```
-- File: DebugPilotCache.Mesa
-- Last edited by
--                Johnsson; August 29, 1978  9:48 AM

  DIRECTORY
    AllocDefs: FROM "allocdefs" USING [MakeDataSegment],
    AltoDefs: FROM "altodefs" USING [PageSize],
    DebugCacheDefs: FROM "debugcachedefs",
    DebugUtilityDefs: FROM "debugutilitydefs" USING [LongREAD, LongWRITE],
    DiskDefs: FROM "diskdefs" USING [
      DA, DC, DH, DS, DSgoodStatus, DSmaskStatus,
      UnrecoverableDiskError],
    InlineDefs: FROM "inlinedefs" USING [
      BITAND, DIVMOD, LongMult],
    MiscDefs: FROM "miscdefs" USING [Zero],
    SegmentDefs: FROM "segmentdefs" USING [
      DataSegmentAddress, DefaultBase, FileHandle,
      FileSegmentAddress, FileSegmentHandle, MoveFileSegment,
      NewFileSegment, Read, SwapIn, SwapUp, Unlock],
    VMMapLog: FROM "vmmaplog" USING [
      Descriptor, Entry, EntryBasePointer, EntryPointer,
      Pilot31Label, PilotFID];

DebugPilotCache: PROGRAM
  IMPORTS AllocDefs, DebugUtilityDefs, DiskDefs, MiscDefs, SegmentDefs
  EXPORTS DebugCacheDefs
  SHARES DiskDefs, SegmentDefs =

  BEGIN


  -- Dealing with the MapLog

  mfMap: DESCRIPTOR FOR ARRAY OF VMMapLog.Entry;
  mfMapSeg: SegmentDefs.FileSegmentHandle;
  mfMapLimit: CARDINAL;
  MFEntrySize: CARDINAL = SIZE[VMMapLog.Entry];

  MapSwapBase: CARDINAL = 16384-256;

  BadMapLogEntry: PUBLIC SIGNAL = CODE;

  InitMapSeg: PUBLIC PROCEDURE [f: SegmentDefs.FileHandle] =
    BEGIN OPEN SegmentDefs;
    mfMapSeg ← NewFileSegment[f,256,1,Read];
    END;

  ReinitMap: PUBLIC PROCEDURE [root: LONG POINTER TO VMMapLog.Descriptor] =
    BEGIN OPEN SegmentDefs;
    entry: VMMapLog.Entry;
    ep: POINTER = @entry;
    i: CARDINAL;
    UNTIL mfMapSeg.lock = 0 DO Unlock[mfMapSeg] ENDLOOP;
    mfMapSeg.write ← FALSE;
    MoveFileSegment[mfMapSeg, mfMapSeg.base, 1];
    mfMapLimit ← 0;
    IF root = NIL THEN RETURN;
    FOR i IN [0..MFEntrySize) DO
     (ep+i)↑ ← DebugUtilityDefs.LongREAD[@root.self+i] ENDLOOP;
    IF entry.page <= 255 THEN entry.page ← entry.page + MapSwapBase;
    AcquireMap[];
    AddMapEntry[@entry];
    ProcessMapLog[root];
    ReleaseMap[];
    END;

  ProcessMapLog: PUBLIC PROCEDURE [root: LONG POINTER TO VMMapLog.Descriptor] =
    BEGIN OPEN DebugUtilityDefs;
    base: VMMapLog.EntryBasePointer;
    entry: VMMapLog.Entry;
    ep: POINTER = @entry;
    freeEntry, i: CARDINAL;
    found: BOOLEAN;
    me: POINTER TO VMMapLog.Entry;
    reader, writer, limit: VMMapLog.EntryPointer;
    AcquireMap[];
```

```
    base ← LOOPHOLE[
      InlineDefs.LongMult[mfMap[0].page, AltoDefs.PageSize]];
    reader ← LongREAD[@root.reader];
    writer ← LongREAD[@root.writer];
    limit ← LongREAD[@root.limit];
    UNTIL reader = writer DO
      FOR i IN [0..MFEntrySize) DO
        (ep+i)↑ ← LongREAD[@base[reader]+i] ENDLOOP;
    IF entry.page <= 255 THEN entry.page ← entry.page + MapSwapBase;
      freeEntry ← 0;
      found ← FALSE;
      FOR i IN [0..mfMapLimit) DO
        me ← @mfMap[i];
        SELECT TRUE FROM
          me.kind = nil =>
            IF freeEntry = 0 THEN freeEntry ← i;
          entry.page IN [me.page..me.page+me.count),
          entry.page+entry.count IN [me.page..me.page+me.count),
          me.page IN [entry.page..entry.page+entry.count) =>
            BEGIN
            IF found THEN me.filePoint ← nil[]
            ELSE
              BEGIN
              found ← TRUE;
              me↑ ← entry;
              mfMapSeg.write ← TRUE;
              IF entry.page = me.page AND entry.count = me.count THEN
                EXIT;
              END;
            END;
          ENDCASE;
        REPEAT FINISHED =>
          IF ~found THEN
            IF freeEntry # 0 THEN mfMap[freeEntry] ← entry
            ELSE AddMapEntry[@entry];
        ENDLOOP;
      IF (reader ← reader + MFEntrySize) = limit THEN
        reader ← LOOPHOLE[0];
      ENDLOOP;
    LongWRITE[@root.reader, reader];
    ReleaseMap[];
    END;

  AddMapEntry: PROCEDURE [e: POINTER TO VMMapLog.Entry] =
    BEGIN
    IF mfMapLimit = LENGTH[mfMap] THEN
      BEGIN
      locks: CARDINAL = mfMapSeg.lock;
      THROUGH [0..locks) DO ReleaseMap[] ENDLOOP;
      SegmentDefs.MoveFileSegment[mfMapSeg, mfMapSeg.base, mfMapSeg.pages+1];
      THROUGH [0..locks) DO AcquireMap[] ENDLOOP;
      END;
    mfMap[mfMapLimit] ← e↑;
    mfMapSeg.write ← TRUE;
    mfMapLimit ← mfMapLimit + 1;
    END;

  LookupMapEntry: PUBLIC PROCEDURE [mempage: CARDINAL]
    RETURNS [me: POINTER TO VMMapLog.Entry] =
    BEGIN
    i: CARDINAL;
    AcquireMap[];
    FOR i IN [0..mfMapLimit) DO
      me ← @mfMap[i];
      IF mempage IN [me.page..me.page+me.count) AND
         me.kind # nil THEN
        RETURN;
      ENDLOOP;
    RETURN[NIL];
    END;

  AcquireMap: PROCEDURE =
    BEGIN
    SegmentDefs.SwapIn[mfMapSeg];
    mfMap ← DESCRIPTOR[
      SegmentDefs.FileSegmentAddress[mfMapSeg],
```

```
    (mfMapSeg.pages*AltoDefs.PageSize)/MFEntrySize + 1];
  END;

ReleaseMap: PUBLIC PROCEDURE =
  BEGIN
  IF mfMapSeg.write AND mfMapSeg.lock = 1 THEN
    BEGIN SegmentDefs.SwapUp[mfMapSeg]; mfMapSeg.write ← FALSE END;
  SegmentDefs.Unlock[mfMapSeg];
  IF mfMapSeg.lock = 0 THEN mfMap ← DESCRIPTOR[NIL, 0];
  END;

NewPilot31Segment: PUBLIC PROCEDURE [
  vda: CARDINAL, fileid: VMMapLog.PilotFID, filepage: CARDINAL]
  RETURNS [p: POINTER] =
  BEGIN OPEN SegmentDefs;
  label: VMMapLog.Pilot31Label;
  p ← DataSegmentAddress[AllocDefs.MakeDataSegment[DefaultBase, 1,
    [0,hard,bottomup,initial,other,TRUE,FALSE]]];
  MiscDefs.Zero[@label, SIZE[VMMapLog.Pilot31Label]];
  label.fid ← fileid;
  label.page ← filepage;
  MoveDiskPage[read, vda, @label, p];
  RETURN
  END;

WritePilot31Page: PUBLIC PROCEDURE [
  address: POINTER, vda: CARDINAL,
  fileid: VMMapLog.PilotFID, filepage: CARDINAL] =
  BEGIN
  label: VMMapLog.Pilot31Label;
  MiscDefs.Zero[@label, SIZE[VMMapLog.Pilot31Label]];
  label.fid ← fileid;
  label.page ← filepage;
  MoveDiskPage[write, vda, @label, address];
  END;

-- low level disk driver

DS: TYPE = DiskDefs.DS;

KCB: TYPE = MACHINE DEPENDENT RECORD [
  next: POINTER TO KCB,
  status: DS,
  command: DiskDefs.DC,
  header: POINTER TO DiskDefs.DH,
  label: POINTER,
  data: POINTER,
  normalbits: WORD,
  errorbits: WORD,
  unused: WORD,
  da: DiskDefs.DA];

nil: POINTER TO KCB = LOOPHOLE[0];

MaskDS: PROCEDURE [DS,DS] RETURNS [DS] = LOOPHOLE[InlineDefs.BITAND];

DSnoStatus: DS = LOOPHOLE[0];

diskCommands: POINTER TO POINTER TO KCB = LOOPHOLE[521B];

MoveDiskPage: PROCEDURE
  [op: {read, write}, vda: CARDINAL, label, data: POINTER] =
  BEGIN OPEN DiskDefs;
  da: DiskDefs.DA = Real31DA[vda];
  header: DH ← [0,da];
  kcb: KCB ← [
    next: nil, status:, command:,
    header: @header, label: label, data: data,
    normalbits: 0, errorbits: 0, unused: 0,
    da:];
  seekonly: DC = [110B,DiskCheck,DiskCheck,DiskCheck,1,1];
  move: DC ← [110B,DiskCheck,DiskCheck,DiskRead,0,1];
  tries: CARDINAL;
  IF op = write THEN move.data ← DiskWrite;
  FOR tries IN [0..5] DO
    UNTIL diskCommands↑ = nil DO NULL ENDLOOP;
```

```
      kcb.status ← DSnoStatus;
      kcb.command ← move;
      kcb.da ← da;
      diskCommands↑ ← @kcb;
      UNTIL kcb.status.done # 0 DO NULL ENDLOOP;
      IF MaskDS[kcb.status,DSmaskStatus] = DSgoodStatus THEN EXIT;
      IF tries = 2 THEN
        BEGIN
        kcb.status ← DSnoStatus;
        kcb.command ← seekonly;
        kcb.da ← [0,0,0,0,1];
        diskCommands↑ ← @kcb;
        END;
      REPEAT FINISHED => ERROR UnrecoverableDiskError[NIL];
      ENDLOOP;
    END;

  Real31DA: PROCEDURE [vda: CARDINAL] RETURNS [da: DiskDefs.DA] =
    BEGIN OPEN InlineDefs;
    da ← [0,0,0,0,0];
    [vda, da.sector] ← DIVMOD[vda, 12];
    [vda, da.head] ← DIVMOD[vda, 2];
    [da.disk, da.track] ← DIVMOD[vda, 203];
    END;

  END...
```