The simplest and safest approach is to perform *disk-locking*. The existing enforced single access on disks opened for write access, is a form of disk-locking. The difference here is that disks are allowed to be attached to drives at more than one client, but with the emulator caches disabled. When the server receives a track read request, it verifies that the disk is not currently locked out, and if not it will grant the node exclusive access. Here the first locking phase is transparent to the client, in that the server can set locks as it receives requests to read or write to pseudo-disks. However, client software must explicitly release a disk when the transaction is finished.

Since read requests are for tracks, the natural approach is to perform *track-based locking*. The server would keep record of the tracks of shared disks currently locked into clients. Once again the locking operation is transparent via the track read operation but a release operation must be performed, when complete.

Sector-locking of the pseudo-disks is another possible approach, since write operations involve sector updates. The transparency of the first locking phase is lost, and an explicit request for a sector must be made to the server, before the disk is actually read. This is necessary, because the region of data being controlled is only a portion of that involved in a read request. When the client reads the disk, a request for an entire track will be sent to the file server. On receiving the request for a track, the server must only return the data for the sectors which the client currently has access to. The rest of the sectors on the track must contain null-data. When finished, a message must be sent to the server releasing the sectors.

Data blocks which are smaller than a sector would be inappropriate, since even though a node will only be given data for the block requested, an entire track must still be sent along the network and read by the client. This inefficiency would rule out locking at the level of database records or fields. Similarly, variable-sized data blocks can be excluded because their boundaries would not coincide with the boundaries of tracks and sectors.

Concurrency control must also solve the **deadlock problem**. Deadlock becomes possible, as soon as one client is allowed exclusive access to more than one data block simultaneously. (Deadlock exists if two applications, each having control of a region, need to wait for the other application to release its region). To overcome this problem, it must be possible for a transaction to abort, so that its locks can be released. For this reason, it is good practice to collect all the locks needed, before any operations are performed on the data [4, pp 144]. Mechanisms to break deadlock, include explicit deadlock detection and time-limited breakable locks [3, pp 359].

The following points should be noted with respect to the accessing of shared disks:

- A single release operation should be performed releasing all currently held data regions, rather than releasing each individually. In track-based locking, this relieves software from knowing which tracks of the pseudo-disk were accessed during the transaction.
- To limit the number of delays on the system, software must use as few data blocks simultaneously as possible. This means that database records should preferably fall on sector or track boundaries.
- If a large number of blocks need to be collected before a transaction can be performed, the request for locks should be made explicitly before reading the blocks. This avoids reading each block and later having to abort the transaction because one block cannot be accessed.

6.6 NODE-TO-NODE MESSAGE PASSING

It is undesirable to allow messages to be passed between users during tests. In addition, it is unnecessary given the application of the network, since all nodes are located in the same area. However, extension may result in the client PC's not all being localized, and if the network is to be applicable outside the given application, it would be necessary that it could support message passing.

Once again, communications has to be performed through the reading and writing of disk sectors. Another communications track (e.g. track 41), could be used for this purpose. Sending the message poses no problems. It merely involves an addition to the functions supported by the file server. The call for this function is sent with the message contents to the server. The server must then send a track containing the message, to the emulator of the destination client.

For a client to receive the message track is more difficult, since under normal operation it is the client which initiates track transfer through disk operations. It would require co-resident software at the client. If possible, when the emulator receives the message track, it would cause an interrupt at the client, to which the co-resident software would respond with a read request to the message track. Alternatively, the message track must be read at regular intervals. If no message has been sent, the emulator must provide an 'empty' track. A third alternative is to leave it to the user to enquire if a message has been received.

7 EVALUATION and CONCLUSIONS

In this chapter, the salient features of the file server design are summed up, in terms of the classifications identified in the literature survey (*cf* section 1.3). The design is then evaluated, considering the requirements for the educational environment given in section 1.2. Finally, I give my conclusions on the efficacy of the network/server solution, and suggest how limitations could be eliminated.

7.1 CLASSIFICATION OF THE PROPOSED FILE SERVER

In chapter one a model was established through which file servers can be classified on the basis of their external characteristics.

The server is *disk- or block-based*, with the basic object it maintains, being the pseudo-disk. This has the form of a file on the server machine, but appears as a floppy disk to the node computer.

The basic *units of data access*, are fixed-size blocks. For data retrieval, the blocks are of track size, while for writing data to the server, the blocks are of sector size. File, page-level and byte-level retrieval are not supported. However, methods have been considered which allow files, and not merely disk blocks, to be obtained from the server (*cf*: Section 6.4).

Object management involves the facilities provided for organization and protection of the objects. A *directory service* is supported. This consists of a pseudo-disk which lists the available applications disks, together with the categories of user which are entitled to use it. *Identity-based access control* was chosen. This is oriented towards allowing categories of users access to disks, but is not suitable for sharing disks amongst particular users.

Atomic transactions are not supported. Concurrent usage is only allowed for read accesses. Write access is essentially single-user. However, chapter six showed it to be possible to achieve *concurrent update*, which is desirable for shared data bases.

The level of the service is essentially that of a low-level file server, or intelligent disk server. However, it was shown to be extendable, to successively high-level functions - allowing it to be more universal in context, supporting both low and high-level usage. However, such extensions will be inefficient, because of the underlying inefficiency in the communications mechanism - namely that communication with the file server has been achieved through disk operations, and requires the reading and writing of entire sectors, even for very short messages.

7.2 EVALUATION

The file server design has dealt with the important issues identified in section 1.2. The achievement in each of these areas will be summed up, and evaluated where necessary.

The shared disk resource has been used to provide *user disk space* and a *common pool of software*. The space limitations due to floppy disk size, have been overcome by allowing a client's drive to have connected to it, any of a large number of pseudo-disks. These pseudo-disks are single files which contain an image of a floppy disk. Mechanisms have been provided for the attachment/detachment of these to/from client machine drives. The size of a pseudo-disk is that of a floppy disk. This restricts the maximum size of any file, and means that a single application may have to be spread over more than one disk.

The user-to-user *data sharing restriction* has been achieved through identity-based, single-access to user disks. A method of allowing the transfer of files between users has been described as an extension in section 6.8. Since the network is a logical star, with all communications routed to the server, this feature can be disabled when necessary - viz. during tests.

The basic mechanisms for the *client interface* - involving data transfer, signing-on or signing-off, the switching in and out of pseudo-disks, and changing operating systems - have been devised. Software which runs on the client computer, is required (as an addition to the operating system) to allow the invocation of the functions offered by the server and to present these functions to the user. These software requirements, comprising the client and user interfaces, have been described.

The network hardware at the file server has shared access to the server machine's memory (in addition to its own local memory). The *interface* between the network and the server, has been implemented through shared resources contained in the server's memory. A method has been suggested, involving semaphore variables in the server's memory, to achieve mutually exclusive access for the Server and Network processors.

Alternatives for the service offered to the *analysis machine* have been given. User files could be received through the analysis machine using existing functions to switch-in a user's disk and then examine the files. Alternatively, a user could submit a file to the server as one which requires analysis, with the analysis machine dequeuing these files for analysis (*cf*: section 6.2).

Extensions to the basic functionality of the server have been proposed (chapter 6). However the extensibility is affected by the method employed for communication between a client and the server. Communication requires the reading/writing of disk tracks/sectors - i.e. entire blocks. This is particularly inefficient for small message transfers. This inefficiency would probably make it undesirable to extend the server to include a file-based service and thus to have full universal file server status. Sufficient extensions have been suggested however, which make it also suitable to a more general network outside of the education requirements.

7.2.1 Performance

Since the clients receive disk data through a floppy disk drive interface, the rate of data transfer is that of the floppy disk controller. Access and disk latency delays are still present, so the performance of the network drives can never exceed that of a floppy disk drive.

In addition though, there is a delay for a track to be requested by the emulator and received from the file server. The probability of this delay has been decreased through caching and prefetching at both the server and emulator. For a file transfer, only the first track transferred should incur such a delay (provided the file is stored contiguously on the pseudo-disk). After that, prefetching should ensure that the emulator has the next track by the time it is required. Thus the delay will comprise a substantial amount of the total transfer time for small files, but be insignificant for very large files - i.e. the perceived performance should approach that of a floppy disk.

In certain respects, the performance of the network drives, could be superior to that of an actual floppy disk drive. Firstly, there is no *motor-on delay*. The necessity for contiguity within the pseudo-disks, to reduce demand for both network transfers and for operations on the server's disk, was emphasized (*cf*: section 2.3.3). If the contiguity of pseudo-disks is maintained by the file server, there would be a favourable side-effect for clients, in that disk performance is substantially better for contiguous files. (Transfer for 100 Kbyte file , stored completely contiguously on a floppy disk, would take about 5 seconds).

At the file server, the critical resource limiting performance, is the disk. This has been supplemented by the cache to improve the efficiency of disk use. Assuming that groups of 6 tracks are fetched to cache during each disk read operation, groups for 12 different pseudo-disks can be brought to cache every second. (A rough estimate based on typical values for disk access and transfer delays, and assuming that the pseudo-disks are stored contiguously on the server's disk). The server's disk should therefore be able to provide, without degradation in performance, the requirements for more than 12 users, simultaneously involved in disk operations. Clearly, not all tracks fetched will be required, but there should also be duplicate requests, where the tracks are already in the cache from a previous transfer.

Under heavy load, it is the network transfer rate of 1Mbit/s which will limit performance. This rate allows transfer of approximately 30 tracks per second. The maximum data requirement at the client, based on the disk transfer rate, is 5 tracks per second. The network transfer rate can therefore only support up to 6 clients, simultaneously involved in disk operations, before load effects become evident. However, except when the cache is empty at start-up time, caching at the emulator should reduce the percentage of the client's disk operations, which require transfers off the network.

7.2.2 Operating System Independence

Currently only MSDOS is used on the computers. Although alternative operating systems have not been investigated, the use of different operating systems has been considered throughout the design. Pseudo-disks belonging to different systems are separated in storage, and only if the client is running the correct system, can a disk be attached. A client can request a change in the operating system. The disk attached to the boot-drive is changed by the server, and code in the user interface at the client causes the client to re-boot. The bootstrap of the computer looks for the boot record in the first sector of the boot-disk. In this way, it should be possible to load any operating system designed for the client computer.

The functions of the file server should not require adaptation, to support new operating systems for the client. However each operating system requires the addition of routines comprising the client and user interfaces. No additions are required to support data transfer operations - since operations involving disk data transfers are trapped in hardware by the disk drive emulator, and requests forwarded to the server.

The track size for pseudo-disks has been assumed to be 9 sectors of 512 bytes. Support for different size sectors or tracks would lead to complications in both the file server cache and the cache of the disk drive emulator.

7.2.3 Protection Mechanisms

Identity-based access to disk storage has been chosen. The user's identity not only establishes the user's personal disks, but to which category of users he belongs. A user's category entitles him to a certain subset of applications. (Application is used loosely, as a group of up to fifteen pseudo-disks to which there is shared access). For commercial software, the number of simultaneous users of an application can be restricted to the number of purchased copies of the product.

The restricted distribution of software obviously relies on the ability to prevent software being copied off the network - i.e. if applications were copied to floppy disk, this would bypass restrictions on the category of user, and on the number of simultaneous users of certain software. Once access has been granted to an application, and the server receives requests for tracks, it is impossible for it to determine for what purpose the track is required - i.e. whether the software is being legitimately executed, or copied. However, protection has been achieved at the client through the ability to attach and detach the applications from drives.

The protection mechanism has been described, and forms part of the requirements for the user interface at the client. It involves executing a protected application immediately after it has been attached. Code remains co-resident with the application, and continues once the application has terminated. It then sends a request for the file server to detach the application.

Inadequacies in this approach are that the user may find a way of halting the application abnormally - in which case, the application would remain attached, and could be copied. In addition, much commercial software allows for the copying of files while executing.

A user might also find a way of by-passing the loading program of the user interface, and sending a request message directly to the file server. He could supply the correct parameters for a protected load, while not performing such a load. It is therefore important that only requests from the user interface, be dealt with by the server. A form of encryption could be used to achieve this - requiring a key, which could reside anywhere on the directory disk (an improper pseudo-disk which does not have file structure). This could be changed at the file server, daily during initialization. A user would have to unscramble code to obtain the key. References [23,24,25] cover security and encryption techniques in detail.

7.3 **CONCLUSIONS**

The file server design has accomplished the aims laid down in section 1.2. However, there are limitations in the performance of the service offered to clients, due to client computers being connected to the network via a floppy disk drive. These limitations involve disk-size, the speed of response, and the extensibility.

Although multiple disks are supported, each is limited to the size of a floppy disk. This restricts the maximum *file size*. In addition, single applications with numerous files, may have to be *spread over many disks*. These disks are simultaneously attached to different drives, and it is necessary that the application be configured to know which files are on which disks. This inconvenience would restrict the use of many software packages.

A fundamental limit on performance, is that the *speed* of data retrieval for a client, is at maximum that of a floppy disk drive - any delays awaiting retrieval from the server, will make the response slower than that of a floppy disk drive.

Since disk related operations are trapped in hardware, it is not known which sectors of a particular track have been requested, for a *read operation*. Therefore an entire track has to be fetched via the network. This track-based retrieval necessitates a heavy emphasis on contiguity, both within the pseudo-disks and on the server's disk.

Communications between a client computer and the server, had to be constructed from the reading and writing of sector data. This is very inefficient for small messages. This inefficiency restricts the ability to include a file-based service and therefore restricts extension toward a truly universal file server.

The severe limitations are mainly due to the system being floppy-disk based, and there are *alternative* means of implementing the network to eliminate these limitations. The most obvious is to perform emulation of a fixed, rather than a floppy, disk drive. The speed of data transfer at the client computer, and the disk-size, would be substantially increased. However, a fixed-disk controller would have to be purchased for each of the client machines, adding substantially to the cost. In addition, data retrieval would still be track-based, and problems associated with communication and contiguity, would remain.

An alternative which eliminates the above limitations of the service, is to intercept a client's disk operations in software, and to route the requests via a communications port to the server. Disk-size need not be limited to a floppy-disk, and transfer rate is now only limited by the speed of the communications port. In software, the sectors rather than just the track required for a read operation can be determined, so that data retrieval could be sector, and not track, based. Communication for higher level operations would be via the communications port, and would not involve entire sector or track data-blocks.

Although the file server design is oriented towards the proposed network, based on floppy-disk-drive emulation, it would still be applicable (with adaptations), if an alternative implementation was chosen.

- The pseudo-disks would no longer necessarily be of floppy disk size. Even without limits on size it is advisable to confine a single application to a disk. This allows the protection mechanisms developed, to be used. When the application is not in use, the disk containing it, is not attached to the drives and cannot be copied.
- If track-based retrieval were eliminated, a sector and not a track should be the basic data unit for the cache. However, group-based retrieval should be maintained, even though groups need not fall on track boundaries.
- Sector-based retrieval will relax the emphasis on contiguity.

The basic structure of the server design should be relevant, no matter which of the above network implementations is chosen. The isolation of the parallel processes and identification of the major resources, could remain unchanged. The resource requiring the biggest alteration would be the Opened Disks/Applications Maps. Since an entire application can fit on a disk, the separate treatment of disks and applications is not required.

From the point of view of the file server then, a floppy-disk based network and track-based retrieval, is *undesirable*. However, the original reasons for adopting such a physical implementation, may still be applicable. These were cost benefits, and the relative transparency which the solution presents, to the existing hardware and software of the client machine (*cf*: section 2.2). Should these factors be seen as overriding the resulting limitations on the File Server, then it could be argued that floppy disk speed is adequate. Also, at peak-load times, performance would be limited by the transfer-rate of the network and not by the transfer-rate at the interface - so that performance at peak-loads would be similar, for all network

implementations which use the same transfer-rate. The disk-based functions offered by the server, and selected extensions, should provide an adequate service, and there is no need to achieve a universal service.

In spite of the afore-mentioned faults, the file server should provide a convenient centralized service in the educational environment. It resolves the difficulties associated with the distribution of floppy disks, and provides the basis for automatic analysis, through collection of student work. Software protection has potential, if the stated limitations can be overcome. This would reduce software costs, by preventing the copying of the software, and through allowing packages with limited usership, to be purchased for the network, without the need to purchase a copy for each machine on the network.

Addendum

As anticipated, the physical interface did lead to limitations on the File Server. During the project, the use of this physical interface was abandoned (and implementation terminated), due to the limitations it placed on the network. Since this was always seen as a possibility, I have tried to evaluate the design as it is, while emphasizing that the design would be applicable to a disk-based approach where client's disk requests are intercepted in software and routed to the File Server through a serial interface, (an approach used in other designs [7]).

Since the inception of this project, certain LAN and File Server systems for IBM PC's have become very popular - such as Novell Netware and MS-NET [26,27]. A commercial network of this sort ought to be far cheaper and more robust, than developing a customized network. However, a customized network is still attractive for the following reasons.

The file server design has a very centralized characteristic, where access to the system, application's software and user files is controlled exclusively by the network manager. Some commercial networks on the other hand, in being more flexible, tend to be orientated towards user-user file sharing. Here the creator of a file specifies who may access the file.

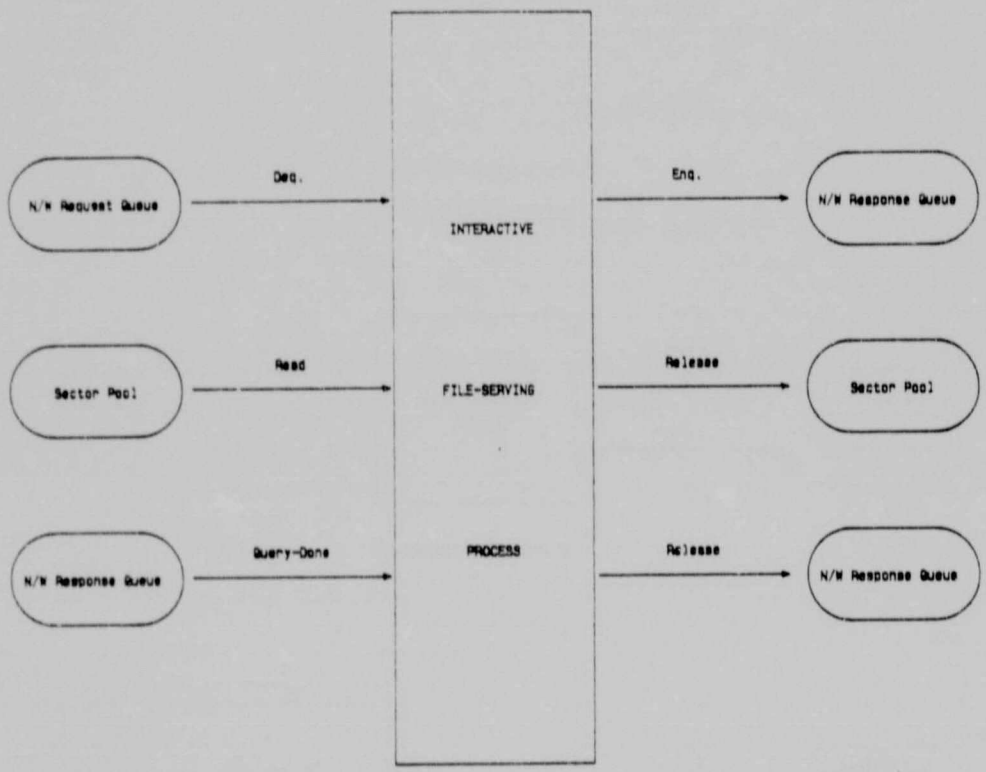
A server with a centralized characteristic at its core, ought to be the most suitable for the specified education purposes - that of a large central file store providing users with their own disks and a selected pool of applications on which to draw. Centralization allows strict control over what software is made available to the user, and over access to this software. The 'safety' of software on the server is however dependent on protection mechanisms once a user is given access to an application. In this respect, the pseudo-disk characteristic (whereby each application/package is confined to a separate logical disk, and has to be 'switched' into a user's drive before use), holds potential.

The advantages offered by a centralized system could yet be sufficient for a customized system. If software companies were satisfied that their products were safe from piracy, they could even see it as an advantage for their products to be in the system's pool, to expose students to their products (who later enter the market place). The possibility would then exist of providing a very large pool of software cheaply, and thereby increasing the service value beyond what could be provided otherwise.

There ought to be sufficient other applications for such a centralized architecture - other universities, or even a software depot service charging clients for the use of a pool of software (thereby increasing accessibility to expensive, less available software). This wider applicability ought to increase the viability of developing a customized network.

APPENDIX A: COLLECTIONS OF PROCESS-RESOURCE DIAGRAMS

APPENDIX A.1 - INTERACTIVE SERVING: 1ST LEVEL



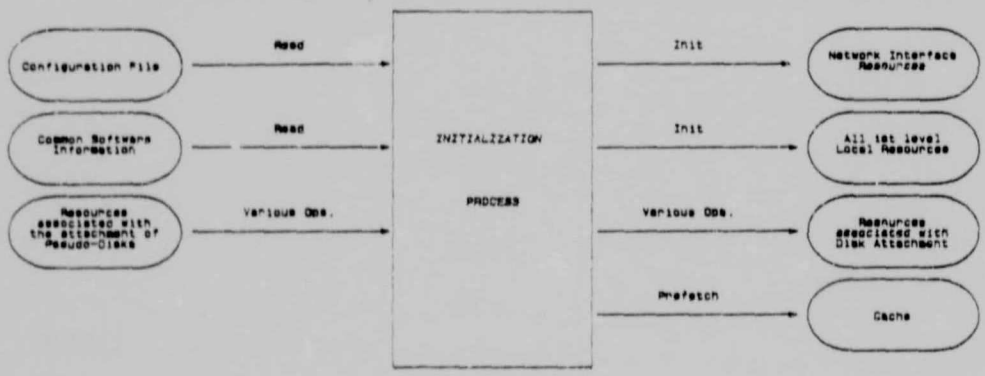
Abbreviations Used in Resource Diagrams

N/W - 'Network'
Enq. - 'Enqueue'
Deq. - 'Dequeue'
Various Ops - 'Multiple Operations - Identified at a lower level'
i/p Par: - 'Input Parameters'
o/p Par: - 'Output Parameters'
WL - 'Write-List'

Resources associated with the attachment of Pseudo-Disks:

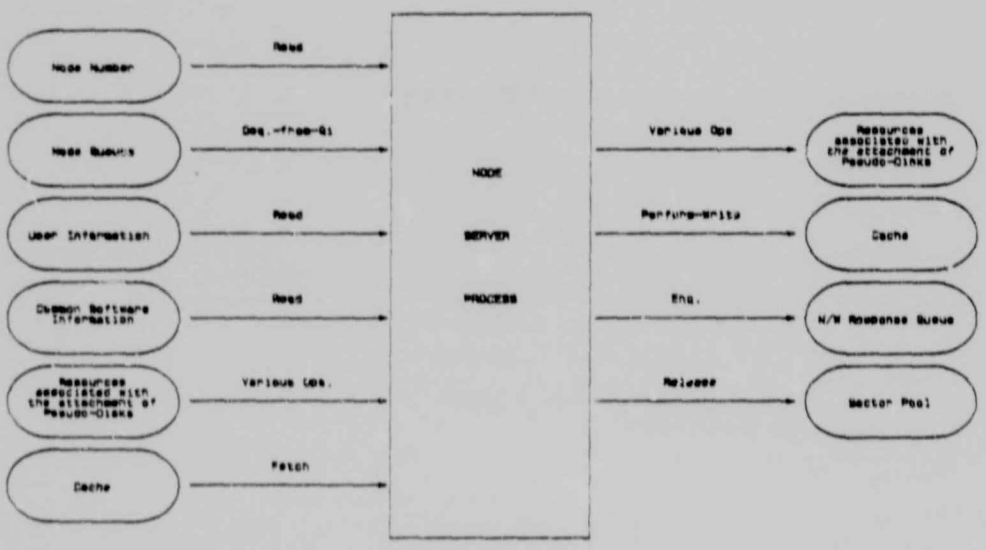
These are the Common Information Resource, MD-Attachment Resource & the Opened-Disks Tables (both Application Map & Disk Map). The various operations on these resources are only identified at the lower levels.

APPENDIX A.2 - INTERACTIVE SERVING: 2ND LEVEL PROCESSES



(a) INITIALIZATION PROCESS

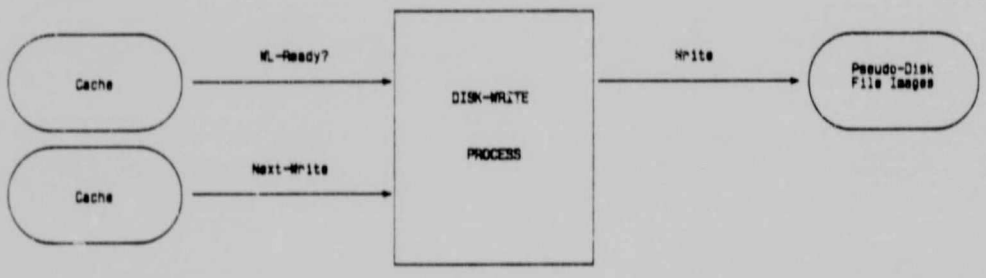
(b) QUEUE-PRODUCING PROCESS



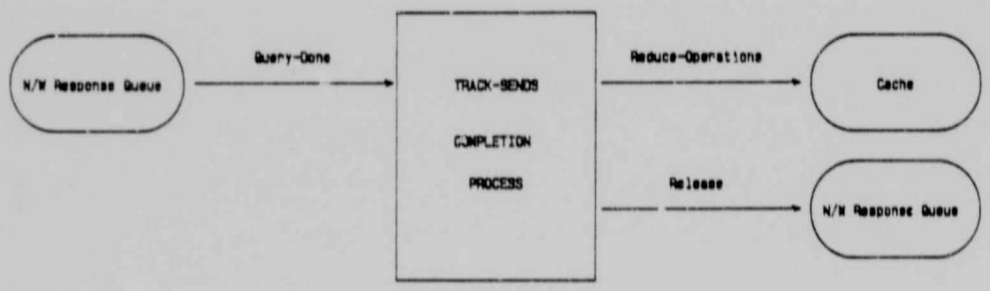
(c) INDIVIDUAL-NODE SERVER PROCESS

APPENDIX A.2 - INTERACTIVE SERVING: 2ND LEVEL PROCESSES

(d) DISK-READ PROCESS



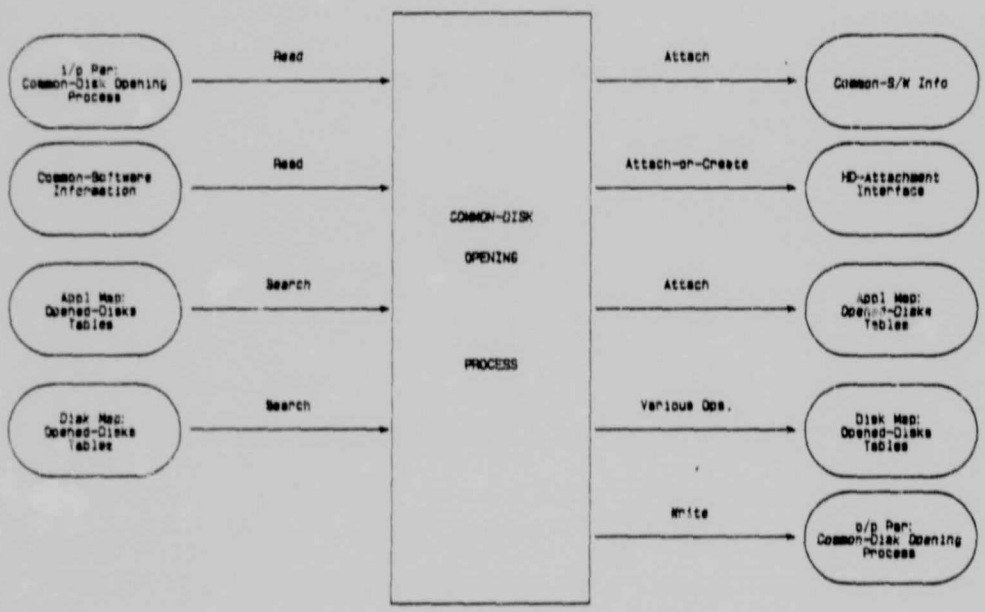
(e) DISK-WRITE PROCESS



(f) COMPLETE -SENDS PROCESS

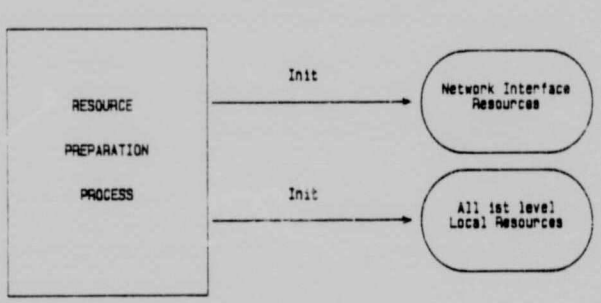
APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.1: THE COMMON-DISK OPENING PROCESS

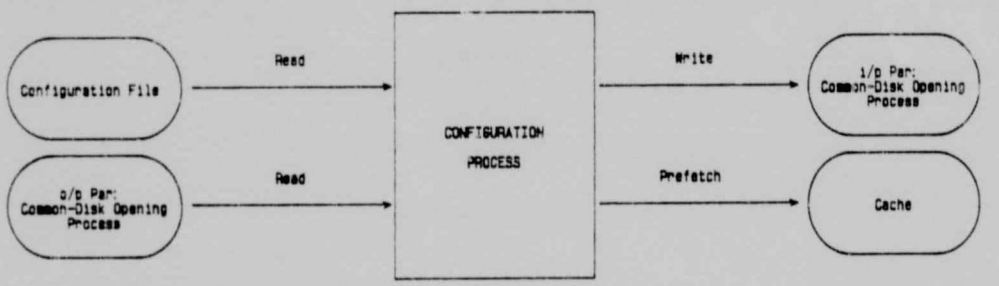


APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.2: DECOMPOSITION OF THE INITIALIZATION PROCESS



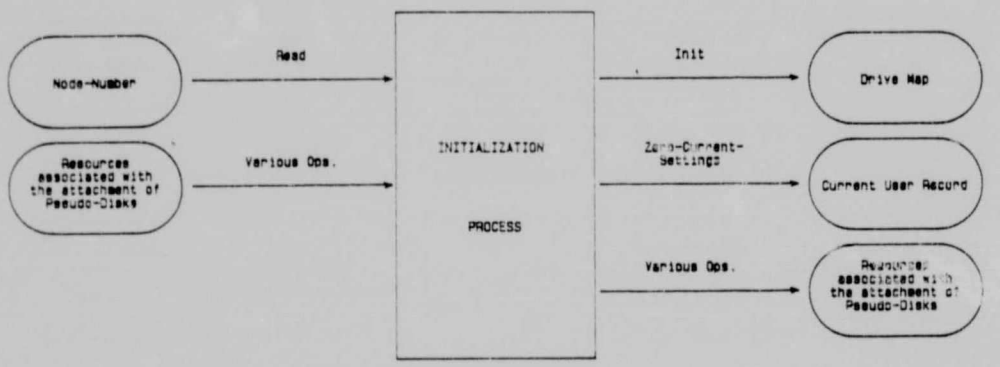
(a) RESOURCE PREPARATION PROCESS



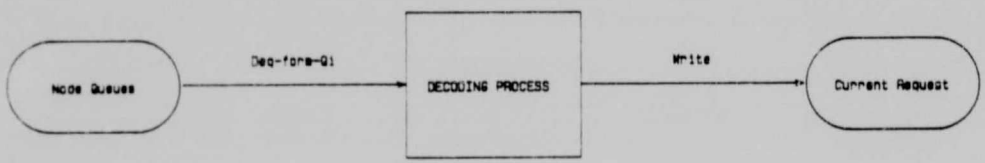
(b) CONFIGURATION PROCESS

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

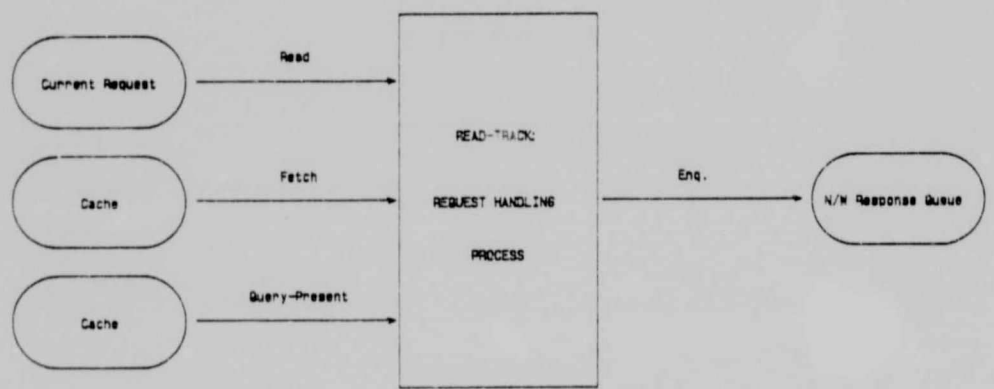
A.3.3: DECOMPOSITION OF THE NODE-SERVER PROCESS



(a) INITIALIZATION PROCESS



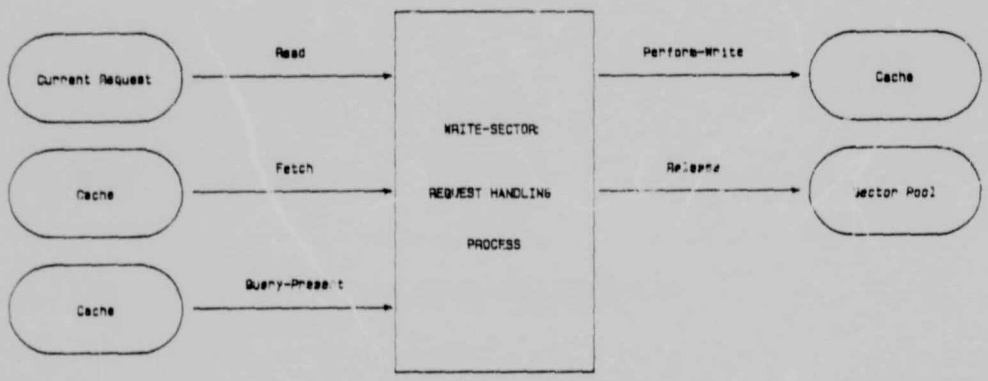
(b) DECODING PROCESS



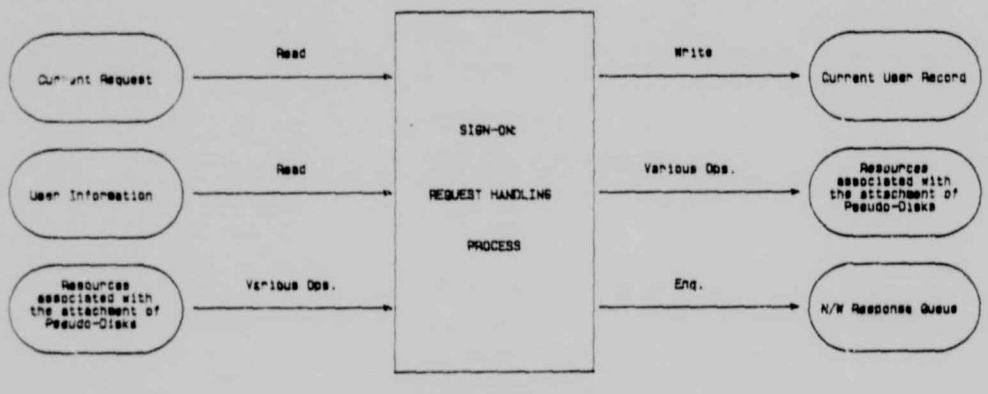
(c) PROCESS REQUEST TO READ A TRACK

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

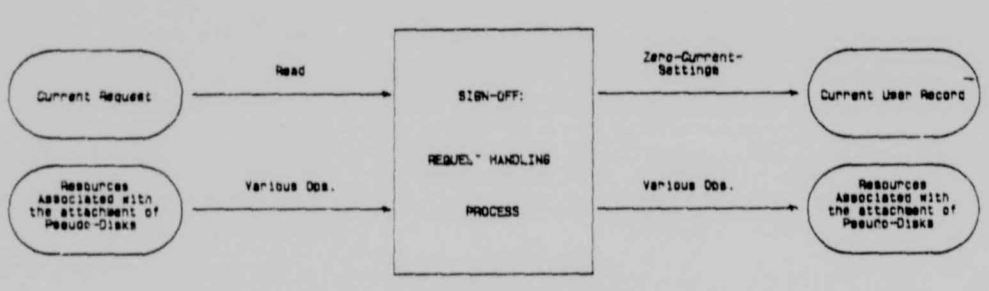
A.3.3: DECOMPOSITION OF THE NODE-SERVER PROCESS



(d) PROCESS REQUEST TO WRITE A SECTOR



(e) PROCESS REQUEST TO SIGN-ON



(f) PROCESS REQUEST TO SIGN-OFF

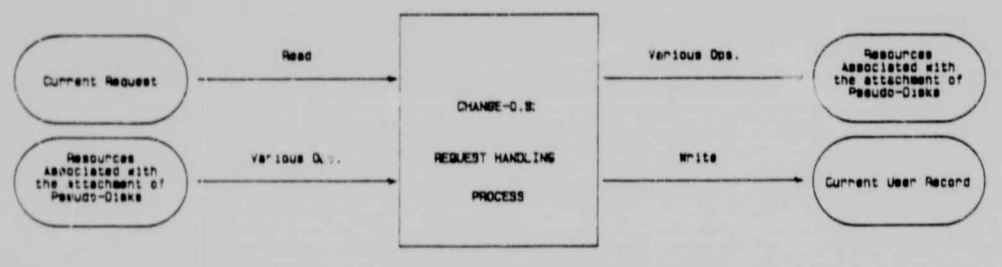
APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.3: DECOMPOSITION OF THE NODE-SERVER PROCESS

(g) PROCESS A REQUEST FOR A COMMON DISK

(h) PROCESS A REQUEST FOR A USER DISK

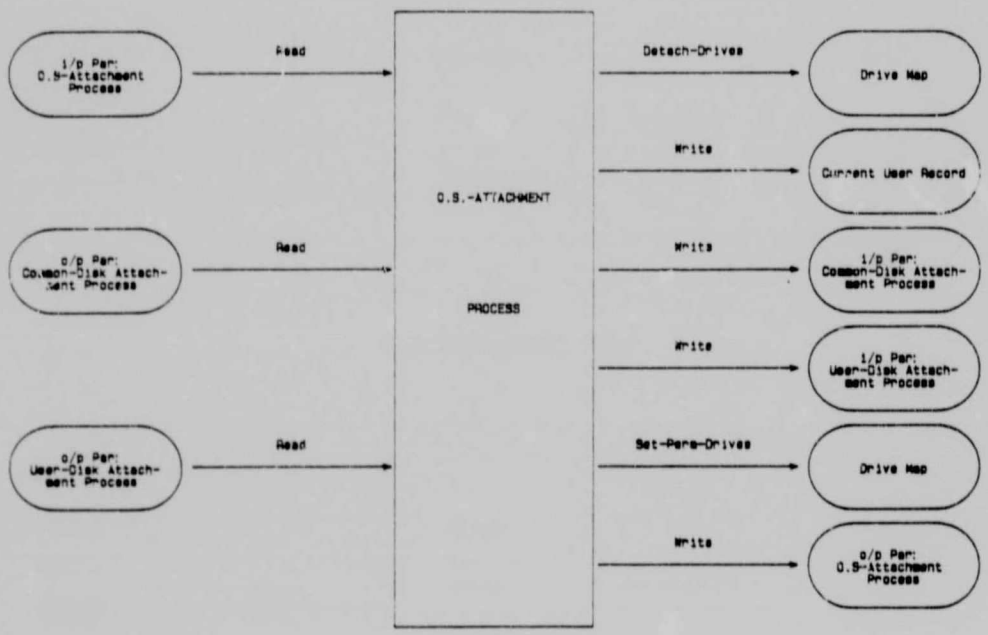
(i) PROCESS A REQUEST TO DETACH DRIVES



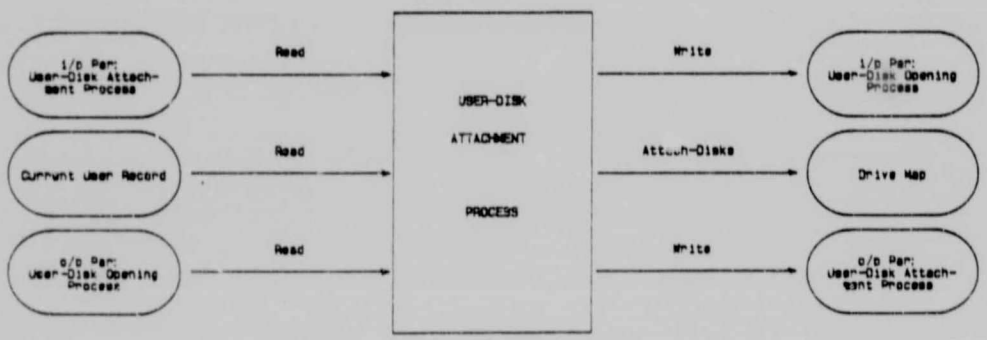
(j) PROCESS A REQUEST FOR AN OPERATING SYSTEM

APPENDIX A.4 - INTERACTIVE SERVING: 4TH LEVEL

FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS



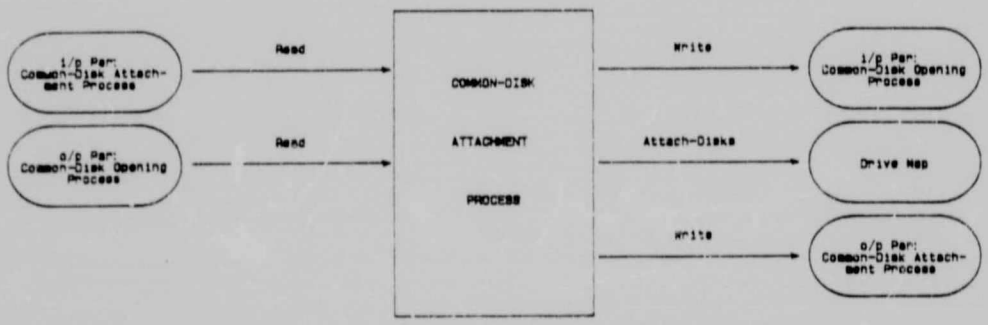
(a) O.S-ATTACHMENT PROCESS



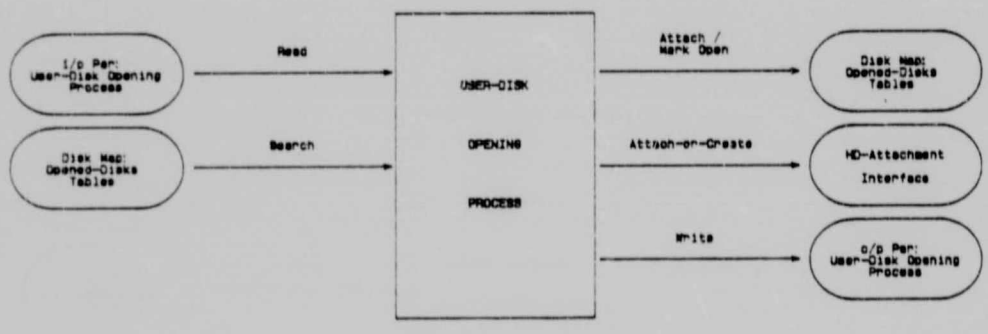
(b) USER-DISK ATTACHMENT PROCESS

APPENDIX A.4 - INTERACTIVE SERVING: 4TH LEVEL

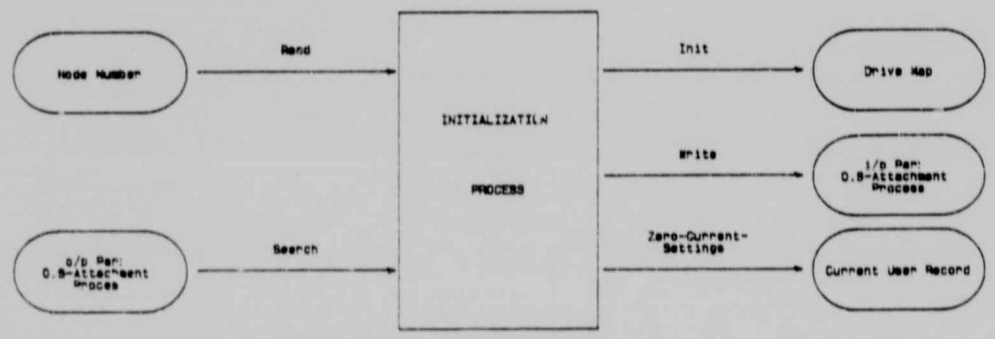
FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS



(c) COMMON-DISK ATTACHMENT PROCESS



(d) USER-DISK OPENING PROCESS

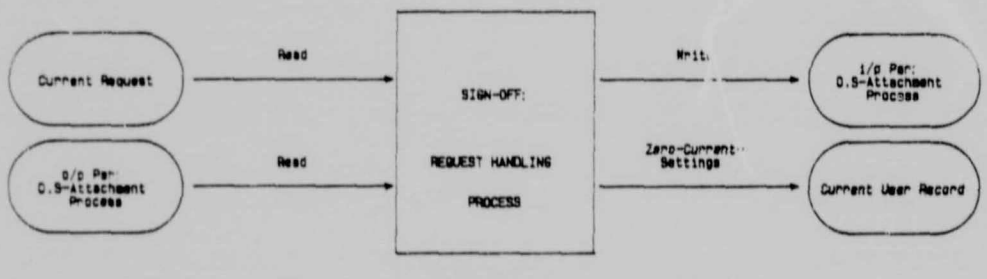


(e) INITIALIZATION PROCESS

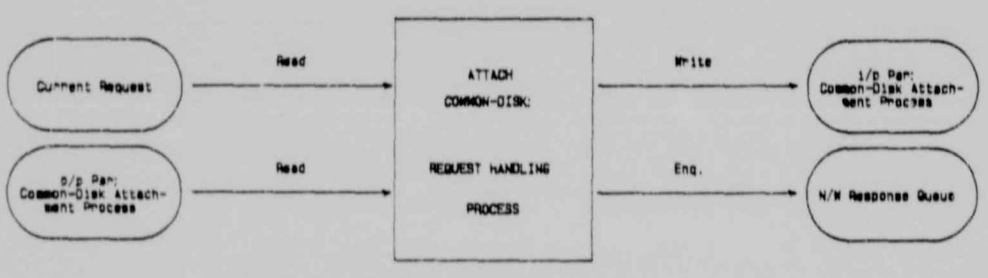
APPENDIX A.4 - INTERACTIVE SERVING: 4TH LEVEL

FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS

(f) PROCESS A REQUEST TO SIGN-ON



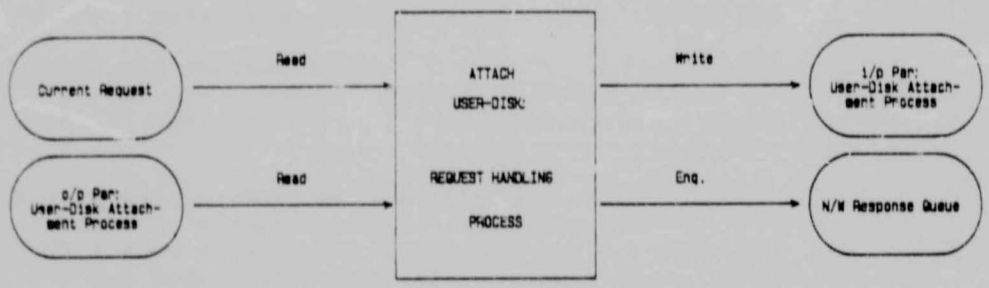
(g) PROCESS REQUEST TO SIGN-OFF



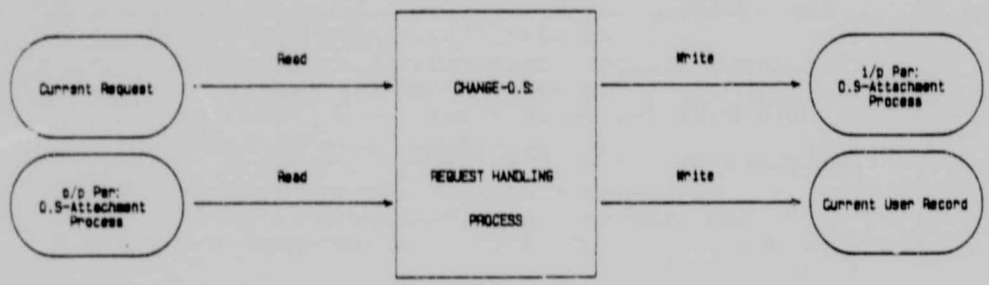
(h) PROCESS A REQUEST FOR A COMMON DISK

APPENDIX A.4 INTERACTIVE SERVING: 4TH LEVEL

FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS



(1) PROCESS A REQUEST FOR A USER DISK



(1) PROCESS A REQUEST FOR AN OPERATING SYSTEM

APPENDIX B: PROGRAM LISTINGS

B.1 EXTERNAL RESOURCES: THE NETWORK-SERVER INTERFACE

The resources comprising the network-server interface, and shared by the server and network software, are first-level, and external to the serving process. However, the resources are created and initialized in the server's code.

B.1.1 Network Request Queue - *File:* NW$INP.RSC

```
/* ------------------------------------------------------------------------------------ */

/* RESOURCE: Network Input */
   /* Requests are written to and read from an input buffer. Input is from the network interface, using
      DMA. This requires some form of mutual exclusion for update of queue variables. */

  /*CONSTs*/
     Declare
       Max$Inputs          literally   '40',
       Rqst$Size           literally   '20';

  /*TYPEs*/
     Declare
       Rqst$Type           literally   '(Rqst$Size) Byte';  /* An Array of Bytes */

  /*VARs*/
     Declare
       NW$Inp_Queue$Size   Byte,
       NW$Inp_Read$Pos     Byte,
       NW$Inp_Write$Pos    Byte,
       NW$Inp_Queue (Max$Inputs)  STRUCTURE ( Node    Node$Type,
                                              Rqst    Rqst$Type);

  /*ACCESS RELATED DATA AND PROCEDURES */
     /*VAR-access*/
        Declare
          NW$Inp_FS        Boolean,    /* File Server has or is requesting access */
          NW$Inp_NW        Boolean,    /* Network Processor has or is requesting access */
          NW$Inp_S3        Boolean;    /* Access to the resource has been granted to one of the Processors */


     /*PROCEDURE */ NW$Inp_FS$Access: Procedure  Boolean;
        /* The File Server is the consumer and is given priority */
        /*BEGIN*/
           IF NOT(NW$Inp_NW)
             THEN DO;
                  NW$Inp_FS = True;
                  DO WHILE (NW$Inp_NW AND (NOT NW$Inp_S3) );
                  END; /* Loop while access not yet decided - gives higher priority */
                  IF (NW$Inp_NW)
                    THEN NW$Inp_FS = False;
                    ELSE NW$Inp_S3 = True;
                END;
          RETURN (NW$Inp_FS);
        END NW$Inp_FS$Access;

     /*PROCEDURE */ NW$Inp_FS$Release: Procedure;
        /*BEGIN*/
          NW$Inp_S3 = False;
```

```
        NVS$Inp_PS : False
      END NVS$Inp_PS$Release;


  /* Routines allowing the Network Processor access, do not reside on the File Server, but are given for completeness */

    /*PROCEDURE  NVS$Inp_NVSAccess: Procedure  Boolean;
      BEGIN
        IF (NOT NVS$Inp_PS)
          THEN DO;
                 NVS$Inp_NV : True;
                 IF (NVS$Inp_PS)
                   THEN NVS$Inp_NV : False;  ** Back-off to allow server the access **
                   ELSE NVS$Inp_S3 : True;
               END;
        RETURN (NVS$Inp_NV);
      END NVS$Inp_NVSAccess; */

    /*PROCEDURE  NVS$Inp_NVSRelease: Procedure;
      BEGIN
        NVS$Inp_S3 : False;
        NVS$Inp_NV : False;
      END NVS$Inp_NVSRelease; */



/*PROCEDURE */ NVS$Inp_Init: Procedure;

  /*BEGIN*/
    NVS$Inp_S3 : False;
    NVS$Inp_PS : False;
    NVS$Inp_NV : False;
    NVS$Inp_Queue$Size : 0;
    NVS$Inp_Write$Pos  : 0;
    NVS$Inp_Read$Pos   : 0;
  END NVS$Inp_Init;


/* The producing action is performed by the Network Processor. The enqueue operator does not reside
     on the File Server, but is given for completeness. */

  /*PROCEDURE  NVS$Inp_Enqueue: Procedure;
    BEGIN
      IF (NVS$Inp_Queue$Size = Max$Inputs)
        THEN Status : Full;
        ELSE
          DO;
            ** Construct message at current write position - i.e. NVS$Inp_Queue (NVS$Inp_Write$Pos);
                 In addition to the original request, the message must contain, the node requesting, & the
                 position of data associated with the request - i.e. Sector Data for a write request **
            Status : Successful;

            ** Get access for the Network processor to shared variables & update queue information **
            DO WHILE ( NOT (NVS$Inp_NVS$Access) );  END;  ** Loop until access is granted **
            NVS$Inp_Write$Pos  : NVS$Inp_Write$Pos + 1;
            IF (NVS$Inp_Write$Pos = Max$Inputs)
              THEN NVS$Inp_Write$Pos : 0;
            NVS$Inp_Queue$Size : NVS$Inp_Queue$Size + 1;
            CALL NVS$Inp_NVS$Release;
          END;
```

```
        END NVS$Inp_Rxqueue; */


    /*PROCEDURE */ NVS$Inp_Dequeue: Procedure (Rqst$Entry$Ptr, Status$Ptr);
                                   Declare  Rqst$Entry$Ptr   Pointer,
                                            Status$Ptr       Pointer;
       /*VAR*/
          Declare
            Status    BASED Status$Ptr    Status$Type;

       /*BEGIN*/
          IF (NVS$Inp_Queue$Size = 0)
            THEN Status = Empty;
            ELSE
              DO;
                /* Copy the request section of the input to the receiving entry */
                CALL MOVB ( @NVS$Inp_Queue(NVS$Inp_Read$Pos).Rqst, Rqst$Entry$Ptr, Rqst$Size);
                Status = Success$;

                /* Get access for the File Server to shared variables & update queue information */
                DO WHILE ( NOT (NVS$Inp_FS$Access) );  END;  /* Loop until access is granted */
                NVS$Inp_Read$Pos = NVS$Inp_Read$Pos + 1;
                IF (NVS$Inp_Read$Pos = Max$Inputs)
                  THEN NVS$Inp_Read$Pos = 0;
                NVS$Inp_Queue$Size = NVS$Inp_Queue$Size - 1;
                CALL NVS$Inp_FS$Release;
              END;
        END NVS$Inp_Dequeue;

/* END  Network Input RESOURCE */

/* ------------------------------------------------------------------------------------------ */
```

B.1.2 Sector Pool - *File:* NW$SECT.RSC

```
/* ------------------------------------------------------------------------------ */

/* RESOURCE: Sector Pool */
   /* This provides entries for sector information passed with write requests. The information must remain
      present until the File Server has successfully performed the write request, to it's cache */

  /*CONST*/
     Declare
       Max$Sects       literally   '20',
       Non$Sect        literally   'Max$Sects';

  /*TYPE*/
     Declare
       Type$Sect$Num   literally   'Byte',
       Sect$Num        literally   'Type$Sect$Num',
       Sect$Entry      literally   'STRUCTURE    ( Sect$Pos   Pointer,
                                                   SL$fp      Sect$Num )';

  /*VAR*/
     Declare
       SP$Bot   (Max$Sects)    Sect$Entry,
       Free$Sects              Sect$Num,
       SP$SLR                  Sect$Num;

  /*ACCESS RELATED DATA AND PROCEDURES */
     /*VAR-access*/
        Declare
          SP_FS        Boolean,        /* File Server has or is requesting access */
          SP_NW        Boolean,        /* Network Processor has or is requesting access */
          SP_S3        Boolean;        /* Access to the resource has been granted to one of the Processors */


     /*PROCEDURE */ SP_FS$Access: Procedure   Boolean;
        /* The File Server is the consumer & is given priority */
        /*BEGIN*/
           IF NOT(SP_NW)
             THEN DO;
                    SP_FS : True;
                    DO WHILE (SP_NW AND (NOT SP_S3) );
                    END; /* Loop while access not yet decided - gives higher priority */
                    IF (SP_NW)
                      THEN SP_FS : False;
                      ELSE SP_S3 : True;
                  END;
          RETURN (SP_FS);
        END SP_FS$Access;

     /*PROCEDURE */ SP_FS$Release: Procedure;
        /*BEGIN*/
          SP_S3 : False;
          SP_FS : False
        END SP_FS$Release;


    /* Routines allowing the Network Processor access, do not reside on the File Server, but are given for completeness */

      /*PROCEDURE  SP_NW$Access: Procedure   Boolean;
```

```
        BEGIN
          IF (NOT SP_FS)
            THEN DO;
                   SP_HW : True;
                   IF (SP_FS)
                     THEN SP_HW : False;      /* Back-off to allow server the access */
                     ELSE SP_SJ : True;
                 END;
          RETURN (SP_HW);
        END SP_HW$Access; */

      /*PROCEDURE  SP_HW$Release: Procedure;
        BEGIN
          SP_SJ : False;
          SP_HW : False
        END SP_HW$Release; */


/*PROCEDURE */ SP_Init: Procedure;
   /*VARS/
      Declare
        Sect        Sect$Num,
        Segment     Token,
        Exception   IO$Exception;
   /*BEGIN*/
      SP_SJ : False;
      SP_FS : False;
      SP_HW : False;
      DO Sect = 0 TO (Max$Sects - 1);
        Segment = RQ$CREATE$SEGMENT (Sector$Size, @Exception);
        SP$Ent(Sect).Sect$Pos = Build$Ptr (0, Segment);
        SP$Ent(Sect).SL$fp    = Sect + 1;
      END;
      Free$Sects = Max$Sects;
    END SP_Init;


/*PROCEDURE */ SP_Release$Sect: Procedure (Sect);
         /* Releases the Sector Pool entry, to the list of free entries */
   /*BEGIN*/
      /* Get access for the File Server to shared variables and update list information */
      DO WHILE ( NOT (SP_FS$Access) );  END;  /* Loop until access is granted */
      IF (SP$SLR = Nom$Sect)
        THEN SP$Ent(Sect).SL$fp = Nom$Sect;
        ELSE SP$Ent(Sect).SL$fp = SP$SLR;
      SP$SLR = Sect;
      Free$Sects = Free$Sects + 1;
      CALL SP_FS$Release;
    END SP_Release$Sect;


/* The acquisition of a sector entry to store sector data, is performed by the Network Processor. The Get$Sector
     operator does not reside on the File Server, but is given for completeness. */

/*PROCEDURE  SP_Get$Sect: Procedure (Sect$Ptr, Status$Ptr);
                        Declare  Sect$Ptr    Pointer,
                                 Status$Ptr  Pointer;
       /* Returns the number of an available sector entry, if there is one */
    VAR
```

```
    Declare
      Status    BASED Status$Ptr    Status$Type,
      Sect      BASED Sect$Ptr      Sect$Num;
    BEGIN
      IF (Free$Sects = 0)
        THEN Status = Empty;
        ELSE
          DO;
            /* Get access for the Network processor to shared variables & update list information */
            DO WHILE ( NOT (SP_NW$Access) );  END;  /* Loop until access is granted */
            Status = Successful;
            Sect   = SP$SLR;
            SP$SLR = SP$Ent(Sect).SL$fp;
            SP$Ent(Sect).SL$fp = Non$Sect;
            Free$Sects = Free$Sects - 1;
            CALL SP_NW$Release;
          END;
      END SP_Get$Sect; */

/* END  Sector Pool RESOURCE */

/* ------------------------------------------------------------------------ */
```

B.1.3 Network Response Queue - File: NW$OUT.RSC

```
/* ------------------------------------------------------------------------------ */

/* RESOURCE: Network Output */
   /* Requests are written to & read from an output buffer. Input is from the server tasks. The queue is read by the
      network interface, using DMA. This requires some form of mutual exclusion for update of queue variables */

  /*CONSTS*/
     Declare
       Max$Outputs          literally     '40',
       Message$Size         literally     '20';

  /*TYPES*/
     Declare
       Message$Type         literally     '(Message$Size)  Byte',   /* An Array of Bytes */

  /*VARS*/
     Declare
       NW$Out$Region        Token,
       NW$Out_Qty$Free      Byte,
       NW$Out_Queue$Size    Byte,
       NW$Out_Read$Pos      Byte,
       NW$Out_Write$Pos     Byte,
       NW$Out_Done$Pos      Byte,
       NW$Out_Queue (Max$Outputs) STRUCTURE ( Done     Boolean,
                                              Node     Node$Type,
                                              Message  Message$Type );


  /*ACCESS RELATED DATA AND PROCEDURES */
     /*VAR-access*/
        Declare
          NW$Out_PS       Boolean,    /* File Server has or is requesting access */
          NW$Out_NW       Boolean,    /* Network Processor has or is requesting access */
          NW$Out_S3       Boolean;    /* Access to the resource has been granted to one of the Processors */

     /*PROCEDURE */ NW$Out_PS$Access: Procedure  Boolean;
        /*BEGIN*/
           IF (NOT NW$Out_NW)
             THEN DO;
                    NW$Out_PS : True;
                    IF (NW$Out_NW)
                      THEN NW$Out_PS : False;  -- Back-off to allow Network Processor the access --
                      ELSE NW$Out_S3 : True;
                  END;
           RETURN (NW$Out_PS);
         END NW$Out_PS$Access;

     /*PROCEDURE */ NW$Out_PS$Release  Procedure;
        /*BEGIN*/
           NW$Out_S3 : False;
           NW$Out_PS : False
         END NW$Out_PS$Release;
```

```
/* Routines allowing the Network Processor access, do not reside on the File Server, but are given for completeness */

  /*PROCEDURE NW$Out_NW$Access: Procedure  Boolean;
     ** The Network Processor is the consumer & is given priority **
     BEGIN
       IF NOT(NW$Out_PS);
          THEN DO;
                 NW$Out_NW : True;
                 DO WHILE (NW$Out_PS AND (NOT NW$Out_S3) );
                 END; ** Loop while access not yet decided - gives higher priority **
                 IF (NW$Out_PS)
                    THEN NW$Out_NW : False;
                    ELSE NW$Out_S3 : True;
               END;
       RETURN (NW$Out_NW);
     END NW$Out_NW$Access; */

  /*PROCEDURE NW$Out_NW$Release: Procedure;
     BEGIN
       NW$Out_S3 : False;
       NW$Out_NW : False
     END NW$Out_NW$Release; */



/*PROCEDURE */ NW$Out_Init: Procedure;
   /*BEGIN*/
      NW$Out_S3 : False;
      NW$Out_PS : False;
      NW$Out_NW : False;
      NW$Out_Qty$Free   : Max$Outputs;
      NW$Out_Queue$Size : 0;
      NW$Out_Write$Pos  : 0;
      NW$Out_Read$Pos   : 0;
      NW$Out_Done$Pos   : 0;
    END NW$Out_Init;


/*PROCEDURE */ NW$Out_Enqueue: Procedure (Message$Ptr, Length, Node, Status$Ptr);
                              Declare  Message$Ptr      Pointer,
                                       Length           Byte,
                                       Node             Node$Type,
                                       Status$Ptr       Pointer;
   /*VARS*/
      Declare
        Status    BASED Status$Ptr   Status$Type;

   /*BEGIN*/
      IF (NW$Out_Qty$Free = 0)
        THEN Status = Full;
        ELSE
          DO;
            /* Assign the destination node (first byte) & the message contents */
            NW$Out_Qty$Free = NW$Out_Qty$Free - 1;
            NW$Out_Queue(NW$Out_Write$Pos).Done = False;
            NW$Out_Queue(NW$Out_Write$Pos).Node = Node;
```

```
    /* Routines allowing the Network Processor access, do not reside on the File Server, but are given for completeness */

      /*PROCEDURE NW$Out_NW$Access: Procedure  Boolean;
         ** The Network Processor is the consumer & is given priority **
         BEGIN
           IF NOT(NW$Out_PS)
              THEN DO;
                   NW$Out_NW : True;
                   DO WHILE (NW$Out_PS AND (NOT NW$Out_S3) );
                   END; ** Loop while access not yet decided - gives higher priority **
                   IF (NW$Out_PS)
                      THEN NW$Out_NW : False;
                      ELSE NW$Out_S3 : True;
                 END;
           RETURN (NW$Out_NW);
         END NW$Out_NW$Access; */

      /*PROCEDURE NW$Out_NW$Release: Procedure;
         BEGIN
           NW$Out_S3 : False;
           NW$Out_NW : False
         END NW$Out_NW$Release; */



/*PROCEDURE */ NW$Out_Init: Procedure;
   /*BEGIN*/
      NW$Out_S3 : False;
      NW$Out_PS : False;
      NW$Out_NW : False;
      NW$Out_Qty$Free  : Max$Outputs;
      NW$Out_Queue$Size : 0;
      NW$Out_Write$Pos  : 0;
      NW$Out_Read$Pos   : 0;
      NW$Out_Done$Pos   : 0;
    END NW$Out_Init;


/*PROCEDURE */ NW$Out_Enqueue: Procedure (Message$Ptr, Length, Node, Status$Ptr);
                              Declare  Message$Ptr     Pointer,
                                       Length          Byte,
                                       Node            Node$Type,
                                       Status$Ptr      Pointer;
   /*VARS*/
      Declare
        Status    BASED Status$Ptr   Status$Type;

   /*BEGIN*/
      IF (NW$Out_Qty$Free : 0)
        THEN Status : Full;
        ELSE
          DO;
            /* Assign the destination node (first byte) & the message contents */
            NW$Out_Qty$Free : NW$Out_Qty$Free - 1;
            NW$Out_Queue(NW$Out_Write$Pos).Done : False;
            NW$Out_Queue(NW$Out_Write$Pos).Node : Node;
```

```
            IF (Length > (Message$Size) )
              THEN Length : Message$Size;
            CALL MOVB ( Message$Ptr, @NW$Out_Queue(NW$Out_Write$Pos).Message, Length);
            Status : Successful;

            /* Get access for the File Server to shared variables & Update queue information */
            DO WHILE ( NOT (NW$Out_FS$Access) ); END;  /* Loop until access is granted */
            NW$Out_Write$Pos : NW$Out_Write$Pos + 1;
            IF (NW$Out_Write$Pos : Max$Outputs)
              THEN NW$Out_Write$Pos : 0;
            NW$Out_Queue$Size : NW$Out_Queue$Size + 1;
            CALL NW$Out_FS$Release;
          END;
      END NW$Out_Enqueue;


/* The consuming action is performed by the Network Processor.  The dequeue operator does not reside
     on the File Server, but is given for completeness.  */

  /*PROCEDURE  NW$Out_Dequeue: Procedure;
      BEGIN
        IF (NW$Out_Queue$Size : 0)
          THEN Status : Empty;
          ELSE
            DO;
              !! Process the response message at the current read position (i.e. NW$Out_Queue (NW$Out_Write$Pos));
                 Send it to the required node - or to the Network processor's memory, & perform actions required,
                 such as transmitting track information referenced in a read response !!
              !! Set second byte of message i.e. Done : True !!
              NW$Out_Queue(NW$Out_Read$Pos).Done : True;
              Status : Successful;

              !! Get access for the Network processor to shared variables & update queue information !!
              DO WHILE ( NOT (NW$Out_NW$Access) ); END;  !! Loop until access is granted !!
              NW$Out_Read$Pos : NW$Out_Read$Pos + 1;
              IF (NW$Out_Read$Pos : Max$Outputs)
                THEN NW$Out_Read$Pos : 0;
              NW$Out_Queue$Size : NW$Out_Queue$Size - 1;
              CALL NW$Out_NW$Release;
              END;
      END NW$Out_Dequeue; */


/*PROCEDURE */ NW$Out_Done: Procedure (Msg$Pos$Ptr) Boolean;
                          Declare  Msg$Pos$Ptr   Pointer;
          /* Determines whether an output message has already been processed by the Network Processor */
    /*VARs/
        Declare
          Msg$Pos    BASED  Msg$Pos$Ptr   Pointer;
    /*BEGIN*/
        Msg$Pos : @NW$Out_Queue(NW$Out_Done$Pos).Message;
        IF (NW$Out_Done$Pos : NW$Out_Read$Pos)
          THEN RETURN (False);
          ELSE RETURN ( NW$Out_Queue(Done$Pos).Done );
      END NW$Out_Done;
```

```
  /*PROCEDURE */ NW$Out_Release: Procedure;
            /* Releases next done entry to the free entries */
      /*BEGIN*/
         NW$Out_Qty$Free = NW$Out_Qty$Free + 1;
         NW$Out_Done$Pos = NW$Out_Done$Pos + 1;
         IF (NW$Out_Done$Pos = Max$Outputs)
           THEN NW$Out_Done$Pos = 0;
       END NW$Out_Done;

/* END  Network Output RESOURCE */

/* ------------------------------------------------------------------------------------------ */
```

B.2 SERVER PROCESS-BODY

The server module is the entire server program comprised from the inclusion of the code for all the external and internal resources, and processes. It also includes the external definitions for iRMX operating system calls (a listing has not been included) and the global literals, constants and types. Procedures for simple string manipulation, were included since PLM does not include a library of string handling routines.

B.2.1 Server Module - File: FSMODULE.SRC

```
FILE$SERVER: DO; /* Beginning of Module */

/* Include Common literals */
$INCLUDE (Common.Lit)

/* Include PLM External Procedure definitions for iBXC Subsystems:
    Basic IO, Extended IO and Nucleus Operating Systems */
$INCLUDE (/BXC5.0/DUTILS/IOS.EXT)
$INCLUDE(/BXC5.0/DUTILS/NUCLUS.EXT)

/* Include libraries of necessary routines */
$INCLUDE (Strings.Lib)

/* Include Resources */
$INCLUDE (Hds$Attach.RSC)
$INCLUDE (Cons$Info.RSC)
$INCLUDE (User$Info.RSC)
$INCLUDE (Ops$Table.RSC)
$INCLUDE (NVS$Inp.RSC)
$INCLUDE (NVS$Out.RSC)
$INCLUDE (NVS$Sect.RSC)
$INCLUDE (Node$Rqst.RSC)
$INCLUDE (Dsk$Req$Q.RSC)
$INCLUDE (Cache.RSC)

/* Include Common Procedures */
$INCLUDE (Cons$Open.RTN)

/* Include Processes */
$INCLUDE (Dsk$Read.PRS)
$INCLUDE (Dsk$Write.PRS)
$INCLUDE (Cpl$Sends.PRS)
$INCLUDE (Server.PRS)
$INCLUDE (Q$Produc.PRS)
$INCLUDE (Init.PRS)

/*BEGIN - FILE$SERVER (Code) */
   /* Setup the Fileserver as the default user of the iBXC Operating System. */
   FS$User = RQ$CREATE$USER (@FS$Id$Entry, @Exception);
   CALL RQ$SET$DEFAULT$USER (Def$Job, FS$User, @Exception);
   /* Start the initial process */
   CALL Initialization;

END FILE$SERVER;
```

B.2.2 Global Literals, Constants & Types - *File*: COMMON.LIT

```
/* COMMON Compilation Literals, Constants & Types */

  /*CONSTs*/
     Declare
       True                literally   '0FFh',
       False               literally   '000h',
       Forever             literally   'While True',
       Max$Nodes           literally   '50';

     /*Status Responses*/
     Declare
       Successful          literally   '0',
       Found               literally   'Successful',
       Ready               literally   'Successful',
       Granted             literally   'Successful',
       User$Denied         literally   '1',
       Appl$Not$Exist      literally   '2',
       Disk$Not$Exist      literally   '2',
       Read$Access$Denied  literally   '3',
       Write$Access$Denied literally   '4',
       Wrong$Loader        literally   '5',
       Wrong$OS            literally   '6',
       Too$Many$Appls      literally   '7',
       Too$Many$Disks      literally   '8',
       No$Disk$Space       literally   '9',
       Cannot$Load$All     literally   '9',
       Open$As$Read        literally   '10',
       Open$As$Write       literally   '11',
       Exceed$Opens        literally   '12',
       Disk$Attach$Error   literally   '13',
       No$Drive$Space      literally   '14',
       Already$Signed$On   literally   '15',
       Full                literally   '16',
       Empty               literally   '17';

  /*TYPEs*/
     Declare
       Token               literally   'Selector',
       Region              literally   'Token',
       Boolean             literally   'Byte',
       Node$Type           literally   'Byte',
       Status$Type         literally   'Byte';
```

```
/* PC & Floppy Disk dependent constants & types */

  /*CONSTS*/
      Declare
        Sect$Size           literally     '200h',
        Sects$In$Track      literally     '9',
        Track$Size          literally     '1200h',
        Qty$Heads           literally     '2',
        Tracks$In$Cache$Track literally '2',
        Sects$In$Cache$Track  literally '18',
        Size$Cache$Track    literally     '2400h';

  /*TYPES*/
      Declare
        Type$PC_Track       literally     'Byte',
        Type$PC_Sect        literally     'Byte';
```

B.2.3 String Manipulation Routines - *File:* STRINGS.LIB

```
/* ------------------------------------------------------------------------------------------ */

/* STRING HANDLING ROUTINES */
    /* Routines for string concatenation, comparison & conversion of
       numbers to decimal number strings */
    /* A string is an array of (Size) characters with the current length of
       the string contained in the first byte */


  /*CONST*/
     Declare
       Empty$String         literally     '0';


  /*PROCEDURE */ String_Compare: Procedure (A$Ptr, B$Ptr) Boolean  REENTRANT;
                                 Declare  A$Ptr  Pointer,
                                          B$Ptr  Pointer;

     /* Compares each byte of two strings including their length bytes
        for equality */

     /*VAR*/
        Declare
          Count  BASED  A$Ptr  Byte;
     /*BEGIN*/
        IF ( CMPB (A$Ptr, B$Ptr, (Count+1)) = 0FFFFh )
          THEN RETURN (True);
          ELSE RETURN (False);
      END String_Compare;


  /*PROCEDURE */ String_Add: Procedure (Sum$Ptr, B$Ptr, Max$Len)  REENTRANT;
                             Declare  Sum$Ptr  Pointer,
                                      B$Ptr    Pointer,
                                      Max$Len  Byte;

     /* Adds the contents of the second string to the first. If Max$Len will be
        exceeded, the resulting string is truncated */

     /*VAR*/
        Declare
          (Sum$    BASED  Sum$Ptr) (1)    Byte,  /* At least 1 element */
          ( B$     BASED  B$Ptr)   (1)    Byte;  /* At least 1 element */
        Declare
          (Sum$Len BASED  Sum$Ptr)        Byte,  /* = Sum$(0) */
          ( B$Len  BASED  B$Ptr)          Byte;  /* = B$(0)   */

     /*BEGIN*/
        IF ( (Sum$Len + B$Len) <= Max$Len )
          THEN
            DO;
              CALL MOVB ( @B$(1), @Sum$(Sum$Len + 1), B$Len);
              Sum$Len = Sum$Len + B$Len;
            END; /* Then */
          ELSE
            DO;
              CALL MOVB ( @B$(1), @Sum$(Sum$Len + 1), (Max$Len - Sum$Len));
```

```
            Sum$Len : Max$Len;
          END; /* Else */
      END String_Add;


/*PROCEDURE */ String_Num: Procedure (Num, Num$Str$Ptr, Size)   REENTRANT;
                          Declare  Num          Word,
                                   Num$Str$Ptr Pointer,
                                   Size         Byte;

    /* A string of (Size) decimal digits is created, without
       zero suppression */

    /*CONST*/
       Declare
         Ascii$Zero    literally    '30h';
    /*VAR*/
       Declare
         (Num$Str  BASED  Num$Str$Ptr)  (1)  Byte,
         Count     Integer;

    /*BEGIN*/
       Num$Str(0) : Size;
       DO Count : Size TO 1  BY (-1);
         Num$Str(Count) : (Num MOD 10) + Ascii$Zero;
         Num : Num / 10;
       END;
     END String_Num;

/* END STRING HANDLING LIBRARY */

/* ------------------------------------------------------------------------------------------ */
```

B. 3 INTERNAL (2ND-LEVEL) RESOURCES

B.3.1 Node Queues - File: NODERQST.RSC

```
/* RESOURCE: Individual Node Request Queues */
     /* A queue of requests exists for each server task */

   /*COMST*/
      Declare
        Max$Rqsts            literally    '50',
        Non$Rqst             literally    'Max$Rqsts';

   /*VAR*/
      Declare
        Node$Rqsts$Region    Token,
        NR_SLR               Byte;    /* Space List Rock - First entry of a list of available entries */

      Declare
        Node$Rqsts (Max$Rqsts)       STRUCTURE  ( Next    Byte,
                                                  Rqst    Rqst$Type),
        Node$Queues (Max$Nodes)      STRUCTURE  ( Server  Token,
                                                  First   Byte,
                                                  Last    Byte );

   /*PROCEDURE */ Node$Rqsts_Init: Procedure;
      /*VAR*/
         Declare
           Count             Node$Type;
      /*BEGIN*/
         /* Zero the base info for each nodes queue, i.e. no task exists & queue is empty */
         DO Count = 0 TO (Max$Nodes - 1);
           Node$Queues(Count).Server = Non$Token;
           Node$Queues(Count).First  = Non$Rqst;
           Node$Queues(Count).Last   = Non$Rqst;
         END;

         /* Initially all request entries are in the space list - i.e. available */
         NR_SLR = 0;
         DO Count = 0 TO (Max$Rqsts - 1);
           Node$Rqsts(Count).Next = Max$Rqsts + 1;
         END;
       END Node$Rqst_Init;


/*PROCEDURE */, Node$Rqsts_Get$Entry: Procedure (Entry$Num$Ptr, Status$Ptr);
                                      Declare  Entry$Num$Ptr  Pointer,
                                               Status$Ptr     Pointer;
   /*VAR*/
      Declare
        Entry$Num    BASED Entry$Num$Ptr     Byte,
        Status       BASED Status$Ptr        Status$Type;

   /*BEGIN*/
      IF (NR_SLR = Non$Rqst)
        THEN Status = Full;
        ELSE
```

```
            DO;
              /* Remove the first entry from the spacelist */
              Entry$Num = NR_SLR;
              NR_SLR = Node$Rqsts(NR_SLR).Next;
              Status = Successful;
            END;
      END Node$Rqsts_Get$Entry;


/*PROCEDURE */ Node$Rqsts_Release$Entry: Procedure (Entry$Num);
                                        Declare  Entry$Num  Byte;

    /*BEGIN*/
       /* Insert the entry at the beginning of the space list */
       Node$Rqsts(Entry$Num).Next = NR_SLR;
       NR_SLR  = Entry$Num;
      END Node$Rqsts_Release$Entry;


/*PROCEDURE */ Node$Rqsts_Enqueue: Procedure (Node, Status$Ptr);
                                  Declare  Node        Node$Type,
                                           Status$Ptr  Pointer;
    /*CONST*/
       Declare
         Queue          literally   'Node$Queues(Node)',  /* Queue for the calling server's node */
         New$Rqst       literally   'Node$Rqsts(Entry$Num)';
    /*VAR*/
       Declare
         Status         BASED Status$Ptr     Status$Type,
         Entry$Num      Byte;

    /*BEGIN*/
       CALL Node$Rqsts_Get$Entry (@Entry$Num, Status$Ptr);
       IF (Status = Successful)
         THEN
           DO;
             /* Dequeue a request from the Network Input Queue */
             CALL NW$Inp_Dequeue ( @New$Rqst.Rqst, Status$Ptr);

             /* Insert the entry at the end of the queue for this node */
             IF ( Queue.First = Non$Rqst)
               THEN  Queue.First = Entry$Num;   /* The node's queue is empty */
               ELSE  Node$Rqsts(Queue.Last).Next = Entry$Num;
             Queue.Last  = Entry$Num;
             New$Rqst.Next = Non$Rqst;
           END;
      END Node$Rqsts_Enqueue;


/*PROCEDURE */ Node$Rqsts_Dequeue: Procedure (Node, Receive$Ptr, Status$Ptr);
                                  Declare  Node        Node$Type,
                                           Receive$Ptr  Pointer,
                                           Status$Ptr   Pointer;


    /*CONST*/
       Declare
         Queue          literally   'Node$Queues(Node)';  /* Queue for the calling server's node */
    /*VAR*/
       Declare
         First$Rqst     Byte,
```

```
        Status          BASED Status$Ptr     Status$Type;

    /* BEGIN */
      IF (Queue.First = Non$Rqst)
        THEN Status = Empty;
        ELSE
          DO;
            /* Copy the first entry's request to the calling task */
            First$Rqst = Queue.First;
            CALL MOVB ( @Node$Rqsts(First$Rqst).Rqst, Receive$Ptr, Rqst$Size);

            /* Remove the first entry from the queue */
            Queue.First = Node$Rqsts(Queue.First).Next;
            IF (Queue.First = Non$Rqst)
              THEN Queue.Last = Non$Rqst;

            /* Release the consumed entry to the space list */
            CALL Node$Rqsts_Release$Entry (First$Rqst);
          END;
      END Node$Rqsts_Dequeue;

/* END  Individual node request queues RESOURCE */
```

B.3.2 HD-Attachment Interface - *File:* HDATT.RSC

```
/* ------------------------------------------------------------------------------------------ */

/* RESOURCE: Hard Disk Attachment */
    /* Includes routines which interface with Operating System functions
        for attaching & creating files & directories */

  /*CONST*/
     Declare
       R$OK                literally   '0',
       Reading             literally   '1',
       Writing             literally   '2',
       Read$And$Wr         literally   '3',
       New$Disk$Size       literally   '1800h',
       Zero$Buffers        literally   '0',
       Non$Token           literally   '0',
       No$Mailbox          literally   'Non$Token',
       Def$User            literally   'Non$Token',
       Def$Prefix          literally   'Non$Token',
       Def$Job             literally   'Non$Token';

  /*TYPES*/
     Declare
       IO$Exception        literally   'Word';

  /*VARS*/
     Declare
       HD$Attach$Region    Region,
       Syst$Prefix         Token,
       Common$Prefix       Token,
       Users$Prefix        Token,
       New$Disk$Buf$Ptr    Pointer,
       Attach$Mbox         Token,
       (New$Disk$Buffer    BASED New$Disk$Buf$Ptr) (New$Disk$Size)  Byte;


  /*PROCEDURE */ Get$Connect$Token: Procedure (Mbox, Token$Ptr, Except$Ptr);
                                     Declare  Mbox       Token,
                                              Token$Ptr  Pointer,
                                              Except$Ptr Pointer;
     /*CONST*/
        Declare
          Segment$Type       literally   '006h',
          Connection$Type    literally   '10ih',
          Wait$Done          literally   '0FFFFh';

     /*VARS*/
        Declare
          Segment$Ptr        Pointer,
          New$Token     BASED Token$Ptr        Token,
          Exception     BASED Except$Ptr       IO$Exception,
         (IO$Segment    BASED Segment$Ptr) (1) Word,
          No$Response        Word,
          Dumm$Except        IO$Exception;

     /*BEGIN*/
        IF (Exception <> R$OK) THEN RETURN; /* Already an error */
        New$Token = RQ$RECEIVE$MESSAGE (Mbox, Wait$Done, @No$Response, Except$Ptr);
```

```
        IF (Exception : RSOK) THEN
          IF ( RQ$GET$TYPE (New$Token, Except$Ptr) : Segment$Type )
            THEN
              DO;
                Segment$Ptr : Build$Ptr (New$Token,0);
                Exception   : IO$Segment(0);
                CALL RQ$DELETE$SEGMENT (New$Token, @Dum$Except);
              END;
          IF (Exception <> RSOK)
            THEN New$Token : Mon$Token;
        END Get$Connect$Token;


/*PROCEDURE */ HD$Attach_Init: Procedure;
    /*CONST*/
        Declare
          Mb$Flags          literally   '0010B',     /* 4-Deep FIFO Queue */
          PS$Syst$Dir       literally   '(20, ''/FILE$SERVER/PS$Syst'')',
          PS$Comm$Dir       literally   '(10, ''^PS$Common'')',
          PS$Users$Dir      literally   '(9, ''^PS$Users'')',
          New$Disks$File    literally   '(8, ''NEWDISKS'')';
    /*VARS*/
        Declare
          Connection        Token,
          Exception         IO$Exception,
          Segment           Token,
          Bytes$Read        Word;

    /*BEGIN*/
        /* Create Mailbox & Prefix Connections */
          Attach$Mbox : RQ$CREATE$MAILBOX (Mb$Flags, @Exception);

        /* Attach prefixes for directories for File Server System, Common &
            Users  & set default prefix as the PS System directory */
          CALL RQ$A$ATTACH$FILE  (Def$User, Def$Prefix, @PS$Syst$Dir, Attach$Mbox, @Exception);
          CALL Get$Connect$Token (Attach$Mbox, @Syst$Prefix, @Exception);
          CALL RQ$SET$DEFAULT$PREFIX (Def$Job, Syst$Prefix, @Exception);

          CALL RQ$A$ATTACH$FILE  (Def$User, Def$Prefix, @PS$Comm$Dir, Attach$Mbox, @Exception);
          CALL Get$Connect$Token (Attach$Mbox, @Common$Prefix, @Exception);
          CALL RQ$A$ATTACH$FILE  (Def$User, Def$Prefix, @PS$Users$Dir, Attach$Mbox, @Exception);
          CALL Get$Connect$Token (Attach$Mbox, @Users$Prefix, @Exception);

        /* Create buffer with first 12 sectors for new dos disk */
          Segment :  RQ$CREATE$SEGMENT  (New$Disk$Size, @Exception);
          New$Disk$Buf$Ptr  : Build$Ptr (Segment,0);
          Connection : RQ$S$ATTACH$FILE (@New$Disks$File, @Exception);
          CALL RQ$S$OPEN    (Connection, Reading, Zero$Buffers, @Exception);
          Bytes$Read : RQ$S$READ$MOVE   (Connection, New$Disk$Buf$Ptr, New$Disk$Size, @Exception);
          CALL  RQ$S$DELETE$CONNECTION  (Connection, @Exception);
      END HD$Attach_Init;



/*PROCEDURE */ HD$Attach_Att$Create$Dir: Procedure
                (A$Ptr, B$Ptr, C$Ptr, Qty$Strings, Prefix, Connect$Ptr, Except$Ptr);
                  Declare
                    ( A$Ptr, B$Ptr, C$Ptr, Connect$Ptr, Except$Ptr )   Pointer,
                      Qty$Strings     Byte,
                      Prefix          Token;
```

```
/*CONSTS*/
    Declare
      RMS$FNFEXIST        literally   '021h',
      String$Size         literally   '31',
      Max$Len             literally   'String$Size-1',
      Full$Access         literally   '1111B';
/*VARS*/
    Declare
      Exception           BASED   Except$Ptr     IO$Exception,
      Reqds$Connect       BASED   Connect$Ptr    Token;
    Declare
      Path (3)            STRUCTURE  ( String(String$Size) Byte ),
      Connection (3)      Token,
      Try                 Byte,
      Count               Byte,
      Except              IO$Exception;

/*BEGIN*/
    DO Count = 1 TO 3;
      Connection(Count)     = Non$Token;
      Path(Count).String(0) = Empty$String;
    END;

    /* Set 3 paths in case files in path do not exist */
    CALL String_Add (@Path(1), A$Ptr, Max$Len);
    IF (Qty$Strings >= 2)
      THEN
        DO;
          CALL String_Add (@Path(2), @Path(1), Max$Len);
          CALL String_Add (@Path(2), @('/')  , Max$Len);
          CALL String_Add (@Path(2), B$ptr   , Max$Len);
        END; /*If Then*/
    IF (Qty$Strings >= 3)
      THEN
        DO;
          CALL String_Add (@Path(3), @Path(2), Max$Len);
          CALL String_Add (@Path(3), @('/')  , Max$Len);
          CALL String_Add (@Path(3), C$ptr   , Max$Len);
        END; /*If Then*/

    /* Attempt to Attach the file - If necessary, create directories in the path */
    Try = Qty$Strings;
    CALL RQ$A$ATTACH$FILE (Def$User, Prefix, @Path(Try), Attach$Mbox, Except$Ptr);
    CALL Get$Connect$Token (Attach$Mbox, @Connection(Try), Except$Ptr);
    Try = Try + 1;
    DO While ( ( (Exception = RMS$FNFEXIST) AND (Try > 1) )
               OR  ( (Exception = RMS$OK) AND (Try < 3) ) );
      IF (Exception = RMS$OK)
        THEN Try = Try + 1;
        ELSE Try = Try - 1;
      CALL RQ$A$CREATE$DIRECTORY (Def$User, Prefix, @Path(Try), Full$Access, Attach$Mbox, Except$Ptr);
      CALL Get$Connect$Token (Attach$Mbox, @Connection(Try), Except$Ptr);
    END; /* While */

    /* Delete Connections of those created & not required */
    DO Count = 1 TO 3;
      IF (Count <> Qty$Strings) THEN
        IF (Connection(Count) <> Non$Token)
          THEN CALL RQ$A$DELETE$CONNECTION (Connection(Count), Nos$Mailbox, @Dums$Except);
    END;
```

```
    Reqd$Connect : Connection(Qty$Strings);
  END HD$Attach_Att$Create$Dir;


/*PROCEDURE */ HD$Attach_Open&Create&File: Procedure
            (Disk, Dir$Connect, Read$Write, Connect$Ptr, Except$Ptr);
                Declare  Disk          Byte,       /* Disk Num relative to it's application */
                         Dir$Connect   Token,
                         Read$Write    Boolean,    /* Mode of opening */
                         Connect$Ptr   Pointer,
                         Except$Ptr    Pointer;
  /*CONST*/
      Declare
        Disk$Str        literally   '(4,''Disk'')',
        E$FNEXIST       literally   '021h',
        Must$Create     literally   'True',
        Full$Access     literally   '1111B',
        Granularity     literally   '2400h';    /* 2 PC-Tracks :  18,  512-byte sectors */
  /*VAR*/
      Declare
        New$Connect     BASED Connect$Ptr   Token,
        Exception       BASED Except$Ptr    IO$Exception;
      Declare
        Disk$Name  (7)  Byte,
        Disk$Num   (3)  Byte,
        Bytes$Written   Word,
        Mode            Byte,
        Dumm$Except     IO$Exception;

  /*BEGIN*/
      /* Form the correct File Name from the given number Disk of the
          particular application */
      New$Connect  = Mom$Token;
      CALL String_Num (Disk, @Disk$Num, 2);
      Disk$Name(0) = Empty$String;
      CALL String_Add (@Disk$Name, @Disk$Str, 6);
      CALL String_Add (@Disk$Name, @Disk$Num, 6);

      /* Attempt to attach the file */
      CALL RQ$A$ATTACH$FILE (Def$User, Dir$Connect, @Disk$Name, Attach$Mbox, Except$Ptr);
      CALL Get$Connect$Token (Attach$Mbox, Connect$Ptr, Except$Ptr);

      /* If the file does not exist, create it and copy the relevant sectors
           for a blank formatted disk */
      IF (Exception = E$FNEXIST)
        THEN
          DO;
            /* Attempt to create the file */
            CALL RQ$A$CREATE$FILE  (Def$User, Dir$Connect, @Disk$Name, Full$Access, Granularity,
                                    New$Disk$Size, Must$Create, Attach$Mbox, Except$Ptr);
            CALL Get$Connect$Token (Attach$Mbox, Connect$Ptr, Except$Ptr);
            IF (Exception = E$OK) THEN
              CALL RQ$S$OPEN ( New$Connect, Writing, Zero$Buffers, Except$Ptr);
            IF (Exception = E$OK) THEN
              Bytes$Written = RQ$S$WRITE$MOVE ( New$Connect, New$Disk$Buf$Ptr,
                                                New$Disk$Size, Except$Ptr);
            IF (Exception = E$OK) THEN
              CALL RQ$S$CLOSE ( New$Connect, Except$Ptr);
            IF (Exception <> E$OK) THEN
```

```
                DO;
                  CALL RQ$A$DELETE$CONNECTION ( New$Connect, No$Mailbox, @Dum$Except);
                  New$Connect = Non$Token;
                END;
            END; /* Then */

        /* If the connection has been attached without error, then open in the required mode */
        IF (Exception = E$OK)
          THEN
            DO;
              IF (Read$Write = True)
                THEN Mode = Read$And$Wr;
                ELSE Mode = Reading;
              CALL RQ$S$OPEN ( New$Connect, Mode, Zero$Buffers, Except$Ptr);
              IF (Exception <> E$OK)
                THEN CALL RQ$A$DELETE$CONNECTION ( New$Connect, No$Mailbox, @Dum$Except);
            END;
      END HD$Attach_Open$Create$File;

/* END HD$ATTACH RESOURCE */

/* ------------------------------------------------------------------------------------------ */
```