```
-- Files.Mesa  Edited by Sandman on August 29, 1977  8:44 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  AltoFileDefs: FROM "altofiledefs",
  BFSDefs: FROM "bfsdefs",
  BootDefs: FROM "bootdefs",
  DirectoryDefs: FROM "directorydefs",
  DiskKDDefs: FROM "diskkddefs",
  InlineDefs: FROM "inlinedefs",
  MiscDefs: FROM "miscdefs",
  SegmentDefs: FROM "segmentdefs";

DEFINITIONS FROM AltoFileDefs, SegmentDefs;

Files: PROGRAM
  IMPORTS BFSDefs, BootDefs, DirectoryDefs, DiskKDDefs, SegmentDefs
  EXPORTS BootDefs, MiscDefs, SegmentDefs SHARES SegmentDefs = BEGIN

  FileError: PUBLIC SIGNAL [file:FileHandle] = CODE;
  FileNameError: PUBLIC SIGNAL [name:STRING] = CODE;
  FileAccessError: PUBLIC SIGNAL [file:FileHandle] = CODE;

  NullFileObject: FileObject = Object [ TRUE,FALSE,
    File [ FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,0,0,0,
    FP[SN[1,0,1,17777B,177777B],eofDA],FA[eofDA,0,0]]];

  NewFile: PUBLIC PROCEDURE [
    name:STRING, access:AccessOptions, version:VersionOptions]
    RETURNS [file:FileHandle] =
    BEGIN OPEN InlineDefs;
    fp: FP;  old, create: BOOLEAN;
    [access,version] ← ValidateOptions[access,version];
    create ← BITAND[version,OldFileOnly]=0;
    old ← DirectoryDefs.DirectoryLookup[@fp,name,create];
    IF (old AND BITAND[version,NewFileOnly]#0)
    OR (~old AND ~create) THEN ERROR FileNameError[name];
    IF (file ← FindFile[@fp]) = NIL THEN
      BEGIN
      file ← AllocateFile[FileTable];
      file↑ ← NullFileObject;
      file.fp ← fp;
      END;
    SetFileAccess[file,access];
    RETURN
    END;

  InsertFile: PUBLIC PROCEDURE [
    fp:POINTER TO FP, access:AccessOptions]
    RETURNS [file:FileHandle] = BEGIN
    [access,] ← ValidateOptions[access,0];
    IF (file ← FindFile[fp]) = NIL THEN
      BEGIN
      file ← AllocateFile[FileTable];
      file↑ ← NullFileObject;
      file.fp ← fp↑;
      END;
    SetFileAccess[file,access];
    RETURN
    END;

  BootFile: PUBLIC PROCEDURE [access:AccessOptions]
    RETURNS [file:FileHandle] = BEGIN
    [access,] ← ValidateOptions[access,0];
    file ← AllocateFile[FileTable];
    file↑ ← NullFileObject;
    SetFileAccess[file,access];
    RETURN
    END;

  ValidateOptions: PROCEDURE [
    access:AccessOptions, version:VersionOptions]
    RETURNS [AccessOptions, VersionOptions] =
    BEGIN OPEN InlineDefs;
    IF access = DefaultAccess THEN access ← Read;
    -- IF version = DefaultVersion THEN version ← 0;
```

```
    IF BITAND[version,NewFileOnly+OldFileOnly] = NewFileOnly+OldFileOnly
    OR (BITAND[version,NewFileOnly]#0 AND BITAND[access,Append]=0)
      THEN ERROR FileAccessError[NIL];
    IF BITAND[access,Append]=0 THEN
      version ← BITOR[version,OldFileOnly];
    RETURN[access,version]
    END;

  GetFileAccess: PUBLIC PROCEDURE [file:FileHandle]
    RETURNS [AccessOptions] = BEGIN
    access: AccessOptions ← 0;
    ValidateFile[FileTable,file];
    IF file.read THEN access ← access+Read;
    IF file.write THEN access ← access+Write;
    IF file.append THEN access ← access+Append;
    RETURN[access]
    END;

  SetFileAccess: PUBLIC PROCEDURE [file:FileHandle, access:AccessOptions] =
    BEGIN OPEN InlineDefs;
    ValidateFile[FileTable,file];
    IF access = DefaultAccess THEN access ← Read;
    file.read ← file.read OR BITAND[access,Read]#0;
    file.write ← file.write OR BITAND[access,Write]#0;
    file.append ← file.append OR BITAND[access,Append]#0;
    RETURN
    END;

  LockFile: PUBLIC PROCEDURE [file:FileHandle] =
    BEGIN OPEN file;
    ValidateFile[FileTable,file];
    IF lock = MaxLocks THEN
      ERROR FileError[file];
    lock ← lock+1;
    RETURN
    END;

  UnlockFile: PUBLIC PROCEDURE [file:FileHandle] =
    BEGIN OPEN file;
    ValidateFile[FileTable,file];
    IF lock = 0 THEN
      ERROR FileError[file];
    lock ← lock-1;
    RETURN
    END;

  ReleaseFile: PUBLIC PROCEDURE [file:FileHandle] =
    BEGIN
    [] ← PurgeFile[file];
    DiskKDDefs.UpdateDiskKD[];
    RETURN
    END;

  DestroyFile: PUBLIC PROCEDURE [file:FileHandle] =
    BEGIN
    seg: DataSegmentHandle;
    fp: FP ← file.fp;
    IF ~PurgeFile[file] OR ~DirectoryDefs.DirectoryPurgeFP[@fp]
      THEN ERROR FileError[file];
    seg ← NewDataSegment[DefaultBase,1];
    BFSDefs.DeletePages [
      DataSegmentAddress[seg],@fp,fp.leaderDA,0
      ! UNWIND => DeleteDataSegment[seg]];
    DeleteDataSegment[seg];
    DiskKDDefs.UpdateDiskKD[];
    RETURN
    END;

  PurgeFile: PROCEDURE [file:FileHandle]
    RETURNS [BOOLEAN] =
    BEGIN OPEN file;
    ValidateFile[FileTable,file];
    IF segcount # 0 THEN ERROR FileError[file];
    IF lock # 0 THEN RETURN[FALSE];
    CloseFile[file];
    LiberateFile[FileTable,file];
```

```
    RETURN[TRUE]
    END;
```

```
-- File length stuff

NormalizeFileIndex: PUBLIC PROCEDURE [
  page:PageNumber, byte:CARDINAL]
  RETURNS [PageNumber, CARDINAL] =
  BEGIN OPEN InlineDefs, AltoDefs;
  delta: PageNumber = byte/BytesPerPage;
  byte ← BITAND[byte,BytesPerPage-1];
  page ← page+delta;
  RETURN[page,byte]
  END;

RoundFileIndex: PUBLIC PROCEDURE [
  page:PageNumber, byte:CARDINAL]
  RETURNS [PageNumber, CARDINAL] =
  BEGIN
  IF byte = AltoDefs.BytesPerPage THEN
    BEGIN  byte ← 0;
    page ← page+1;
    END;
  RETURN[page,byte]
  END;

TruncateFileIndex: PUBLIC PROCEDURE [
  page:PageNumber, byte:CARDINAL]
  RETURNS [PageNumber, CARDINAL] = BEGIN
  IF page > 0 AND byte = 0 THEN
    BEGIN  page ← page-1;
    byte ← AltoDefs.BytesPerPage END;
  RETURN[page,byte]
  END;

GetEndOfFile: PUBLIC PROCEDURE [file:FileHandle]
  RETURNS [page:PageNumber, byte:CARDINAL] =
  BEGIN OPEN file;
  cfa: CFA;  seg: DataSegmentHandle;
  ValidateFile[FileTable,file];
  IF ~open THEN OpenFile[file];
  IF ~lengthvalid THEN
    BEGIN
    seg ← NewDataSegment[DefaultBase,1];
    cfa ← CFA[fp,eof];
    [] ← JumpToPage [
      @cfa,AltoDefs.MaxFilePage,DataSegmentAddress[seg]
      ! UNWIND => DeleteDataSegment[seg]];
    DeleteDataSegment[seg];
    UpdateFileLength[file,@cfa.fa];
    END;
  [page,byte] ← TruncateFileIndex[eof.page,eof.byte];
  RETURN
  END;

SetEndOfFile: PUBLIC PROCEDURE [
  file:FileHandle, page:PageNumber, byte:CARDINAL] =
  BEGIN  da: vDA;
  cfa: CFA ← CFA[file.fp,file.eof];
  seg: DataSegmentHandle = NewDataSegment[DefaultBase,1];
  buf: POINTER = DataSegmentAddress[seg];
  BEGIN ENABLE UNWIND => DeleteDataSegment[seg];
    ValidateFile[FileTable,file];
    IF ~file.open THEN OpenFile[file];
    [page,byte] ← NormalizeFileIndex[page,byte];
    [page,byte] ← RoundFileIndex[page,byte];
    IF page=0 THEN ERROR FileError[file];
    [,da] ← JumpToPage[@cfa,page,buf];
    SELECT cfa.fa.page FROM
      = page =>
        SELECT cfa.fa.byte FROM
          > byte => IF ~file.write
            THEN ERROR FileAccessError[file];
          < byte => IF ~file.append
            THEN ERROR FileAccessError[file];
          ENDCASE =>
            IF da=eofDA THEN GO TO done
            ELSE ERROR FileError[file];
      < page =>
```

```
          BEGIN  da ← eofDA;
          IF ~file.append THEN
            ERROR FileAccessError[file];
          END;
        ENDCASE => ERROR FileError[file];
      BFSDefs.CreatePages[buf,@cfa,page,byte];
      IF da # eofDA THEN BFSDefs.DeletePages [
        buf,@cfa.fp,da,page+1];
      EXITS
        done => NULL;
      END;
    DeleteDataSegment[seg];
    UpdateFileLength[file,@cfa.fa];
    RETURN
    END;

  UpdateFileLength: PUBLIC PROCEDURE [
    file:FileHandle, fa:POINTER TO FA] =
    BEGIN OPEN file;
    ValidateFile[FileTable,file];
    IF eof # fa↑ THEN
      BEGIN  eof ← fa↑;
      lengthchanged ← TRUE;
      END;
    lengthvalid ← TRUE;
    RETURN
    END;
```

```
-- Open and Close (leader page stuff)

MakePageZeroSeg: PROCEDURE [file:FileHandle]
  RETURNS [seg:FileSegmentHandle] = BEGIN
  temp: FileHandle = BootFile[Read+Write];
  temp.fp ← file.fp;  temp.open ← TRUE;
  seg ← NewFileSegment[temp,0,1,Read+Write
    ! UNWIND => ReleaseFile[temp]];
  SwapIn[seg ! UNWIND =>
    DeletePageZeroSeg[seg]];
  RETURN
  END;

DeletePageZeroSeg: PROCEDURE [seg:FileSegmentHandle] =
  BEGIN
  IF seg.swappedin THEN Unlock[seg];
  DeleteFileSegment[seg];
  RETURN
  END;

SecondsClock: POINTER TO TIME = LOOPHOLE[572B];

DAYTIME: PUBLIC PROCEDURE RETURNS [TIME] =
  BEGIN
  RETURN[SecondsClock↑]
  END;

OpenFile: PUBLIC PROCEDURE [file:FileHandle] =
  BEGIN OPEN file;
  ld: POINTER TO LD;
  seg: FileSegmentHandle;
  ValidateFile[FileTable,file];
  IF ~open THEN
    BEGIN
    seg ← MakePageZeroSeg[file];
    ld ← FileSegmentAddress[seg];
    eof ← ld.eofFA;
    -- PATCH for OS versions 5 & up
    IF eof.da = 0 THEN eof.da ← eofDA;
    IF read THEN ld.read ← DAYTIME[];
    IF write OR append THEN ld.written ← DAYTIME[];
    DeletePageZeroSeg[seg];
    open ← TRUE;
    END;
  RETURN
  END;

CloseFile: PUBLIC PROCEDURE [file:FileHandle] =
  BEGIN OPEN file;
  ld: POINTER TO LD;
  seg: FileSegmentHandle;
  ValidateFile[FileTable,file];
  IF swapcount # 0 THEN
    SIGNAL FileError[file];
  IF open AND lengthchanged THEN
    BEGIN
    seg ← MakePageZeroSeg[file];
    ld ← FileSegmentAddress[seg];
    ld.eofFA ← eof;
    DeletePageZeroSeg[seg];
    lengthchanged ← FALSE;
    END;
  open ← FALSE;
  RETURN
  END;
```

```
  -- Managing File Objects

  TableHandle: TYPE = BootDefs.TableHandle;
  FileObjects: BootDefs.Table ← [SIZE[FileObject],NIL];
  FileTable: TableHandle = @FileObjects;

  GetFileTable: PUBLIC PROCEDURE RETURNS [TableHandle] =
    BEGIN RETURN[FileTable] END;

  -- Procedures are bound before this initialization code is run
  AllocateFile: PROCEDURE [TableHandle] RETURNS [FileHandle] ← LOOPHOLE[BootDefs.AllocateObject];
  ValidateFile: PROCEDURE [TableHandle,FileHandle] ← LOOPHOLE[BootDefs.ValidateObject];
  LiberateFile: PROCEDURE [TableHandle,FileHandle] ← LOOPHOLE[BootDefs.LiberateObject];

  EnumerateFiles: PUBLIC PROCEDURE [
    proc: PROCEDURE [FileHandle] RETURNS [BOOLEAN]]
    RETURNS [FileHandle] =
    BEGIN RETURN[LOOPHOLE[
      BootDefs.EnumerateObjects[FileTable,LOOPHOLE[proc]]]]
    END;

  FindFile: PUBLIC PROCEDURE [fp:POINTER TO FP] RETURNS [FileHandle] =
    BEGIN
    MatchFP: PROCEDURE [file:FileHandle] RETURNS [BOOLEAN] =
      BEGIN
      RETURN [
        file.fp.serial = fp.serial
        AND file.fp.leaderDA = fp.leaderDA]
      END;
    RETURN[EnumerateFiles[MatchFP]]
    END;

  GetFileFP: PUBLIC PROCEDURE [file:FileHandle, fp:POINTER TO FP] =
    BEGIN
    ValidateFile[FileTable,file];
    fp↑ ← file.fp;
    RETURN
    END;

  -- Need to start worrying about Sys.Log.
  -- Cleanup procedure should reset LengthValid

  END.
```