```
-- File: ExternalCache.Mesa
-- Last edited by
--                Johnsson; July 20, 1978  2:22 PM
--                Barbara; July 7, 1978  12:17 PM

  DIRECTORY
    AllocDefs: FROM "allocdefs" USING [
      AddSwapStrategy, CantSwap, SwappingProcedure, SwapStrategy],
    AltoDefs: FROM "altodefs" USING [PageSize],
    AltoFileDefs: FROM "altofiledefs" USING [fillinDA, FP, vDA],
    BFSDefs: FROM "bfsdefs" USING [ActOnPages, GetNextDA],
    ControlDefs: FROM "controldefs" USING [ControlLink],
    CoreSwapDefs: FROM "coreswapdefs",
    DebugData: FROM "debugdata" USING [altoXM, debugPilot, mdsContext, onDO],
    DebuggerDefs: FROM "debuggerdefs" USING [LA, LP],
    DebugMiscDefs: FROM "debugmiscdefs",
    DebugUtilityDefs: FROM "debugutilitydefs",
    DebugXMDefs: FROM "debugxmdefs" USING [XMRead, XMWrite],
    DiskDefs: FROM "diskdefs" USING [
      DA, DC, DH, DiskRequest, DS, DSgoodStatus, DSmaskStatus,
      UnrecoverableDiskError],
    ImageDefs: FROM "imagedefs",
    InlineDefs: FROM "inlinedefs" USING [
      BITAND, COPY, DIVMOD, LongDivMod, LongMult],
    MiscDefs: FROM "miscdefs" USING [Zero],
    Mopcodes: FROM "mopcodes" USING [zMISC, zRBL, zWBL],
    SegmentDefs: FROM "segmentdefs" USING [
      DeleteFileSegment, FileHandle, FileSegmentAddress, FileSegmentHandle,
      GetFileSegmentDA, InsertFile, LockFile, NewFileSegment, Read,
      SetFileSegmentDA, SwapIn, Unlock],
    SystemDefs: FROM "systemdefs" USING [AllocatePages, FreePages];

ExternalCache: PROGRAM
  IMPORTS AllocDefs, BFSDefs, DDptr: DebugData, DebugXMDefs, DiskDefs, MiscDefs, SystemDefs, SegmentDef
**s
  EXPORTS DebugMiscDefs, DebugUtilityDefs
  SHARES DiskDefs, SegmentDefs =

  BEGIN

  -- move this to a utility somewhere

  Bound: PUBLIC PROCEDURE [p: UNSPECIFIED] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[LOOPHOLE[p, ControlDefs.ControlLink].tag # unbound]
    END;

  LA: TYPE = DebuggerDefs.LA;
  LP: TYPE = DebuggerDefs.LP;

  -- DO/Pilot Stuff

  PilotFID: TYPE = RECORD [a,b: UNSPECIFIED];

  PilotDiskLabel: TYPE = RECORD [
    fid: PilotFID,
    page: CARDINAL,
    fill: ARRAY [0..5) OF WORD];

  MapEntry: TYPE = MACHINE DEPENDENT RECORD [
    flags: MapFlags,
    realPage: [0..7777B]];

  MapFlags: TYPE = MACHINE DEPENDENT RECORD [
    LogSE, W, D, Ref: BOOLEAN];

  Vacant: MapEntry = [[FALSE, TRUE, TRUE, FALSE],0];
  Clean: MapEntry = [[FALSE, FALSE, FALSE, FALSE],0];

  MapSwapBase: CARDINAL = 4096-256;

  SETF: PROCEDURE [CARDINAL, MapEntry] RETURNS [MapEntry] =
    MACHINE CODE BEGIN Mopcodes.zMISC, 1 END;

  RBL: PROCEDURE [LA] RETURNS [UNSPECIFIED] =
    MACHINE CODE BEGIN Mopcodes.zRBL, 0 END;
```

```
WBL: PROCEDURE [UNSPECIFIED, LA] =
  MACHINE CODE BEGIN Mopcodes.zWBL, 0 END;

-- Cache of user memory pages

CoreSegmentObject: TYPE = RECORD [
  address: POINTER,
  lastused: CARDINAL,
  mempage: CARDINAL,
  inuse: BOOLEAN,
  dirty: BOOLEAN,
  body:
    SELECT type: * FROM
      alto => [
        segment: SegmentDefs.FileSegmentHandle],
      pilot31 => [
        vda: CARDINAL,
        fileid: PilotFID,
        filepage: CARDINAL],
      ENDCASE];

CoreSegment: TYPE = POINTER TO CoreSegmentObject;

maxsegments: CARDINAL = 8;        -- number of pages to keep in core

CS: ARRAY [0..maxsegments) OF CoreSegmentObject;

CurrentUseValue: CARDINAL;
CoreFile: SegmentDefs.FileHandle;
DAs: ARRAY [-1..256] OF AltoFileDefs.vDA;
DiskAddresses: DESCRIPTOR FOR ARRAY OF AltoFileDefs.vDA ←
  DESCRIPTOR[@DAs[0],256];
CacheSwap: AllocDefs.SwapStrategy ←
  AllocDefs.SwapStrategy[link:,proc:AllocDefs.CantSwap];

InitCoreCache: PROCEDURE [file: SegmentDefs.FileHandle] =
  BEGIN
  i: CARDINAL;
  CurrentUseValue ← 0;
  FOR i IN [0..maxsegments) DO CS[i].inuse ← FALSE ENDLOOP;
  CoreFile ← file;
  SegmentDefs.LockFile[CoreFile];
  AllocDefs.AddSwapStrategy[@CacheSwap];
  END;


FlushCoreCache: PUBLIC AllocDefs.SwappingProcedure =
  BEGIN OPEN SegmentDefs;
  did: BOOLEAN ← FALSE;
  i: CARDINAL ← 0;
  cs: CoreSegment;
  CacheSwap.proc ← AllocDefs.CantSwap;
  FOR i IN [0..maxsegments) DO
    cs←@CS[i];
    IF cs.inuse THEN
      BEGIN
      FlushCS[cs];
      did ← TRUE;
      END;
    ENDLOOP;
  CurrentUseValue←0;
  RETURN[did]
  END;


FlushCS: PROCEDURE [cs: CoreSegment] =
  BEGIN OPEN SegmentDefs;
  WITH cs SELECT FROM
    alto =>
      BEGIN
      IF dirty THEN segment.write ← TRUE;
      Unlock[segment];
      DeleteFileSegment[segment];
      END;
    pilot31 =>
```

```
      BEGIN
      IF dirty THEN
        WritePilot31Page[address, vda, fileid, filepage];
      SystemDefs.FreePages[address];
      END;
        ENDCASE;
  cs.inuse ← FALSE;
  END;


NewSwateeSegment: PROCEDURE [mempage: CARDINAL, cs: CoreSegment] =
  BEGIN OPEN SegmentDefs;
  filepage: CARDINAL = SELECT mempage FROM
    IN[2..253] => mempage, 1 => 254, ENDCASE => 255;
  seg: FileSegmentHandle = NewFileSegment[CoreFile, filepage, 1, Read];
  SetFileSegmentDA[seg, DiskAddresses[filepage]];
  SwapIn[seg];
  DiskAddresses[filepage] ← GetFileSegmentDA[seg];
  cs↑ ← [
    address: FileSegmentAddress[seg],
    inuse: TRUE,
    dirty: FALSE,
    lastused:,
    mempage: mempage,
    body: alto[seg]];
  RETURN
  END;


NonExistentMemoryPage: PUBLIC SIGNAL [page: CARDINAL] = CODE;
InvalidAddress: PUBLIC SIGNAL [a: LA] = CODE;

NewVMSegment: PROCEDURE [mempage: CARDINAL, cs: CoreSegment] =
  BEGIN OPEN SegmentDefs;
  ERROR NonExistentMemoryPage[mempage];
  -- find file info
  -- call NewAltoSegment or NewPilot31Segment
  END;


NewAltoSegment: PROCEDURE [fp: POINTER TO AltoFileDefs.FP, filepage: CARDINAL]
  RETURNS [seg: SegmentDefs.FileSegmentHandle] =
  BEGIN OPEN SegmentDefs;
  seg ← NewFileSegment[InsertFile[fp, Read], filepage, 1, Read];
  SwapIn[seg];
  RETURN
  END;


NewPilot31Segment: PROCEDURE [
  vda: CARDINAL, fileid: PilotFID, filepage: CARDINAL]
  RETURNS [p: POINTER] =
  BEGIN
  label: PilotDiskLabel;
  p ← SystemDefs.AllocatePages[1];
  MiscDefs.Zero[@label, SIZE[PilotDiskLabel]];
  label.fid ← fileid;
  label.page ← filepage;
  MoveDiskPage[read, Real31DA[vda], @label, p];
  RETURN
  END;


GetCS: PROCEDURE [mempage: CARDINAL] RETURNS [sp: CoreSegment] =
  BEGIN
  minUseVal: CARDINAL ← CurrentUseValue;
  minUseIndex: CARDINAL ← 0;
  i: CARDINAL;

  BEGIN
  FOR i IN [0..maxsegments) DO
    sp ← @CS[i];
    IF ~sp.inuse THEN GO TO newseg;
    IF sp.mempage = mempage THEN EXIT;
    IF sp.lastused < minUseVal THEN
      BEGIN minUseVal←sp.lastused; minUseIndex←i END;
```

```
    REPEAT FINISHED =>
      BEGIN
      FOR i IN [0..maxsegments) DO
        CS[i].lastused ← CS[i].lastused - minUseVal;
        ENDLOOP;
      CurrentUseValue ← CurrentUseValue - minUseVal;
      FlushCS[sp ← @CS[minUseIndex]];
      GO TO newseg;
      END
    ENDLOOP;
  EXITS newseg =>
    BEGIN
    cso: CoreSegmentObject;
    IF mempage IN [0..253] THEN NewSwateeSegment[mempage, @cso]
    ELSE NewVMSegment[mempage, @cso];
    FOR i IN [0..maxsegments) DO
      sp ← @CS[i];
      IF ~sp.inuse THEN BEGIN sp↑ ← cso; EXIT END;
      REPEAT FINISHED => ERROR
      ENDLOOP;
    END;
  END;
  sp.lastused ← CurrentUseValue ← CurrentUseValue+1;
  CacheSwap.proc ← FlushCoreCache;
  RETURN[sp];
  END;


LongREAD: PUBLIC PROCEDURE [a: LA] RETURNS [UNSPECIFIED] =
  BEGIN OPEN AltoDefs, InlineDefs;
  mempage, offset: CARDINAL;
  [mempage, offset] ← LongDivMod[LOOPHOLE[a],PageSize];
  IF mempage > 255 THEN
    BEGIN
    IF RealMemory[mempage] THEN RETURN[ReadVM[mempage, a]];
    END
  ELSE
    BEGIN
    IF DDptr.debugPilot THEN
      BEGIN
      m: MapEntry = ReadMap[MapSwapBase+mempage];
      vp: CARDINAL;
      IF m = Vacant THEN ERROR InvalidAddress[a];
      FOR vp IN [0..253] DO
        IF ReadMap[vp].realPage = m.realPage THEN
          BEGIN mempage ← vp; EXIT END;
        REPEAT FINISHED => RETURN[ReadVM[mempage, a]];
        ENDLOOP;
      END;
    IF mempage > 253 THEN RETURN[MEMORY[mempage*PageSize+offset]];
    END;
  RETURN [(GetCS[mempage].address + offset)↑];
  END;


LongWRITE: PUBLIC PROCEDURE [a: LA, v: UNSPECIFIED] =
  BEGIN OPEN AltoDefs, InlineDefs;
  mempage, offset: CARDINAL;
  s: CoreSegment;
  [mempage, offset] ← LongDivMod[LOOPHOLE[a],PageSize];
  IF mempage > 255 THEN
    BEGIN
    IF RealMemory[mempage] THEN
      BEGIN WriteVM[mempage, a, v]; RETURN END;
    END
  ELSE
    BEGIN
    IF DDptr.debugPilot THEN
      BEGIN
      m: MapEntry = ReadMap[MapSwapBase+mempage];
      vp: CARDINAL;
      IF m = Vacant THEN ERROR InvalidAddress[a];
      FOR vp IN [0..253] DO
        IF ReadMap[vp].realPage = m.realPage THEN
          BEGIN mempage ← vp; EXIT END;
        REPEAT FINISHED => BEGIN WriteVM[mempage, a, v]; RETURN END;
```

```
        ENDLOOP;
      END;
    IF mempage > 253 THEN
      BEGIN MEMORY[mempage*PageSize+offset] ← v; RETURN END;
    END;
  ((s←GetCS[mempage]).address + offset)↑ ← v;
  s.dirty ← TRUE;
  RETURN
  END;

AREAD, MREAD, SREAD: PUBLIC PROCEDURE [a: UNSPECIFIED] RETURNS [UNSPECIFIED] =
  BEGIN OPEN AltoDefs, InlineDefs;
  mds: LP = LOOPHOLE[LongMult[DDptr.mdsContext, PageSize]];
  RETURN[LongREAD[LOOPHOLE[mds+CARDINAL[a],LA]]]
  END;


AWRITE, MWRITE, SWRITE: PUBLIC PROCEDURE [a: UNSPECIFIED, v: UNSPECIFIED] =
  BEGIN OPEN AltoDefs, InlineDefs;
  mds: LP = LOOPHOLE[LongMult[DDptr.mdsContext, PageSize]];
  LongWRITE[LOOPHOLE[mds+CARDINAL[a],LA], v];
  RETURN
  END;

RealMemory: PROCEDURE [mempage: CARDINAL] RETURNS [BOOLEAN] =
  BEGIN
  IF DDptr.onDO THEN RETURN[ReadMap[mempage] # Vacant];
  IF DDptr.altoXM THEN RETURN[mempage < 1024];
  ERROR NonExistentMemoryPage[mempage];
  END;

ReadVM: PROCEDURE [mempage: CARDINAL, p: LA] RETURNS [UNSPECIFIED] =
  BEGIN
  IF DDptr.onDO THEN RETURN[RBL[p]];
  IF DDptr.altoXM AND Bound[DebugXMDefs.XMRead] THEN
    RETURN[DebugXMDefs.XMRead[LOOPHOLE[p]]];
  ERROR InvalidAddress[p];
  END;

WriteVM: PROCEDURE [mempage: CARDINAL, p: LA, v: UNSPECIFIED] =
  BEGIN
  IF DDptr.onDO THEN
    BEGIN
    oldm: MapEntry ← SETF[mempage, Vacant];
    [] ← SETF[mempage, Clean];
    WBL[v, p];
    IF ~oldm.flags.W THEN oldm.flags.D ← TRUE;  -- wait for hardware change
    oldm.flags.Ref ← TRUE;
    [] ← SETF[mempage, oldm];
    RETURN
    END;
  IF DDptr.altoXM AND Bound[DebugXMDefs.XMWrite] THEN
    BEGIN DebugXMDefs.XMWrite[LOOPHOLE[p], v]; RETURN END;
  ERROR InvalidAddress[p];
  END;

WritePilot31Page: PROCEDURE [address: POINTER, vda: CARDINAL, fileid: PilotFID, filepage: CARDINAL] =
**
  BEGIN
  label: PilotDiskLabel;
  MiscDefs.Zero[@label, SIZE[PilotDiskLabel]];
  label.fid ← fileid;
  label.page ← filepage;
  MoveDiskPage[write, Real31DA[vda], @label, address];
  END;

ReadMap: PROCEDURE [mempage: CARDINAL] RETURNS [MapEntry] =
  BEGIN
  m: MapEntry = SETF[mempage, Vacant];
  [] ← SETF[mempage, m];
  RETURN[m]
  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];

  DiskOp: TYPE = {read, write};

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

  MoveDiskPage: PROCEDURE
    [op: DiskOp, da: DiskDefs.DA, label, data: POINTER] =
    BEGIN OPEN DiskDefs;
    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;


  -- initialization

  InitializeDebuggerFiles: PUBLIC PROCEDURE [debuggee: SegmentDefs.FileHandle] =
    BEGIN OPEN AltoFileDefs, DiskDefs, SegmentDefs;
    p: POINTER;
    diskrequest: DiskRequest;
    diskrequest ← DiskRequest [
      ca: SystemDefs.AllocatePages[1],
      fixedCA: TRUE,
      da: @DiskAddresses[0],
      fp: @debuggee.fp,
```

```
    firstPage: 0,
    lastPage: 255,
    action: ReadD,
    lastAction: ReadD,
    signalCheckError: FALSE,
    option: update[BFSDefs.GetNextDA]];
  p ← LOOPHOLE[@DiskAddresses[0]-1];
  p↑ ← fillinDA;
  InlineDefs.COPY[from: p, to: p+1, nwords: 257];
  DiskAddresses[0] ← debuggee.fp.leaderDA;
  [] ← BFSDefs.ActOnPages[LOOPHOLE[@diskrequest]];
  SystemDefs.FreePages[diskrequest.ca];
  InitCoreCache[debuggee];
  END;

END...
```