```
-- MakeImage.Mesa  Edited by Sandman on October 6, 1977  5:35 PM

DIRECTORY
  AltoDefs: FROM "altodefs",
  AltoFileDefs: FROM "altofiledefs",
  BcdDefs: FROM "BcdDefs",
  BcdMergeDefs: FROM "BcdMergeDefs",
  BcdTabDefs: FROM "bcdtabdefs",
  BcdTableDefs: FROM "bcdtabledefs",
  BFSDefs: FROM "bfsdefs",
  BootDefs: FROM "bootdefs",
  ControlDefs: FROM "controldefs",
  CoreSwapDefs: FROM "coreswapdefs",
  DirectoryDefs: FROM "directorydefs",
  DiskDefs: FROM "diskdefs",
  DiskKDDefs: FROM "diskkddefs",
  FrameDefs: FROM "framedefs",
  ImageDefs: FROM "imagedefs",
  InlineDefs: FROM "inlinedefs",
  LoaderBcdUtilDefs: FROM "LoaderBcdUtilDefs",
  LoadStateDefs: FROM "LoadStateDefs",
  MakeImageUtilDefs: FROM "makeimageutildefs",
  MiscDefs: FROM "miscdefs",
  OsStaticDefs: FROM "osstaticdefs",
  ProcessDefs: FROM "processdefs",
  SegmentDefs: FROM "segmentdefs",
  StreamDefs: FROM "streamdefs",
  StringDefs: FROM "stringdefs",
  SystemDefs: FROM "systemdefs",
  TimeDefs: FROM "timedefs";

DEFINITIONS FROM
  LoadStateDefs, DiskDefs, ImageDefs, ControlDefs, SegmentDefs, MakeImageUtilDefs;

MakeImage: PROGRAM
  IMPORTS BcdMergeDefs, BcdTabDefs, BcdTableDefs, BFSDefs, BootDefs, CoreSwapDefs,
    DirectoryDefs, DiskDefs, DiskKDDefs, FrameDefs, ImageDefs, LoaderBcdUtilDefs,
    LoadStateDefs, MiscDefs, SegmentDefs, StreamDefs, StringDefs, SystemDefs,
    MakeImageUtilDefs
  EXPORTS ImageDefs SHARES ProcessDefs, DiskDefs, SegmentDefs, ControlDefs, ImageDefs =
  BEGIN

  CFA: TYPE = AltoFileDefs.CFA;
  DataSegmentHandle: TYPE = SegmentDefs.DataSegmentHandle;
  FP: TYPE = AltoFileDefs.FP;
  FileHandle: TYPE = SegmentDefs.FileHandle;
  FileSegmentHandle: TYPE = SegmentDefs.FileSegmentHandle;
  PageCount: TYPE = AltoDefs.PageCount;
  PageNumber: TYPE = AltoDefs.PageNumber;
  ProcessRegister: TYPE = ProcessDefs.ProcessRegister;
  ProcessHandle: TYPE = ProcessDefs.ProcessHandle;
  ProcessVector: TYPE = ProcessDefs.ProcessVector;
  shortFileRequest: TYPE = short ImageDefs.FileRequest;
  vDA: TYPE = AltoFileDefs.vDA;
  GlobalFrameHandle: TYPE = ControlDefs.GlobalFrameHandle;
  LoadStateGFT: TYPE = LoadStateDefs.LoadStateGFT;
  ConfigIndex: TYPE = LoadStateDefs.ConfigIndex;
  StreamHandle: TYPE = StreamDefs.StreamHandle;
  ProcDesc: TYPE = ControlDefs.ProcDesc;

-- Bcd Merging Management

MergeAllBcds: PROCEDURE [initialgft: LoadStateGFT, code, symbols: BOOLEAN,
  names: DESCRIPTOR FOR ARRAY OF STRING] =
  BEGIN OPEN LoadStateDefs, BcdMergeDefs;
  MergeLoadedBcds: PROCEDURE [config: ConfigIndex, addr: BcdAddress] RETURNS [BOOLEAN] =
    BEGIN OPEN LoaderBcdUtilDefs, LoadStateDefs;
    rel: Relocation ← InitializeRelocation[config];
    bcdseg: FileSegmentHandle ← BcdSegFromLoadState[config];
    bcd: BcdBase ← SetUpBcd[bcdseg];
    MergeBcd[bcd, rel, 0, initialgft, code, symbols, names[config]];
    ReleaseBcdSeg[bcdseg];
    ReleaseRelocation[rel];
    RETURN [FALSE];
    END;
  MergeCopiedFrames: PROCEDURE [frame: GlobalFrameHandle] RETURNS [BOOLEAN] =
    BEGIN
    copied: GlobalFrameHandle;
    config: ConfigIndex;
    ModuleCopiedFrom: PROCEDURE [f: GlobalFrameHandle] RETURNS [BOOLEAN] =
      BEGIN
      RETURN [f # frame AND f.codesegment = frame.codesegment AND
        (config ← MapRealToConfig[f.gftindex.gftindex].config) # ConfigNull];
      END;
    IF MapRealToConfig[frame.gftindex.gftindex].config # ConfigNull THEN RETURN [FALSE];
    IF (copied←FrameDefs.EnumerateGlobalFrames[ModuleCopiedFrom]) # NULLFrame THEN
      BEGIN
      MergeModule[frame, copied, initialgft];
      RETURN [FALSE];
      END;
    RETURN [FALSE];
    END;

  InitializeMerge[TableSize, NumberGFIInConfig[initialgft, 0]];
  [] ← EnumerateLoadStateBcds[recentlast, MergeLoadedBcds];
  [] ← FrameDefs.EnumerateGlobalFrames[MergeCopiedFrames];
  [] ← MergedBcdSize[];
  WriteMergedBcd[MoveWords];
  FinalizeMerge[];
  END;

MergeABcd: PROCEDURE [config: ConfigIndex, initgft: LoadStateGFT, code, symbols: BOOLEAN,
  names: DESCRIPTOR FOR ARRAY OF STRING] =
  BEGIN OPEN BcdMergeDefs, LoaderBcdUtilDefs, LoadStateDefs;
  rel: Relocation ← InitializeRelocation[config];
  bcdseg: FileSegmentHandle ← BcdSegFromLoadState[config];
  bcd: BcdBase ← SetUpBcd[bcdseg];
  InitializeMerge[bcdseg.pages+1, NumberGFIInConfig[initgft, config]];
  MergeBcd[bcd, rel, config, initgft, code, symbols, names[config]];
  ReleaseBcdSeg[bcdseg];
  ReleaseRelocation[rel];
  [] ← MergedBcdSize[];
  WriteMergedBcd[MoveWords];
  FinalizeMerge[];
  END;

MoveWords: PROCEDURE [source: POINTER, nwords: CARDINAL] =
  BEGIN
  IF nwords # StreamDefs.WriteBlock[stream: bcdstream, address: source, words: nwords]
    THEN ERROR;
  END;

NewBcdSegmentFromStream: PROCEDURE [stream: StreamDefs.DiskHandle, page: PageNumber]
  RETURNS [newpage: PageNumber, seg: FileSegmentHandle] =
  BEGIN
  index: StreamDefs.StreamIndex;
  index ← StreamDefs.GetIndex[stream];
  IF index.byte # 0 THEN
    BEGIN
    index.byte ← 0;
    index.page ← index.page + 1;
    StreamDefs.SetIndex[stream, index];
    END;
  seg ← NewFileSegment[stream.file, page+1, index.page-page, Read+Write];
  seg.class ← bcd;
  maxbcdsize ← MAX[maxbcdsize, seg.pages];
  newpage ← index.page;
  RETURN
  END;

MapSegmentsInBcd: PROCEDURE [
 initialGFT: LoadStateGFT, config: ConfigIndex, bcdseg: FileSegmentHandle]
  RETURNS [unresolved, exports: BOOLEAN] =
  BEGIN OPEN LoaderBcdUtilDefs, LoadStateDefs;
  bcd: BcdBase ← SetUpBcd[bcdseg];
  MapSegments: PROCEDURE [mtb: CARDINAL, mti: BcdDefs.MTIndex] RETURNS [BOOLEAN] =
    BEGIN OPEN m: mtb+mti;
    frame: GlobalFrameHandle;
    rgfi: GFTIndex;
    FOR rgfi IN [0..(REGISTER[SDreg]+sGFTLength)↑) DO
      IF initialGFT[rgfi] = [config: config, gfi: m.gfi] THEN EXIT;
      ENDLOOP;
    IF m.cseg.file = BcdDefs.FTSelf THEN
      BEGIN
      frame ← LOOPHOLE[REGISTER[GFTreg], gftp]↑[rgfi].frame;
      m.cseg.base ← frame.codesegment.base;
      END;
    IF m.sseg.file = BcdDefs.FTSelf THEN
      BEGIN
      frame ← LOOPHOLE[REGISTER[GFTreg], gftp]↑[rgfi].frame;
      IF frame.symbolsegment # NIL THEN m.sseg.base ← frame.symbolsegment.base;
      END;
    RETURN[FALSE];
    END;
  [] ← EnumerateModuleTable[bcd, MapSegments];
  unresolved ← bcd.nImports # 0;
  exports ← bcd.nExports # 0;
  Unlock[bcdseg];
  SwapOut[bcdseg];
  END;

TableSize: CARDINAL = 15*AltoDefs.PageSize;
bcdstream: StreamDefs.DiskHandle;
maxbcdsize: CARDINAL ← 0;

DisplayHeader: POINTER TO WORD = LOOPHOLE[420B];
DIW: POINTER TO WORD = LOOPHOLE[421B];

SwapTrapDuringMakeImage: PUBLIC SIGNAL = CODE;

SwapErrorDuringMakeImage: PUBLIC SIGNAL = CODE;

SwapOutDuringMakeImage: PUBLIC SIGNAL = CODE;

SwapTrapError: PROCEDURE [dest: ControlDefs.ControlLink] =
  BEGIN
  s: ControlDefs.StateVector;
  s ← STATE;
  SIGNAL SwapTrapDuringMakeImage;
  RETURN WITH s;
  END;

SwapOutError: SegmentDefs.SwappingProcedure =
  BEGIN
  SIGNAL SwapOutDuringMakeImage;
  RETURN[TRUE];
  END;

-- File Segment Transfer Routines

bufferseg: DataSegmentHandle;
buffer: POINTER;
BufferPages: PageCount;

SwapDR: TYPE = POINTER TO swap DiskRequest;

TransferPages: PROCEDURE [
 da: vDA, base: PageNumber, pages: PageCount, fp: POINTER TO FP, sdr: SwapDR]
  RETURNS [next: vDA] =
  BEGIN OPEN DiskDefs;
  sdr.da ← @da;
  sdr.firstPage ← base;
  sdr.lastPage ← base+pages-1;
  sdr.fp ← fp;
  IF SwapPages[sdr].page # base+pages-1 THEN SIGNAL SwapErrorDuringMakeImage;
  next ← sdr.desc.next;
  RETURN[next];
  END;

TransferFileSegment: PROCEDURE [
 buffer: POINTER, seg: FileSegmentHandle, file: FileHandle, base: PageNumber, fileda: vDA]
  RETURNS [vDA] =
  BEGIN
  dpd: DiskPageDesc;
  sdr: swap DiskRequest;
  old: FileHandle ← seg.file;
  segbase: PageNumber ← seg.base;
  pages: PageCount ← seg.pages;
  segda: vDA ← seg.hint.da;
  seg.base ← base;
  sdr ← [ca: buffer, da:, firstPage:, lastPage:, fp:, fixedCA: FALSE, action:,
    lastAction:, signalCheckError: FALSE, option: swap[desc: @dpd]];
  IF seg.swappedin THEN
    BEGIN
    sdr.ca ← SegmentDefs.AddressFromPage[seg.VMpage];
    sdr.action ← sdr.lastAction ← WriteD;
    fileda ← TransferPages[fileda, base, pages, @file.fp, @sdr];
    old.swapcount ← old.swapcount - 1;
    file.swapcount ← file.swapcount + 1;
    END
  ELSE
    BEGIN
    WHILE BufferPages < pages DO
      pages ← pages - BufferPages;
      sdr.action ← sdr.lastAction ← ReadD;
      segda ← TransferPages[segda, segbase, BufferPages, @old.fp, @sdr];
      sdr.action ← sdr.lastAction ← WriteD;
      fileda ← TransferPages[fileda, base, BufferPages, @file.fp, @sdr];
      segbase ← segbase + BufferPages;
      base ← base + BufferPages;
      ENDLOOP;
    sdr.action ← sdr.lastAction ← ReadD;
    segda ← TransferPages[segda, segbase, pages, @old.fp, @sdr];
    sdr.action ← sdr.lastAction ← WriteD;
    fileda ← TransferPages[fileda, base, pages, @file.fp, @sdr];
    END;
  old.segcount ← old.segcount - 1;
  seg.file ← file;
  seg.hint ← FileHint[AltoFileDefs.eofDA, 0];
  file.segcount ← file.segcount + 1;
  IF old.segcount = 0 THEN ReleaseFile[old];
  RETURN [fileda];
  END;

EnumerateNeededModules: PROCEDURE [proc: PROCEDURE [ProcDesc]] =
  BEGIN
  proc[LOOPHOLE[EnumerateNeededModules]];
  proc[LOOPHOLE[MakeImageUtilDefs.AddFileRequest]];
  proc[LOOPHOLE[BFSDefs.ActOnPages]];
  proc[LOOPHOLE[SegmentDefs.MapFileSegment]];
  proc[LOOPHOLE[SegmentDefs.CloseFile]];
  proc[LOOPHOLE[DiskKDDefs.CloseDiskKD]];
  proc[LOOPHOLE[ImageDefs.UserCleanupProc]];
  proc[LOOPHOLE[DirectoryDefs.EnumerateDirectory]];
  proc[LOOPHOLE[StreamDefs.CreateWordStream]];
  proc[LOOPHOLE[StringDefs.EquivalentString]];
  proc[LOOPHOLE[LoadStateDefs.InputLoadState]];
  END;

SwapOutMakeImageCode: PROCEDURE =
  BEGIN OPEN FrameDefs;
  SwapOutCode[GlobalFrame[MakeImageUtilDefs.AddFileRequest]];
  SwapOutCode[GlobalFrame[BcdTableDefs.Allocate]];
  SwapOutCode[GlobalFrame[BcdTabDefs.FindString]];
  SwapOutCode[GlobalFrame[LoaderBcdUtilDefs.EnumerateModuleTable]];
  SwapOutCode[GlobalFrame[LoadStateDefs.InputLoadState]];
  SwapOutCode[GlobalFrame[BcdMergeDefs.MergeBcd]];
  END;
```

```
InstallImage: PROCEDURE [name: STRING, merge, code, symbols: BOOLEAN] =
  BEGIN OPEN AltoFileDefs, DiskDefs;
  AV: POINTER = REGISTER[AVreg];
  SD: POINTER TO ARRAY [0..256) OF UNSPECIFIED = REGISTER[SDreg];
  GFT: POINTER = REGISTER[GFTreg];
  WDC: CARDINAL;
  diskrequest: DiskRequest;
  lpn: PageNumber; numChars: CARDINAL;
  savealloctrap, savealloc, saveswaptrap: ControlLink;
  auxtrapFrame: FrameHandle;
  nextpage: PageNumber;
  swappedinfilepages, swappedoutfilepages, datapages: PageCount ← 0;
  SwapOutErrorStrategy: SegmentDefs.SwapStrategy ← [link:,proc:SwapOutError];
  mapindex: MapIndexType ← 0;
  maxFileSegPages: CARDINAL ← 0;
  endofdatamapindex: MapIndexType;
  ptSeg: DataSegmentHandle;
  HeaderSeg: DataSegmentHandle;
  Image: POINTER TO ImageHeader;
  imageDA, HeaderDA: vDA;
  ImageFile: FileHandle;
  diskKD: FileSegmentHandle;
  saveAP, saveRP: ProcessRegister;
  saveDIW: WORD;
  savePV: ProcessVector;
  bcdstreampage: PageNumber;
  bcdnames: DESCRIPTOR FOR ARRAY OF STRING;
  bcds: DESCRIPTOR FOR ARRAY OF BcdItem;
  BcdItem: TYPE = RECORD [
    bcdseg: FileSegmentHandle,
    unresolved, exports: BOOLEAN];
  con, nbcds: ConfigIndex;
  time: AltoFileDefs.TIME;
  initgft: LoadStateGFT;
  net: CARDINAL ← MiscDefs.GetNetworkNumber[];

  SaveProcesses: PROCEDURE =
    BEGIN OPEN ProcessDefs;
    saveAP ← AP↑;
    saveRP ← RP↑;
    saveDIW ← DIW↑;
    savePV ← PV↑;
    DIW↑ ← 2;
    WakeupsWaiting↑ ← 0;
    END;
  RestoreProcesses: PROCEDURE =
    BEGIN OPEN ProcessDefs;
    ActiveWord↑ ← AP↑ ← saveAP;
    WakeupsWaiting↑ ← RP↑ ← saveRP;
    DIW↑ ← saveDIW;
    PV↑ ← savePV;
    END;
  EnterMapItem: PROCEDURE [vmpage: PageNumber, pages: PageCount] =
    BEGIN
    Image.map[mapindex] ← MapItem[vmpage,pages];
    mapindex ← mapindex + 1;
    END;
  CountFileSegments: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF ~symbols AND s.class=symbols THEN RETURN[FALSE];
    IF s # diskKD THEN
      BEGIN
      [] ← BootDefs.PositionSeg[s, FALSE];
      IF s.swappedin THEN
        BEGIN
        swappedinfilepages ← swappedinfilepages + s.pages;
        IF s.class=code THEN
          maxFileSegPages ← MAX[maxFileSegPages, s.pages];
        END
      ELSE
        BEGIN
        swappedoutfilepages ← swappedoutfilepages + s.pages;
        maxFileSegPages ← MAX[maxFileSegPages, s.pages];
        END
      END;
    RETURN[FALSE];
    END;
  CountDataSegments: PROCEDURE [s: DataSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF s # bufferseg THEN datapages ← datapages + s.pages;
    RETURN[FALSE];
    END;
  MapDataSegments: PROCEDURE [s: DataSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF s # HeaderSeg AND s # bufferseg THEN
      BEGIN
      EnterMapItem[s.VMpage, s.pages];
      nextpage ← nextpage + s.pages;
      END;
    RETURN[FALSE];
    END;
  WriteSwappedIn: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF s.swappedin THEN
      BEGIN
      imageDA ← TransferFileSegment[buffer, s, ImageFile, nextpage, imageDA];
      EnterMapItem[s.VMpage, s.pages];
      nextpage ← nextpage + s.pages;
      END;
    RETURN[FALSE];
    END;
  WriteSwappedOutBcd: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF ~s.swappedin AND s.class = bcd THEN
      BEGIN
      imageDA ← TransferFileSegment[buffer, s, ImageFile, nextpage, imageDA];
      nextpage ← nextpage + s.pages;
      END;
    RETURN[FALSE];
    END;
  WriteSwappedOutCode: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF ~s.swappedin AND s.class = code THEN
      BEGIN
      imageDA ← TransferFileSegment[buffer, s, ImageFile, nextpage, imageDA];
      nextpage ← nextpage + s.pages;
      END;
    RETURN[FALSE];
    END;
  WriteSwappedOutNonCode: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF ~symbols AND s.class=symbols THEN RETURN[FALSE];
    IF ~s.swappedin AND s.class # code AND s.class # bcd AND s # diskKD THEN
      BEGIN
      imageDA ← TransferFileSegment[buffer, s, ImageFile, nextpage, imageDA];
      nextpage ← nextpage + s.pages;
      END;
    RETURN[FALSE];
    END;
  SaveBcd: PROCEDURE [config: ConfigIndex, addr: BcdAddress] RETURNS [BOOLEAN] =
    BEGIN
    bcds[config].bcdseg ← LoadStateDefs.BcdSegFromLoadState[config];
    RETURN [FALSE];
    END;

  SD[sAddFileRequest] ← AddFileRequest;
  ImageFile ← NewFile[name, Read+Write+Append, DefaultVersion];
  diskKD ← KDSegment[];
  ProcessDefs.DisableInterrupts[];
  WDC ← ControlDefs.ReadWDC[];
  CoreSwapDefs.SetLevel[-1];
  SaveProcesses[];
  ImageDefs.UserCleanupProc[Save];

-- handle bcds

  initgft ← DESCRIPTOR[SystemDefs.AllocateSegment[SD[sGFTLength]], SD[sGFTLength]];
  bcdstream ← StreamDefs.NewWordStream["makeimage.scratch$", Read+Write+Append];
  nbcds ← LoadStateDefs.InputLoadState[];  -- bring it in for first time
  bcdnames ← GetBcdFileNames[nbcds];
  nbcds ← IF merge THEN 1 ELSE nbcds;
```

```
   bcds ← DESCRIPTOR[GetSpace[nbcds*SIZE[BcdItem]], nbcds];
   bcdstreampage ← 0;
   InitLoadStateGFT[initgft, merge, nbcds];
   IF merge THEN
     BEGIN
     MergeAllBcds[initgft, code, symbols, bcdnames];
     [bcdstreampage, bcds[0].bcdseg] ← NewBcdSegmentFromStream[bcdstream, bcdstreampage];
     END
   ELSE
     BEGIN
     [] ← LoadStateDefs.EnumerateLoadStateBcds[recentlast, SaveBcd];
     FOR con IN [0..nbcds) DO
       MergeABcd[con, initgft, code, symbols, bcdnames];
       [bcdstreampage, bcds[con].bcdseg] ←
         NewBcdSegmentFromStream[bcdstream, bcdstreampage];
       ENDLOOP;
     END;
   bcdstream.destroy[bcdstream];
-- LoadStateDefs.ReleaseLoadState[];
   IF merge THEN PatchUpGFT[];

   [] ← SystemDefs.PruneHeap[];

   SetupAuxStorage[];
   EnumerateNeededModules[LockCodeSegment];
   HeaderDA ← DAofPage[ImageFile, 1];
   -- [] ← FrameDefs.EnumerateGlobalFrames[SwapOutUnlockedCode];
   -- [] ← EnumerateFileSegments[SwapOutUnlocked];

   -- set up private frame allocation trap
   ControlDefs.Free[ControlDefs.Alloc[0]]; -- flush large frames
   savealloctrap ← SD[sAllocListEmpty];
   SD[sAllocListEmpty] ← auxtrapFrame ← auxtrap[];
   savealloc ← SD[sAlloc]; SD[sAlloc] ← myalloc;
   REGISTER[AVreg] ← DataSegmentAddress[AuxSeg];

   BufferPages ← maxbcdsize+LoadStateDefs.GetLoadState[].pages;
   bufferseg ← NewDataSegment[DefaultBase, BufferPages];
   [] ← EnumerateDataSegments[CountDataSegments];
   swappedinfilepages ← swappedoutfilepages ← 0;
   [] ← EnumerateFileSegments[CountFileSegments];
   SetEndOfFile[ImageFile,
     datapages + swappedinfilepages + swappedoutfilepages + FirstImageDataPage - 1,
     AltoDefs.BytesPerPage];
   [] ← DiskKDDefs.CloseDiskKD[];

   HeaderSeg ← NewDataSegment[DefaultBase, 1];
   Image ← DataSegmentAddress[HeaderSeg];
   MiscDefs.Zero[Image,SIZE[ImageHeader]];
   Image.versionident ← ImageDefs.VersionID; Image.options ← 0;
--Image.state.stk[0] ← Image.state.stk[1] ← 0;
   Image.state.stkptr ← 2;
   Image.state.X ← REGISTER[Lreg];
   Image.av ← AV;
   Image.gft ← GFT;
   Image.sd ← SD;
   time ← MiscDefs.DAYTIME[];
   Image.version ← BcdDefs.VersionStamp[
     time: TimeDefs.PackedTime[lowbits: time.low, highbits: time.high],
     zapped: FALSE,
     net: net,
     host: OsStaticDefs.OsStatics.SerialNumber];
   Image.creator ← ImageDefs.ImageVersion[]; -- version stamp of currently running image

   nextpage ← FirstImageDataPage;
   [] ← EnumerateDataSegments[MapDataSegments];
   IF nextpage # FirstImageDataPage+datapages THEN ERROR;
   endofdatamapindex ← mapindex;

   -- now disable swapping
   saveswaptrap ← SD[sCsegSwappedOut];
   SD[sCsegSwappedOut] ← SwapTrapError;
   AddSwapStrategy[@SwapOutErrorStrategy];
   imageDA ← DAofPage[ImageFile, nextpage];
   buffer ← DataSegmentAddress[bufferseg];
   [] ← EnumerateFileSegments[WriteSwappedIn];
```

```
  IF nextpage # FirstImageDataPage+datapages+swappedinfilepages THEN ERROR;
  [] ← EnumerateFileSegments[WriteSwappedOutBcd];
  [] ← EnumerateFileSegments[WriteSwappedOutCode];
  [] ← EnumerateFileSegments[WriteSwappedOutNonCode];
  DeleteDataSegment[bufferseg];

  SegmentDefs.CloseFile[ImageFile ! SegmentDefs.FileError => RESUME];
  ImageFile.write ← ImageFile.append ← FALSE;

--  [] ← LoadStateDefs.InputLoadState[];
  FOR con IN [0..nbcds) DO
    [bcds[con].unresolved, bcds[con].exports] ← MapSegmentsInBcd[initgft, con, bcds[con].bcdseg];
    ENDLOOP;
--  LoadStateDefs.ReleaseLoadState[];

  diskrequest ← DiskRequest[
    ca: auxalloc[datapages+3],
    da: auxalloc[datapages+3],
    fixedCA: FALSE,
    fp: auxalloc[SIZE[FP]],
    firstPage: FirstImageDataPage-1,
    lastPage: FirstImageDataPage+datapages-1,
    action: WriteD,
    lastAction: WriteD,
    signalCheckError: FALSE,
    option: update[BFSDefs.GetNextDA]];

  diskrequest.fp↑ ← ImageFile.fp;
  [] ← SegmentDefs.EnumerateFileSegments[BashHint];
  [] ← SegmentDefs.EnumerateFiles[BashFile];
  (diskrequest.ca+1)↑ ← Image;
  FillInCAs[Image, endofdatamapindex, diskrequest.ca+2];
  MiscDefs.SetBlock[diskrequest.da,fillinDA,datapages+3];
  (diskrequest.da+1)↑ ← HeaderDA;

  [lpn,numChars] ← BFSDefs.ActOnPages[LOOPHOLE[@diskrequest]];
  IF lpn # 0 OR numChars # 0 THEN
    BEGIN
    DisplayHeader↑ ← SD[sGoingAway] ← 0;
    ImageDefs.StopMesa[];
    END;
  REGISTER[AVreg] ← AV;
  REGISTER[SDreg] ← SD;
  REGISTER[GFTreg] ← GFT;
  ControlDefs.WriteWDC[WDC];
  SD[sAllocListEmpty] ← savealloctrap;
  SD[sAlloc] ← savealloc;
  SD[ControlDefs.sAddFileRequest] ← 0;
  Free[auxtrapFrame];
  DeleteDataSegment[HeaderSeg];
  ptSeg ← NewDataSegment[PageFromAddress[ptPointer↑],1];
  [] ← DiskDefs.ResetDisk[];
  DiskKDDefs.InitializeDiskKD[];
  BootPageTable[ImageFile, ptPointer↑];
  DeleteDataSegment[ptSeg];

  -- turn swapping back on
  RemoveSwapStrategy[@SwapOutErrorStrategy];
  SD[sCsegSwappedOut] ← saveswaptrap;

  RestoreProcesses[];
  ProcessDefs.EnableInterrupts[];
  ProcessFileRequests[];

--  [] ← LoadStateDefs.InputLoadState[];
  LoadStateDefs.ResetLoadState[initgft];
  FOR con IN [0..nbcds) DO
    LoadStateDefs.UpdateLoadState[
      con, bcds[con].bcdseg, bcds[con].unresolved, bcds[con].exports];
    DeleteFileSegment[bcds[con].bcdseg];
    ENDLOOP;
  LoadStateDefs.ReleaseLoadState[];
  SystemDefs.FreeSegment[BASE[initgft]];
  DeleteDataSegment[AuxSeg];

  FreeAllSpace[];
  EnumerateNeededModules[UnlockCodeSegment];
  SwapOutMakeImageCode[];
  ImageDefs.UserCleanupProc[Restore];
  RETURN
  END;

-- auxillary storage for frames and non-saved items
AuxSeg: DataSegmentHandle;
freepointer: POINTER;
wordsleft: CARDINAL;

SetupAuxStorage: PROCEDURE =
  BEGIN
  av : POINTER;
  i: CARDINAL;
  AuxSeg ← NewDataSegment[DefaultBase,10];
  av ← freepointer ← DataSegmentAddress[AuxSeg];
  wordsleft ← 10*AltoDefs.PageSize;
  [] ← auxalloc[AllocationVectorSize];
  freepointer ← freepointer+3; wordsleft ← wordsleft-3;
  FOR i IN [0..maxallocslot) DO
    (av+i)↑ ← (i+1)*4+2;
    ENDLOOP;
  (av+6)↑ ← (av+maxallocslot)↑ ← (av+maxallocslot+1)↑ ← 1;
  END;

auxalloc: PROCEDURE [n: CARDINAL] RETURNS [p: POINTER] =
  BEGIN -- allocate in multiples of 4 words
  p ← freepointer;
  n ← InlineDefs.BITAND[n+3,177774B];
  freepointer ← freepointer+n;
  IF wordsleft < n THEN ImageDefs.PuntMesa[];
  wordsleft ← wordsleft-n;
  RETURN
  END;

myalloc: PROCEDURE [n: CARDINAL] RETURNS [p: POINTER] =
  BEGIN -- replaces the normal alloc procedure
  IF n >= maxallocslot THEN ImageDefs.PuntMesa[];
  p ← auxalloc[FrameDefs.FrameSize[n]] + 1;
  (p-1)↑ ← n;
  RETURN
  END;

framevec: ARRAY [0..18] OF INTEGER =
    [7,11,15,19,23,27,31,39,47,55,67,79,95,111,127,147,171,199,231];

pgft: TYPE = POINTER TO ARRAY [0..0) OF ControlDefs.GFTItem;

auxtrap: PROCEDURE RETURNS [myframe: FrameHandle] =
  BEGIN
  state: StateVector;
  newframe: FrameHandle;
  eventry: POINTER TO EntryVectorItem;
  fsize, findex: CARDINAL;
  newG: GlobalFrameHandle;
  dest, tempdest: representation ControlLink;
  alloc: BOOLEAN;
  gfi: GFTIndex;
  ep: CARDINAL;
  pcsegp: TYPE = POINTER TO CsegPrefix;

  myframe ← REGISTER[Lreg];
  state.X ← myframe.returnlink; state.Y ← 0;
  state.instbyte←0;
  state.stk[0]←myframe;
  state.stkptr←1;

  ProcessDefs.DisableInterrupts[];

  DO
    ProcessDefs.EnableInterrupts[];
    TRANSFER WITH state;

    state←STATE;  -- interrupts turned off by trap
    myframe.returnlink ← state.Y;
    dest ← state.stk[state.stkptr-1];
    state.stkptr ← state.stkptr-1;

    tempdest ← dest;
    DO
      SELECT tempdest.type FROM
        frametag =>
          BEGIN
          alloc ← TRUE;
          findex ← LOOPHOLE[tempdest, CARDINAL]/4;
          EXIT
          END;
        procdesctag =>
          BEGIN OPEN proc: LOOPHOLE[tempdest, ProcDesc];
          [gftindex: gfi, epoffset: ep, tag:] ← proc;
          [frame: newG, epbase: findex] ← LOOPHOLE[REGISTER[GFTreg], pgft][gfi];
          eventry ← @LOOPHOLE[newG.codebase, pcsegp].EntryVector[findex+ep];
          findex ← eventry.framesize;
          alloc ← FALSE;
          EXIT
          END;
        indirecttag => tempdest ← MEMORY[LOOPHOLE[tempdest]];
        ENDCASE => ImageDefs.PuntMesa[];
      ENDLOOP;

    IF findex >= maxallocslot THEN ImageDefs.PuntMesa[]
    ELSE
      BEGIN
      fsize ← framevec[findex]+1;  -- includes overhead word
      newframe ← LOOPHOLE[freepointer+1];
      freepointer↑ ← findex;
      freepointer ← freepointer + fsize;
      IF wordsleft < fsize THEN ImageDefs.PuntMesa[] ELSE wordsleft ← wordsleft - fsize;
      END;

    IF alloc THEN
      BEGIN
      state.X ← myframe.returnlink;
      state.stk[state.stkptr] ← newframe;
      state.stkptr ← state.stkptr+1;
      END
    ELSE
      BEGIN
      IF dest.type # indirecttag THEN
        BEGIN
        state.X ← newframe;
        newframe.accesslink ← newG;
        newframe.pc ← eventry.initialpc;
        newframe.returnlink ← myframe.returnlink;
        END
      ELSE
        BEGIN
        IF findex = maxallocslot THEN ImageDefs.PuntMesa[];
        state.X ← dest;
        newframe.accesslink ← (REGISTER[AVreg]+findex)↑;
        (REGISTER[AVreg]+findex)↑ ← newframe;
        END;
      state.Y ← myframe.returnlink;
      END;

    ENDLOOP;
    END;

  PageTable: TYPE = MACHINE DEPENDENT RECORD [
    fp: AltoFileDefs.CFP,
    firstpage: CARDINAL,
    table: ARRAY [0..1) OF DiskDefs.DA];
  ptPointer: POINTER TO POINTER TO PageTable = LOOPHOLE[24B];

  BootPageTable: PROCEDURE [file:FileHandle, pt:POINTER TO PageTable] =
    BEGIN OPEN AltoFileDefs;
    lastpage: PageNumber;
    PlugHint: PROCEDURE [seg:FileSegmentHandle] RETURNS [BOOLEAN] =
      BEGIN
      IF seg.file = file  AND seg.base IN[pt.firstpage..lastpage] THEN
```

```
        seg.hint ← FileHint [
          page: seg.base,
          da: DiskDefs.VirtualDA[pt.table[seg.base-pt.firstpage]]];
      RETURN[FALSE]
      END;
    DropFileRequest[file];
    file.open ← TRUE;
    file.fp ← FP[serial: pt.fp.serial, leaderDA: pt.fp.leaderDA];
    FOR lastpage ← 0, lastpage+1
    UNTIL pt.table[lastpage] = DiskDefs.DA[0,0,0,0,0]
      DO NULL ENDLOOP;
    IF lastpage = 0 THEN RETURN;
    lastpage ← lastpage+pt.firstpage-1;
    [] ← EnumerateFileSegments[PlugHint];
    RETURN
    END;

-- The driver

  MakeImage: PUBLIC PROCEDURE [name: STRING, symbolsToImage: BOOLEAN] =
    BEGIN
    s: StateVector;
    InitFileRequest[];
    InitSpace[];
    IF ~symbolsToImage THEN RequestSymbolFiles[];
    s.stk[0] ← REGISTER[Greg];
    s.stkptr ← 1;
    s.instbyte ← 0;
    s.X ← FrameDefs.SwapOutCode;
    s.Y ← GetReturnLink[];
    InstallImage[name, TRUE, TRUE, FALSE];
    RETURN WITH s;
    END;

  MakeUnMergedImage: PUBLIC PROCEDURE [name: STRING, symbolsToImage: BOOLEAN] =
    BEGIN
    s: StateVector;
    InitFileRequest[];
    InitSpace[];
    IF ~symbolsToImage THEN RequestSymbolFiles[];
    s.stk[0] ← REGISTER[Greg];
    s.stkptr ← 1;
    s.instbyte ← 0;
    s.X ← FrameDefs.SwapOutCode;
    s.Y ← GetReturnLink[];
    InstallImage[name, FALSE, TRUE, FALSE];
    RETURN WITH s;
    END;

  END.
```