```
-- Segments.Mesa  Edited by Sandman on May 12, 1978  3:07 PM

DIRECTORY
  AllocDefs: FROM "allocdefs" USING [
    AllocHandle, AllocInfo, GetAllocationObject, MakeDataSegment],
  AltoDefs: FROM "altodefs" USING [MaxVMPage, PageSize],
  AltoFileDefs: FROM "altofiledefs" USING [CFA, eofDA, FA, FP, vDA],
  BootDefs: FROM "bootdefs" USING [
    AllocateObject, EnumerateObjects, LiberateObject, MapVM, ValidateObject],
  DiskDefs: FROM "diskdefs" USING [
    DiskCheckError, DiskPageDesc, DiskRequest, nSectors, SwapPages],
  InlineDefs: FROM "inlinedefs" USING [BITAND, COPY],
  NucleusDefs: FROM "nucleusdefs",
  ProcessDefs: FROM "processdefs" USING [DisableInterrupts, EnableInterrupts],
  SegmentDefs: FROM "segmentdefs" USING [
    AccessOptions, AddressFromPage, Append, DataSegmentAddress,
    DataSegmentHandle, DefaultAccess, DefaultBase, DefaultPages,
    DeleteDataSegment, FileAccessError, FileError, FileHandle, FileHint,
    FileSegmentHandle, GetEndOfFile, MaxRefs, MaxSegs, NewDataSegment,
    Object, OpenFile, PageCount, PageNumber, PageFromAddress, Read,
    ReleaseFile, SegmentHandle, SetFileAccess, SwapError, SwapIn, SwapOut,
    SwapUp, Unlock, Write],
  SystemDefs: FROM "systemdefs";

DEFINITIONS FROM AltoFileDefs, BootDefs, SegmentDefs;

Segments: PROGRAM
  IMPORTS AllocDefs, BootDefs, DiskDefs, SegmentDefs
  EXPORTS BootDefs, NucleusDefs, 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[];
    BEGIN ENABLE UNWIND => LiberateFileSegment[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[FALSE, segment[file[FALSE, BITAND[access,Read]#0,
      BITAND[access,Write]#0, other, 0, file, base, pages, 0,
      disk[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
    seg ← NewFileSegment[file,base,pages,access];
    IF addr # NIL THEN
      BEGIN
      seg.VMpage ← PageFromAddress[addr];
      -- DisableInterrupts[];
      IF ~PagesBusy[seg.VMpage, pages] THEN ERROR;
      seg.swappedin ← TRUE;
      seg.lock ← seg.lock+1;
      file.swapcount ← file.swapcount+1;
      -- EnableInterrupts[];
      AllocDefs.GetAllocationObject[].update[seg.VMpage, pages, inuse, seg];
      END;
    RETURN
    END;

  PagesBusy: PROCEDURE [base: PageNumber, pages: PageCount] RETURNS [BOOLEAN] =
    BEGIN OPEN AllocDefs;
    object: AllocHandle ← GetAllocationObject[];
    FOR base IN [base..base+pages) DO
      IF object.status[base].status # busy THEN RETURN[FALSE];
      ENDLOOP;
    RETURN[TRUE]
    END;

  DeleteFileSegment: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
    BEGIN
    file: FileHandle ← seg.file;
    ValidateFileSegment[seg];
    SwapOut[seg];
    LiberateFileSegment[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[seg];
    IF base = DefaultBase THEN base ← 1;
    IF pages = DefaultPages THEN pages ← GetEndOfFile[seg.file].page-base+1;
    IF pages ~IN (0..AltoDefs.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[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;
    WITH s: seg SELECT FROM
      disk => s.hint ← FileHint[eofDA,0];
      ENDCASE;
    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;
  WITH s: seg SELECT FROM
    disk =>
      BEGIN
      IF s.hint.da = eofDA AND s.base > 8
        AND s.file.segcount > 1 THEN FindSegHint[@s];
      IF s.hint.da = eofDA OR s.hint.page # s.base THEN
        BEGIN
        buffer ←
          IF useseg THEN AddressFromPage[s.VMpage]
          ELSE DataSegmentAddress[buf ← NewDataSegment[DefaultBase,1]];
        cfa.fp ← s.file.fp;
        cfa.fa ← FA[s.hint.da,s.hint.page,0];
        [] ← JumpToPage[@cfa,s.base,buffer
          ! UNWIND => IF ~useseg THEN DeleteDataSegment[buf]];
        IF ~useseg THEN DeleteDataSegment[buf];
        IF cfa.fa.page # s.base THEN ERROR SwapError[@s];
        s.hint ← FileHint[cfa.fa.da,cfa.fa.page];
        RETURN[useseg AND cfa.fa.byte#0];
        END;
      END;
    ENDCASE;
  RETURN[FALSE]
  END;

FindSegHint: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN
  CheckHint: PROCEDURE [other:FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    WITH o: other SELECT FROM
      disk =>
        BEGIN
        IF o.file = seg.file AND o.hint.da # eofDA
          AND o.hint.page IN (hint.page..seg.base] THEN hint ← o.hint;
        RETURN[hint.page=seg.base]
        END;
      ENDCASE;
    RETURN[FALSE]
    END;
  hint: FileHint;
  WITH s: seg SELECT FROM
    disk =>
      BEGIN
      hint ← s.hint;
      [] ← EnumerateFileSegments[CheckHint];
      s.hint ← hint;
      END;
    ENDCASE;
  RETURN
  END;

GetFileSegmentDA: PUBLIC PROCEDURE [seg:FileSegmentHandle] RETURNS [vDA] =
  BEGIN
  WITH s: seg SELECT FROM
    disk =>
      BEGIN
      [] ← PositionSeg[seg,FALSE];
      RETURN[s.hint.da];
      END;
    ENDCASE;
  RETURN[AltoFileDefs.eofDA]
  END;

SetFileSegmentDA: PUBLIC PROCEDURE [seg:FileSegmentHandle, da:vDA] =
  BEGIN
  WITH s: seg SELECT FROM
    disk => s.hint ← FileHint[da,s.base];
    ENDCASE;
  RETURN
  END;
```

```
-- Segment Initialization

CopyDataToFileSegment: PUBLIC PROCEDURE [
  dataseg: DataSegmentHandle, fileseg: FileSegmentHandle] =
  BEGIN
  waslocked: BOOLEAN;
  IF dataseg.pages # fileseg.pages THEN SwapError[fileseg];
  IF fileseg.swappedin OR fileseg.loc = remote THEN
    BEGIN
    SwapIn[fileseg];
    waslocked ← fileseg.lock # 1;
    InlineDefs.COPY[
      from: DataSegmentAddress[dataseg],
      to: FileSegmentAddress[fileseg],
      nwords: dataseg.pages*AltoDefs.PageSize];
    IF ~waslocked THEN Unlock[fileseg];
    IF ~waslocked AND fileseg.loc = remote THEN SwapOut[fileseg];
    END
  ELSE
    WITH s: fileseg SELECT FROM
      disk =>
        BEGIN
        s.VMpage ← dataseg.VMpage;
        IF s.hint.page # s.base OR s.hint.da = eofDA THEN
          [] ← PositionSeg[@s, FALSE];
        MapVM[@s, WriteD];
        END;
      ENDCASE;
  END;

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

AllocateResidentPages: PUBLIC PROCEDURE [npages:CARDINAL]
  RETURNS [POINTER] =
  BEGIN OPEN AllocDefs;
  info: AllocInfo = [0, hard, topdown, initial, other, TRUE, FALSE];
  RETURN[DataSegmentAddress[MakeDataSegment[DefaultBase, npages, info]]]
  END;

AllocateResidentSegment: PUBLIC PROCEDURE [nwords:CARDINAL]
  RETURNS [POINTER] =
  BEGIN
  RETURN[AllocateResidentPages[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;

ValidateFileSegment: PROCEDURE [FileSegmentHandle];
LiberateFileSegment: PROCEDURE [FileSegmentHandle];

AllocateFileSegment: PROCEDURE RETURNS [seg: FileSegmentHandle] =
  BEGIN
  seg ← LOOPHOLE[AllocateObject[SIZE[file segment Object]]];
  seg↑ ← Object [FALSE, segment[file[, , , , , , , , , disk[]]]];
  RETURN
  END;

EnumerateFileSegments: PUBLIC PROCEDURE [
  proc: PROCEDURE [FileSegmentHandle] RETURNS [BOOLEAN]]
  RETURNS [FileSegmentHandle] =
  BEGIN OPEN BootDefs;
  CheckSegment: PROCEDURE [seg: SegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[WITH s: seg SELECT FROM
      file => proc[@s],
      ENDCASE => FALSE]
    END;
  RETURN[LOOPHOLE[EnumerateObjects[segment, LOOPHOLE[CheckSegment]]]];
  END;

VMtoDataSegment: PUBLIC PROCEDURE [a:POINTER] RETURNS [DataSegmentHandle] =
  BEGIN OPEN AllocDefs;
  seg: SegmentHandle ← VMtoSegment[a];
  IF seg = NIL THEN RETURN[NIL];
  WITH s: seg SELECT FROM data => RETURN[@s]; ENDCASE;
  RETURN[NIL];
  END;

VMtoFileSegment: PUBLIC PROCEDURE [a:POINTER] RETURNS [FileSegmentHandle] =
  BEGIN
```

```
    seg: SegmentHandle ← VMtoSegment[a];
    IF seg = NIL THEN RETURN[NIL];
    WITH s: seg SELECT FROM file => RETURN[@s]; ENDCASE;
    RETURN[NIL];
    END;

  VMtoSegment: PUBLIC PROCEDURE [a:POINTER] RETURNS [SegmentHandle] =
    BEGIN OPEN AllocDefs;
    pg: PageNumber = PageFromAddress[a];
    RETURN[GetAllocationObject[].status[pg].seg];
    END;

  SegmentAddress: PUBLIC PROCEDURE [seg:SegmentHandle] RETURNS [POINTER] =
    BEGIN
    page: PageNumber;
    WITH s: seg SELECT FROM
      data => page ← s.VMpage;
      file => IF ~s.swappedin THEN RETURN[NIL] ELSE page ← s.VMpage;
      ENDCASE => RETURN[NIL];
    RETURN[AddressFromPage[page]]
    END;

  EnumerateDataSegments: PUBLIC PROCEDURE [
    proc:PROCEDURE [DataSegmentHandle] RETURNS [BOOLEAN]]
    RETURNS [DataSegmentHandle] =
    BEGIN
    seg: SegmentHandle;
    i: CARDINAL ← 0;
    WHILE i  < AltoDefs.PageSize DO
      seg ← AllocDefs.GetAllocationObject[].status[i].seg;
      IF seg # NIL THEN
        WITH s: seg SELECT FROM
          data =>
            BEGIN
            IF proc[@s] THEN RETURN [@s];
            i ← i + s.pages;
            END;
          file => i ← i + s.pages;
          ENDCASE
      ELSE i ← i + 1;
      ENDLOOP;
    RETURN[NIL];
    END;

  PagesFree: PUBLIC PROCEDURE [base: PageNumber, pages: PageCount]
    RETURNS [BOOLEAN] =
    BEGIN
    FOR base IN [base..base+pages) DO
      IF ~PageFree[base] THEN RETURN[FALSE];
      ENDLOOP;
    RETURN[TRUE]
    END;

  PageFree: PUBLIC PROCEDURE [page: PageNumber] RETURNS [BOOLEAN] =
    BEGIN OPEN AllocDefs;
    RETURN[GetAllocationObject[].status[page].status = free]
    END;

  -- Main Body

  ValidateFileSegment ← LOOPHOLE[ValidateObject];
  LiberateFileSegment ← LOOPHOLE[LiberateObject];

  END.
```