```
-- UserSegments.Mesa
-- Edited by:
--             Johnsson on August 30, 1978  12:02 PM
--             Barbara on July 31, 1978  4:28 PM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [BYTE, BytesPerPage, PageNumber, PageSize],
  AltoFileDefs: FROM "altofiledefs" USING [eofDA, FP],
  BcdDefs: FROM "bcddefs" USING [MTHandle, MTIndex],
  BootDefs: FROM "bootdefs" USING [PageMap, SystemTableHandle],
  ControlDefs: FROM "controldefs" USING [
    BytePC, FrameHandle, GlobalFrameHandle, InstWord],
  DebugBreakptDefs: FROM "debugbreakptdefs" USING [CodeObject],
  DebugCacheDefs: FROM "debugcachedefs" USING [LongWRITEClean, SwappedIn],
  DebugContextDefs: FROM "debugcontextdefs" USING [
    DAcquireBcd, DReleaseBcd, InvalidGlobalFrame,
    MapRC],
  DebugData: FROM "debugdata" USING [
    altoXM, config, DebuggeeFH, debugPilot, ESV, initBCD, mdsContext, onDO],
  DebuggerDefs: FROM "debuggerdefs" USING [LA],
  DebugMiscDefs: FROM "debugmiscdefs" USING [CommandNotAllowed],
  DebugUsefulDefs: FROM "debugusefuldefs",
  DebugUtilityDefs: FROM "debugutilitydefs" USING [
    Bound, FindOriginal, LoadStateInvalid, LongREAD, LongWRITE,
    SREAD, SWRITE, VirtualGlobalFrame],
  DebugXMDefs: FROM "debugxmdefs" USING [XMAllocOnDrum, XMFreeOnDrum],
  FrameDefs: FROM "framedefs",
  InlineDefs: FROM "inlinedefs" USING [LongDiv, LongMult],
  LoaderBcdUtilDefs: FROM "loaderbcdutildefs" USING [
    BcdBase, EnumerateModuleTable, ReleaseBcdSeg, SetUpBcd],
  LoadStateDefs: FROM "loadstatedefs" USING [
    BcdSegFromLoadState, ConfigNull, GFTIndex,
    InputLoadState, ReleaseLoadState],
  SegmentDefs: FROM "segmentdefs" USING [
    AddressFromPage, DeleteFileSegment, FileHandle,
    FileHint, FileObject, FileSegmentAddress, FileSegmentHandle,
    FileSegmentObject, InsertFile, NewFileSegment,
    PageFromAddress, Read, ReleaseFile, SegmentFault,
    SetEndOfFile, SwapIn, SwapOut, Unlock],
  SystemDefs: FROM "systemdefs" USING [AllocateHeapNode, FreeHeapNode],
  VMMapLog: FROM "vmmaplog" USING [
    Descriptor, PatchTable, PatchTableEntry,
    PatchTableEntryBasePointer, PatchTableEntryPointer];

DEFINITIONS FROM DebugUtilityDefs;

UserSegments: PROGRAM
  IMPORTS DDptr: DebugData, DebugCacheDefs, DebugContextDefs, DebugMiscDefs,
    DebugUtilityDefs, DebugXMDefs, LoaderBcdUtilDefs, LoadStateDefs,
    SegmentDefs, SystemDefs
  EXPORTS DebugMiscDefs, DebugUsefulDefs, DebugUtilityDefs
  SHARES SegmentDefs, ControlDefs =

BEGIN

BytePC: TYPE = ControlDefs.BytePC;
BYTE: TYPE = AltoDefs.BYTE;
FrameHandle: TYPE = ControlDefs.FrameHandle;
GlobalFrameHandle: TYPE = ControlDefs.GlobalFrameHandle;
FileHandle: TYPE = SegmentDefs.FileHandle;
InstWord: TYPE = ControlDefs.InstWord;
CodeObject: TYPE = DebugBreakptDefs.CodeObject;
LA: TYPE = DebuggerDefs.LA;


FileSegmentHandle: TYPE = SegmentDefs.FileSegmentHandle;


-- Utilities

CopyRead: PUBLIC PROCEDURE [to, from: POINTER, nwords: CARDINAL] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..nwords) DO
    (to+i)↑ ← SREAD[from+i];
    ENDLOOP;
```

```
  RETURN
  END;

CopyWrite: PUBLIC PROCEDURE [to, from: POINTER, nwords: CARDINAL] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..nwords) DO
    SWRITE[to+i,(from+i)↑];
    ENDLOOP;
  RETURN
  END;

LongCopyRead: PUBLIC PROCEDURE [to: POINTER, from: LONG POINTER, nwords: CARDINAL] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..nwords) DO
    (to+i)↑ ← LongREAD[from+i];
    ENDLOOP;
  RETURN
  END;

LongCopyWrite: PUBLIC PROCEDURE [to: LONG POINTER, from: POINTER, nwords: CARDINAL] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..nwords) DO
    LongWRITE[to+i,(from+i)↑];
    ENDLOOP;
  RETURN
  END;

UserSegment: SegmentDefs.FileSegmentObject;

UserFileSegmentAddress: PROCEDURE[useg: FileSegmentHandle]
  RETURNS[POINTER] =
  BEGIN
  RETURN[SegmentDefs.AddressFromPage[ReadUserSegment[useg].VMpage]]
  END;

ReadUserSegment: PROCEDURE [s: FileSegmentHandle] RETURNS [FileSegmentHandle] =
  BEGIN
  CopyRead[to: @UserSegment, from: s,
    nwords: SIZE[SegmentDefs.FileSegmentObject]];
  RETURN [@UserSegment]
  END;

WriteUserSegment: PROCEDURE [s: FileSegmentHandle] =
  BEGIN
  CopyWrite[ to: s, from: @UserSegment,
    nwords: SIZE[SegmentDefs.FileSegmentObject]];
  END;

-- "Swapping Drum" and user code manipulation

  DrumItemHandle: TYPE = POINTER TO DrumItem;
  DrumItem: TYPE = RECORD [
    next: DrumItemHandle,
    dseg: FileSegmentHandle,    -- for segment on drum
    co: CodeObject,
    useg: FileSegmentHandle,    -- in user space (Alto)
    oldBase: AltoDefs.PageNumber,
    oldFile: FileHandle,
    oldHint: SegmentDefs.FileHint];
  diHead: DrumItemHandle ← NIL;

  endHint: SegmentDefs.FileHint;
  endPage: AltoDefs.PageNumber;
  drumFile: FileHandle;

  MoveToDrum: PROCEDURE [f: GlobalFrameHandle, co: CodeObject] =
    BEGIN
    di: DrumItemHandle;
    LocateCode[f];
    IF gfCache.seg # NIL THEN AllocOnDrum[gfCache.seg].di.co ← co
    ELSE IF DDptr.altoXM AND
            DebugUtilityDefs.Bound[DebugXMDefs.XMAllocOnDrum] THEN
      BEGIN
```

```
    IF (di←DebugXMDefs.XMAllocOnDrum[f]) # NIL THEN di.co ← co;
    END;
  FlushCodeCache[];
  RETURN
  END;

AllocOnDrum: PUBLIC PROCEDURE [useg: FileSegmentHandle]
  RETURNS [di: DrumItemHandle] =
  BEGIN OPEN SegmentDefs;
  p: DrumItemHandle;
  lfo: FileObject;
  tfile: FileHandle = @lfo; -- copy of user file object
  tseg: FileSegmentHandle; -- copy of user segment
  dseg: FileSegmentHandle = MapUserSegment[useg];
  old: FileHandle = dseg.file;
  di ← SystemDefs.AllocateHeapNode[SIZE[DrumItem]];
  di.next ← NIL;
  di.dseg ← dseg;
  -- copy values from user segment
  tseg ← ReadUserSegment[di.useg ← useg];
  di.oldBase ← tseg.base;
  di.oldFile ← tseg.file;
  WITH t: tseg SELECT FROM
    disk => di.oldHint ← t.hint;
    ENDCASE => ERROR RemoteSegment[useg
      ! UNWIND => SystemDefs.FreeHeapNode[di]];
  -- remove segment from user's file object
  CopyRead[to: tfile, from: tseg.file, nwords: SIZE[FileObject]];
  tfile.lock ← tfile.lock + 1;
  tfile.segcount ← tfile.segcount - 1;
  IF tseg.swappedin THEN tfile.swapcount ← tfile.swapcount - 1;
  CopyWrite[from: tfile, to: tseg.file, nwords: SIZE[FileObject]];
  -- move user segment to drum file
  tseg.file ← DDptr.ESV.drumFile;
  tseg.base ← endPage;
  -- reflect new seg and swap counts in users drum file object
  CopyRead[to: tfile, from: DDptr.ESV.drumFile, nwords: SIZE[FileObject]];
  tfile.segcount ← tfile.segcount + 1;
  IF tseg.swappedin THEN tfile.swapcount ← tfile.swapcount + 1;
  CopyWrite[from: tfile, to: DDptr.ESV.drumFile, nwords: SIZE[FileObject]];
  SwapIn[dseg];
  dseg.write ← TRUE;
  -- update seg and swap counts for debugger's files
  old.swapcount ← old.swapcount - 1;
  IF (old.segcount ← old.segcount - 1) = 0 THEN
    ReleaseFile[old];
  drumFile.segcount ← drumFile.segcount + 1;
  drumFile.swapcount ← drumFile.swapcount + 1;
  -- move drum segment to drum file
  dseg.file ← drumFile;
  dseg.base ← endPage;
  WITH d: dseg SELECT FROM
    disk => d.hint ← endHint;
    ENDCASE;
  endPage ← endPage + dseg.pages;
  Unlock[dseg];
  SwapOut[dseg !
    SegmentFault =>
      BEGIN
      SetEndOfFile[drumFile,endPage-1,AltoDefs.BytesPerPage];
      RETRY
      END];
  WITH d: dseg SELECT FROM
    disk => endHint ← d.hint;
    ENDCASE;
  WITH t: tseg SELECT FROM
    disk => t.hint ← endHint;
    ENDCASE;
  WriteUserSegment[useg];
  dseg.write ← FALSE;
  -- add new item to end of list
  IF diHead = NIL THEN diHead ← di
  ELSE FOR p ← diHead, p.next UNTIL p.next = NIL DO
    NULL;
    REPEAT FINISHED => p.next ← di;
    ENDLOOP;
```

```
    RETURN
    END;

  FreeOnDrum: PUBLIC PROCEDURE [f: GlobalFrameHandle] =
    BEGIN
    IF DDptr.debugPilot THEN RETURN;  -- Pilot code not on drum
    LocateCode[f];
    IF gfCache.seg # NIL THEN RemoveFromDrum[gfCache.seg]
    ELSE IF DDptr.altoXM AND DebugUtilityDefs.Bound[DebugXMDefs.XMFreeOnDrum] THEN
      BEGIN
      DebugXMDefs.XMFreeOnDrum[f];
      END;
    FlushCodeCache[];
    RETURN
    END;

  RemoveFromDrum: PUBLIC PROCEDURE [useg: FileSegmentHandle] =
    BEGIN OPEN SegmentDefs;
    lfo: FileObject;
    tfile: FileHandle = @lfo; -- copy of user file object
    tseg: FileSegmentHandle; -- copy of user segment
    prev, di: DrumItemHandle;
    -- find item on the list
    prev ← NIL;
    FOR di ← diHead, di.next UNTIL di = NIL DO
      IF di.useg = useg THEN EXIT;
      prev ← di;
      REPEAT FINISHED => RETURN
      ENDLOOP;
    IF prev = NIL THEN diHead ← di.next
    ELSE prev.next ← di.next;
    -- put old values back into user segment
    tseg ← ReadUserSegment[useg];
    tseg.file ← di.oldFile;
    tseg.base ← di.oldBase;
    WITH t: tseg SELECT FROM
      disk => t.hint ← di.oldHint;
      ENDCASE;
    -- add segment to original file
    CopyRead[to: tfile, from: tseg.file, nwords: SIZE[FileObject]];
    tfile.lock ← tfile.lock - 1;
    tfile.segcount ← tfile.segcount + 1;
    IF tseg.swappedin THEN tfile.swapcount ← tfile.swapcount + 1;
    CopyWrite[from: tfile, to: tseg.file, nwords: SIZE[FileObject]];
    -- remove segment from drum file
    CopyRead[to: tfile, from: DDptr.ESV.drumFile, nwords: SIZE[FileObject]];
    tfile.segcount ← tfile.segcount - 1;
    IF tseg.swappedin THEN tfile.swapcount ← tfile.swapcount - 1;
    CopyWrite[from: tfile, to: DDptr.ESV.drumFile, nwords: SIZE[FileObject]];
    WriteUserSegment[useg];
    -- update end values and shuffle
    WITH s: di.dseg SELECT FROM
      disk => endHint ← s.hint;
      ENDCASE;
    endPage ← di.dseg.base;
    DeleteFileSegment[di.dseg]; -- delete the real debugger segment
    ShuffleDrum[di.next];
    SystemDefs.FreeHeapNode[di];
    RETURN
    END;

  CodeOnDrum: PROCEDURE [co: CodeObject] RETURNS [BOOLEAN] =
    BEGIN
    di: DrumItemHandle;
    FOR di ← diHead, di.next UNTIL di = NIL DO
      IF di.co = co THEN RETURN[TRUE];
      ENDLOOP;
    RETURN[FALSE];
    END;

  ShuffleDrum: PROCEDURE [di: DrumItemHandle] =
    -- Starting with di, shuffle segments to lower addresses on the drum
    -- and update the user's copies
    BEGIN OPEN SegmentDefs;
    seg: FileSegmentHandle;
    useg: FileSegmentHandle;
```

```
    UNTIL di = NIL DO
      SwapIn[seg ← di.dseg];
      useg ← ReadUserSegment[di.useg];
      useg.base ← seg.base ← endPage;
      WITH s: seg SELECT FROM
        disk => s.hint ← endHint;
        ENDCASE;
      WITH u: useg SELECT FROM
        disk => u.hint ← endHint;
        ENDCASE;
      WriteUserSegment[di.useg];
      endPage ← endPage + seg.pages;
      Unlock[seg];
      SwapOut[seg];
      WITH s: seg SELECT FROM
        disk => endHint ← s.hint;
        ENDCASE;
      di ← di.next;
      ENDLOOP;
    END;

  RemoteSegment: PUBLIC SIGNAL [seg: FileSegmentHandle] = CODE;

  MapUserSegment: PUBLIC PROCEDURE [useg: FileSegmentHandle] RETURNS [seg: FileSegmentHandle]=
    -- Return a segment in the debugger space for the given user segment
    BEGIN OPEN SegmentDefs;
    tempseg:  FileSegmentHandle;
    localfp: AltoFileDefs.FP;
    tempseg ← ReadUserSegment[useg];
    CopyRead[
      from: @tempseg.file.fp,
      to: @localfp,
      nwords: SIZE[AltoFileDefs.FP]];
    seg ← NewFileSegment[
      InsertFile[@localfp, Read],tempseg.base,tempseg.pages,Read];
    WITH s: seg SELECT FROM
      disk =>
        s.hint ← WITH t: tempseg SELECT FROM
          disk => t.hint,
          ENDCASE => FileHint[AltoFileDefs.eofDA, 0];
      ENDCASE;
    RETURN
    END;

  InitializeDrum: PUBLIC PROCEDURE =
    BEGIN
    next: DrumItemHandle;
    UNTIL diHead = NIL DO
      next ← diHead.next;
      SegmentDefs.DeleteFileSegment[diHead.dseg];
      SystemDefs.FreeHeapNode[diHead];
      diHead ← next;
      ENDLOOP;
    drumFile ← DDptr.DebuggeeFH;
    endHint ← [AltoFileDefs.eofDA, 0];
    endPage ← 256;  -- after core image
    SegmentDefs.SetEndOfFile[drumFile,endPage+19,AltoDefs.BytesPerPage];
    RETURN
    END;

  ReadCodeByte: PUBLIC PROCEDURE [gframe: GlobalFrameHandle, pc: BytePC]
    RETURNS [BYTE] =
    BEGIN
    iword: InstWord;
    lpc: LONG POINTER;
    patched: BOOLEAN ← FALSE;
    LocateCode[gframe];
    lpc ← gfCache.p.lp+pc/2;
    IF DDptr.onDO AND DDptr.ESV.extension.type = pilot THEN
      [patched, iword] ← CheckPatchTable[lpc];
    IF (gfCache.in OR gfCache.seg = NIL) AND ~patched THEN
      iword ← LongREAD[lpc]
    ELSE
      IF gfCache.seg # NIL THEN
        BEGIN OPEN SegmentDefs;
        useg: FileSegmentHandle = ReadUserSegment[gfCache.seg];
```

```
      WITH useg SELECT FROM
        remote => ERROR RemoteSegment[gfCache.seg];
        ENDCASE;
      SwapIn[gfCache.dseg];
      iword ← (FileSegmentAddress[gfCache.dseg]+gfCache.offset+pc/2)↑;
      Unlock[gfCache.dseg];
      END;
  RETURN[IF pc MOD 2 = 0 THEN iword.evenbyte ELSE iword.oddbyte]
  END;


WriteCodeByte: PUBLIC PROCEDURE [
  gframe: GlobalFrameHandle, pc: BytePC, b: BYTE] =
  BEGIN
  iword: InstWord;
  even: BOOLEAN;
  pi: POINTER TO InstWord;
  co: CodeObject;
  IF DDptr.onDO AND DDptr.ESV.extension.type = pilot THEN
    WritePilotCodeByte[gframe, pc, b];
  even ← pc MOD 2 = 0;
  co ← GFtoCode[gframe];
  IF ~CodeOnDrum[co] THEN MoveToDrum[gframe, co];
  LocateCode[gframe];
  IF gfCache.in OR gfCache.seg = NIL THEN
    BEGIN
    iword ← LongREAD[gfCache.p.lp+pc/2];
    IF even THEN iword.evenbyte ← b ELSE iword.oddbyte ← b;
    LongWRITE[gfCache.p.lp+pc/2, iword];
    END;
  IF gfCache.seg # NIL THEN
    BEGIN OPEN SegmentDefs;
    useg: FileSegmentHandle = ReadUserSegment[gfCache.seg];
    WITH useg SELECT FROM
      remote => ERROR RemoteSegment[gfCache.seg];
      ENDCASE;
    gfCache.dseg.write ← TRUE;
    SwapIn[gfCache.dseg];
    pi ← FileSegmentAddress[gfCache.dseg]+gfCache.offset+pc/2;
    IF even THEN pi.evenbyte ← b ELSE pi.oddbyte ← b;
    Unlock[gfCache.dseg];
    END;
  RETURN
  END;

WritePilotCodeByte: PUBLIC PROCEDURE [
  gframe: GlobalFrameHandle, pc: BytePC, b: BYTE] =
  BEGIN OPEN VMMapLog;
  iword: InstWord;
  even: BOOLEAN = pc MOD 2 = 0;
  lpc: LONG POINTER;
  mempage: CARDINAL;
  patched: BOOLEAN;
  pte: PatchTableEntry;
  pti, ptlimit, ptmaxlimit: PatchTableEntryPointer;
  desc: LONG POINTER TO Descriptor ← DDptr.ESV.mapLog;
  pt: LONG POINTER TO PatchTable;
  lbp: PatchTableEntryBasePointer;
  LongCopyRead[to:@pt, from:@desc.patchTable, nwords: SIZE[LONG POINTER]];
  lbp ← LOOPHOLE[@pt.entries[0]];
  ptlimit ← LongREAD[@pt.limit];
  ptmaxlimit ← LongREAD[@pt.maxLimit];
  LocateCode[gframe];
  lpc ← gfCache.p.lp+pc/2;
  [patched,iword] ← CheckPatchTable[lpc];
  IF ~patched THEN iword ← LongREAD[lpc];
  IF even THEN iword.evenbyte ← b ELSE iword.oddbyte ← b;
  FOR pti ← FIRST[PatchTableEntryPointer], pti+SIZE[PatchTableEntry]
    UNTIL pti = ptlimit DO
    LongCopyRead[to:@pte, from:@lbp[pti], nwords:SIZE[PatchTableEntry]];
    IF pte.address = lpc THEN EXIT;
    REPEAT FINISHED =>
      BEGIN
      IF ptlimit = ptmaxlimit THEN ERROR DebugMiscDefs.CommandNotAllowed;
      LongWRITE[@pt.limit, ptlimit+SIZE[PatchTableEntry]];
      pte.address ← lpc;
```

```
      END;
    ENDLOOP;
  pte.value ← iword;
  LongCopyWrite[from:@pte, to:@lbp[pti], nwords:SIZE[PatchTableEntry]];
  mempage ← InlineDefs.LongDiv[LOOPHOLE[lpc], AltoDefs.PageSize];
  IF DebugCacheDefs.SwappedIn[mempage] THEN
    DebugCacheDefs.LongWRITEClean[lpc, iword];
  RETURN
  END;

CheckPatchTable: PUBLIC PROCEDURE [lp: LONG POINTER]
  RETURNS [BOOLEAN, InstWord] =
  BEGIN OPEN VMMapLog;
  pte: PatchTableEntry;
  pti, ptlimit: PatchTableEntryPointer;
  desc: LONG POINTER TO Descriptor ← DDptr.ESV.mapLog;
  pt: LONG POINTER TO PatchTable;
  lbp: PatchTableEntryBasePointer;
  LongCopyRead[to:@pt, from:@desc.patchTable, nwords: SIZE[LONG POINTER]];
  lbp ← LOOPHOLE[@pt.entries[0]];
  ptlimit ← LongREAD[@pt.limit];
  FOR pti ← FIRST[PatchTableEntryPointer], pti+SIZE[PatchTableEntry] UNTIL pti = ptlimit DO
    LongCopyRead[to:@pte, from:@lbp[pti], nwords:SIZE[PatchTableEntry]];
    IF pte.address = lp THEN RETURN[TRUE, pte.value];
    ENDLOOP;
  RETURN[FALSE,[0,0]]
  END;

COCacheObject: TYPE = RECORD [
  gf: GlobalFrameHandle,
  code: CodeObject];

coCache: COCacheObject ← [NIL,];

GFtoCode: PUBLIC PROCEDURE [f: GlobalFrameHandle] RETURNS [CodeObject] =
  BEGIN OPEN LoadStateDefs, coCache;
  cgfi: GFTIndex;
  bcdseg: FileSegmentHandle;
  bcd: LoaderBcdUtilDefs.BcdBase;

  FindModuleSeg: PROCEDURE [mth: BcdDefs.MTHandle, mti: BcdDefs.MTIndex]
    RETURNS [BOOLEAN] =
    BEGIN
    IF cgfi IN[mth.gfi..mth.gfi+mth.ngfi) THEN
      BEGIN code.seg ← mth.code.sgi; RETURN[TRUE]; END;
    RETURN[FALSE];
    END;

  BEGIN OPEN DebugContextDefs, LoaderBcdUtilDefs;
  IF coCache.gf = f THEN RETURN[code];
  [] ← InputLoadState[ ! LoadStateInvalid => GOTO noContext];
  [cgfi,code.config] ← MapRC[
    IF VirtualGlobalFrame[f].copied THEN FindOriginal[f] ELSE f];
  IF code.config = ConfigNull THEN ERROR InvalidGlobalFrame[f];
  IF code.config # DDptr.config OR DDptr.initBCD THEN
    BEGIN
    bcd ← SetUpBcd[bcdseg ← BcdSegFromLoadState[code.config]];
    [] ← EnumerateModuleTable[bcd, FindModuleSeg];
    ReleaseBcdSeg[bcdseg];
    END
  ELSE
    BEGIN
    bcd ← DAcquireBcd[];
    [] ← EnumerateModuleTable[bcd, FindModuleSeg];
    DReleaseBcd[];
    END;
  ReleaseLoadState[];
  EXITS
    noContext => ERROR DebugMiscDefs.CommandNotAllowed;
  END;
  coCache.gf ← f;
  RETURN[code]
  END;


FrameCacheObject: TYPE = RECORD [
```

```
    gf: GlobalFrameHandle,
    seg: FileSegmentHandle,
    p: LA,
    in: BOOLEAN,
    offset: CARDINAL,
    dseg: FileSegmentHandle];

  gfCache: FrameCacheObject ← [NIL,,,,,];

  FlushCodeCache: PROCEDURE =
    BEGIN
    IF gfCache.gf # NIL AND gfCache.dseg # NIL THEN
      SegmentDefs.DeleteFileSegment[gfCache.dseg];
    gfCache.gf ← NIL;
    RETURN;
    END;

  FlushCodeSegmentCache: PUBLIC PROCEDURE =
    BEGIN
    FlushCodeCache[];
    coCache.gf ← NIL;
    RETURN;
    END;

  -- copied from GlobalFrameDefs.mesa

  GlobalFrame: TYPE = MACHINE DEPENDENT RECORD [
    gfi: [0..777B],
    unused: [0..1], -- reserved for future gfi expansion
    copied, alloced, shared, started: BOOLEAN,
    trapxfers, codelinks: BOOLEAN,
    code: FrameCodeBase,
    global: ARRAY [0..0) OF UNSPECIFIED];

  FrameCodeBase: TYPE = MACHINE DEPENDENT RECORD [
    SELECT OVERLAID * FROM
      in => [
        SELECT OVERLAID * FROM
          codebase => [
            codebase: LONG POINTER],
          shortCodebase => [
            shortCodebase: UNSPECIFIED,
            highHalf: CARDINAL],
          ENDCASE],
      out => [
        offset: CARDINAL,
        handle: POINTER],
      either => [
        fill1: [0..77777B],
        swappedout: BOOLEAN,
        highByte, topByteOfLongPointer: [0..377B]],
      ENDCASE];

  CodeFile: PUBLIC PROCEDURE [f: GlobalFrameHandle] RETURNS [FileHandle] =
    BEGIN
    co: CodeObject ← GFtoCode[f];
    di: DrumItemHandle;
    fp: AltoFileDefs.FP;
    LocateCode[f];
    IF gfCache.dseg = NIL THEN RETURN[NIL];
    FOR di ← diHead, di.next UNTIL di = NIL DO
      IF di.co = co THEN
        BEGIN OPEN SegmentDefs;
        IF di.oldFile = NIL THEN RETURN[NIL];
        CopyRead[from: @di.oldFile.fp, to: @fp,
          nwords: SIZE[AltoFileDefs.FP]];
        RETURN[InsertFile[@fp, Read]]
        END;
      ENDLOOP;
    RETURN[gfCache.dseg.file]
    END;

  LocateCode: PROCEDURE [f: GlobalFrameHandle] =
    BEGIN OPEN SegmentDefs, gfCache;
    gf: GlobalFrame;
    IF gfCache.gf = f THEN RETURN;
```

```
    FlushCodeCache[];
    gfCache.gf ← f;
    in ← TRUE;
    p ← LOOPHOLE[InlineDefs.LongMult[DDptr.mdsContext, AltoDefs.PageSize]];
    seg ← NIL;
    offset ← 0;
    CopyRead[from: f, to: @gf, nwords: SIZE[GlobalFrame]];
    IF gf.code.swappedout THEN gf.code.swappedout ← in ← FALSE;
    IF gf.code.highByte # 0 THEN
      BEGIN
      seg ← gf.code.handle;
      IF in THEN
        BEGIN
        p.low ← gf.code.shortCodebase;
        offset ← gf.code.shortCodebase - LOOPHOLE[AddressFromPage[ReadUserSegment[seg].VMpage], CARDINA
**L];
        END
      ELSE offset ← gf.code.offset;
      END
    ELSE
      BEGIN
      table: BootDefs.SystemTableHandle = DDptr.ESV.tables;
      p.lp ← gf.code.codebase;
      IF table # NIL AND gf.code.topByteOfLongPointer = 0 THEN
        BEGIN
        pagemap: POINTER TO BootDefs.PageMap ← SREAD[@table.pagemap];
        page: CARDINAL = PageFromAddress[gf.code.shortCodebase];
        seg ← SREAD[@pagemap[page]];
        in ← TRUE;
        offset ←
          gf.code.shortCodebase - LOOPHOLE[AddressFromPage[ReadUserSegment[seg].VMpage], CARDINAL];
        END;
      END;
    dseg ← IF seg # NIL THEN MapUserSegment[seg] ELSE NIL;
    END;


END...
```