```
-- Segments.Mesa  Edited by Sandman on August 16, 1977  8:41 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  AltoFileDefs: FROM "altofiledefs",
  BootDefs: FROM "bootdefs",
  DiskDefs: FROM "diskdefs",
  InlineDefs: FROM "inlinedefs",
  ProcessDefs: FROM "processdefs",
  SegmentDefs: FROM "segmentdefs",
  SystemDefs: FROM "systemdefs";

DEFINITIONS FROM AltoFileDefs, BootDefs, SegmentDefs;

Segments: PROGRAM
  IMPORTS BootDefs, DiskDefs, SegmentDefs
  EXPORTS BootDefs, SegmentDefs, SystemDefs SHARES SegmentDefs = BEGIN

  InvalidSegmentSize: PUBLIC SIGNAL [pages:PageCount] = CODE;

  NewFileSegment: PUBLIC PROCEDURE [
    file:FileHandle, base:PageNumber, pages:PageCount, access:AccessOptions]
    RETURNS [seg:FileSegmentHandle] =
    BEGIN OPEN InlineDefs;
    IF access = DefaultAccess THEN access ← Read;
    IF file.segcount = MaxSegs THEN ERROR FileError[file];
    IF BITAND[access,Append]#0 THEN ERROR FileAccessError[file];
    seg ← AllocateFileSegment[FileSegmentTable];
    BEGIN ENABLE UNWIND =>
        LiberateFileSegment[FileSegmentTable,seg];
      IF base = DefaultBase THEN base ← 1;
      IF pages = DefaultPages THEN
        pages ← GetEndOfFile[file].page-base+1;
      IF pages ~IN (0..AltoDefs.MaxVMPage+1] THEN
        ERROR InvalidSegmentSize[pages];
      SetFileAccess[file,access];
      END;
    seg↑ ← Object [ TRUE, FALSE,
      Segment [ FALSE, BITAND[access,Read]#0,
      BITAND[access,Write]#0, other, 0, pages,
      0, file, base, FileHint[eofDA,0]]];
    file.segcount ← file.segcount+1;
    RETURN
    END;

  BootFileSegment: PUBLIC PROCEDURE [
    file:FileHandle, base:PageNumber,
    pages:PageCount, access:AccessOptions, addr:POINTER]
    RETURNS [seg:FileSegmentHandle] = BEGIN
    vm: PageNumber;
    seg ← NewFileSegment[file,base,pages,access];
    IF addr # NIL THEN
      BEGIN
      seg.VMpage ← vm ← PageFromAddress[addr];
      -- DisableInterrupts[];
      FOR vm IN [vm..vm+pages) DO
        IF PageFree[vm] THEN ERROR;
        ENDLOOP;
      seg.swappedin ← TRUE;
      seg.lock ← seg.lock+1;
      file.swapcount ← file.swapcount+1;
      -- EnableInterrupts[];
      END;
    RETURN
    END;

  DeleteFileSegment: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
    BEGIN
    file: FileHandle ← seg.file;
    ValidateFileSegment[FileSegmentTable,seg];
    SwapOut[seg];
    LiberateFileSegment[FileSegmentTable,seg];
    file.segcount ← file.segcount-1;
    IF file.segcount = 0 THEN ReleaseFile[file];
    RETURN
    END;
```

```
FileSegmentAddress: PUBLIC PROCEDURE [seg:FileSegmentHandle]
  RETURNS [POINTER] = BEGIN
  IF ~seg.swappedin THEN ERROR SwapError[seg];
  RETURN[AddressFromPage[seg.VMpage]]
  END;


-- Window Segments (such as they are)

MoveFileSegment: PUBLIC PROCEDURE [
  seg:FileSegmentHandle, base:PageNumber, pages:PageCount] =
  BEGIN ValidateFileSegment[FileSegmentTable,seg];
  IF base = DefaultBase THEN base ← 1;
  IF pages = DefaultPages THEN
    pages ← GetEndOfFile[seg.file].page-base+1;
  IF pages ~IN (0..Al+oDefs.MaxVMPage+1] THEN
    ERROR InvalidSegmentSize[pages];
  SwapOut[seg];  seg.base ← base;
  seg.pages ← pages;
  RETURN
  END;

MapFileSegment: PUBLIC PROCEDURE [
  seg:FileSegmentHandle, file:FileHandle, base:PageNumber] =
  BEGIN
  wasin, waswrite: BOOLEAN;
  old: FileHandle = seg.file;
  ValidateFileSegment[FileSegmentTable,seg];
  IF ~old.read THEN ERROR FileAccessError[old];
  IF ~file.write THEN ERROR FileAccessError[file];
  IF base = DefaultBase THEN base ← 1;
  wasin ← seg.swappedin;  waswrite ← seg.write;
  IF ~wasin THEN SwapIn[seg];
  -- DisableInterrupts[];
  old.swapcount ← old.swapcount-1;
  old.segcount ← old.segcount-1;
  seg.file ← file;  seg.base ← base;
  seg.hint ← FileHint[eofDA,0];
  seg.write ← TRUE;
  file.segcount ← file.segcount+1;
  file.swapcount ← file.swapcount+1;
  -- EnableInterrupts[];
  IF wasin OR ~waswrite THEN SwapUp[seg];
  seg.write ← waswrite;
  IF ~wasin THEN
    BEGIN Unlock[seg]; SwapOut[seg] END;
  IF old.segcount=0 THEN ReleaseFile[old];
  RETURN
  END;
```

```
-- Segment Positioning

PositionSeg: PUBLIC PROCEDURE [seg:FileSegmentHandle, useseg:BOOLEAN]
  RETURNS [BOOLEAN] = BEGIN
  -- returns TRUE if it read a non-null page into the segment.
  cfa: CFA;  buf: DataSegmentHandle;  buffer: POINTER;
  IF seg.hint.da = eofDA AND seg.base > 8
  AND seg.file.segcount > 1 THEN FindSegHint[seg];
  IF seg.hint.da = eofDA OR seg.hint.page # seg.base THEN
    BEGIN
    buffer ←
      IF useseg THEN AddressFromPage[seg.VMpage]
      ELSE DataSegmentAddress[buf ← NewDataSegment[DefaultBase,1]];
    cfa.fp ← seg.file.fp;
    cfa.fa ← FA[seg.hint.da,seg.hint.page,0];
    [] ← JumpToPage[@cfa,seg.base,buffer
      ! UNWIND => IF ~useseg THEN DeleteDataSegment[buf]];
    IF ~useseg THEN DeleteDataSegment[buf];
    IF cfa.fa.page # seg.base THEN ERROR SwapError[seg];
    seg.hint ← FileHint[cfa.fa.da,cfa.fa.page];
    RETURN[useseg AND cfa.fa.byte#0];
    END;
  RETURN[FALSE]
  END;

FindSegHint: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN
  hint: FileHint ← seg.hint;
  CheckHint: PROCEDURE [other:FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF other.file = seg.file AND other.hint.da # eofDA
    AND other.hint.page IN (hint.page..seg.base] THEN
      hint ← other.hint;
    RETURN[hint.page=seg.base]
    END;
  [] ← EnumerateFileSegments[CheckHint];
  seg.hint ← hint;
  RETURN
  END;

GetFileSegmentDA: PUBLIC PROCEDURE [seg:FileSegmentHandle] RETURNS [vDA] =
  BEGIN
  [] ← PositionSeg[seg,FALSE];
  RETURN[seg.hint.da]
  END;

SetFileSegmentDA: PUBLIC PROCEDURE [seg:FileSegmentHandle, da:vDA] =
  BEGIN
  seg.hint ← FileHint[da,seg.base];
  RETURN
  END;
```

```
-- Segment Initialization

CopyDataToFileSegment: PUBLIC PROCEDURE [
  dataseg: DataSegmentHandle, fileseg: FileSegmentHandle] =
  BEGIN
  waslocked: BOOLEAN ← TRUE;
  IF dataseg.pages # fileseg.pages THEN SwapError[fileseg];
  IF fileseg.swappedin THEN
    BEGIN
    IF fileseg.lock = 0 THEN
      BEGIN
      SwapIn[fileseg];
      waslocked ← FALSE;
      END;
    InlineDefs.COPY[
      from: DataSegmentAddress[dataseg],
      to: FileSegmentAddress[fileseg],
      nwords: dataseg.pages*AltoDefs.PageSize];
    IF ~waslocked THEN Unlock[fileseg];
    END
  ELSE
    BEGIN
    fileseg.VMpage ← dataseg.VMpage;
    IF fileseg.hint.page # fileseg.base OR fileseg.hint.da = eofDA THEN
      [] ← PositionSeg[fileseg, FALSE];
    MapVM[fileseg, WriteD];
    END;
  END;

CopyFileToDataSegment: PUBLIC PROCEDURE [
  fileseg: FileSegmentHandle, dataseg: DataSegmentHandle] =
  BEGIN
  waslocked: BOOLEAN ← TRUE;
  IF dataseg.pages # fileseg.pages THEN SwapError[fileseg];
  IF fileseg.swappedin THEN
    BEGIN
    IF fileseg.lock = 0 THEN
      BEGIN
      SwapIn[fileseg];
      waslocked ← FALSE;
      END;
    InlineDefs.COPY[
      from: FileSegmentAddress[fileseg],
      to: DataSegmentAddress[dataseg],
      nwords: dataseg.pages*AltoDefs.PageSize];
    IF ~waslocked THEN Unlock[fileseg];
    END
  ELSE
    BEGIN
    fileseg.VMpage ← dataseg.VMpage;
    IF (fileseg.hint.page # fileseg.base OR fileseg.hint.da = eofDA)
     AND PositionSeg[fileseg, TRUE] AND fileseg.pages = 1
      THEN NULL ELSE MapVM[fileseg, ReadD];
    END;
  END;

ChangeDataToFileSegment: PUBLIC PROCEDURE [
  dataseg: DataSegmentHandle, fileseg: FileSegmentHandle] =
  BEGIN
  IF dataseg.pages # fileseg.pages OR ~fileseg.write OR fileseg.swappedin
    OR fileseg.file.swapcount = MaxRefs THEN SIGNAL SwapError[fileseg];
  IF ~fileseg.file.open THEN OpenFile[fileseg.file];
  ProcessDefs.DisableInterrupts[];
  fileseg.swappedin ← TRUE;
  fileseg.VMpage ← dataseg.VMpage;
  fileseg.lock ← fileseg.lock+1;
  fileseg.file.swapcount ← fileseg.file.swapcount + 1;
  ProcessDefs.EnableInterrupts[];
  BootDefs.LiberateObject[BootDefs.GetDataSegmentTable[], dataseg];
  END;
```

```
-- File Positioning

jump: INTEGER = 10*DiskDefs.nSectors;

InvalidFP: PUBLIC SIGNAL [fp:POINTER TO FP] = CODE;

JumpToPage: PUBLIC PROCEDURE [
  cfa:POINTER TO CFA, page:PageNumber, buf:POINTER]
  RETURNS [prev,next:vDA] =
  BEGIN OPEN DiskDefs;
  desc: DiskPageDesc;
  da: vDA ← cfa.fa.da;
  startpage: PageNumber;
  direction: INTEGER ← 1;
  firstpage: PageNumber ← cfa.fa.page;
  arg: swap DiskRequest ← DiskRequest [
    buf,@da,,,@cfa.fp,TRUE,ReadD,ReadD,TRUE,swap[@desc]];
  BEGIN
    IF da=eofDA THEN GO TO reset;
    SELECT firstpage-page FROM
      <= 0  => NULL;
      = 1, < firstpage/10 => direction ← -1;
      ENDCASE => GO TO reset;
    EXITS reset =>
      BEGIN
      firstpage ← 0;
      da ← cfa.fp.leaderDA;
      END;
    END;
  BEGIN
    ENABLE DiskCheckError--[page]-- =>
      BEGIN
      IF page # startpage THEN RESUME;
      IF startpage=0 THEN GO TO failed;
      firstpage ← 0;
      da ← cfa.fp.leaderDA;
      direction ← 1;
      RETRY;
      END;
    IF da=eofDA THEN GO TO failed;
    startpage ← firstpage;
    UNTIL da=eofDA DO
      arg.firstPage ← firstpage;
      arg.lastPage ←
        IF direction<0 THEN firstpage
        ELSE MIN[page,firstpage+jump-1];
      [] ← SwapPages[@arg];
      IF desc.page=page THEN EXIT;
      da ← IF direction<0 THEN desc.prev ELSE desc.next;
      firstpage ← desc.page+direction;
      ENDLOOP;
    cfa.fa ← FA[desc.this,desc.page,desc.bytes];
    RETURN [desc.prev,desc.next];
    EXITS
      failed => ERROR InvalidFP[@cfa.fp];
    END;
  END;
```

```
-- Simplified Data Segments

AllocatePages: PUBLIC PROCEDURE [npages:CARDINAL]
  RETURNS [POINTER] = BEGIN
  RETURN[DataSegmentAddress[NewDataSegment[DefaultBase,npages]]]
  END;

AllocateSegment: PUBLIC PROCEDURE [nwords:CARDINAL]
  RETURNS [POINTER] = BEGIN
  RETURN[AllocatePages[PagesForWords[nwords]]]
  END;

SegmentSize: PUBLIC PROCEDURE [base:POINTER]
  RETURNS [CARDINAL] = BEGIN
  seg: DataSegmentHandle = VMtoDataSegment[base];
  RETURN [
    IF seg = NIL THEN 0
    ELSE seg.pages*AltoDefs.PageSize]
  END;

FreeSegment, FreePages: PUBLIC PROCEDURE [base:POINTER] =
  BEGIN
  seg: DataSegmentHandle = VMtoDataSegment[base];
  IF seg # NIL THEN DeleteDataSegment[seg];
  RETURN
  END;

PagesForWords: PUBLIC PROCEDURE [nwords: CARDINAL] RETURNS [CARDINAL] =
  BEGIN
  RETURN[(nwords + (AltoDefs.PageSize-1))/AltoDefs.PageSize]
  END;


-- Managing File Segment Objects

FileSegmentObjects: Table ← Table[SIZE[FileSegmentObject],NIL];
FileSegmentTable: TableHandle ← @FileSegmentObjects;

GetFileSegmentTable: PUBLIC PROCEDURE RETURNS [TableHandle] =
  BEGIN RETURN[FileSegmentTable] END;

AllocateFileSegment: PROCEDURE [TableHandle] RETURNS [FileSegmentHandle];
ValidateFileSegment: PROCEDURE [TableHandle,FileSegmentHandle];
LiberateFileSegment: PROCEDURE [TableHandle,FileSegmentHandle];

EnumerateFileSegments: PUBLIC PROCEDURE [
  proc: PROCEDURE [FileSegmentHandle] RETURNS [BOOLEAN]]
  RETURNS [FileSegmentHandle] = BEGIN
  RETURN[LOOPHOLE[
    EnumerateObjects[FileSegmentTable,LOOPHOLE[proc]]]]
  END;

VMtoFileSegment: PUBLIC PROCEDURE [a:POINTER] RETURNS [FileSegmentHandle] =
  BEGIN
  pg: PageNumber ← PageFromAddress[a];
  MatchPage: PROCEDURE [seg:FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[seg.swappedin AND pg IN
      [seg.VMpage..seg.VMpage+seg.pages-1]]
    END;
  RETURN[EnumerateFileSegments[MatchPage]]
  END;


-- Managing System Objects

SystemObjects: SystemTable;  -- initialized below

GetSystemTable: PUBLIC PROCEDURE RETURNS [SystemTableHandle] =
  BEGIN RETURN[@SystemObjects] END;


-- Main Body

AllocateFileSegment ← LOOPHOLE[AllocateObject];
ValidateFileSegment ← LOOPHOLE[ValidateObject];
```

```
LiberateFileSegment ← LOOPHOLE[LiberateObject];

SystemObjects ← SystemTable [
  pagemap: GetPageMap[],
  datasegs: GetDataSegmentTable[],
  filesegs: GetFileSegmentTable[],
  files: GetFileTable[]];

END.
```