```
-- Files.Mesa  Edited by Sandman on May 12, 1978  4:40 PM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [BytesPerPage, MaxFilePage],
  AltoFileDefs: FROM "altofiledefs" USING [
    CFA, eofDA, FA, FP, LD, SN, TIME, vDA],
  BFSDefs: FROM "bfsdefs" USING [CreatePages, DeletePages],
  BootDefs: FROM "bootdefs" USING [
    AllocateObject, EnumerateObjects, LiberateObject, ValidateObject],
  DirectoryDefs: FROM "directorydefs" USING [
    DirectoryLookup, DirectoryPurgeFP],
  DiskKDDefs: FROM "diskkddefs" USING [UpdateDiskKD],
  InlineDefs: FROM "inlinedefs" USING [BITAND, BITOR],
  MiscDefs: FROM "miscdefs",
  NucleusDefs: FROM "nucleusdefs",
  SegmentDefs: FROM "segmentdefs" USING [
    AccessOptions, Append, DataSegmentAddress, DataSegmentHandle,
    DefaultAccess, DefaultBase, DeleteDataSegment, DeleteFileSegment,
    FileHandle, FileObject, FileSegmentAddress, FileSegmentHandle, JumpToPage,
    LengthHandle, LengthObject, MaxLocks, NewDataSegment, NewFileOnly,
    NewFileSegment, Object, OldFileOnly, PageNumber, Read, SwapIn, Unlock,
    VersionOptions, Write];

DEFINITIONS FROM AltoFileDefs, SegmentDefs;

Files: PROGRAM
  IMPORTS BFSDefs, BootDefs, DirectoryDefs, DiskKDDefs, SegmentDefs
  EXPORTS BootDefs, MiscDefs, NucleusDefs, 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 [FALSE,
    file [FALSE,FALSE,FALSE,FALSE,FALSE,FALSE,0,FALSE,0,0,0,
    FP[SN[1,0,1,17777B,177777B],eofDA]]];

  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[];
      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[];
      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[];
```

```
    file↑ ← NullFileObject;
    SetFileAccess[file,access];
    RETURN
    END;

  InsertFileLength: PUBLIC PROCEDURE [file: FileHandle, fa: POINTER TO FA] =
    BEGIN
    lh: LengthHandle ← AllocateLength[file];
    [da: lh.da, page: lh.page, byte: lh.byte] ← fa↑;
    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[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[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[file];
    IF lock = MaxLocks THEN ERROR FileError[file];
    lock ← lock+1;
    RETURN
    END;

  UnlockFile: PUBLIC PROCEDURE [file:FileHandle] =
    BEGIN OPEN file;
    ValidateFile[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[file];
  IF segcount # 0 THEN ERROR FileError[file];
  IF lock # 0 THEN RETURN[FALSE];
  CloseFile[file];
  IF file.length THEN LiberateLength[FindLength[file]];
  LiberateFile[file];
  RETURN[TRUE]
  END;


-- File length stuff

NormalizeFileIndex: PUBLIC PROCEDURE [page:PageNumber, byte:CARDINAL]
  RETURNS [PageNumber, CARDINAL] =
  BEGIN
  page ← page + byte/AltoDefs.BytesPerPage;
  byte ← byte MOD AltoDefs.BytesPerPage;
  RETURN[page,byte]
  END;

RoundFileIndex: PUBLIC PROCEDURE [page:PageNumber, byte:CARDINAL]
  RETURNS [PageNumber, CARDINAL] =
  BEGIN
  [page, byte] ← NormalizeFileIndex[page, byte];
  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
  [page, byte] ← NormalizeFileIndex[page, byte];
  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;
  lh: LengthHandle;
  cfa: CFA;  seg: DataSegmentHandle;
  ValidateFile[file];
  lh ← AllocateLength[file];
  IF ~lengthvalid THEN
    BEGIN
    IF ~open THEN OpenFile[file];
    IF lh.da = eofDA THEN GetLengthHint[lh];
    seg ← NewDataSegment[DefaultBase,1];
    cfa ← CFA[fp,FA[lh.da, lh.page, lh.byte]];
    [] ← JumpToPage [@cfa,AltoDefs.MaxFilePage,DataSegmentAddress[seg]
      ! UNWIND => DeleteDataSegment[seg]];
    DeleteDataSegment[seg];
    ChangeFileLength[@cfa.fa,lh];
    END;
  [page, byte] ← TruncateFileIndex[lh.page,lh.byte];
  RETURN
  END;

SetEndOfFile: PUBLIC PROCEDURE [
  file:FileHandle, page:PageNumber, byte:CARDINAL] =
  BEGIN  da: vDA;
  lh: LengthHandle;
  cfa: CFA;
  seg: DataSegmentHandle = NewDataSegment[DefaultBase,1];
  buf: POINTER = DataSegmentAddress[seg];
  BEGIN ENABLE UNWIND => DeleteDataSegment[seg];
```

```
    ValidateFile[file];
    lh ← AllocateLength[file];
    IF ~file.open THEN OpenFile[file];
    IF lh.da = eofDA THEN GetLengthHint[lh];
    cfa ← CFA[fp: file.fp, fa: [da: lh.da, page: lh.page, byte: lh.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];
  ChangeFileLength[@cfa.fa,lh];
  RETURN
  END;

ChangeFileLength: PROCEDURE [fa:POINTER TO FA, lh: LengthHandle] =
  BEGIN
  currentlength: FA ← [da: lh.da, page: lh.page, byte: lh.byte];
  IF currentlength # fa↑ THEN
    BEGIN
    lh.file.lengthchanged ← TRUE;
    [da: lh.da, page: lh.page, byte: lh.byte] ← fa↑;
    END;
  lh.file.lengthvalid ← TRUE;
  RETURN
  END;

SetFileLength, UpdateFileLength: PUBLIC PROCEDURE [
  file:FileHandle, fa:POINTER TO FA] =
  BEGIN OPEN file;
  lh: LengthHandle;
  ValidateFile[file];
  lh ← AllocateLength[file];
  ChangeFileLength[fa, lh];
  RETURN
  END;

GetFileLength: PUBLIC PROCEDURE [file:FileHandle, fa:POINTER TO FA] =
  BEGIN
  lh: LengthHandle;
  ValidateFile[file];
  IF ~file.length THEN
    BEGIN
    fa↑ ← [da: eofDA, page: 0, byte: 0];
    RETURN
    END;
  lh ← FindLength[file];
  fa↑ ← [da: lh.da, page: lh.page, byte: lh.byte];
  RETURN
  END;


-- Open and Close (leader page stuff)

MakePageZeroSeg: PROCEDURE [file:FileHandle, access: AccessOptions]
 RETURNS [seg:FileSegmentHandle] =
  BEGIN
```

```
    temp: FileHandle = BootFile[access];
    temp.fp ← file.fp;  temp.open ← TRUE;
    seg ← NewFileSegment[temp, 0, 1, access
      ! 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;
    lh: LengthHandle;
    ld: POINTER TO LD;
    seg: FileSegmentHandle;
    ValidateFile[file];
    IF ~open THEN
      BEGIN
      seg ← MakePageZeroSeg[file, Read+Write];
      ld ← FileSegmentAddress[seg];
      IF length THEN
        BEGIN
        lh ← FindLength[file];
        [page: lh.page, byte: lh.byte, da: lh.da] ← ld.eofFA;
        -- PATCH for OS versions 5 & up
        IF lh.da = 0 THEN lh.da ← eofDA;
        END;
      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;
    lh: LengthHandle;
    seg: FileSegmentHandle;
    ValidateFile[file];
    IF swapcount # 0 THEN
      SIGNAL FileError[file];
    IF open AND length AND lengthchanged THEN
      BEGIN
      seg ← MakePageZeroSeg[file, Read+Write];
      ld ← FileSegmentAddress[seg];
      lh ← FindLength[file];
      ld.eofFA ← FA[byte: lh.byte, page: lh.page, da: lh.da];
      DeletePageZeroSeg[seg];
      lengthchanged ← FALSE;
      END;
    open ← FALSE;
    RETURN
    END;

  GetLengthHint: PROCEDURE [lh: LengthHandle] =
    BEGIN
    ld: POINTER TO LD;
    seg: FileSegmentHandle;
    seg ← MakePageZeroSeg[lh.file, Read];
    ld ← FileSegmentAddress[seg];
    [page: lh.page, byte: lh.byte, da: lh.da] ← ld.eofFA;
    -- PATCH for OS versions 5 & up
```

```
    IF lh.da = 0 THEN lh.da ← eofDA;
    DeletePageZeroSeg[seg];
    RETURN
    END;


  -- Managing File Objects

  -- Procedures are bound before this initialization code is run
  ValidateFile: PROCEDURE [FileHandle] = LOOPHOLE[BootDefs.ValidateObject];
  LiberateFile: PROCEDURE [FileHandle] = LOOPHOLE[BootDefs.LiberateObject];
  AllocateFile: PROCEDURE RETURNS [FileHandle] =
    BEGIN
    RETURN[LOOPHOLE[BootDefs.AllocateObject[SIZE[file Object]]]]
    END;

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

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

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


  -- Managing Length Objects

  -- Procedures are bound before this initialization code is run
  ValidateLength: PROCEDURE [LengthHandle] =
    LOOPHOLE[BootDefs.ValidateObject];
  LiberateLength: PROCEDURE [LengthHandle] =
    LOOPHOLE[BootDefs.LiberateObject];

  FindLength: PROCEDURE [file: FileHandle] RETURNS [LengthHandle] =
    BEGIN
    FindLength: PROCEDURE [lh: LengthHandle] RETURNS [BOOLEAN] =
      BEGIN RETURN[lh.file = file] END;
    RETURN[LOOPHOLE[BootDefs.EnumerateObjects[length, LOOPHOLE[FindLength]]]]
    END;

  AllocateLength: PROCEDURE [file:FileHandle] RETURNS [lh:LengthHandle] =
    BEGIN
    IF file.length THEN RETURN[FindLength[file]];
    lh ← LOOPHOLE[BootDefs.AllocateObject[SIZE[LengthObject]]];
    lh↑ ← [FALSE, length[0,0,0,file,eofDA]];
    file.length ← TRUE;
    file.lengthvalid ← FALSE;
    RETURN
    END;

  END.
```