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:
CHAPTER 6 - Possible Extensions to the File Server

- 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 PCs 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.
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 application 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.
CHAPTER 7 - Evaluations & Conclusions

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 pseudo-disks 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 6.4. 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 5). 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.
CHAPTER 7 - Evaluations & Conclusions

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

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

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.1: THE COMMON-DISK OPENING PROCESS
A.3.2: DECOMPOSITION OF THE INITIALIZATION PROCESS
A.3.3: DECOMPOSITION OF THE NODE-SERVER PROCESS

APPENDIX A.4 - INTERACTIVE SERVING: 4TH LEVEL

A.4.1 FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS
APPENDIX A - Process-Resource Diagrams

APPENDIX A.1 - INTERACTIVE SERVING: 1ST LEVEL

Abbreviations used in Resource Diagrams

H/W - 'Hardware'
Eng. - 'Emuware'
Des. - 'Dequeue'
Var. Opn. - 'Multiple Operations - Identified at a lower level'
I/O Par. - 'Input Parameters'
O/I Par. - 'Output Parameters'
WL - 'White-List'

Resources associated with the attachment of Pseudo-Disk:

These are the Common Information Resources, HD-attachment Resource &
the Pseudo-Disk Tables (both Application HD & Disk) and the various
operations on these resources are only identified at the lower levels.

- A.2 -
APPENDIX A - Process-Resource Diagrams

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

(a) INITIALIZATION PROCESS

(b) QUEUE-PRODUCING PROCESS

(c) INDIVIDUAL-NODE SERVER PROCESS
APPENDIX A - Process-Resource Diagrams

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

[Diagram of Disk-Read Process]

[Diagram of Disk-Write Process]

[Diagram of Track-Send Process]

- A.4 -
APPENDIX A - Process-Resource Diagrams

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.1: THE COMMON-DISK OPENING PROCESS
APPENDIX A - Process-Resource Diagrams

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.2: DECOMPOSITION OF THE INITIALIZATION PROCESS

(a) RESOURCE PREPARATION PROCESS

(b) CONFIGURATION PROCESS
APPENDIX A - Process-Resource Diagrams

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 - Process-Resource Diagrams

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.3: DECOMPOSITION OF THE NODE-SERVER PROCESS

[Diagram of process-resource interactions]

(a) PROCESS REQUEST TO WRITE A SECTOR

(b) PROCESS REQUEST TO SIGN-ON

(c) PROCESS REQUEST TO SIGN-OFF
APPENDIX A - Process-Resource Diagrams

APPENDIX A.3 - INTERACTIVE SERVING: 3RD LEVEL

A.3.3: DECOMPOSITION OF THE NODE-SERVER PROCESS

(1) PROCESS A REQUEST FOR A COMMON DISK

(2) PROCESS A REQUEST FOR A USER DISK

(3) PROCESS A REQUEST TO DETACH DRIVES

(4) PROCESS A REQUEST FOR AN OPERATING SYSTEM
APPENDIX A - Process-Resource Diagrams

APPENDIX A.4 - INTERACTIVE SERVING: 4TH LEVEL

FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS

[Diagram of process and resource decomposition]

(a) O.S-ATTACHMENT PROCESS

(b) USER-DISK ATTACHMENT PROCESS

- A.10 -
APPENDIX A - Interactive Serving: 4th Level

Further Decomposition of the Node-Server Process

FIG. COMMON-DISK ATTACHMENT PROCESS

FIG. USER-DISK OPENING PROCESS

FIG. INITIALIZATION PROCESS
APPENDIX A - Process-Resource Diagrams

APPENDIX A.4 - INTERACTIVE SERVING: 4TH LEVEL

FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS

1) PROCESS A REQUEST TO SIGN-ON

2) PROCESS REQUEST TO SIGN-OFF

3) PROCESS A REQUEST FOR A COMMON DISK
APPENDIX A - Process-Resource Diagrams

APPENDIX A.4 INTERACTIVE SERVING: 4TH LEVEL

FURTHER DECOMPOSITION OF THE NODE-SERVER PROCESS

1. PROCESS A REQUEST FOR A USER DISK

2. PROCESS A REQUEST FOR AN OPERATING SYSTEM
APPENDIX B: PROGRAM LISTINGS

B.1 EXTERNAL RESOURCES: THE NETWORK-SERVER INTERFACE
B.1.1 Network Request Queue - File: NW$INP.RSC
B.1.2 Sector Pool - File: NW$SECT.RSC
B.1.3 Network Response Queue - File: NW$OUT.RSC

B.2 SERVER PROCESS-BODY
B.2.1 Server Module - File: FSMODULE.SRC
B.2.2 Global Literals, Constants, Types - File: COMMON.LIT
B.2.3 String Manipulation Routines - File: STRINGS.LIB

B.3 INTERNAL (2ND-LEVEL) RESOURCES
B.3.1 Node Queues - File: NODERQST.RSC
B.3.2 HD-Attachment Interface - File: HDATT.RSC
B.3.3 Common Software Information - File: CMINFO.RSC
B.3.4 User Information - File: UINFO.RSC
B.3.5 Opened Disks Tables - File: OP$TABLE.RSC
B.3.6 Cache Resource - File: CACHE.RSC
B.3.7 Disk-Read Queue - File: DSKREADQ.RSC

B.4 2ND-LEVEL PROCESSES
B.4.1 Initialization Process - File: INIT.PRS
B.4.2 Queue-Producing Process - File: Q$PRODUC.PRS
B.4.3 Individual-Node Server Process
   Node-Server Process Body - File: SERVER.PRS
   Drive-Map - File: DRIVE.RSC
   3rd-Level Processes - File: RGSTPROC.RTN
   Sub-processes, Common to 3rd-level Processes - File: SERVPROC.RTN
B.4.4 Disk-Read Process - File: DSKREAD.PRS
B.4.5 Disk-Write Process - File: DSKWRITE.PRS
B.4.6 Track-Send Completion Process - File: CPLSENGS.PRS

B.5 SUB-PROCESSES COMMON TO 2ND LEVEL PROCESES
B.5.1 Disk-Open Process - File: COM$OPEN.RTN
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: WM10IP.BSC

!/------------------------------------------------------------------!/ 
!/ RESOURCE: Network Input x/ 
!/ 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. x/ 

!/CONSTs/ 
Declare
MaxInputs     literally '40'; 
Rqst$Size     literally '76'; 

!/TYPEs/ 
Declare
Rqst$Type     literally 'B(16$Size) Byte'; /x An Array of Bytes x/ 

!/VARY/ 
Declare
WM1OIP.Stderr$Size     Byte, 
WM1OIP.ReadByte$Pos     Byte, 
WM1OIP.Write$Pos        Byte, 
WM1OIP.Queue(MaxInputs) STRUCTURE (Node No$Type, 
Rqst Rqst$Type); 

!/ACCESS RELATED DATA AND PROCEDURES x/ 
!/VARY-access/ 
Declare
WM1OIP_FS     Boolean, /x File Server has or is requesting access x/ 
WM1OIP_NV     Boolean, /x Network Processor has or is requesting access x/ 
WM1OIP_S3     Boolean; /x Access to the resource has been granted to one of the Processors x/ 

!/PROCEDURE x/ WM1OIP_FSAccess: Procedure Boolean; 
!/x The File Server is the consumer and is given priority x/ 
!/BEGIN/ 
IF NOT(WM1OIP_NV) 
THEN DO; 
WM1OIP_FS : True; 
DO WHILE (WM1OIP_NV AND (NOT WM1OIP_S3 )); 
END; /x Loop while access not yet decided - gives higher priority x/ 
IF (WM1OIP_NV) 
THEN WM1OIP_FS : False; 
ELSE WM1OIP_S3 : True; 
END; 
RETURN (WM1OIP_FS); 
END WM1OIP_FSAccess; 

!/PROCEDURE x/ WM1OIP_FSRelease: Procedure; 
!/BEGIN/ 
WM1OIP_S3 : False;
/* Routines allowing the Network Processor access, do not reside on the File Server, but are given for completeness */

PROCEDURE Wtlnp_NVAccess: Procedure Boolean;
BEGIN
  IF (NOT Wtlnp_PS) THEN DO;
    Wtlnp_NV = True;
  END;
  IF (Wtlnp_PS) THEN Wtlnp_NV = False; /* Back-off to allow server the access */
  ELSE Wtlnp_S3 = True;
END;

PROCEDURE Wtlnp_NVRelease: Procedure;
BEGIN
  Wtlnp_S3 = False;
  Wtlnp_NV = False;
END Wtlnp_NVRelease;

PROCEDURE */ Wtlnp_Init: Procedure;
BEGIN
  Wtlnp_S3 = False;
  Wtlnp_PS = False;
  Wtlnp_NV = False;
  Wtlnp_QueueSize = 0;
  Wtlnp_WritePos = 0;
  Wtlnp_ReadPos = 0;
END Wtlnp_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 */ Wtlnp_Queue: Procedure;
BEGIN
  IF (Wtlnp_QueueSize = Maxinputs) THEN Status = Fail;
  ELSE DO;
    /* Construct message at current write position - i.e. Wtlnp_Queue (Wtlnp_WritePos); 
      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 (Wtlnp_NVAccess)); /* INDO Loop until access is granted */
      Wtlnp_WritePos = Wtlnp_WritePos + 1;
      IF (Wtlnp_WritePos = Maxinputs) THEN Wtlnp_WritePos = 0;
      Wtlnp_QueueSize = Wtlnp_QueueSize + 1;
      CALL Wtlnp_NVRelease;
    END;
  END;
END;
PROCEDURE #VWInput_Dequeue: Procedure (RqstEntryPtr, StatusPtr);
    Declare RqstEntryPtr  Pointer,
    StatusPtr Pointer;

DECLARE
    Status BASED StatusPtr StatusType;

IF (#VWInput_QueueSize : 0)
    THEN Status : Empty,
    ELSE
        DO:
            /* Copy the request section of the input to the receiving entry */
            CALL #VWInput_Dequeue(#VWInput_ReadPos, Rqst, RqstEntryPtr, RqstSize);
            Status : Success;

        /* Get access for the File Server to shared variables & update queue information */
        DO WHILE NOT (#VWInput_PSAccess);
            /* Loop until access is granted */
            #VWInput_ReadPos : #VWInput_ReadPos + 1;
            IF (#VWInput_ReadPos : MaxInputs)
                THEN #VWInput_ReadPos : 0;
                #VWInput_QueueSize : #VWInput_QueueSize - 1;
                CALL #VWInput_PSRelease;
            END;
        END;
    END;

END #VWInput_Dequeue;

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

/ * DEFINE */
Declare
MaxSects literally '20',
NonSect literally 'MaxSects';

/ * TYPE */
Declare
TypeSectNum literally 'Byte',
Sect#Num literally 'TypeSectNum',
SectSEntry literally 'STRUCTURE ( SectPos Pointer,
SL#p Sect#Num )';

/ * VARY */
Declare
SPBit (MaxSects) SectSEntry,
FreeSects Sect#Num,
SPPSL Sect#Num;

/ * ACCESS RELATED DATA AND PROCEDURES */
/ * VARY-access */
Declare
SP_PS Boolean,  /* Pile Server has or is requesting access */
SP_NV 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_PSAccess: Procedure  Boolean; */
/ * NORTH */
IF NOT (SP_NV)
THEN DO;
SP_PS : True;
DO WHILE (SP_NV AND (NOT SP_S3));
END;  /* Loop while access not yet decided - gives higher priority */
IF (SP_NV)
THEN SP_PS := False;
ELSE SP_S3 := True;
END;
RETURN (SP_PS);
END SP_PSAccess;

/ * PROCEDURE */
/ * SP_PSRelease: Procedure; */
/ * NORTH */
SP_S3 := False;
SP_PS := False
END SP_PSRelease;

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

/ * PROCEDURE */
/ * SP_NVAccess: 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_JS : True;
  END;
  RETURN (SP_HW);
  ENd SP_HWAccess; \

/*PROCEDURE SP_HW_Release: Procedure;*/
BEGIN
  SP_JS : False;
  SP_HW : False
  ENd SP_HW_Release; \

/*PROCEDURE */
/* SP_Init: Procedure;*/
/*VAR*/
Declare
  Sect SectNum;
  Segment Token;
  Exception IOException;
/*BEGIN*/
  SP_FS : False;
  SP_JS : False;
  SP_HW : False;
  DO Sect = 0 TO (MaxSects - 1);
    Segment = DATA,strlen(SECTION) (SectorSize, @Exception);
    SP_Enter(Sect), SectorPos : BuildPtr (0, Sector);
    SP_Enter(Sect), SectorPos : Sector + 1;
  END;
  FreeSects : MaxSects;
END SP_Init;

/*PROCEDURE */
/* SP_ReleaseSect: 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_FSAccess)); END; \* Loop until access is granted \*
IF (SP_SLR : MemSect) THEN SP_Enter(Sect), SectorPtr : MemSect;
ELSE SP_Enter(Sect), SectorPtr : SP_SLR;
SP_SLR : Sect;
FreeSects : FreeSects + 1;
CALL SP_FS_Release;
END SP_ReleaseSect;

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

/*PROCEDURE */
/* SP_GetSect: Procedure (SectorPtr, StatusPtr);*/
Declare SectorPtr, StatusPtr;
/* VAR */

Declare
Status BASED StatusPtr StatusType
Sect BASED SectPtr SectNum

BEGIN
IF (FreeSets = 0)
THEN Status := Empty;
ELSE
   ** Get access for the Network processor to shared variables & update list information **
   DO WHILE NOT (SP_MVMAccess) ; END; ** Loop until access is granted **
   Status := Successful;
   Sect := SPASLR;
   SPASLR := SPASLR(Sect),SALp;
   SPASLR(Sect),SALp := NonSet;
   FreeSets := FreeSets - 1;
   CALL SP_MVMRelease;
END;
END SP_GetSet; */

/* END Sector Pool RESOURCE */

/ * ------------------------------------------------------------------------ */
8.1.3 Network Response Queue - File NWOUT.RSC

/--------------------------------------------------------------------------/;

/ RESOURCE: Network Output /;
/ Requests are written to a 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 /;

/CONSTANTS/;
Declare
   MaxOutputs literally '40';
   MessageSize literally '20';

/PRIVATE/;
Declare
   MessageType literally (MessageSize) Byte; / An Array of Bytes /;
   /AVAR/;
Declare
      NWOutRegion Token,
      NWOut_ByteFree Byte,
      NWOut_QueueSize Byte,
      NWOut_ReadPos Byte,
      NWOut_WritePos Byte,
      NWOut_DonePos Byte,
      NWOut_Queue(MaxOutputs) STRUCTURE ( Done Boolean,
                                   Node NodeType,
                                   Message MessageType );

/ACCESS RELATED DATA AND PROCEDURES/;
/AVAR-access/;
Declare
   NWOut_PS Boolean; / File Server has or is requesting access /;
   NWOut_NW Boolean; / Network Processor has or is requesting access /;
   NWOut_S3 Boolean; / Access to the resource has been granted to one of the Processors /;

/PROCEDURE */ NWOut_PSAccess: Procedure Boolean;

/VERSION/;
   IF ( NOT NWOut_NW )
      THEN DO;
         NWOut_PS := True;
         IF ( NWOut_NW )
            THEN NWOut_PS := False; // Back-off to allow Network Processor the access
            ELSE NWOut_S3 := True;
      END;
   RETURN NWOut_PS;
   END NWOut_PSAccess;

/PROCEDURE */ NWOut_PSRelease: Procedure;

/VERSION/;
   NWOut_S3 := False;
   NWOut_PS := False;
   END NWOut_PSRelease;
These routines allow the network processor access do not reside on the file server, but are given for completeness /

PROCEDURE NVOut_NVAccess: Procedure Boolean;
  // The network processor is the consumer A is given priority =
  BEGIN
    IF NOT NVOut_PS;
    THEN DO:
      NVOut_NV := True;
      DO WHILE (NVOut_PS AND (NOT NVOut_SJ));
      END: Loop while access not yet decided A gives higher priority =
      IF (NVOut_PS);
      THEN NVOut_NV := False;
      ELSE NVOut_SJ := True;
      END:
      RETURN NVOut_NV;
    END NVOut_NVAccess: /

PROCEDURE NVOut_NVRelease: Procedure;
  BEGIN
    NVOut_SJ := False;
    NVOut_NV := False
    END NVOut_NVRelease: /

PROCEDURE NVOut_Init: Procedure;
  BEGIN
    NVOut_SJ := False;
    NVOut_PS := False;
    NVOut_NV := False;
    NVOut_QtyFree := MaxOutputs;
    NVOut_QueAddr := 0
    NVOut_WritePos := Q;
    NVOut_ReadPos := Q
    NVOut_DonePos := Q
    END NVOut_Init;

PROCEDURE NVOut_Queue: Procedure (MessagesPtr, Length, Node, StatusPtr);
  DECLARE MessagesPtr Pointer,
    Length Byte,
    Node NodeType,
    StatusPtr Pointer;
  BEGIN
    Status := STATUS_NVOut_Init;

    IF NVOut_QtyFree := 1
    THEN Status := Full;
    ELSE DO:
      // Assign the destination node (first byte) & the message contents /
      NVOut_QtyFree := NVOut_QtyFree - 1;
      NVOut_QueAddr := NVOut_WritePos := 1
      IF NVOut_DonePos := False;
      NVOut_DonePos := Node;
    END:
  END NVOut_Queue:

- B9 -
PROCEDURE NVOut_NVAccess: Procedure: Boolean;
  The Network Processor is the consumer & is given priority #
BEGIN
  IF H(M(NVOut_PS)) THEN DO:
    NVOut_NV := True;
    DO WHILE (NVOut_PS AND (NOT NVOut_S1)) ;
    END: # Loop while access not yet decided: gives higher priority #
    IF (NVOut_PS) THEN NVOut_NV := False;
    ELSE NVOut_S1 := True;
  END:
  RETURN (NVOut_NV);
END NVOut_NVAccess; /*

PROCEDURE NVOut_NVRelease: Procedure;
BEGIN
  NVOut_S1 := False;
  NVOut_NV := False;
END NVOut_NVRelease; /*

PROCEDURE / NVOut_Init: Procedure:
BEGIN
  NVOut_S1 := False;
  NVOut_PS := False;
  NVOut_NV := False;
  NVOut_QtyFree := MAXOutputs;
  NVOut_QueueSize := Q;
  NVOut_WritePos := Q;
  NVOut_ReadPos := Q;
  NVOut_DonePos := Q;
END NVOut_Init;

PROCEDURE / NVOut_Queue: Procedure (MessagesPtr, Length, Node, StatusPtr);
Declare MessagesPtr Pointer, Length Byte, Node NodesType, StatusPtr Pointer;
BEGIN
  IF NVOut_QtyFree = 0 THEN Status := Fail, ELSE DO:
    /* Assign the destination node (first byte) & the message contents */
    NVOut_QtyFree := NVOut_QtyFree - 1;
    NVOut_Queue(NVOut_WritePos).Done := False;
    NVOut_Queue(NVOut_WritePos).Node := Node;
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 NVKOut_Dequeue: Procedure;
BEGIN
  IF NVKOut_QueueSize = 0
  THEN Status = Empty;
  ELSE
    Do
      Process the response message at the current read position; i.e. NVKOut_Queue (NVKOut_WritePos);
      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; if
      Set second byte of message i.e. Done := True;
      NVKOut_Queue(NVKOut_ReadPos). Done := True;
      Status := Successful;
      Get access for the Network processor to shared variables & update queue information; DO WHILE NOT (NVKOut_PSAccess); END; Loop until access is granted;
      NVKOut_ReadPos := NVKOut_ReadPos + 1;
      IF (NVKOut_ReadPos > MaxOutputs)
        THEN NVKOut_ReadPos := 0;
      NVKOut_QueueSize := NVKOut_QueueSize - 1;
      CALL NVKOut_NVRRelease;
    END
END NVKOut_Dequeue; /

PROCEDURE NVKOut_Done: Procedure (MsgPosPtr) Boolean;
DECLARE MsgPosPtr Pointer;
BEGIN
  IF NOT NVKOut_Done
  THEN RETURN (False);
  ELSE RETURN (NVKOut_Done); Done
END NVKOut_Done;
/*PROCEDURE */ NWSOut_Release: Procedure;
  /* Releases next done entry to the free entries */
  /*BEGIN*/
  NWSOut_QtyFree = NWSOut_QtyFree + 1;
  NWSOut_DonePos = NWSOut_DonePos + 1;
  IF (NWSOut_DonePos = MaxOutputs)
    THEN NWSOut_DonePos = 0;
  END NWSOut_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

FILESERVER: DO; /x Beginning of Module x/

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

/x Include PLM External Procedure definitions for iRMX Subsystems:
    Basic 10, Extended 10 and Nucleus Operating Systems x/
$INCLUDE (/RM10.0/OUTL0.10S.EXT)
$INCLUDE (/RM10.0/OUTL0/MESS.EXT)

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

/x Include Resources x/
$INCLUDE (MesAttach.RSC)
$INCLUDE (CommInfo.RSC)
$INCLUDE (UserDef.RSC)
$INCLUDE (OptTable.RSC)
$INCLUDE (WMInfo.RSC)
$INCLUDE (WMPsect.RSC)
$INCLUDE (ModInst.RSC)
$INCLUDE (EmrRej.RSC)
$INCLUDE (Cache.RSC)

/x Include Common Procedures x/
$INCLUDE (ComOpen.RTW)

/x Include Processes x/
$INCLUDE (DsRead.PRS)
$INCLUDE (DsWrite.PRS)
$INCLUDE (CpsSends.PRS)
$INCLUDE (Server.PRS)
$INCLUDE (GpProd.PRS)
$INCLUDE (Init.PRS)

/xBEGIN – FILESERVER (Code) x/
/x Setup the Fileserver as the default user of the iRMX Operating System x/
FSUser = 'RBCREATEUSER (@$SYSIDEntry, #Exception);
CALL RBCSETDEFAULTUER (DefJob, FSSysUser, #Exception);
/x Start the initial process x/
CALL Initialization;

END FILESERVER;
### COMMON Compilation Literals, Constants & Types

 DECLARE

 _True_ literally OFFh',
 _False_ literally 000h',
 _Forever_ literally While True',
 _MaxRows_ literally '50';

 DECLARE

 _Successful_ literally '0',
 _Found_ literally 'Successful',
 _Ready_ literally 'Successful',
 _Granted_ literally 'Successful',
 _Denied_ literally '1',
 _AppStatExist_ literally '2',
 _DiskNotExist_ literally '2',
 _ReqSuccessDenied_ literally '3',
 _WriteAccessDenied_ literally '4',
 _WrongLoader_ literally '5',
 _WrongBOS_ literally '6',
 _TooManyApps_ literally '7',
 _TooManyDisks_ literally '8',
 _NoDiskSpace_ literally '9',
 _CannotLoadAll_ literally '9',
 _OpenInvalid_ literally '10',
 _OpenInvalidWrite_ literally '11',
 _ExceedOpens_ literally '12',
 _DiskNotAccessed_ literally '13',
 _NoDriveSpace_ literally '14',
 _AlreadySignedOn_ literally '15',
 _Full_ literally '16',
 _Empty_ literally '17';

 DECLARE

 _Token_ literally 'Selector',
 _Region_ literally 'Token',
 _Boolean_ literally 'Byte',
 _NodeType_ literally 'Byte',
 _StatusType_ literally 'Byte';
/* PC & Floppy Disk dependent constants & types */

/\CONSTs/
Declare
SectSize literally '200h',
SectsInTrack literally '9',
TrackSize literally '1200h',
UtsHeads literally '2',
TracksInCachesTrack literally '2',
SectsInCachesTrack literally '18',
SidesCachesTrack literally '2400h';

/\TYPER/
Declare
Type&PC_Track literally 'Byte',
Type&PC_Sect literally 'Byte';
/* 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 */

/*CONSTs/ Declare
EmptyString literally '0';

/*PROCEDURE*/ String_Compare: Procedure (APtr, BPtr) Boolean RETURN;
Declare APtr Pointer,
BPtr Pointer;

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

/*VARs/ Declare
Count BASED APtr Byte;

/*BEGIN*/ IF (CMPB (APtr, BPtr, (Count+1)) = 0FFFFh )
THEN RETURN (True);
ELSE RETURN (False);
END String_Compare;

/*PROCEDURE*/ String_Add: Procedure (SumPtr, APtr, MaxLen) RETURN;
Declare SumPtr Pointer,
APtr Pointer,
MaxLen Byte;

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

/*VARs/ Declare
(Sum BASED SumPtr) (1) Byte, /* At least 1 element */
(B BASED BPtr) (1) Byte, /* At least 1 element */

 Declare
(SumLen BASED SumPtr) Byte, /* Sum len */
(BLen BASED BPtr) Byte, /* B len */

/*BEGIN*/ IF ( (SumLen + BLen) <= MaxLen )
THEN DO;
CALL MOVB ( #0h(1), @Sum(SumLen + i), BLen);
SumLen = SumLen + BLen;
END; /* Then */
ELSE DO;
CALL MOVB ( #0h(1), @Sum(SumLen + i), (MaxLen - SumLen));

- B15 -
/PROCEEDURES  */ String Num: Procedure (Num, NumStrPtr, Size)  RESULTANT: 
Declare Num Word, NumStrPtr Pointer, Size Byte;

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

/CONST/ 
Declare AsciiZero literally '30h';

/VARY/ 
Declare (NumStr BASE NumStrPtr) (1) Byte, Count Integer;

/HIGHLIGHT/ 
NumStr(0) = Size;
DO Count : Size TO 1 BY (-1);
   NumStr(Count) = (Num MOD 10) + AsciiZero;
   Num = Num / 10;
END,
END String Num.

*/ END STRING HANDLING LIBRARY */

/ * !-------------------------------------------------------------------------- */
B.3 INTERNAL (2ND-LEVEL) RESOURCES

B.3.1 Node Queues - File NODEQ2Q.ESC

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

/* VARS */
Declare
MaxRequests literally '50',
NonRequests literally 'MaxRequests';

/* VARS */
Declare
NodeRequests Region Token,
HE_SLR Byte; /* Space List Rock - First entry of a list of available entries */

Declare
NodeRequests (MaxRequests) STRUCTURE (Next Byte,
                      Rqst RqstType),
NodeRequests (MaxRequests) STRUCTURE (Server Token,
                      First Byte,
                      Last Byte);

/* PROCEDURE */
NodeRequests_Init: Procedure;

/* VARS */
Declare
Count NodeType;

/* PROCEDURE */
HE_SLR Init:
/* Zero the base info for each nodes queue, i.e. no task exists & queue is empty */
DO Count : 0 TO (MaxRequests - 1);
    NodeRequests(Count).Server = NonToken;
    NodeRequests(Count).First = NonRequests;
    NodeRequests(Count).Last = NonRequests;
END;

/* Initially all request entries are in the space list - i.e. available */
HE_SLR = 0;
DO Count : 0 TO (MaxRequests - 1);
    NodeRequests(Count).Next = MaxRequests + 1;
END;
END NodeRequests_Init;

/* PROCEDURE */
NodeRequests_getEntry: Procedure (EntryNumPtr, StatusPtr);
    Declare EntryNumPtr Pointer,
            StatusPtr Pointer;

/* VARS */
Declare
EntryNum BASED EntryNumPtr Byte,
Status BASED StatusPtr StatusType;

/* VARS */
IF (HE_SLR = NonRequests)
    THEN Status = Poll;
ELSE
    /* Do nothing */
END IF;

- 17 -
DO:
  /* Remove the first entry from the space list */
  EntryNum : NK_SLR;
  NK_SLR : Node$Rqsts(NK_SLR).Next;
  Status : Successful;
END;
END Node$Rqsts_ReleaseEntry;

PROCEDURE x/ Node$Rqsts_ReinsertEntry: Procedure (Entry$Num);
  Declare Entry$Num Byte;

BEGIN:
  /* Insert the entry at the beginning of the space list */
  Node$Rqsts(Entry$Num).Next : NK_SLR;
  NK_SLR : Entry$Num;
END Node$Rqsts_ReleaseEntry;

PROCEDURE x/ Node$Rqsts_Insert: Procedure (Node, Status$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$Rqsts_GetEntry (Entry$Num, Status$Ptr);
  IF (Status : Successful)
    THEN DO:
      /* Dequeue a request from the network input queue */
      CALL NV$Inq_Dequeue (New$Rqst, Status$Ptr);

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

PROCEDURE x/ 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,
/* RBC10x */
IF (Queue.First : #NonQst)
   THEN Status : Empty;
   ELSE
      /* Copy the first entry's request to the calling task */
      FirstQst : Queue.First;
      CALL NEW ( #NodeQstis(FirstQst).Qst, ReceivesPtr, #QstSize);
      /* Remove the first entry from the queue */
      Queue.First : #NodeQstis(Queue.First).Next;
      IF (Queue.First : #NonQst)
         THEN Queue.Last : #NonQst;
      /* Release the consumed entry to the space list */
      CALL NodeQstis_ReleaseEntry (FirstQst);
      END;
END NodeQstis_Dequeue;

/* END individual node request queues RBC10x */
FILE: HDATT.RC

/* RESOURCE: Hard Disk Attachment */

// Includes routines which interface with Operating System functions
// for attaching & creating files & directories */

/COMMTs/
Declare
RACE literally '0',
Reading literally '1',
Writing literally '2',
ReadAdvPtr literally '3',
NewDisk&Size literally '1000',
ZeroBuffers literally '0',
RootToken literally '0',
Mailbox literally 'RootToken',
Default literally 'RootToken',
DefPrefix literally 'RootToken',
DefJob literally 'RootToken';

/ITYPES/
Declare
IOException literally 'Word';

/YAVAI/
Declare
HDMAttachRegion Region,
SysPrefix Token,
CommonPrefix Token,
UsersPrefix Token,
NewDiskBuff Pointer,
AttachMailbox Token,
(NewDiskBuffer BASED NewDiskBuff) (NewDisk&Size) Byte;

/PROC/ GetConnectToken Procedure (Hbox, TokenPtr, ExceptPtr);
Declare Hbox Token,
TokenPtr Pointer,
ExceptPtr Pointer;

/COMMTs/
Declare
SegmentType literally '0006',
ConnectionType literally '101b',
WaitDone literally '000000';

/YAVAI/
Declare
SegmentPtr Pointer,
NewToken BASED TokenPtr Token,
Exception BASED ExceptPtr IOExcept,
(10SegmBASED ExceptPtr) (1) Word,
HasResponse Word,
DoneExcept IOExcept;

/EXECUT/
IF (Exception <> HACE) THEN RETURN; // Already an error
NewToken := MKSECRETVMESSAGE (Hbox, WaitDone, HasResponse, ExceptPtr);
IF (Exception = REXX) THEN
    IF (MROUTETYPES (NewToken, ExceptionPtr) = SegmentType) THEN
        DO;
            SegmentPtr = BuildPtr (NewToken, 0);
            Error : = IDSegment (0);
            CALL PAGERESLETSEGMENT (NewToken, #DomExcept);
        END;
    IF (Exception = REXX) THEN NewToken : = NewToken;
END SetConnectToken:

//PROCEDURE / HDMAttach_Init: Procedure;
//VARS/
   Declare
   MDFlags  literally '00100', /* 4-Deep FIFO Queue */
   PSASysDir literally '(10, '\"PSASysDir\"'),
   PSCommonDir literally '(10, '\"PSCommon\"'),
   PSUsersDir literally '(10, '\"PSUsers\"'),
   NewDiskSize literally '(8, '\"NewDisk\"');
//VARS/
   Declare
   Connection  Token.
   Exception  #DomException.
   Segment  Token.
   BytesRead  Word.

//VARS/
   /\ Create Mailbox & Prefix Connections /\,
   AttachAsBox  MD_CONNECT_MAILBOX (MDFlags, #Exception);

   // Attach prefixes for directories for File Server System, Common 1
   Users & set default prefix as the FS System directory /\,
   CALL PFSATTACHFILE (DefUser, DefPrefix, PSASysDir, AttachAsBox, #Exception);
   CALL GetConnectToken (AttachAsBox, #SysPrefix, #Exception);
   CALL PAGETRESLETPREFIX (DefJob, SysPrefix, #Exception);

   CALL PFSATTACHFILE (DefUser, DefPrefix, PSCommonDir, AttachAsBox, #Exception);
   CALL GetConnectToken (AttachAsBox, #CommonPrefix, #Exception);
   CALL PFSATTACHFILE (DefUser, DefPrefix, PSUsersDir, AttachAsBox, #Exception);
   CALL GetConnectToken (AttachAsBox, #UsersPrefix, #Exception);

   // Create buffer with first 12 sectors for new disk /\,
   Segment  MDCONNECT_SEGMENTS (NewDiskSize, #Exception);
   NewDiskSize : = BuildPtr (Segment, 0);
   Connection  MD_CONNECTFILE (NewDiskSize, #Exception);
   CALL PROBTO (Connection, Reading, ZeroBuffers, #Exception);
   BytesRead  PROBTO (Connection, NewDiskSize, #Exception);
   CALL MDDELETECONNECTION (Connection, #Exception);
END HDMAttach_Init:

//PROCEDURE / HDMAttach_H CreateDir: Procedure
   (APItr, BPtr, CPtr, QtyStrings, Prefix, ConnectPtr, ExceptPtr);
   Declare
   ( APItr, BPtr, CPtr, ConnectPtr, ExceptPtr ) Pointer,
   QtyStrings  Byte,
   Prefix  Token;
```c
/CONST/
Declare
BMPHSEXIST literally "0x01b",
StringSize literally "31",
MaxLen literally "StringSize-1",
FullPathAccess literally "1111b";
/VAR/
Declare
Exception BASED ExceptPtr 10hException,
ReqdConnect BASED ConnectPtr Token;
Declare
Path (3) STRUCTURE { String(StringSize) Byte 1,
Connection (3) Token,
Try Byte,
Count Byte,
except 10hException;
-BEGIN/
SD Count = 1 TO 3;
Connection(Count) = NonToken;
Path(Count) String(0) = EmptyString;
END;

/SET:
Set 3 paths in case files in path do not exist
CALL String Add (#Path(1), AsPtr, MaxLen); IF (#Strings < 2) THEN DO:
CALL String Add (#Path(2), #Path(1), MaxLen);
CALL String Add (#Path(2), #"" , MaxLen);
CALL String Add (#Path(2), #Ptr , MaxLen);
END; /if Them/ IF (#Strings < 3) THEN DO:
CALL String Add (#Path(3), #Path(2), MaxLen);
CALL String Add (#Path(3), #"" , MaxLen);
CALL String Add (#Path(3), #Ptr , MaxLen);
END; /if Them/ /

/ATT:
Attempt to Attach the file - If necessary, create directories in the path
Try = #Strings;
CALL ROMAATTACHFILE (DefUser, Prefix, #Path(Try), AttachMBBox, ExceptPtr);
CALL GetConnectsToken (AttachBox, #Connection(Try), ExceptPtr);
Try = Try + 1;
DO While (( (Exception = BMPHSEXIST) AND (Try < 3))
OR ( (Exception = ROMA) AND (Try < 3) )); IF (Exception = ROMA) THEN Try = Try + 1;
ELSE Try = Try - 1;
CALL ROMADELETEDIRECTORY (DefUser, Prefix, #Path(Try), PullAccess, AttachMBBox, ExceptPtr);
CALL GetConnectsToken (AttachBox, #Connection(Try), ExceptPtr);
END; /While/ /

/DEL:
Delete Connections of those created & not required
DO Count = 1 TO 3;
IP (Count < #Strings) THEN
IP (Connection(Count) = NonToken)
THEN CALL ROMADELETECONNECTION (Connection(Count), NonMailBox, #DefUserExcept);
END;
```
PROCEDURE / MDAttach_Open&CreateFile: Procedure
(DisK, DiskConnect, Read&Write, Connect&Ptr, Except&Ptr);

DECLARE DISK DISK Num relative to it's application /;

DECLARE DISK-Token,
Read&Write Boolean, / Mode of opening /;

DECLARE Connect&Ptr Pointer,
Except&Ptr Pointer;

DECLARE DISKStr literally "'(D, 'Disk')'",
EXIST literal '02hh'
Must&Create literally 'True',
Full&Access literally '1111B',
Granularity literally '2400h'; / 2 PC-Tracks : 10, 512-byte sectors x/

DECLARE NewConnect BASED Connect&Ptr Token,
Exception BASED Except&Ptr I&OException;

DECLARE DismName (1) Byte,
Dism&Num (2) Byte,
Bytes&Written Word,
Mode Byte,
Dum&Except I&OException;

BEGIN /

/ Format the correct file name from the given number of the
particular application /

NewConnect : Num&Token;
CALL String&Num (DisK, &Dism&Num, 2);
Dism&Name (0) : Empty&String;
CALL String&Add (&Dism&Name, &DismStr, 6);
CALL String&Add (&Dism&Name, &Dism&Num, 6);

/ Attempt to attach the file /
CALL MDAS ATTACH&FILE (DefUser, Dir&Connect, &Dism&Name, Attach&Box, Except&Ptr);
CALL Get&Connect&Token (Attach&Box, Connect&Ptr, Except&Ptr);

/ If the file does not exist, create it and copy the relevant sectors
for a blank formatted disk /
IF (Exception : EXIST) THEN
THEN

/ Attempt to create the file /
CALL MDASCREATE&FILE (DefUser, Dir&Connect, &Dism&Name, Full&Access, Granularity,
NewDisk&Size, Must&Create, Attach&Box, Except&Ptr);
CALL Get&Connect&Token (Attach&Box, Connect&Ptr, Except&Ptr);
IF (Exception : E&BOX) THEN
CALL MDAS&OPEN (NewConnect, Writing, Zero&Buffers, Except&Ptr);
IF (Exception : E&BOX) THEN
Bytes&Written : MDASWRITE&MOVE (NewConnect, New&Disk&Buffer,
NewDisk&Size, Except&Ptr);
IF (Exception : E&BOX) THEN
CALL MDAS&CLOSE (NewConnect, Except&Ptr);
IF (Exception () E&BOX) THEN
DO;
  CALL ROMADELETECONNECTION (NewConnect, No$Mailbox, #Do$Except);
  NewConnect := Non$Token;
END;
END; /* Then */

/* If the connection has been attached without error, then open in the required mode */
IF (Exception : BREAK)
  THEN
    IF (ReadWrite : True)
      THEN Mode := ReadAndWrite;
      ELSE Mode := Reading;
      CALL ROMAOPEN (NewConnect, Mode, ZeroBuffers, Except$Ptr);
      IF (Exception : BREAK)
        THEN CALL ROMADELETECONNECTION (NewConnect, No$Mailbox, #Do$Except);
      END;
    END;
END HMAAttach_OpenedCreatedFile;

/* END HMAATTACH RESOURCES */

/* ----------------------------------------------------------------------------------------------- */
Author  Hildyard Mark William

PUBLISHER:
University of the Witwatersrand, Johannesburg
©2013

LEGAL NOTICES:

Copyright Notice: All materials on the University of the Witwatersrand, Johannesburg Library website are protected by South African copyright law and may not be distributed, transmitted, displayed, or otherwise published in any format, without the prior written permission of the copyright owner.

Disclaimer and Terms of Use: Provided that you maintain all copyright and other notices contained therein, you may download material (one machine readable copy and one print copy per page) for your personal and/or educational non-commercial use only.

The University of the Witwatersrand, Johannesburg, is not responsible for any errors or omissions and excludes any and all liability for any errors in or omissions from the information on the Library website.