THE FILE SYSTEM

December 7, 1984

2600 Liberty Avenue
P. O. Box 2600
Pittsburgh, PA 15230
(412) 355-0900

Accent is a trademark of Carnegie-Mellon University.

Accent and many of its subsystems and support programs were originally developed by the CMU Computer Science Department as part of its Spice Project.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

PERQ, PERQ2, LINQ, and Qnix are trademarks of PERQ Systems Corporation.

Table of Contents Page

List of Figures — Page

1. Theory

The file system that runs under the Accent operating system provides a directory-oriented structure that allows the user to store and manipulate files on disks. The system supports a variety of hard disks. On the PERQ workstation, the standard disk is a Shugart 24 megabyte disk. On the PERQ2 workstation, the standard disk is a Winchester-type 8" 35 megabyte disk or a 5.25" Winchester-type disk. The 5.25" disk is one of the following capacities: 34 megabytes, 81 megabytes, or 110 megabytes. (All disk sizes are formatted capacities.) Optionally, on a PERQ2 with a 5.25" disk a second disk of the same capacity may be added.

The hardware addressing of devices occurs through microcode, using a physical address. Software addresses the devices through the microcode using a logical address.

Note that on a PERQ system, the low-order word precedes the high-order word in memory. All values represented in this document are in memory order. Bits are numbered as follows:

```
    High-byte         |     Low-byte
15 14 13 12 11 10 9 8 | 7 6 5 4 3 2 1 0
```

The disk addresses represented in this document are octal values.

The current implementation of the Accent file system consists of data transmission and name manipulation primitives, plus temporary primitives needed to retain disk structure compatibility with the PERQ Systems Corporation's POS operating system.

The syntax of Accent file names reflects the hierarchical

organization of the Accent file system as follows:

/ disk / partition / directory / file

The mass storage devices (hard disks) form the base of the hierarchy. The file system divides each device into a number of sections, known as partitions. Each partition can contain any number of directories. Directories are special format files that can contain names and addresses of files as well as other directories; you can nest directories and thus form a multi-level directory structure. In this file system, all files can be noncontiguous. However, files, including directories, cannot cross partition boundaries; portions of files can only be scattered throughout the partition in which they reside.

The remainder of this document is organized as follows:

Chapter 2, Use, describes the fundamental file system operations, syntax, and semantics that allow you to create, read, write, delete, and close files. Other functions require one or more of these fundamental operations.

The format and composition of the Accent file system structure is discussed in detail in the following chapters:

Chapter 3, Definitions, lists all the routines.

Chapter 4, Device Information, describes the distinct (physical) and standard (logical) addresses. The chapter also describes the logical structures that provide the file system interface between physical and logical structures.

Chapter 5, File System Data Structures, details the data structures of the logical structures. This chapter describes the significance of each word in the file system logical structures.

Chapter 6, File Formats, defines the format of two important types of files: segment (.Seg) files produced by a compiler; and directory files that form the keystone of the file system. This chapter also describes the method in which files are entered or listed in the directories. The chapter does not attempt to define the format of executable (.Run) files nor system bootstrap (.Boot) files. (The module RunDefs.Pas defines .Run file format.)

2. Use

The file system provides a file server with a naming convention that allows files on a remote machine to be referenced by Accent.

Each machine should have a file named boot:Sysname that contains the name of that machine. The first line of this file is an ASCII name by which the file system is known. To reference a file on a remote machine a user must know the ASCII name of that machine (as defined in the remote SysName file), and then reference the remote file by its absolute pathname, starting with the remote machine name.

For example, if a machine has the name mrt, remote users can refer to a file on that machine named /Sys/User/Visitor/foo.pas (if the user permits read access) by typing:

/mrt/User/Visitor/foo.pas

Users can control remote machine access to their file system. Ownership and access privileges are discussed in the document "Basic Operations" in the *Accent User's Manual.*

The exported Pascal procedures for the file system are found in the file SesameUser.Pas; the interface routines are in the file PathName.Pas; and the Environment Manager procedures (to set search lists) are in the file EnvMgrUser.Pas. The type definition files are SesameDefs.Pas and EnvMgrDefs.Pas, respectively.The Sesame server port SesPort must be imported from the file PascalInit.Pas.

The definitions throughout this document are given in Pascal. If

you are programming in the C language, please refer also to the document "C System Interfaces" in the *Accent Languages Manual*. If you are programming in the Lisp language, see the document "Lisp Interaction with the Accent Operating System" in the *Accent Lisp Manual*. When FORTRAN becomes available under Accent, the definitions will be the same as the C language.

2.1. Names

File system names are composed entirely of 7-bit ASCII characters.

2.1.1. Syntax characters

The syntax characters are:

/ Directory Delimiter
Version Number Delimiter
: Logical Name Delimiter
(postfix notation)
\ Quote Next Character

2.1.2. Constituent characters

The constituent characters are:

Letters (case preserved,
case insensitive matching)

Digits

The Special characters:
. _ + - $

2.1.3. Pattern characters

The pattern characters are:

* Match zero or more arbitrary characters

? Match exactly one arbitrary character

Pattern characters used outside of a pattern context are illegal unless quoted.

2.1.4. Other characters

Names are always returned from the file system in such a manner that they can be correctly passed as is to the file system (i.e.: all non-constituent characters in names are quoted).

3. Definitions

3.1. Type Definitions

3.1.1. Sesame Definitions

```
imports AccentType from AccentType;
imports TimeDefs   from TimeDefs;

{}
{ APath_Name:      A full Name Server pathname string.
{ Wild_APath_Name: A full Name Server pathname string
                   allowing wildcarding.
{ Entry_Name:      A single component of a pathname string.
{}

const
    Path_Name_Size      = 255;  { Number of characters in a
                                  Path_Name }
    Entry_Name_Size     = 80;   { Number of characters in an
                                  Entry_Name }
type
    APath_Name          = string[Path_Name_Size];
                          { An abs. pathname }
    Wild_APath_Name     = string[Path_Name_Size];
                          { A wild abs. pathname }
    Entry_Name          = string[Entry_Name_Size];
                          { A pathname component }

{}
{ Name_Flags:  Flags giving desired treatment of names in
{              Name Server calls. Note that specific flag
{              values may be illegal for certain calls, and
{              must be zero.
{}

const
    NFlag_Deleted   = #000001;  { Allow deleted names }
    NFlag_NoNormal  = #000002;  { Disallow normal (not
                                  deleted) names }
    NFlag_RESERVED  = #177774;  { These bits reserved for
                                  expansion }
type
    Name_Flags      = 0 .. #3;

{}
{ Name_Status:  Flags useful for determining the disposition
{               of a name in the name data base.
{}
```

```
const
    NStat_Deleted   = #000001;  { Set if name is deleted }
    NStat_High      = #000002;  { Set if name is highest
                                  undeleted version }
    NStat_Low       = #000004;  { Set if name is lowest
                                  undeleted version }
    NStat_RESERVED  = #177770;  { These bits reserved
                                  for expansion }
type
    Name_Status     = 0 .. #7;

{}
{ Entry_Type:  The kinds of objects which can be in the
{              name data base.
{}

const
    Entry_All           = 0;    { Special value referencing
                                  all entry types }
    Entry_File          = 1;    { Entry_Data is a File_ID }
    Entry_Directory     = 2;    { Name refers to another level
                                  of the name hierarchy.
                                  Entry_Data is empty. }
    Entry_Port          = 3;    { Entry_Data is a port }
type
    Entry_RESERVED      = 4 .. #377; { These values reserved
                                       for expansion }
    Entry_UserDefined   = #400 .. #77777;  { Values available
                                             to the user }
type
    Entry_Type          = 0 .. #77777;

{}
{ Entry_Data:  The variant data record dependent upon the
{              Entry_Type value.
{}

type
    Entry_Data = record case Entry_Type of
        Entry_File      : ( {EDFileID : File_ID  --
                             Not Yet Implemented} );
        Entry_Directory : ( );
        Entry_Port      : ( EDPort : Port );
        #400            : ( EDBytes : packed array
                            [0..255] of bit8 );
        #401            : ( EDWords : array [0..127]
                            of integer );
        #402            : ( EDLongs : array [0..63]
                            of long );
        #403            : ( EDString: string[255]);
        end;

{}
{ Entry_List_Record:  ScanNames returns array of
{                     Entry_List_Record.
{}
```

```
    Entry_List_Record   = record
        EntryName       : Entry_Name;
        EntryVersion    : long;
        EntryType       : Entry_Type;
        NameStatus      : Name_Status;
        end;

    Entry_List_Array    = array [0..0] of
                          Entry_List_Record;
                          { hack }
    Entry_List          = Entry_List_Array;

{}
{ Valid_Name_Chars:  Those 7-bit ascii characters which can
{                    occur in an entry name without being
{                    quoted.  Note that to match uppercase,
{                    uppercase letters also have to be
{                    quoted.
{}
const
    Valid_Name_Chars = {Put in a string since can't put
                        in a set }
        '$-.+0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ_
         abcdefghijklmnopqrstuvwxyz';

{}
{ Separator characters for parts of Path_Name syntax.
{}
const
    Dir_Separator    =  '/';    { Directory separator }
    Ver_Separator    =  '#';    { Version number separator }
    Quote_Char       =  '\';    { Quote character for "funny"
                                  characters }

    Up_one_Directory =  '../';
    This_Directory   =  './';


{}
{ Group_ID:  An identifier for a User Group
{}

type
    Group_ID    = long;

{}
{ File_Data:  A pointer to a file mapped into memory
{}

type
    File_Data   = pointer;  { A pointer to data for file calls }

{}
{ Print_Name:  A short string name in the file header.
{}
```

```
const
    Print_Name_Size = 80;    { Number of characters in a file
                               Print_Name }
type
    Print_Name  = string[Print_Name_Size]; { A print name
                               string }

{}
{ Data_Format:  A user-defined longword for storing data
{       format information. System-defined values are given below.
{       User-defined values must have a non-zero high word
{       (#200000 or greater).  These values are intended to be
{       used by programs which need to know about data formats
{       in order to convert from one format to another (such
{       as FTP).
{}
const
    DForm_Unspecified   = 0;             { Unspecified data
                                           format }

    DForm_8_Bit         = 8;             { 8 bit binary data }
    DForm_16_Bit        = 16;            { 16 bit binary data }
    DForm_32_Bit        = 32;            { 32 bit binary data }
    DForm_36_Bit        = 36;            { 36 bit binary data }

    DForm_CRLF_Text     = #413;          { CRLF delimited
                                           text }
    DForm_LF_Text       = #410;          { LF delimited text }

    DForm_Press         = #1000;         { Press file format data }
type
    Data_Format = long;

{}
{ File_Header:  How the user perceives the file header
{       information. It is actually generated by the
{       GetFileHeader style calls from the real header.
{}

type
    File_Header = packed record
        FileSize        : long;          { Size of file in bytes }
        DataFormat      : long;          { Advisory data format }
        PrintName       : Print_Name;    { The file print name }
        Author          : Group_ID;      { Who wrote the file }
        CreationDate    : Internal_Time;{ When it was written }
        AccessID        : Group_ID;      { Who last accessed the
                                           file }
        AccessDate      : Internal_Time;{ When it was last
                                           accessed }
        FHdr_RESERVED   : array [56..64] of integer;
                                        { Pad for expansion }
        end;

{}
{ Error return values
```

```
{}

const
    Sesame_Error_Base   = 1200;

    NameNotFound        = Sesame_Error_Base + 1;
    DirectoryNotFound   = Sesame_Error_Base + 2;
    DirectoryNotEmpty   = Sesame_Error_Base + 3;
    BadName             = Sesame_Error_Base + 4;
    InvalidVersion      = Sesame_Error_Base + 5;
    InvalidDirectoryVersion
                        = Sesame_Error_Base + 6;
    BadWildName         = Sesame_Error_Base + 7;
    NotAFile            = Sesame_Error_Base + 8;
    NoAccess            = Sesame_Error_Base + 9;
    NotSamePartition    = Sesame_Error_Base + 10;
    ImproperEntryType   = Sesame_Error_Base + 11;
    NotADirectory       = Sesame_Error_Base + 12;
```

3.1.2. Definitions for Sesame Disk Mount routines

```
imports AccentType from AccentType;
imports AuthDefs   from AuthDefs;       { User_ID }

{}
{ Partition list
{}

type

  PartData     = record          {entry in the PartTable}
    PartName   : DevPartString;  {name of this partition}
    DevName    : DevPartString;  {which disk this part is on}
    PartRootDir: SegID;          {SegID of Root Directory}
    PartNumFree: long;           {HINT of how many free pages}
    PartMounted: boolean;        {this partition is mounted}
    PartStart  : DiskAddr;       {Disk Address of 1st page}
    PartEnd    : DiskAddr;       {Disk Address of last page}
    PartKind   : PartitionType;  {Root or Leaf}
    PartExUse  : boolean;        {Opened exclusively }
   end;

    PartDList    = array[1..MAXPARTITIONS] of PartData;
    ptrPartDList = ^PartDList;


{}
{ Error return codes
{}

const
    SesDisk_Error_Base = 29400;

    BadPartitionName   = SesDisk_Error_Base + 1;
```

```
    BadPartitionType    = SesDisk_Error_Base + 2;

    AuthorizationServerGone = SesDisk_Error_Base + 3;

{}
{ Temporary Entry_Type values
{}

const
    Entry_Foreign       = 6;

{}
{ Temporary Access Control flags
{}
const
    Owner_Read_Access   = #1;
    Owner_Write_Access  = #2;

    All_Read_Access     = #100;
    All_Write_Access    = #200;

type
    Access_Rights       = integer;

{}
{ The temporary access control flags are returned in one of
{ the "reserved"
{ fields in the file header.
{}

const
    FHdr_Access_Rights_Offset = 56;
```

3.2. File I/O Primitives

3.2.1. Reading data from a file to memory (short form)

```
Function SubReadFile(
            ServPort        : port;
            APathName       : APath_Name;
        var Data            : FileData;
        var Data_Cnt        : long)
      : GeneralReturn;
```

Abstract:

The SubReadFile request is used to map the data associated with the given filename into memory. Note that it always maps in the entire file, thus there are no arguments specifying the size or position of the data to be read. You can get away with mapping the whole file since you actually will only bring in the pages that are touched, the rest being backed to secondary storage. Holes

in the file are mapped in as valid zero pages that will be created if written to.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The absolute pathname to read the data from.

Data A pointer to the data read in memory.

Data_Cnt The total number of bytes read.

Returns:

Success The data was successfully mapped into memory.

NameNotFound
APathName was not found in the directory (but the directory exists).

NoAccess The requesting user tried to read a file for which the user did not possess Read Access rights.

NotAFile The entry found under APathName was not a file.

DirectoryNotFound
One of the directories in the pathname does not exist.

BadName APathName is incorrectly formatted (invalid syntax).

3.2.2. Reading data from a file to memory (long form)

```
Function SesReadFile(
            ServPort       : port;
        var APathName      : APath_Name;
        var Data           : File_Data;
        var Data_Cnt       : long;
        var DataFormat     : Data_Format;
        var CreationDate   : Internal_Time;
        var NameStatus     : Name_Status)
    : GeneralReturn;
```

Abstract:

The SesReadFile request is a long form of the SubReadFile call. It returns several parameter values that the short form does not.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The absolute pathname to read the data from. Changed to be the name actually read from.

Data A pointer to the data read in memory.

Data_Cnt The total number of bytes read.

DataFormat One of {unspecified, text, bit8, bit16, bit32, bit36, press, . . .}.

CreationDate
The date and time the file was written.

NameStatus Deleted / Undeleted, Low version / high version.

Returns:

Success The data was successfully mapped into memory.

NameNotFound
APathName was not found in the directory (but the directory exists).

NoAccess The requesting user tried to read a file for which the user did not possess rights.

NotAFile The entry found under APathName was not a file.

DirectoryNotFound
One of the directories in the pathname does not exist.

3.2.3. Writing data from memory to a file

```
Function SubWriteFile(
        ServPort          : Port;
    var APathName         : APath_Name;
        Data              : File_Data;
        DataCnt           : long;
        DataFormat        : Data_Format;
    var CreationDate      : Internal_Time
      ): General Return;
```

Abstract:

The SubWriteFile request is used to enter a new name into the directory structure and write a file under that name. This short form of the call picks up defaults for unspecified parameters from previous versions or directory defaults. The user must have write access on the directory and on the previous version. Empty pages need no disk pages assigned to them.

Parameters:

ServPort* \ File system service port to your machine (SesPort).

APathName The absolute pathname to which to write the data.

Data The data to be written.

DataCnt The number of bytes of data to be written.

DataFormat One of {unspecified, text, bit8, bit16, bit32, bit36, press, . . .}

CreationDate
The date and time the file was written.

Returns:

Success The data was written under the name returned.

NoAccess The requesting user tried to write a file for which the user did not possess write access.

DirectoryNotFound
One of the directories in APathName does not exist.

InvalidVersion
The explicit version number was less than or equal to that of an existing version.

AlignmentError
The data was not on a page boundary.

3.3. File Header Manipulation Primitives

3.3.1. Returning file header information

```
Function SesGetFileHeader(
            ServPort          : Port;
            APathName         : APath_Name;
          var FileHeader      : File_Header
          ): General Return;
```

Abstract:

The SesGetFileHeader request is used to return fields from the file header.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The absolute pathname with which to find the file.

FileHeader A data structure containing the fields of the file header.

Returns:

Success The header data was successfully returned.

BadName APathName is incorrectly formatted (invalid syntax).

NoAccess The requesting user tried to read a file header for which the user did not possess rights.

NotAFile The entry found under APathName was not a file.

DirectoryNotFound
One of the directories in the pathname does not exist.

3.3.2. Returning file data and header information

```
Function SesReadBoth(
            ServPort          : Port;
        var APathName         : APath_Name;
        var Data              : File_Date;
        var DataCnt           : long;
        var FileHeader        : File_Header;
        var NameStatus        : NameStatus
          ): General Return;
```

Abstract:

The SesReadBoth request is used to return both the data from a file and fields from its header.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The absolute pathname with which to find the file.

Data A pointer to the data read in memory.

DataCnt The total number of bytes read.

FileHeader A data structure containing the fields of the file header.

NameStatus Flags denoting deleted / undeleted, low version / high version.

Returns:

Success The header data was successfully returned.

BadName APathName is incorrectly formatted (invalid syntax).

NoAccess The requesting user tried to read a file header for which the user did not possess rights.

NotAFile The entry found under APathName was not a file.

DirectoryNotFound
One of the directories in APathName does not exist.

3.4. Name Server Primitives

3.4.1. Looking up names

```
Function SubLookUpName(
            ServPort          : Port;
        var APathName         : APath_Name;
        var EntryType         : Entry_Type;
        var EntryData         : Entry_Data;
        var NameStatus        : Name_Status
          ): General Return;
```

Abstract:

This function provides a simple way to look up a name and return the data stored within it. If the final result is of type IPC Port, the corresponding IPC port is returned. If it is of type Directory no entry data is returned, but entry type specified the fact that a lookup on a directory name was done. Note that SubLookupName is invalid for files. SubTestName should be used to test for the existence of a file.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The name to be looked up (may not contain wildcard characters).

EntryType The type value of the entry found. Some example values are: File, Directory, and Port.

EntryData A variant field dependent upon the entry type. An IPC port will be returned here for a Port entry, whereas there will be no data returned for a Directory -- all that this call will tell you about a directory is that it exists.

NameStatus Flags denoting deleted/undeleted, low version/high version.

Returns:

Success The name was successfully looked up.

NameNotFound
APathName was not found in the directory (but the directory

exists).

NoAccess The user does not have sufficient rights to look up the given name.

3.4.2. Determining the type of a name

```
Function SubTestName(
            ServPort          : Port;
        var APathNAme         : APath_Name;
            EntryType         : Entry_Type;
        var NameStatus        : Name_Status
          ): General Return;
```

Abstract

SubTestName is like SubLookUpName except that it never returns the entry data associated with a name; only the entry type. This call should be used when all that is desired is to test for the existence of a name and to determine its type.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The name to be looked up (may not contain wildcard characters).

EntryType The type value of the entry found. Some example values are File, Directory, and Port.

NameStatus Flags denoting deleted / undeleted, low version / high version.

Returns:

Success The name was successfully found.

NameNotFound
APathName was not found in the directory (but the directory exists).

NoAccess The user does not have real rights on the name's directory.

3.4.3. Entering an object into a directory

```
Function SubEnterName(
        ServPort          : Port;
    var APathName         : APath_Name;
        EntryType         : Entry_Type;
        EntryData         : Entry_Data
      ): General Return;
```

Abstract:

The SubEnterName request is used to place a name and entry value pair in the directory structure.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The name to be entered into the directory structure.

EntryType The type value of the object to be entered. Some example values are Directory, and Port.

EntryData A variant field dependent upon the entry type. For instance, this field must contain an IPC port for type Port. For type Directory, this field is left empty, since the user can't write any directory data directly. Use of this call with entry type directory enters a new directory; no special call is needed for that purpose.

Returns:

Success APathName was successfully entered.

NoAccess The user does not possess write rights in the specified directory.

InvalidEntryType
SubEnterName cannot be used to enter a file.

3.4.4. Deleting a name in a directory

```
Function SubDeleteName(
          ServPort          : Port;
          APathName         : APath_Name
         ): General Return;
```

Abstract:

The SubDeleteName request is used to delete a name from a directory. It requires write permission in the directory and on the name.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The name to be deleted.

Returns:

Success The name was successfully deleted.

NameNotFound
APathName was not found in the directory (but the directory exists).

NoAccess The user does not possess write rights on the name specified.

DirectoryNotEmpty
Illegal to delete a non-empty directory.

3.4.5. Changing the name of an entry

```
Function SubReName(
            ServPort          : Port;
            OldPathName       : APath_Name;
        var NewPathName       : APath_Name
          ): General Return;
```

Abstract:

SubRename enters the object specified by Old APathName into the global namespace as New APathName and then removes the Old APathName. The access rights of the object are moved with it.

Parameters:

ServPort File system service port to your machine (SesPort).

OldPathName
The absolute pathname of the object to be renamed.

NewPathName
The new name to enter the object under.

Returns:

Success The name was successfully changed.

NoAccess The user either does not have the proper rights to delete old APathName or to enter new APathName.

3.4.6. Scanning directories for names

```
Function SesScanNames(
            ServPort          : Port;
            WildAPathName     : Wild_APath_Name;
            NameFlags         : Name_Flags;
            EntryType         : Entry_Type;
        var DirectoryName     : APath_Name;
        var EntryList         : Entry_List;
        var EntryList_Cnt     : long
          ): General Return;
```

Abstract:

The SesScanNames call is used to search for a given pattern in a specified directory. It will sort and return all the matches to the given pattern in the directory. Optionally, names only of a specific entry type can be scanned for. The version field for directory will be returned as zero. Note that wildcarding is permitted only in the terminal component of wild APathName. This call requires Read access on the directory being scanned, in which case all matches are returned.

Parameters:

ServPort File system service port to your machine (SesPort).

WildAPathName
The absolute pathname to be scanned (may contain wildcard characters in the terminal component).

NameFlags (optional)
The flags to inhibit / allow deleted / undeleted names.

EntryType The entry type being scanned for. The special type designator All may be given, in which case names of all entry types are returned.

DirectoryName
The absolute pathname of the directory in which matches occured.

EntryList pointer to an array of records containing the matching entries in the directory.

EntryList_Cnt
The actual number of list elements returned; zero if no match occurred.

Returns:

Success The given wild absolute pathname was successfully scanned.

NoAccess The user does not posses Read rights on the directory to be scanned.

BadWildName
Wildcards were found in wild APathName at other than the terminal component.

3.5. Interface Operations

3.5.1. Reading a file

```
Function ReadFile(
          InOut PathName              : Path_Name;
            Out Data                  : File_Data;
            Out ByteCount             : long)
        : GeneralReturn;
```

Abstract:

The ReadFile call is equivalent to calling FindFileName and then using the returned absolute pathname in a call to SubReadFile.

Parameters:

PathName
Path name of the file to be read. It is returned set to the absolute pathname of the file read.

Data Pointer to the returned data.

ByteCount
The number of bytes read.

Returns:

Success The data was successfully mapped into memory.

NameNotFound
APathName was not found in the directory (but the directory exists).

NoAccess The requesting user tried to read a file for which the user did not possess rights.

NotAFile The entry found under APathName was not a file.

DirectoryNotFound
One of the directories in the pathname does not exist.

3.5.2. Reading one of a list of files

```
Function ReadExtendedFile(
          InOut PathName                : Path_Name;
                ExtensionList           : Extension_List;
                ImplicitSearchList      : Env_Var_Name;
                Out Data                : File_Data;
                Out ByteCount           : long)
            : GeneralReturn;
```

Abstract:

The ReadExtendedFile call is equivalent to calling FindExtendedFileName followed by a call to SubReadFile with the returned absolute pathname and its extension.

Parameters:

Path Name
: The pathname of the file to be read. Returned set to the absolute pathname of the file read.

ExtensionList
: A list of possible filename extensions.

ImplicitSearchList
: Name of the search list to use if a partial pathname is supplied. If blank, the default list is used.

Data
: Pointer to the returned data.

ByteCount
: The number of bytes read.

Returns:

Success
: The data was successfully mapped into memory.

BadName
: APathName is incorrectly formatted (invalid syntax).No entry was found under a pathname.

NoAccess
: The requesting user tried to read a file for which the user did not possess rights.

NotAFile
: The entry found under APathName was not a file.

DirectoryNotFound
: The file associated with the entered ID could not be found.

3.5.3. Writing a file

```
Function WriteFile(
          InOut PathName              : Path_Name;
                Data                  : File_Data;
                ByteCount             : long)
        : GeneralReturn;
```

Abstract:

The WriteFile call is equivalent to calling ExpandPathName followed by a call to SubWriteFile with the returned absolute pathname.

Parameters:

PathName
: The pathname of the file to be written. Returned set to the absolute pathname of the file written.

Data
: A pointer to the data to be written.

ByteCount
: The number of bytes to write.

Returns:

Success
: The data was written under the name returned.

BadName
: APathName is incorrectly formatted (invalid syntax).

NoAccess
: The requesting user tried to write a file for which the user did not possess rights.

InvalidVersion
: The explicit version number was less than or equal to that of an existing version.

AlignmentError
: The data was not on a page boundary.

3.5.4. Checking on quoting of wildcards

```
Function IsQuotedChar(
             S                          : Wild_Path_Name;
             Index                      : integer)
         : boolean;
```

Abstract:

Checks whether Wild_Path_Name(Index) is quoted by preceding \ characters.

Parameters:

S — The string to check.

Index — The index of the desired character in that string.

Returns:

TRUE — If the character is quoted by preceding \ characters.

FALSE — Otherwise.

3.5.5. Finding first occurrence of unquoted character

```
Function IndexlUnquoted(
               S                        : Wild_Path_Name;
               C                        : char)
         : integer;
```

Abstract:

Find the first occurrence of C in S that is not quoted.

Parameters:

S — The string to search.

C — The character to find; it should not be '\'.

Returns:

The index of the matching character, or zero, if none.

3.5.6. Expanding a pathname into an absolute pathname

```
Function ExpandPathName(
          InOut WildPathName              : Wild_Path_Name;
                ImplicitSearchList        : Env_Var_Name)
        : GeneralReturn;
```

Abstract:

The ExpandPathName call accepts a pathname relative to an implicit logical name, and returns the corresponding absolute pathname after the logical name is expanded. If no implicit logical name is specified in the call and the pathname is a relative pathname, then the expansion is done relative to the logical name default (resulting in an expansion in the current directory). The user may override the default expansion by specifying either an absolute pathname or an explicit logical name in the pathname parameter. Logical names will be recursively flattened as necessary to complete the expansion. Undefined logical names will cause the expansion to fail with an empty search list error.

This call differs from FindPathName in that only a macro expansion is done, and no attempt is made to verify that the name exists or is even valid. Whereas FindPathName should normally be used before a LookupName type of operation, ExpandPathName should normally be used before an EnterName type of operation. Note that only the first element in the logical name search list is used by this call.

Parameters:

WildPathName
: The (potentially) relative pathname to be expanded. Changed to absolute pathname.

ImplicitSearchList
: The pathname is to be interpreted as relative to this logical name. If this parameter is blank, the logical name default is used.

Returns:

Success — The name search was successful.

FirstItemNotDefined

The specified logical name was not defined.

SearchlistNotFound
: The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
: A recursively defined logical name resulted in a logical name expansion loop.

3.5.7. Completing a filename

```
Function CompletePathName(
      var WildPathName                  : Wild_Path_Name;
          ImplicitSearchList            : Env_Var_Name;
          FirstOnly                     : boolean;
      var Cursor                        : integer)
    : long;
```

Abstract:

Do filename completion in the filename indicated by WildPathName.

Parameters

WildPathName
: The partial filename to be expanded. Changes to indicate the (partially) completed filename.

ImplicitSearchList
: Name of the search list to use if a partial path name is supplied.

FirstOnly
: If true, only look in the first item of the search list. Should ordinarily be false.

Cursor
: Position in WildPathName *after* which the implicit '*' should be inserted. Changed to indicate the corresponding position in the expanded WildPathName. The corresponding position is *forced* to be at the *end* of an entry name. Most common usage is: Cursor = length(WildPathName).

Returns:

Number of names that matched WildPathName.

0 = no matches; WildPathName unchanged.
1 = unique match; WildPathName is the expanded name.
n = several matches; WildPathName is the part that matches them all.

3.5.8. Completing a filename relative to EnvMgr connection

```
Function EnvCompletePathName(
                EnvConnection           : Port;
            var WildPathName            : Wild_Path_Name;
                ImplicitSearchList      : Env_Var_Name;
                FirstOnly               : boolean;
            var Cursor                  : integer)
          : long;
```

Abstract:

Do filename completion on the filename indicated by WildPathName, relative to a supplied EnvMgr connection.

Parameters:

EnvConnection
: The Environment Manager connection to use.

WildPathName
: The partial filename to be expanded. Changed to indicate the (partially) completed filename.

ImplicitSearchList
: Name of the search list to use if a partial path name is supplied.

FirstOnly
: If true, only look in the first item of the search list. Should ordinarily be false.

Cursor
: Position in WildPathName *after* which the implicit '*' should be inserted. Changed to indicate the corresponding position in the expanded WildPathName. The corresponding position will never be less than the original poisition. Most common usage is: Cursor = length(WildPathName).

Returns:

Number of names that matched WildPathName.

0 = no matches; WildPathName unchanged.
1 = unique match; WildPathName is the expanded name.
n = several matches; WildPathName is the part that matches them all.

3.5.9. Performing a name search for any type of name

```
Function FindPathName(
         InOut PathName                : Path_Name;
               ImplicitSearchList      : Env_Var_Name;
               FirstOnly               : boolean;
           Out EntryType               : Entry_Type;
           Out NameStatus              : Name_Status
       ): GeneralReturn;
```

Abstract:

The FindPathName call searchs for a pathname relative to the implicit logical name, returning the name found and the entry type associated with it. If no logical name is specified in the call and the pathname is a relative pathname, then the search is done using the logical name default. The user may override the implicit logical name given in the call by specifying either an absolute pathname or an explicit logical name in the pathname parameter. Logical names are recursively expanded as necessary to complete the search. Except for the case that the top-level logical name is undefined, errors in the search will not terminate it. Thus, if a name in a list is undefined, it is effectively ignored.

In summary, there are three cases of pathnames that are handled:

1. absolute pathname - only a TestName on the name is necessary to get the entry type. An example of this type is "/Sys/Accent/Accent.Boot".

2. relative pathname - requires the flattening of the implicit logical name, and then iterating down the search list testing for pathname with TestName until it is found. An example of this type is "Accent/Accent.Boot", after using "path/Sys".

3. logical pathname - requires the parsing off the logical name part, the flattening of the logical name, and then iterating down the search list testing for pathname with TestName until it is found. An example of this type is "current:Accent/Accent.Boot", where "current" is defined to be /sys/.

Parameters:

PathName
: The relative, absolute, or logical pathname to be search for. It is returned set to the absolute pathname that was found.

ImplicitSearchList
: If pathname was a relative pathname then the pathname is to be interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly
: If true, only the first name in the search list is used; if false, a complete search is performed.

EntryType
: The type value of the entry found.

NameStatus
: Deleted / undeleted, Low version / high version.

Returns:

Success
: The name search was successful.

NameNotFound
: APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
: The specified logical name was not defined.

SearchlistNotFound
: The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
: A recursively defined logical name resulted in a logical name expansion loop.

3.5.10. Performing a name search for a file name

```
Function FindFileName(
          InOut FileName                : Path_Name;
                ImplicitSearchList      : Env_Var_Name;
                FirstOnly               : boolean)
        : GeneralReturn;
```

Abstract:

The FindFileName call is the same as FindPathName, but looks only for entries of type file.

Parameters:

FileName The relative, absolute, or logical pathname to be searched for. It is returned set to the absolute pathname that was found.

ImplicitSearchList
If pathname was a relative pathname then the pathname is to be interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly If true, only the first name in the search list will be used; if false, a completed search is performed.

Returns:

Success The name search was successful.

NameNotFound
APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
The specified logical name was not defined.

SearchlistNotFound
The logical name expanded to an empty search list due to references to undefined logical names.

NotAFile The name did not refer to a file.

SearchlistLoop
A recursively defined logical name resulted in a logical name expansion loop.

3.5.11. Performing a name search with extensions

```
Function FindExtendedPathName(
          InOut PathName              : Path_Name;
                ExtensionList         : Extension_List;
                ImplicitSearchList    : Env_Var_Name;
                FirstOnly             : boolean;
            Out EntryType             : Entry_Type;
            Out NameStatus            : Name_Status)
        : GeneralReturn;
```

Abstract:

The FindExtendedPathName call is like the FindPathName call with the additional function that it performs a two-dimensional search: first down the extension list and then down the directory search list. Each extension string is

successively concatenated to the terminal component of the name and tried successively in each search list directory (if an absolute pathname was not specified) until a match is found.

Parameters:

PathName The relative, absolute, or logical pathname to be searched for. Changed to the name actually found.

ExtensionList
The list of extensions. Each name in the extension list is terminated by a semicolon (';'). A null name implies a null extension. An empty list means that no extensions are applied.

ImplicitSearchList
If pathname was a relative pathname then the pathname is interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly If true, only the first name in the search list will be used; if false, a complete search is performed.

NameStatus Deleted / undeleted, Low version / high version.

Returns:

Success The name search was successful.

NameNotFound
APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
The specified logical name was not defined.

SearchlistNotFound
The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
A recursively defined logical name resulted in a logical name expansion loop.

3.5.12. Performing a file name search with extensions

```
Function FindExtendedFileName(
          InOut FileName              : Path_Name;
                ExtensionList         : Extension_List;
                ImplicitSearchList    : Env_Var_Name;
                First Only            : boolean)
        : GeneralReturn;
```

Abstract:

The FindExtendedFileName is the same as the FindExtendedPathName except that it only searches for names of files.

Parameters:

FileName The relative, absolute, or logical pathname for which to search. Changed to the name actually found.

ExtensionList
The list of extensions. Each name in the extension list is terminated by a semicolon (';'). A null name implies a null extension. An empty list means that no extensions are applied.

ImplicitSearchList
If pathname was a relative pathname then the pathname is to be interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly If true, only the first name in the search list will be used; if false, a completed search is performed.

Returns:

Success The name search was successful.

NameNotFound
APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
The specified logical name was not defined.

SearchlistNotFound
The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
A recursively defined logical name resulted in a logical name

expansion loop.

NotAFile The name was found but it was not a file.

3.5.13. Performing a name search for any specific type of name with an optional list of extensions

```
Function FindTypedName(
         InOut PathName              : Path_Name;
               ExtensionList         : Extension_List;
               ImplicitSearchList    : Env_Var_Name;
               FirstOnly             : boolean;
         InOut EntryType             : Entry_Type;
           Out NameStatus            : Name_Status)
       : GeneralReturn;
```

Abstract:

The FindTypedName call is like the FindExtendedPathName call except that it searches for a specific type of name. The extension list may be empty.

Parameters:

PathName The relative, absolute, or logical pathname to be searched for. It is returned set to the absolute pathname that was found.

ExtensionList
The list of extensions. Each name in the extension list is terminated by a semicolon (';'). A null name implies a null extension. An empty list means that no extensions are applied.

ImplicitSearchList
If pathname was a relative pathname then the pathname is to be interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly If true, only the first name in the search list will be used; if false, a complete search is performed.

EntryType
Entry type being search for. Entry_All finds the first one, and sets EntryType to the type found.

NameStatus Deleted / undeleted, Low version / high version.

Returns:

Success The name search was successful.

NameNotFound
: APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
: The specified logical name was not defined.

SearchlistNotFound
: The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
: A recursively defined logical name resulted in a logical name expansion loop.

ImproperEntryType
: Name was found but was of wrong type.

3.5.14. Finding a wild-carded name using a search list

```
Function FindWildPathnames(
          InOut WildPathName          : Path_Name;
                ImplicitSearchList    : Env_Var_Name;
                FirstOnly             : boolean;
                NameFlags             : Name_Flags;
                EntryType             : Entry_Type;
            Out FoundInFirst          : boolean;
            Out DirName               : APath_Name;
            Out EntryList             : Entry_List;
            Out EntryListCnt          : long)
        : GeneralReturn;
```

Abstract:

Finds matches for a wild-carded name using a search list. For each item in the search list, the name is looked up using SesScanNames. If any matches are found, the search stops and the results from SesScanNames are returned. Note that this routine may not be quite what you want -- it returns the first non-empty match-set, NOT the union of all the match-sets. Think about the distinction before deciding whether this routine is appropriate for your application!

Parameters:

WildPathName
: The relative, absolute, or logical pathname to be searched for. Only the terminal name component may have wild-card characters.

It is returned set to the absolute pathname that was found.

ImplicitSearchList
: If pathname was a relative pathname then the pathname is to be interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly
: If true, only the first name in the search list will be used; if false, a complete search is performed.

NameFlags
: The flags to inhibit or allow deleted / undeleted names.

EntryType
: The type value of the entries to return. Entry_All causes all types of entries to be returned.

FoundInFirst
: Returned TRUE if and only if the first item in the search list produced the match.

DirName
: Returned by SesScanNames, set to the absolute pathname of the directory in which the matches were found.

EntryList
: List of entry names, types, version and status as returned by SesScanNames.

EntryListCnt
: The number of list elements returned; zero if no match occurred.

Returns:

Success
: The name search was successful.

NameNotFound
: APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
: The specified logical name was not defined.

SearchlistNotFound
: The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
: A recursively defined logical name resulted in a logical name expansion loop.

3.5.15. Finding a wild-carded name using a search list and the Environment Port

```
Function EnvFindWildPathNames(
                EnvConnection          : Port;
            var WildPathName           : Path_Name;
                ImplicitSearchList     : Env_Var_Name;
                FirstOnly              : boolean;
                NameFlags              : Name_Flags;
                EntryType              : Entry_Type;
            var FoundInFirst           : boolean;
            var DirName                : APath_Name;
            var EntryList              : Entry_List;
            var EntryList_Cnt          : long)
          : General Return;
```

Abstract:

The EnvFindWildPathNames call is identical to the FindWildPathNames call except that the Environment Port can be explicitly specified.

Parameters:

EnvConnection

WildPathName
: The relative, absolute, or logical pathname to be searched for. Only the terminal name component may have wild-card characters. It is returned set to the absolute pathname that was found.

ImplicitSearchList
: If pathname was a relative pathname, then the pathname is to be interpreted as relative to this search list. If this parameter is blank, the logical name default is used.

FirstOnly
: If true, only the first name in the search list will be used; if false, a complete search is performed.

NameFlags
: The flags to inhibit or allow deleted / undeleted names.

EntryType
: The type value of the entries to return. Entry_All causes all types of entries to be returned.

FoundInFirst
: Returned TRUE if and only if the first item in the search list produced the match.

DirName
: Returned set to the absolute pathname of the directory in which the

matches were found.

EntryList List of entry names, types, version and status as returned by SesScanNames.

EntryList_Cnt
The number of list elements returned; zero if no match occurred.

Returns

Success The name search was successful.

NameNotFound
APathName was not found in the directory (but the directory exists).

FirstItemNotDefined
The specified logical name was not defined.

SearchlistNotFound
The logical name expanded to an empty search list due to references to undefined logical names.

SearchlistLoop
A recursively defined logical name resulted in a logical name expansion loop.

3.5.16. Finding the terminal component and version number of a pathname

```
Procedure ExtractSimpleName(
              Name                  : Path_Name;
          Out StartTerminal         : integer;
          Out StartVersion          : integer);
```

Abstract:

The ExtractSimpleName call returns the indices of the terminal component of the name and the version number.

Parameters:

Name The pathname to check.

StartTerminal
Returns the index of the first character of the terminal component of the name. It will be length(Name) + 1 if the name is a directory name (ends with '/' or ':').

StartVersion

Returns the index of the version suffix of name (at the '#'). It will be length(Name) + 1 if the name has no version.

3.5.17. Returning the terminal versionless component of a pathname

```
Function SimpleName(
                PathName            : Path_Name)
        : Entry_Name;
```

Abstract:

Returns the terminal node of a name without a version number.

Parameters:

PathName

The path name you want the terminal node for.

Result:

EntryName The terminal Entry_Name for the Path_Name.

3.5.18. Contracting an absolute pathname to a relative pathname when possible

```
Function StripCurrent(
          InOut WildPathName          : Wild_Path_Name)
        : General Return;
```

Abstract:

This routine may be called by an application in order to try to shorten an absolute pathname suitable for typeout to the user. If the initial path components of WildPathName matches the first entry of the logical name Current, then the relative pathname with the initial components removed is returned. Otherwise, the absolute pathname is returned. A version number is included.

Parameters:

WildPathName

The absolute pathname to be converted.

Returns:

Success The absolute pathname was processed correctly.

NoCurrent
The logical name Current was undefined.

PathnameError
Pathname syntax error.

FirstItemNotDefined
The specified logical name was not defined.

3.5.19. Adding an extension to a pathname

```
Procedure AddExtension(
          InOut FileName          : Path_Name;
                Extension         : String);
```

Abstract:

Adds an extension to a file name if it is not already there.

Parameters:

FileName Name to check. It is changed if the extension is added.

Extension The extension to add.

3.5.20. Changing an extension of a pathname

```
Procedure ChangeExtensions(
          InOut Name              : Path_Name;
                EList             : Extension_List;
                NewExt            : string);
```

Abstract:

The ChangeExtensions call removes any of a list of extensions from a file name if they are there and then adds NewExt to the end.

Parameters:

Name The path name to be modified.

EList The list of extensions to check for and replace.

NewExt The extension to add.

3.5.21. Finding the next extension in a list

```
Function NextExtension(
         InOut EList            : Extension_List)
        : string;
```

Abstract:

The NextExtension call retrieves the next extension from a list of them which are terminated by semicolons. Removes the extension from the list.

Parameters:

Elist The list of extensions to be modified.

Result:

Extension The next extension.

3.5.22. Removing extensions

```
Procedure RemoveExtension(
         InOut FileName          : Path_Name;
               Extension         : String);
```

Abstract:

The RemoveExtension call removes an extension from a file name if it is there.

Parameters:

FileName The file name to check. It is altered to remove the extension if it is there.

Extension The extension to remove.

3.6. Sesame routines

3.6.1. Mounting a partition

```
Function SesDiskMount(
      ServPort          : Port;
      PartName          : string
     ): General Return;
```

Abstract:

Mounts a partition for the exclusive use of a particular file server.

Parameters:

ServPort File system service port for your machine (SesPort).

PartName The name of the partition to be mounted. Format is: *device:partition*. May end with a backslash (/).

Returns:

Success The partition is mounted.

NotAPart The name of the partition is incorrect.

PartNotAvail
The partition is already mounted for another file server.

3.6.2. Dismounting a partition

```
Function SesDiskDisMount(
      ServPort          : Port;
      PartName          : string
     ): General Return;
```

Abstract:

The partition referred to by PartName is dismounted.

Parameters:

ServPort File system service port to your machine (SesPort).

PartName The name of the partition to be dismounted. The format is: *device:partition*. May end with a backslash (/).

Returns:

Success The partition is dismounted. The file server will no longer find,

read, or write any files on that partition.

PartNotMounted
: The partition was not mounted.

3.6.3. Mounting a physical device (disk)

```
Function SesMountDevice(
      ServPort          : Port;
      Interface         : DiskInterface;
      Log_Unit          : InterfaceInfo;
  var UnitNum           : Integer;
  var PartL             : PtrPartDList;
  var NumElts           : Integer
    ): General Return;
```

Abstract:

SesMountDevice gets the information about all the partitions on a physical device (a disk) and mounts that device if it is not already mounted.

Parameters:

ServPort
: File system service port to your machine (SesPort).

Interface
: The type of interface that the disk is attached to.

Log_Unit
: Information to select a disk on that interface.

UnitNum
: Returns the logical disk number where that disk was mounted by the kernel.

PartL
: Returns a pointer to an array of PartitionInformation blocks. The array is allocated by this routine.

NumElts
: Returns the number of partitions about which information is returned.

Returns:

Success
: The disk is mounted.

Failure
: The disk was not mounted.

3.6.4. Getting partitions for all mounted devices

```
Function SesGetDiskPartitions(
      ServPort          : Port;
  var PartL             : ptrPartDList;
  var NumElts           : Integer
    ): General Return;
```

Abstract:

SesGetDiskPartitions gets the disk partitions for all mounted devices. This routine returns the same information as SesMountDevice.

Parameters:

ServPort — File system service port to your machine (SesPort).

PartL — Returns a pointer to an array of PartitionInformation blocks. The array is allocated by this routine.

NumElts — Returns the number of partitions about which information is returned.

Returns:

Success — Information is returned.

3.6.5. Setting the creation date of a file

```
Function SesSetPOSWriteDate(
      ServPort          : Port;
      APathName         : APath_Name;
      WriteDate         : Internal_Time
    ): General Return;
```

Abstract:

SesSetPOSWriteDate sets the CreationDate file header field for an existing file.

Parameters:

ServPort — File system service port to your machine (SesPort).

APathName — The fully or partially specified name of the file.

WriteDate — Sets the time the file was *created* for all other calls.

Returns:

Success — Creation date is set.

NameNotFound
: APathName was not found in the directory (but the directory exists).

DirectoryNotFound
: One of the directories in the pathname does not exist.

NoAccess : The user does not have write access on the file.

3.6.6. Establishing a link to a foreign File System process

```
Function SesEnterForeignSesamoid(
      ServPort          : Port;
      APathName         : APath_Name;
      ForeignPort       : Port;
      ForeignPrefix     : APath_Name
    ): General Return;
```

Abstract:

Enters a link to a foreign File System process. No checking is done on the foreign Port or Prefix.

Parameters:

ServPort : File system service port to your machine (SesPort).

APathName : The path name under which to enter the foreign File System link. It is treated as a directory name.

ForeignPort : The Net access port to the foreign File System.

ForeignPrefix
: The path name for the directory in the foreign File System where this link should point.

Returns:

Success : The link to the foreign File System process is entered.

DirectoryNotFound
: One of the directories in the path name does not exist.

NoAccess : The user does not have write access to the directory.

3.6.7. Reading a link to a foreign File System process

```
Function SesLookupForeignSesamoid(
      ServPort          : Port;
      APathName         : APath_Name;
  var ForeignPort       : Port;
  var ForeignPrefix     : APath_Name
    ): General Return;
```

Abstract:

Reads a link to a foreign File System process.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The path name to read.

ForeignPort Returns the Net access port to the foreign File System.

ForeignPrefix
Returns the path name for the directory in the foreign File System where this link should point.

Returns:

Success The path name is read.

DirectoryNotFound
One of the directories in the path name does not exist.

NoAccess User does not have read access to the directory.

ImproperEntryType
Entry is not for a foreign file system.

3.6.8. Providing a message server port to Sesamoid

```
Function SesMsgServerPort(
      ServPort          : Port;
      MsgServerPort     : Port
    ): General Return;
```

Abstract

Provides a message server port to Sesamoid. This call is for use by system processes only.

Parameters:

ServPort Service port to Sesamoid - must be PrivPort.

MsgServerPort
Message server port to set.

Returns:

Success Message server port is set.

NoAccess The ServPort was not PrivPort.

3.6.9. Returning a public access port

```
Function SesGetNetPort(
      ServPort          : Port;
  var SesNetPort        : Port
    (: General Return;
```

Abstract:

SesGetNetPort returns the public access port to Sesamoid's net service. This call is for use by system processes only.

Parameters:

ServPort Service port to Sesamoid - must be PrivPort.

SesNetPort Returns Sesamoid's public connection port.

Returns:

Success The port is returned.

NoAccess The ServPort was not PrivPort.

3.6.10. Getting the segment ID of a file.

```
Function SesGetSegID(
      ServPort          : Port;
  var APathName         : APath_Name;
  var SegmentID         : SegID
    ): General Return;
```

Abstract:

SesGetSegID gets the segment ID for a file. This is used by BindBoot to get the actual address of a boct file. The segment cannot be on a remote machine.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName File name to look for. Changed to the name actually found.

SegmentID Returns the segment ID (disk address) of the file.

Returns:

Success Segment ID is returned.

NameNotFound
APathName was not found in the directory (but the directory exists).

DirectoryNotFound
One of the directories in the path name does not exist.

NoAccess User does not have read access to the file.

NotAFile The entry found under APathName is not a file.

3.6.11. Entering a segment into the directory system

```
Function SesEnterSegID(
      ServPort          : Port;
  var APathName         : APath_Name;
      SegmentID         : SegID
    ): General Return;
```

Abstract:

SesEnterSegID enters a segment into the directory system under a path name. The segment cannot be on a remote machine. This call is used by DirScavenge to rebuild directories. This call assumes that the segment represents an existing file that should already be in the directory. Thus, it will not replace an existing directory entry.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The name to enter the segment under. Changed to the name actually used.

SegmentID The segment to enter.

Returns:

Success The segment is entered.

DirectoryNotFound
One of the directories in the path name does not exist.

NoAccess The user does not have write access to the directory.

Failure SegmentID is not a valid segment ID.

3.6.12. Returning a full path name for a segment

```
Function SesGetSegName(
      ServPort          : Port;
      SegmentID         : SegID;
  var APathName         : APath_Name
    ): General Return;
```

Abstract:

SesGetSegmentName returns the full path name for a segment. The segment must be on the local machine.

Parameters:

ServPort File system service port to your machine (SesPort).

SegmentID Segment whose path name is sought.

APathName Returns the full path name for the desired segment.

Returns:

Success Full path name is returned.

NameNotFound
APathName was not found in the directory (but the directory exists).

DirectoryNotFound
One of the directories in the path name does not exist.

NoAccess User does not have read access to the partition.

3.6.13. Returning the access rights for an entry

```
Function SesGetAccessRights(
      ServPort          : Port;
  var APathName         : APath_Name;
  var Owner             : User_ID;
  var Rights            : Access_Rights
    ): General Return;
```

Abstract:

SesGetAccessRights returns the access rights for an entry.

Parameters:

ServPort — File system service port to your machine (SesPort).

APathName — Fully specified path name of the entry. Changed to the name actually found.

Owner — Returns the name of the owner for the entry.

Rights — Returns the access rights for the entry.

Returns:

Success — Access rights for the entry are returned.

DirectoryNotFound — One of the directories in the path name does not exist.

NameNotFound — APathName was not found in the directory (but the directory exists.

3.6.14. Changing the access rights for any entry

```
Function SesSetAccessRights(
      ServPort          : Port;
  var APathName         : APath_Name;
      NewOwner          : User_ID;
      NewRights         : Access_Rights
    ): General Return;
```

Abstract:

SesSetAccessRights changes the access rights for a particular entry.

Parameters:

ServPort File system service port to your machine (SesPort).

APathName The fully specified path name of th entry.

NewOwner The new owner for the entry.

NewRights The new access rights for the entry.

Returns:

Success The access rights are changed.

Failure NewOwner or NewRights is invalid (out of range).

DirectoryNotFound
One of the directories in the path name does not exist.

NameNotFound
APathName was not found in the directory (but the directory exists).

NoAccess The uyser does not have owner access to the directory.

3.6.15. Getting the default access rights mask

```
Function SesGetDefaultAccess(
      ServPort          : Port;
  var DefaultAccess     : Access_Rights
    ): General Return;
```

Abstract:

Returns the default access rights mask used to create new entries.

Parameters:

ServPort File system service port to your machine (SesPort).

DefaultAccess
Returns access mask.

Returns:

Success Access mask is returned.

3.6.16. Setting the default access rights mask

```
Function SesSetDefaultAccess(
      ServPort          : Port;
      NewDefaultAccess  : Access_Rights
    ): General Return;
```

Abstract:

This call sets the default access rights mask used to create new entries.

Parameters:

ServPort File system service port to your machine (SesPort).

NewDefaultAccess
New access mask to set.

Returns:

Success New access mask is set.

3.6.17. Authenticating ServerPort

```
Function SesAuthServerPort(
      ServPort          : Port;
      AuthServerPort    : Access_Rights
    ): General Return;
```

Abstract:

For use by system processes only.

3.6.18. Changing access rights associated with SesPort

```
Function SesConnect(
      ServPort          : Port;
      RegPort           : Port
    ): General Return;
```

Abstract:

This call makes the connection for this machine's user. Changes the access rights associated with SesPort. For use by system processes only.

Parameters:

ServPort File system service port to your machine (SesPort).

AuthServerPort
Registered port for user.

Returns:

Success Connection made.

InvalidUser User does not have access right to this machine.

3.6.19. Performing direct disk I/O

```
Function SesDirectIO(
      ServPort          : Port;
  var CmdBlk            : DirectIOArgs;
  var DataHdr           : Header;
  var Data              : DiskBuffer
    ): General Return;
```

Abstract:

This call performs disk I/O direectly to the disk, bypassing the file and segment systems. This call may cause great damage to the disk structures.

Parameters:

:

ServPort File system service port to your machine (SesPort).

CmdBlk The disk command to be executed.

DataHdr The 16-byte disk header.

Data A 512-byte data block.

Returns:

Success Disk operation successfully performed.

Otherwise I/O error. The return code indicates the specific disk error.

3.7. Version Numbers

The following routines, found in SesDiskUser.Pas and SesameUser.Pas, return version numbers.

3.7.1. Returning version number of disk handling system

```
Function SesDisk_Version(
             ServPort           : Port;
         var VerString          : string
    ): GeneralReturn;
```

Abstract:

Returns the version number of the disk handling system.

Parameters:

ServPort Port for the current process

VerString Returns a string containing the version number

Returns:

Success

3.7.2. Returning version number of file system

```
Function Sesame_Version(
            ServPort            : Port;
        var VersionStr          : string
    ): GeneralReturn;
```

Abstract:

Returns the version number of the file system.

Parameters:

ServPort Port for the current process

VersionStr Returns a string containing the version number

Returns:

Success

4. Device Information

4.1. Physical Format of Devices

Each device consists of discrete positions known as cylinders. The device's read/write heads divide the cylinders into tracks. Each track is divided into sectors. Figure 1 lists the number of tracks per cylinder and the number of sectors per track for Micropolis and Shugart hard disks.

Figure 1: Cylinders, Tracks, and Sectors for Volumes

Drive/ capacity	Cylinders	Tracks/ cylinder	Sectors/ track
Micropolis 1303 35-MB (5.25")	830 (0 - 829)	5 (0 - 4)	16 (0 - 15)
Shugart 4002 24-MB	202 (0 - 201)	8 (0 - 7)	30 (0 - 29)
Micropolis 35 35-MB	580 (0 - 579)	5 (0 - 4)	24 (0 - 23)

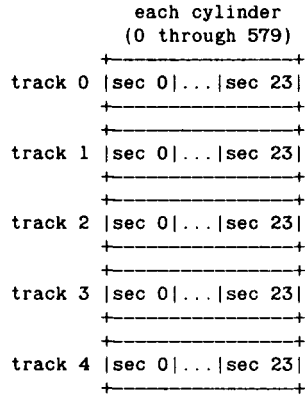
The Micropolis 1303 5.25" disk consists of 830 cylinders numbered from 0 through 829. It contains five tracks per cylinder (tracks 0 through 4). The tracks on the 5.25" Micropolis disk are divided into 16 sectors, numbered 0 through 15.

The Shugart 4002 disk consists of 202 cylinders numbered from 0 through 201. A 24-MB Shugart contains eight tracks per cylinder (tracks 0 through 7). The tracks on a Shugart disk are divided into 30 sectors, numbered 0 through 29.

The Micropolis 35 disk consists of 580 cylinders numbered from 0 through 579 and uses five read/write heads. Thus, a Micropolis disk contains five tracks per cylinder (tracks 0 through 4). The tracks on a Micropolis disk are divided into 24 sectors, numbered 0 through 23.

Figure 2 illustrates the cylinder, track, and sector divisions of a 35-MB Micropolis disk.

Figure 2: 35-MB Micropolis Disk Organization



The actual maximum usable space on any given disk will be less once it is formatted. Although a disk may be said to be 35-MB, it, in fact, may have only 33-MB of "usable" space. The table below gives the unformatted and formatted disk sizes of the different disks available with the PERQ.

Disk	Unformatted	Formatted
14" Shugart	N/A	24-MB
8" Micropolis	39-MB	35-MB
5 1/4" Micropolis	43-MB	34-MB
5 1/4" Maxtor	105-MB	81-MB
5 1/4" Maxtor	143-MB	110-MB

Each sector on a device minimally contains a data block and a three-word (6-byte) physical header.

As its name implies, a data block is the data area of a sector. On a hard disk, the data block is an array of 256 16-bit words that contain data.

The physical header permits the disk controller to verify head positioning and uniquely identifies each sector on the device; it contains the cylinder number, the track number within the cylinder, and the sector number within the track.

Physical Disk Addresses (PDAs) specify the exact physical location of a sector; the PDA defines sector location by cylinder number, track number within that cylinder, and sector number within that track. To communicate with a device, microcode passes the PDA to the specific hard disk.

A PDA is a 32-bit (2-word) value. The Shugart disk controllers expect the cylinder, track, and sector specification in a single word, while the Micropolis disk controllers expect the specification in two words.

In a Shugart disk PDA, the low order word specifies the cylinder, track, and sector number as an octal value. The high byte (bits 8 - 15) of the low order word contains the cylinder number. Bits 5 through 7 of this word contain the track number. Bits 0 through 4 contain the sector number. Thus, the first Physical Disk Address (expressed in octal) of a 24-MB Shugart disk is:

Low High

000000 000000

(specifies cylinder 0,

track 0, sector 0)

The highest Physical Disk Address for a 24-MB Shugart disk is:

Low High

144775 000000

(specifies cylinder 201,
track 7, sector 29)

The high order word of a Shugart disk PDA must be all zeroes.

In a Micropolis disk PDA, both the low and high order words are significant. The low byte (bits 0 - 7) of the low order word specifies the sector and the high byte (bits 8 - 15) of the low order word specifies the track. The high order word specifies the cylinder number. Thus, the first Physical Disk Address (expressed in octal) of a 35-MB Micropolis disk is:

Low High

000000 000000

(specifies cylinder 0,
track 0, sector 0)

The highest Physical Disk Address for a Micropolis 35 disk is:

Low High

002027 001103

(specifies cylinder 579,
track 4, sector 23)

On a hard disk, each sector also contains a separate 8-word data area, known as a logical header. This area is used by the file system to verify to which file the data belongs and to permit a recovery operation in the event of a failure.

4.2. Logical Format of Devices

The microcode views a disk as a cluster of uniquely addressable sectors. Since the disks are different, the microcode provides different cylinder, track, and sector abstractions of the devices. Conversely, the file system views a disk simply as an array of data area pairs addressed by sequential integers in the range 0 through n. (Note that this view of a disk is implicit and does not directly correspond to any module or procedure in the file system.)

The file system refers to the data area pairs as a logical block. A logical block consists of an eight-word logical header and a 256-word data block.

The hard disk hardware supports the data area pair concept directly. The implementation of logical blocks (logical header/data block) is straightforward on hard disks; each physical sector on a hard disk that is accessible to the file system can be viewed as a logical block.

The file system assigns a sequential number to each logical block on a disk, known as the Logical Block Number or LBN. The logical blocks on a disk are numbered consecutively from 0 to n-1, where the disk contains n logical blocks. A Logical Block Number identifies each data area pair on a disk that is accessible to the file system.

A file system volume contains a microprogram that runs diagnostics and reads the .Boot and .MBoot files (the .Boot file contains the operating system and the .MBoot file contains the

QCode interpreter and IO microcode). This microprogram resides on disk in physical space reserved for it, known as the boot area. The boot microcode in ROM accesses the reserved area to initiate the boot sequence. The code is maintained outside the logical disk structures and is not accessible to the file system.

On a Shugart disk, the first track (cylinder 0, track 0, sectors 0 through 29) is reserved for the bootstrap code. The first logical block on a Shugart disk, LBN 0, is the first physical sector following the boot code (cylinder 0, track 1, sector 0). Thus, Shugart disk LBNs range from 0 through n-1 where n is the maximum number of sectors on the disk, minus 30.

On a Micropolis 35MB disk, the first two tracks (cylinder 0, tracks 0 and 1) are reserved for the bootstrap code. The first logical block on a Micropolis disk, LBN 0, is at physical location cylinder 0, track 2, sector 0. Thus, Micropolis disk LBNs range from 0 through n-1 where n is the file system's maximum number of sectors on a disk, minus 48.

The file system supports a single address space for the logical blocks on hard disks, known as a Logical Disk Address or LDA.

Like a Physical Disk Address, Logical Disk Addresses are 32-bit (2-word) values, but the values are distinctly different. A PDA uniquely identifies a sector on a given device by cylinder number, track number within the cylinder, and sector number within the track. An LDA encodes a disk specifier and a positive number representing the disk specific Logical Block Number.

Both words of an LDA are significant for hard disks. The following describes the significance of each bit that forms an LDA.

Low word:

Bits 0 through 7 are all zeroes. Bits 8 through 15 form the low eight bits of the LBN.

High word:

Bits 0 through 11 form the high twelve bits of the LBN.

Bits 12 and 13 specify the volume.

Bits 14 and 15 indicate whether the address is on disk, and therefore in permanent storage, or in virtual memory. Under the current system, these bits are always set to indicate permanent storage.

Most I/O operations must know the contents of the block's logical header. The logical header contains the following information:

File serial number (2 words)
Logical block number (1 word)
Filler word (1 word)
Next Logical Disk Address (2 words)
Previous Logical Disk Address (2 words)

The serial number is the Logical Disk Address of the File Information Block (FIB). Section 2.4 describes the FIB in detail. Note that the serial number is written as a Physical Disk Address and converted to an LDA when the header is read.

The Logical block number in the logical header represents the block number within a series of blocks, not the sequential LBN of the block. This document refers to the logical block number of a block within a series of blocks as the relative logical block number.

The filler word is used by the file system in its free block allocation scheme. When the block is allocated, the filler word is unchanged.

The Next Logical Disk Address and the Previous Logical Disk

Address form doubly linked lists of blocks. The values are the LDAs of the next block and the preceding block. For both pointers, a zero value terminates the link. The values for these pointers are actually written as Physical Disk Addresses and converted to LDAs when the header is read for processing purposes.

To access a logical block, the block's LBN is first extracted from the LDA and then converted into a cylinder, track, sector specification. For example, assume the file system must access a logical block on a 24-MB Shugart disk at the following Logical Disk Address:

```
151000 140000
```

First, it extracts the Logical Block Number. The high byte of the low order word forms the low eight bits of the LBN.

```
Low-order word
    151000

high-byte|low-byte
 11010010|00000000
```

Bits 0 through 11 of the high order word form the high twelve bits of the LBN.

```
High-order word
    140000

high-byte|low-byte
 11000000|00000000
```

Thus, the binary representation of the LBN is:

```
00000000000011010010
```

which converts to 322(8) or LBN 210. Since the numbering scheme for logical blocks on a Shugart disk omits the first 30 sectors, Shugart disk LBN 210 is actually physical sector 240. The cylinder, track, sector specification for physical sector number 240 is cylinder 1, track 0, sector 0 (derived from 30 sectors per track and 8 tracks per cylinder).

4.3. Device Composition

The device that carries the file system structure is referred to as a volume. A volume is simply an ordered set of logical blocks.

A Device Information Block (DIB) identifies a volume as an Accent file structured volume. The DIB is the first logical block and has a defined physical location on the volume. It contains all the fixed information about the volume; the DIB describes the number of logical blocks on the volume, specifies the physical addresses for the Qcode and microcode boots, and identifies the volume with a label (an ASCII string of one to eight characters). The DIB serves as the foundation of the volume structure. Section 5.2 describes the DIB in further detail.

The file system divides the set of logical blocks on a volume into physically contiguous sections known as partitions. Partitions are created on a volume using the Partition program. (The Partition program accepts input for partition name and size, and permits you to specify the number of partitions. The Partition program is described in detail in the "User Facilities" document of the *Accent User's Manual*.) The DIB contains pointers to the first block of each physically contiguous disk area.

The first block of each partition is the Partition Information Block (PIB), which details partition specific information. Section 5.3 describes the PIB in detail.

Any data of interest in a partition (that is, all blocks not available

for allocation) are contained in segments. A segment is a cluster of logical blocks linked together by pointers in each block's logical header. Each segment in a partition has a block that describes the segment, known as the information block. Section 5.4 describes the information block in detail.

The information block specifies the type of segment and, most importantly, contains pointers to the blocks that form the segment. All blocks that form the segment contain the Logical Disk Address of the information block in the serial number of their logical header. The LDA of the information block identifies all blocks that belong to a given segment. The file system refers to the LDA of a segment's information block as the SegID. The SegID is used in all references to a segment.

Named segments form files. A file name consists of four portions and takes the form:

/ device / partition / directory1 / ...directory8 / filename

A complete file specification is referred to as a full pathname; it specifies the complete path for a file. Since the file system applies defaults for device, partition, and directories, users need not specify a full pathname. A file specification that omits all portions except the file's name is referred to as a simple file specification.

Each file in a partition has an entry in a directory and the information block from the segment. The directory entry contains a pointer to the information block of the file. Section 6.2 describes directory entries. The file's information block contains the same type and pointer data as the information block from a segment with additional file system specific information. Thus, all files are segments, but because a file has a directory entry and additional information in its information block, not all segments

are files.

The information block of a segment and the information block of a file share the same data structure, known as the File Information Block or FIB. Section 5.4 describes the FIB in detail.

Since a segment is the origin of a file, the serial number in the logical header of all blocks that form the file contains the LDA of the FIB (the SegID).

The logical address of the File Information Block is sufficient to locate a file uniquely in a partition; the address is hardly mnemonic. A directory is a file that contains the names and pointers to the FIBs of each file within the directory; directories associate symbolic names with the address of the FIB.

Since directories are simply files that contain pointers to other files, the files contained in a directory can be other directories. Thus, directories can be nested to form a hierarchical, multi-directory structure. You can construct directory hierarchies of arbitrary depth and complexity to structure files in whatever manner is convenient.

The main directory in a partition (the base of the multi-directory structure) is the root directory for that partition. Subsequent levels are referred to as subdirectories. Note that since a directory is a file, an FIB exists for each directory.

5. File System Data Structures

This section describes the format of the logical structures on a file system volume.

5.1. Overview

The Device Information Block (DIB) identifies a file structured volume and contains pointers to Partition Information Blocks (PIBs).

Each PIB defines the limits of physically contiguous areas on the volume and contains a pointer to the File Information Block (FIB) of each partition's root directory.

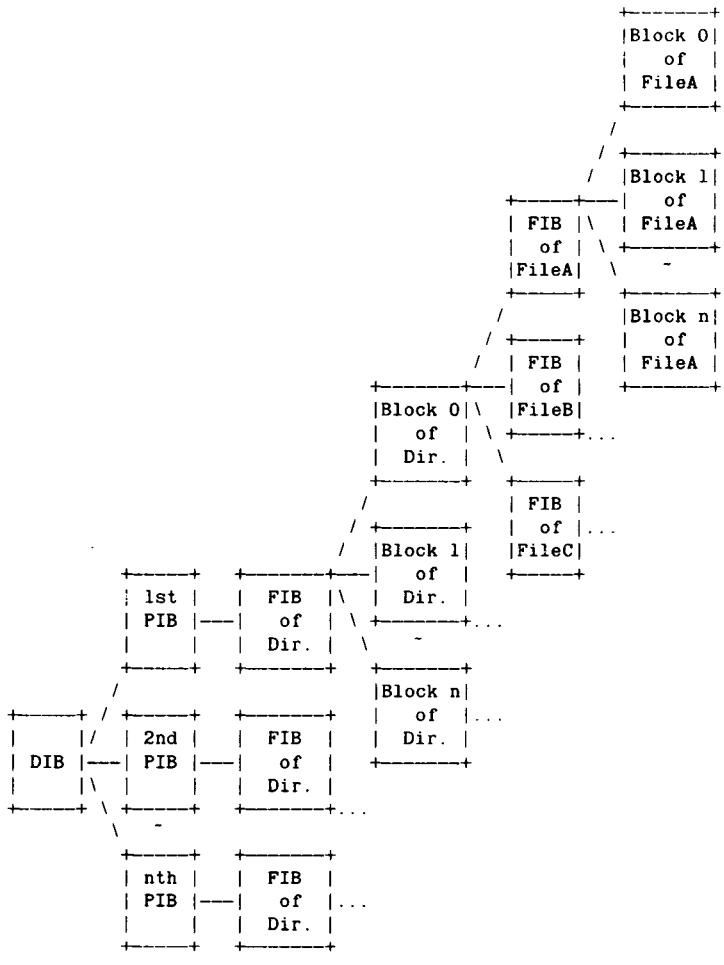
The FIB of the root directory, as well as the FIB of all other directories, contains pointers to each of the blocks that form the directory.

Each block of a directory contains pointers to the FIBs of the files that comprise the directory.

The FIB of a file contains pointers to the blocks that form the file.

Figure 3 depicts the data area links of the logical structure. Note that directory entries are hash coded by file name. The hash function specifies the block number of the directory to contain the entry. Therefore, directory entries are not necessarily accurately represented in Figure 3. However, Figure 3 accurately depicts placement if you assume that the file names hash to the same values. Chapter 6 provides more specifics on directory entries and the hash function.

Figure 3: Logical Links For On-Disk Structure



The sections that follow provide complete details on the logical structures.

5.2. Device Information Block

The Device Information Block (DIB) identifies a device for use with the Accent file system and serves as the ground zero entry point into the device's file structure. The DIB also contains a table that defines a mapping of 26 characters to 26 pairs of interpreter and system boot files. This table is read by the microprogram in the boot area.

The DIB is the first logical block on a volume (LBN 0) and contains pointers to each Partition Information Block (PIB).

The sections that follow provide a detailed description of the DIB. Note that the DIB and all Partition Information Blocks (PIBs) use the same format; some words in the structure are relevant only in a DIB structure and some only in a PIB structure.

Section 5.2.14 shows a graphic representation of the DIB.

5.2.1. Size of boot area

This word defines the number of tracks at a well-known place on the disk. The system boot loader is kept here for use during the boot process. This word is used only by the 5.25" disk.

5.2.2. Number of sectors

This word holds the number of sectors per track, which is the number of data blocks on a single track; typically 16 on a 5.25" disk. This word is used only by the 5.25" disk.

5.2.3. Number of heads

This word specifies the number of read/write heads on the device. It is used only by the 5.25" disk.

5.2.4. Number of cylinders

This word specifies the total number of cylinders on the entire disk. It is used only by the 5.25" disk.

5.2.5. Write pre-compensation cylinder

This next word is turned on in the inner track of the disk. This word specifies the cylinder number where the software needs to turn pre-compensation on. It is used only with the 5.25" disk.

5.2.6. BootTable

This 52-word array contains physical addresses for Qcode boots. Each double word of the 52-word array corresponds, successively, to the lowercase letters a through z for each .Boot file on a hard disk. For example, if a system contains an a and b boot, the first two words (words 0 and 1) of this array contain the physical address of the operating system to boot when lowercase a is booted, and the third and fourth words (words 2 and 3) contain the physical address of the operating system to boot when lowercase b is booted.

5.2.7. InterpreterTable

This 52-word array contains physical addresses for microcode boots. Each double word of the 52-word array corresponds, successively, to the letters for each .MBoot file. (MBoot files contain the Qcode interpreter microcode. MBoot is described in detail in the "Pascal/C Machine Reference" document in the *Accent Languages Manual.*)

5.2.8. PartitionName

This quad word contains the name (one to eight characters) of the partition. In the DIB, the partition name is interpreted as the name of the device (for example, SYS).

5.2.9. PartitionStart

This double word contains the LDA of the start of the partition. In the DIB, the partition start is interpreted as the first logical block (LBN 0) on the device. This double word always points to LBN 0 LDA 000000 140000.

5.2.10. PartitionEnd

This double word contains the LDA of the end of the partition. In the DIB, the partition end is interpreted as the last logical block on the device. This double word always points to the highest block number of the device.

5.2.11. PartitionLists

This 128-word array contains the LDA of the Partition Information Block (PIB) for each partition (thus, an array of 64 LDAs, one for each possible partition). As you create each partition, the Partition program (described in the "User Facilities" document of the *Accent User's Manual*) writes the address of its first block successively in this array.

5.2.12. PartitionRoot

This double word contains the LDA of the root partition (that is, the DIB) and always points to LBN 0.

5.2.13. PartitionType / DeviceType

This word specifies partition type (root, unused, or leaf) and device type (Winchester 24-MB, Winchester 5.25", unused1, unused2). In a DIB structure, the partition type is always root. The Partition program (described in the "User Facilities" document of the *Accent User's Manual*) writes the device type in this word when you partition the volume.

5.2.14. Disk information block layout

The Disk Information Block (DIB) is shown in Figure 4. Note that words 253, 254 and 255 are not used.

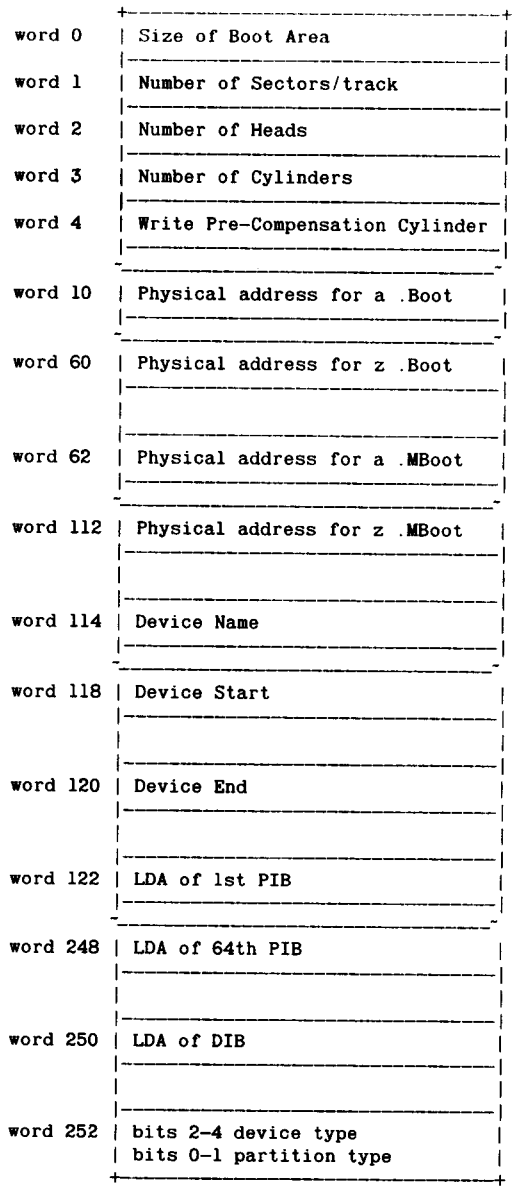
5.3. Partition Information Block

A partition is a physically contiguous area of logical blocks. The fundamental operations on a partition are allocation and deallocation of blocks from and to a pool of blocks in the partition. The pool of blocks is known as the partition's free block list. You allocate blocks to form segments, that can in turn form files, and de-allocate blocks to destroy segments. Since segments can form files that are organized by a directory structured name space, a partition also provides a root directory for all files allocated in the partition.

The first block of a partition contains the name and limits of the partition, information describing the free block pool of the partition, and the Logical Disk Address of the partition's root directory. This block is known as the Partition Information Block (PIB).

The PIB contains two pointers (Logical Disk Addresses) to the head (first) and tail (last) blocks on the free list. When a partition is mounted, its PIB is read into a table (PartTable) in memory. As blocks are allocated or deallocated, information in the PartTable is updated. The only dynamically updated information

Figure 4: Disk Information Block (DIB)



in the PIB is that which describes the partition's free block list. This information is updated in PartTable after every block allocation or deallocation, but is only updated intermittently in the copy of the PIB on disk. Consequently, if the system terminates abnormally, the disk copy may be incorrect. If the pointer to the head of the free list is incorrect, the system finds the actual start by following the filler word in the logical header.

The PIB replicates the DIB structure described in Section 5.2. However, the structure for the DIB defines the file system entry point for the device while the PIB structure defines the physically contiguous area of disk addresses for each partition. Thus, one PIB exists for each partition on the volume while exactly one DIB exists for the volume.

Partitions always start on cylinder boundaries. The Partition program (described in the "User Facilities" document of the *Accent User's Manual*) writes the PIB in the first block of a partition. Therefore, a PIB always starts on a cylinder boundary. The LBN of the first PIB depends on the volume.

The first PIB always is at physical address cylinder 1, track 0, sector 0. For a 24-MB Shugart disk, the first PIB is LBN 210 (LDA 151000 140000). For a 35-MB Micropolis disk the first PIB is LBN 72 (LDA 044000 140000).

The location of the PIBs for subsequent partitions depends on the number of blocks you allocate to each partition. A typical allocation for hard disks is 10080 blocks per partition.

Each PIB contains a pointer to the partition's root directory File Information Block (FIB). The root directory contains Logical Disk Addresses of the FIBs for files, including directories. Since directories are hierarchically structured, the root directory effectively points to all files within a partition.

The following sections provide a detailed description of the PIB. Note that the DIB and all PIBs use the same format; some words in the PIB structure are not relevant in a DIB structure.

5.3.1. FreeHead

This double word contains the LDA for the start of the partition's free block list. The FreeHead entry points to a block within the partition. This block is the actual start of the free list when the serial number and the previous pointer in its header are -1.

If the serial number and the previous pointer are not -1, the actual start of the free list is found by following the filler word until both are -1.

5.3.2. FreeTail

This double word contains the LDA for the end of the partition's free block list. Like the FreeHead entry, the FreeTail pointer in the PIB may not be accurate.

The FreeTail entry points to a block within the partition. This block is the actual end of the free list when the next pointer in its header is -1 (-1 indicates a nil pointer). If not, the actual end of the free list is found by following the next address pointers to -1.

5.3.3. NumberFree

This double word contains an integer value that suggests the number of free blocks in a partition. This value may not be accurate for the same reason that the accuracy of the FreeHead and FreeTail pointers is not guaranteed.

5.3.4. RootDirectoryID

This double word contains the LDA for the partition's root directory File Information Block (FIB). Section 5.4 describes the FIB. The root directory is a file that contains other directories and/or files.

5.3.5. BadSegmentID

Each partition has a linked list of bad blocks. The bad blocks are withheld from the free list and are not used by the file system. This double word contains the LDA for a partition's bad segment.

5.3.6. FillerField

The filler field of the logical header is used by the file allocation scheme to allocate blocks for files. This word indicates whether the filler should be interpreted as a disk block number (disk relative), or as an offset from the start of the partition (partition relative).

5.3.7. InterpreterTable

This 52-word array contains physical addresses for microcode boots and is therefore not relevant to the PIB. The word values are zero to indicate nil pointers. Section 5.2.7 describes the significance of these words.

5.3.8. PartitionName

This quad word array contains the name (up to eight characters) of the partition. You specify this name as input to the Partition program. The partition name can be the same as the device name, but each partition must have a distinct name.

5.3.9. PartitionStart

This double word contains the LDA of the first block of the partition. Since the PIB is the first block of a partition, the entry points to the partition's PIB.

5.3.10. PartitionEnd

This double word contains the LDA of the last block of the partition.

The entries for PartStart and PartEnd specify the size of each partition as physically contiguous disk addresses. You enter the number of blocks (size) for each partition as input to the Partition program. The maximum size of a partition is 65040 blocks.

5.3.11. PartitionLists

This 128-word array contains the LDA of the PIBs for each partition and is therefore not relevant to any PIB itself. Section 5.2.11 describes the significance of these words.

5.3.12. PartitionRoot

This double word contains the LDA of the root partition or DIB (LBN 0).

5.3.13. PartitionType / DeviceType

This word specifies partition type (root, unused, or leaf) and device type (Winchester 24-MB, Winchester 5.25", unused1, unused2). In a PIB structure, the partition type is always leaf.

5.3.14. Partition information block layout

The Partition Information Block (PIB) is illustrated in Figure 5.

5.4. File Information Block

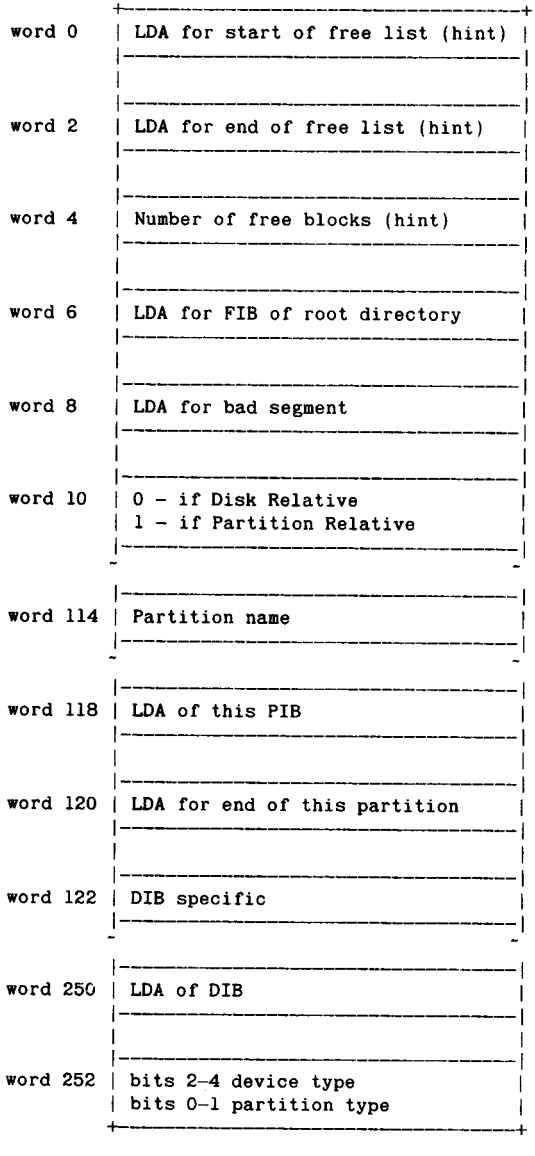
A single block of a segment or a file (including directories) describes each segment or file on an Accent file structured volume and is referred to as a File Information Block (FIB). The FIB is relative logical block number -1. The blocks that form the usable part of a segment or file have relative logical block numbers ranging from 0 through 32767. The FIB identifies to which particular segment or file a specific block belongs; the LDA of the FIB is the segment number for all the blocks of a segment or file.

The FIB contains all the information necessary to access an individual segment or file, as well as pointers to the blocks that form the segment or file. (Actually, the FIB only allows access to a segment's blocks; file access requires the DIB, PIB, and directory structures.)

Some entries in the FIB are not relevant when the structure describes only a segment. Specifically, the FIB of a segment has zero values for the first 52 words, while the FIB of a file has relevant values for these words.

The following sections provide a detailed description of the FIB. Section 5.4.9 supplies a graphic representation of the FIB.

Figure 5: Partition Information Block (PIB)



words 253, 254, and 255 are unused

5.4.1. File system data

This 52-word packed record contains basic file system information, including the file name. The file system uses this record to access and maintain a file. The record is not used by the segment system. Module IFileDefs.Pas provides the Pascal type definition for this packed record.

The first word (word 0) contains the number of blocks allocated to the file.

The second word (word 1) specifies the number of bits in the last block of the file and whether or not the file can be sparse.

The low order 12 bits (bits 0 through 11) specify the number of bits in the last block of the file. For example, when you allocate a block from the free list for use as the last block of a file and do not fill the entire block with data, the remainder of the block may contain random data or garbage. These 12 bits signal the end of significant data.

Bit 12 specifies whether or not the file can be sparse. Sparse files are files that contain unallocated blocks between allocated blocks.

Bits 13 through 15 are not used.

The third and fourth word (words 2 and 3) contain the date and time the file was created. The time is expressed in the old internal time stamp format, that is: numeric fields for year, month, day, hour, minute, and second.

Words four and five contain the date and time (also expressed in the standard time stamp format) the file was last deaccessed after being accessed for a write operation.

The seventh and eighth words (words 6 and 7) contain the date and time of the last file access.

Word eight contains the file type, the access rights, and the file's owner, as follows:

Bits 0...1: File Type

0 Ordinary file.
1 Not used.
2 Not used.
3 Directory

Bits 2...5: Access Rights

2 Read by owner NOT allowed.
(Default: ALLOWED.)
3 Overwrite, delete, rename by owner NOT allowed.
(Default: ALLOWED.)
4 Read by other users NOT allowed.
(Default: ALLOWED.)
5 Overwrite, delete, rename by other users allowed.
(Default: NOT ALLOWED.)

Bits 6...15: User ID of File's Owner

0 Owned by "no user".

The tenth and eleventh words (words 9 and 10) indicate the format of data in the file. (Optional; 0 if not set.) SesameDefs.Pas defines the currently used values.

Words 11 through 51 contain the partial file name (includes all directories, omits the partition and device specification). Partition and device specifications are not required since files cannot cross partition boundaries; the file must be in the same partition as its FIB, the FIB in the same partition as the PIB, and the PIB in the device's DIB. Note that the file name is a string; the low byte of the first word of a string always specifies the length of the string. While pathnames can be up to 255 characters, each component of the pathname can have only 80 characters, except filenames which can have only 25 characters.

Figure 6 is a graphical representation of the file system data portion of the FIB.

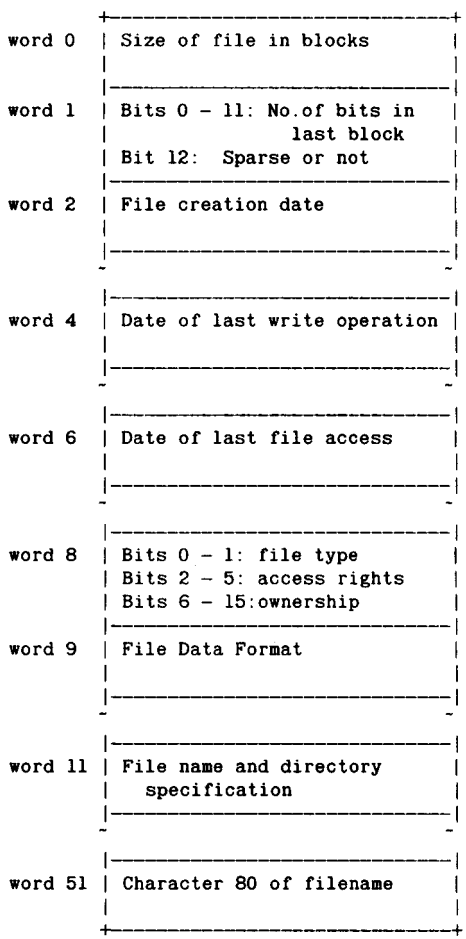
5.4.2. Random index

The random index provides a list of the disk addresses of the blocks that form the segment or file. It consists of three parts: the direct index; the indirect index; and the double indirect index.

The direct index is contained within the FIB itself (FIB words 52 through 179) and references logical disk addresses of the first 64 blocks of a segment or file (relative logical blocks 0 through 63).

The indirect index is also contained within the FIB (FIB words 180 through 243), but the entries do not point to blocks within the segment or file. Rather, the indirect index entries point to 32 blocks that each contain 128 disk addresses of blocks in the segment or file. Thus, these entries form a one-level addressing

Figure 6: Graphic Representation of the FIB



scheme. The indirect index can address 4096 blocks of a segment or file (relative logical blocks 64 through 4160, since the direct index addresses 0 through 63).

Like the indirect index, the double indirect index is contained within the FIB (FIB words 244 through 247). The indirect index points to blocks which point to blocks in the segment or file. The double indirect index entries contain disk addresses of two blocks. Each of these blocks point to 128 other blocks that each contain 128 disk addresses of blocks in the segment or file. Thus, these entries form a two-level addressing scheme. In theory, the double indirect index can address 32768 blocks of a segment or file (relative logical blocks 4161 through 36929). However, 32767 is the largest relative LBN possible. Furthermore, an overhead of 291 blocks is required to address the theoretical maximum. Therefore, the maximum size of a single segment or file in a partition is governed by the size limits of the partition and the number of overhead blocks required to address the blocks of the segment or file.

The blocks in the random index have negative file logical block numbers as follows:

Direct index: block -1 (within the FIB)
Points to relative logical blocks 0 through 63.

Indirect index: blocks -4 through -35
Points to relative logical blocks 64 through 4160

Double indirect index: blocks -2 and -3
Block -2 points to blocks -36 through -163 (blocks -36 through -163 point to relative logical blocks 4161 through 20545)

Block -3 points to blocks -164 through -257 (blocks -164 through -257 point to relative logical blocks 20546 through 32767, the theoretical maximum)

5.4.3. Type of segment

This word specifies whether a segment is permanent, temporary, or bad. A permanent segment persists until explicitly destroyed. Temporary segments (for example, segments used for swapping) exist only while the process that creates the temporary segment exists. Bad segments are malformed segments and are not readable.

5.4.4. Number of blocks in use

Since a file can be sparse (block n may exist when block n-1 has never been allocated), this word specifies the number of blocks that are actually in use by the file.

5.4.5. File relative LBN of largest block

This word specifies the file relative logical block number of the largest block allocated to the file. Remember that since files can be sparse, this word does not specify the total number of blocks in use by the file, refer to Section 5.4.4.

5.4.6. Last block address

This double word specifies the Logical Block Number of the last block allocated to the file.

5.4.7. File relative LBN of last pointer block

This word specifies the file relative logical block number of the file's pointer block with the largest absolute value (the smallest or most negative number).

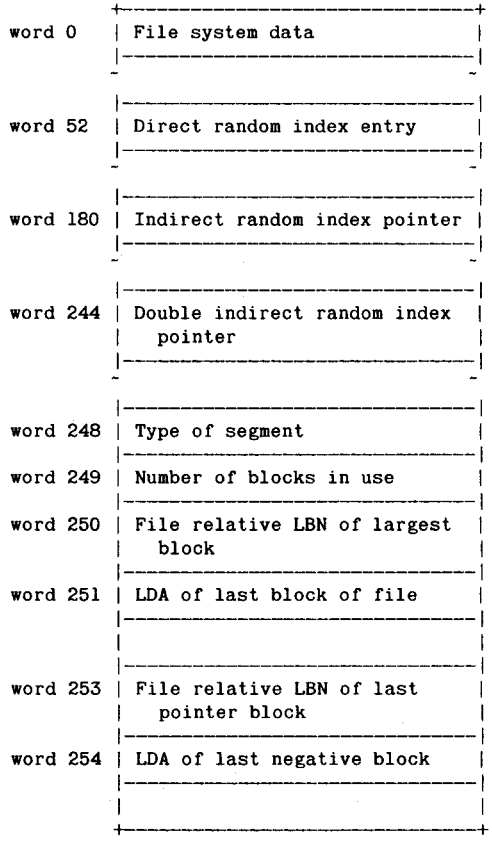
5.4.8. Last pointer block address

This double word specifies the logical disk address of the last pointer block for the file.

5.4.9. File information block layout

The layout is shown in Figure 7.

Figure 7: FIB Layout



6. File Formats

This section details the format of Segment (.Seg) and Directory files.

6.1. Segment Files

A segment file is produced when a program is compiled by the Pascal or FORTRAN compiler. The segment file is used primarily to hold the actual QCodes which the PERQ executes. In addition to the QCodes, the segment file includes information used by the linker, loader and debugger. By default, segment files use .Seg as the extension.

Segment files must conform to certain formatting conventions. A segment file consists of 512 byte blocks organized in five groups: a header block; one or more code blocks; a routine dictionary; an import list; and a routine name list.

The header block contains information about the size and contents of the module. The routine dictionary contains information necessary to the execution of routines. The import list contains module names and file names of imported modules. The routine name list contains names for the routines defined in the module.

6.1.1. Header block

The first portion of the segment file contains 14 fields, as shown in Figure 8 (refer to Section 6.1.5 for precise sizes of the fields).

The first field is a byte containing flags set by the compiler. Bit 0, the least significant bit, indicates whether the segment is a program or module. (A program is a special instance of a module, that includes a main body.) If ON, the segment is a

program. Bit 1, if ON, indicates that the routine names (only) contained in the segment file are 14 characters long. If bit 1 is OFF, the names are 8 characters long. Bit 2, if ON, indicates there is symbolic debugging. Bit 3, if ON, indicates that equal optimize is selected. If Bit 4 is ON, and bit 1 is ON, it indicates that the routine names are 30 characters long (rather than the 8 or 14-character long names indicated by bit 1 alone). The remaining 3 bits are reserved for future use.

The high-order byte of this word contains the version number of the QCodes (the compiler generates the version number).

The next field (starting on the next word) contains the name of the module (as it appears in the source file). Module names are currently unique to 8 characters. The name of the source file from which the segment file was generated follows the module name. The filename string contains the full pathname of the source file from which the segment file was generated.

The fifth field gives the number of imported segments. This is followed by the import block number which is the number of the block containing the import list (described below) and the size of the global data block (referred to as the GDB in the document "Pascal/C Machine Reference" in the *Accent Microprogramming Manual*. (With no imported segments, the import block number is undefined.) The "version" and "copyright" fields follow. These strings may be specified using compiler switches or comments in the source file. The next field indicates the language of the source file.

The next three fields are block numbers of the unresolved reference information (described in "Pre-segment Files" below), the routine descriptor information and the diagnostic information, respectively. If these blocks do not exist, these block numbers have the value zero. The last 3 fields are defined for internal use,

and the remainder of the header block is reserved for future use.

The first two words of the second block (block 1) complete the header information. Word 0 points to the routine dictionary that follows the code. The value in word 0 is offset (in words) from the beginning of block 1 to the first word of the routine dictionary. Word 1 contains the number of routines in the segment.

6.1.2. Code blocksD

The second entity is the code portion of the segment file (QCodes). The QCodes start with the third word (word 2) of the first block. The QCodes for each routine follow those of the previous routine on the next word boundary. The code section of the segment file may occupy several consecutive blocks.

The routine dictionary is aligned on the next quad-word boundary following the end of the QCodes. The dictionary contains an entry for each routine in the module and is padded to the end of the current block. A more complete description of the routine dictionary, as well as the format of an entry in the routine dictionary, may be found in the document "Pascal/C Machine Reference" in the *Accent Microprogramming Manual.*

6.1.3. Import list

Imports refer to segments that are external to the current module. The import list begins in the block following the routine dictionary (procedures, functions and exceptions in Pascal). The number of this block is given in the header block. The import list can take any number of blocks (including zero for no imports). The import list is ordered by segment number (the compiler-generated ISN [internal segment number]). Each entry in the import list consists of the 8 character module name followed by the name of the source file (for example, foo {from} foo.pas).

Figure 8: Header Block 0

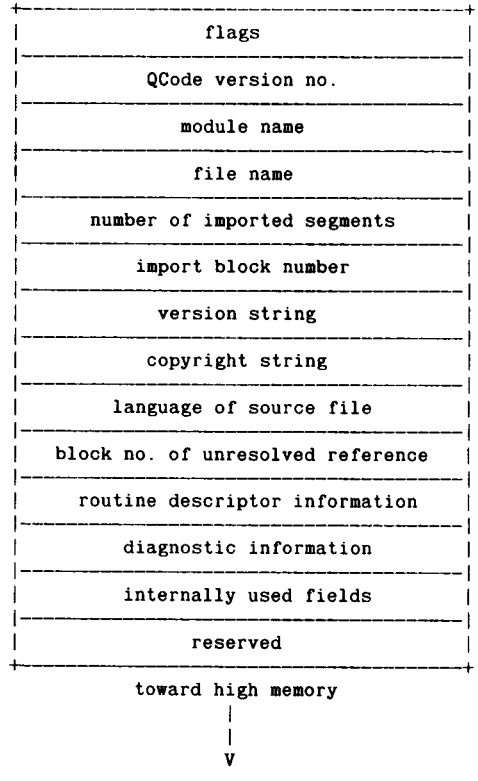
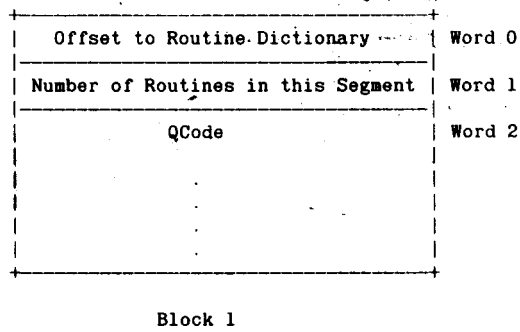


Figure 9: Code Block



Entries are word aligned and have constant length. Enough room is reserved for a 100 character file name (51 words). If a module entry needs less than the allotted space, it is padded with blanks.

6.1.4. Routine name list

After the list of imported segments, the compiler provides the names of the routines defined in the module. The first routine name begins immediately after the last import entry. The entries in the routine name list are 8, 14 or 30 characters long, are word-aligned, and are ordered by routine number (the order of their appearance in the source). Routine names may or may not fit entirely in the same block as the end of the import list.

6.1.5. Field definitions

Users who need to manipulate segment files are encouraged to import the declarations found in the file "SegDefs.Pas". The necessary definitions to access portions of segment and run files may be found in this file.

6.2. Directory Files

Directories can be nested to form a hierarchical, multi- directory structure. You can construct directory hierarchies to structure files in whatever manner is convenient.

A directory entry is a 16-word packed record that points to the FIB of a file within the directory. Module SegDefs.Pas defines the format of a directory entry.

The first word of each directory entry specifies whether or not the entry is in use; bit 0 is set if the directory entry is valid. Bit 1 is reserved. Bits 2 through 15 are reserved for future use. Note that reserved bits must be zero, but programs reading reserved bits should not assume that the bits are in fact zero.

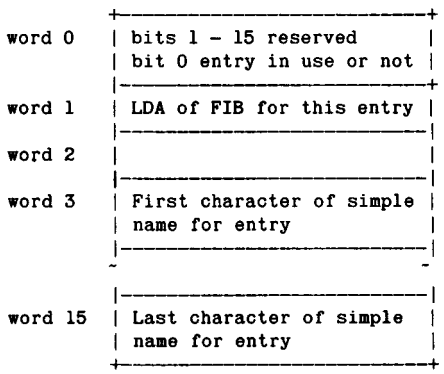
The second and third words of a directory entry contain the logical disk address of the FIB of this entry.

The remaining 13 words contain the file name portion of a file specification. Since a directory can only contain entries for files within the partition, and thus device, where the directory itself is located, these words do not include the device, partition, and directory portions of a file specification. A file name that omits the device, partition, and directory portions is known as a SimpleName. Note that the file name is a string; the low byte of the first word of a string always specifies the length of the string. Therefore, a simple file name cannot exceed 25 characters.

Figure 10 is a graphical representation of a directory entry.

Directory entries are hash coded by file name to decrease the lookup time. The hash function specifies the block of the directory in which the file name and the address of the file's FIB are written. When a block of the directory is full (it contains 16

Figure 10: Directory Entry



directory entries), successive or overflow directory entries destined for that block are written in block n + 31.

Since the hash function specifies the block of the directory to contain the entry, directories often contain unallocated blocks between allocated blocks.

PERQ

CUSTOMER REPORT FORM
FOR SOFTWARE PROBLEMS OR SUGGESTIONS

Equipment Information

Machine: ___ PERQ
___ PERQZ (portrait)
___ PERQZ (landscape)

Disk: ___ 14" Shugart
___ 8" Micropolis
___ 5.25" Micropolis

Processor: ___ 4K CPU
___ 16K CPU

Memory: ___ 1 Megabyte
___ 2 Megabyte

Operating System Name and Version No. _____

Program, Sub-system, Module, or Manual _____

___ Problem - Problem appears to be:
___ Random ___ Caused by sequence of events (please describe below)

___ Suggestion for enhancement

Description of problem (including system activity at the time of the problem) or suggestion for enhancement.

_____

_____

_____

_____

_____

_____

_____

_____

Date _____

Name _____

Title _____

Company _____

Address _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 112 PITTSBURGH, PA

POSTAGE WILL BE PAID BY ADDRESSEE

PERQ Systems Corporation
2600 Liberty Avenue
Box 2600
Pittsburgh, PA 15230

PERQ

READER'S COMMENTS

ON DOCUMENTATION

PERQ Systems Corporation wishes to provide you with documents that are accurate, complete, and clear. You can help us provide you with excellent documentation by taking a few minutes to report any inaccuracies you discover, improvements you feel are needed, or features you find especially helpful.

Name of document _____

If part of larger manual, name of manual _____

Date of document _____ Section No., if applicable _____

Comments:

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

_____

Date _____

Name _____

Title _____

Company _____

Address _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 112 PITTSBURGH, PA

POSTAGE WILL BE PAID BY ADDRESSEE

PERQ Systems Corporation
2600 Liberty Avenue
Box 2600
Pittsburgh, PA 15230