example. The compiled version was linked with DRIVEC2.OBJ.
*PIPES.PAS *PIPES.EXE	An example program, written in Microsoft Pascal, showing how to send disk commands. It uses the pipes commands for the example. The compiled version was linked with DRIVEC2.OBJ.
DRIVEC2.ASM DRIVEC2.OBJ BDRIVEC2.OBJ	This is the source for the machine language module used to send drive commands. This version works with MSDOS 1.0, 1.1, and 2.x; it works for both flat cable and Omninet, because it calls the Corvus disk driver to send the command. The OBJ files provided are conditionally assembled for MS Pascal and MS Basic compiler respectively.
DRIVEIO2.ASM DRIVEIO2.OBJ BDRVIO2.OBJ	This is the source for a machine language module used to send drive commands via the flat cable interface card. This version will work for the IBM-PC and TI-PC; some I/O port equates must be changed for other interface cards. The OBJ files provided are conditionally assembled for MS Pascal and MS Basic compiler respectively.
ODRIVIO2.ASM ODRIVIO2.OBJ BODRVIO2.OBJ	This is the source for a machine language module used to send drive commands via the Omninet transporter. This version will work for the IBM-PC and TI-PC. The OBJ files provided are

conditionally assembled for MS Pascal and MS Basic compiler respectively.

IMPORTANT NOTE: The ODRIVIO2 routine may NOT be used on a PC which has the Corvus Constellation II driver installed.

*SEMA4ASM.ASM
*SEMA4ASM.OBJ This is a machine language module which supports the semaphore functions SemLock, SemUnlock, and SemStatus. This version is written to interface to Microsoft Pascal.

*PIPESASM.ASM
*PIPESASM.OBJ This is a machine language module which supports the pipes functions PipeOpRd, PipeOpWr, PipeRead, PipeWrite, PipeClRd, PipeClWr, PipePurge, and PipeStatus. This version is written to interface to Microsoft Pascal.

* These files are not yet available.

Versions supported are:

IBM-PC MSDOS 1.0, 1.1, 2.0, 2.1
TI Professional MSDOS 1.25, 2.0
DEC Rainbow MSDOS
Zenith Z-100 MSDOS

Formats available are:

IBM-PC 8-sector single-sided

CP/M 80 CONSTELLATION II

The following files are contained on the standard distribution floppies for Constellation II:

SEMA4.COM SEMA4.PAS SEMA4.CMD	An example program, written in Pascal MT+, showing how to send disk commands. It uses the semaphore commands as an example.
CPMIO.DOC CPMIO.ERL	A document file describing the support services provided by the driver interface unit CPMIO.ERL.

CP/M 86 CONSTELLATION II

The following files are contained on the standard distribution floppies for Constellation II:

SEMA4.CMD SEMA4.PAS SEMA4.KMD	An example program, written in Pascal MT86+, showing how to send disk commands. It uses the semaphore commands as an example.
CPMIO86.DOC CPMIO86.R86	A document file describing the support services provided by the driver interface unit CPMIO86.R86.

CP/M 80 (Flat cable only; not Constellation II)

A Software Developer's diskette is available from Corvus customer server. It contains the following files:

MIRROR.ASM	Source for the Corvus Mirror program. Shows how to send drive commands for flat cable interface.
CDIAGNOS.ASM	Source for the Corvus CDIAGNOS program. Shows how to send drive commands for flat cable interface.

Versions supported are:

S-100
TRS 80 Model II
Zenith H-89, H-90
Xerox 820
Sony

Formats available are:

S-100 8" single-sided, single-density
Northstar 5 1/4"
Vector Graphics 5 1/4"
Zenith H-89
Zenith H-90
Xerox 820
Sony

APPLE PASCAL CONSTELLATION I

The following files are contained on the standard Apple floppies for Constellation I:

CORVUS.LIBRARY	Contains units for sending drive commands (OMNISEND, DRIVEIO), using semaphores (SEMA4), and using pipes (PIPES).
SPOOL.TEXT SPOOL.CODE	An example program showing how to use pipes.
SHARE.TEXT SHARE.CODE	An example program showing how to use semaphores.

APPLE DOS CONSTELLATION I

The following files are contained on the standard Apple floppies for Constellation I:

BCI.OBJ OMNIBCI.OBJ	A machine language interface for sending disk commands.
SPOOL	An Applesoft program showing how to use pipes.
SHARE	An Applesoft program showing how to use semaphores.

This page intentionally left blank.

INDEX