```
-- BcdWrite.mesa; edited by Johnsson on August 30, 1978  9:07 PM

DIRECTORY
  AltoDefs: FROM "altodefs",
  BcdControlDefs: FROM "bcdcontroldefs",
  BcdDefs: FROM "bcddefs",
  BcdErrorDefs: FROM "bcderrordefs",
  BcdFileDefs: FROM "bcdfiledefs",
  BcdHeapDefs: FROM "bcdheapdefs",
  BcdTabDefs: FROM "bcdtabdefs",
  BcdTreeDefs: FROM "bcdtreedefs",
  BcdUtilDefs: FROM "bcdutildefs",
  IODefs: FROM "iodefs",
  InlineDefs: FROM "inlinedefs",
  MiscDefs: FROM "miscdefs",
  SegmentDefs: FROM "segmentdefs",
  StreamDefs: FROM "streamdefs",
  StringDefs: FROM "stringdefs",
  SystemDefs: FROM "systemdefs",
  SymbolCompressorDefs: FROM "symbolcompressordefs",
  TableDefs: FROM "tabledefs",
  TimeDefs: FROM "timedefs";

DEFINITIONS FROM BcdDefs;

BcdWrite: PROGRAM [data: BcdControlDefs.BinderData]
  IMPORTS BcdErrorDefs, BcdFileDefs, BcdHeapDefs, BcdTabDefs, BcdTreeDefs, BcdUtilDefs, IODefs, MiscDef
**s, SegmentDefs, StreamDefs, StringDefs, SystemDefs, SymbolCompressorDefs, TimeDefs, TableDefs
  EXPORTS BcdControlDefs =
  BEGIN

  Alignment: CARDINAL = 4;  -- Code Segments must start at 0 MOD Alignment

  StreamHandle: TYPE = StreamDefs.StreamHandle;
  FileSegmentHandle: TYPE = SegmentDefs.FileSegmentHandle;
  empty: BcdTreeDefs.TreeLink = BcdTreeDefs.empty;

  BcdWriteError: PUBLIC SIGNAL = CODE;
  error: PROCEDURE = BEGIN SIGNAL BcdWriteError END;

  tb, stb, ctb, mtb, etb, itb, sgb, ftb, ntb: TableDefs.TableBase;
  ssb: BcdDefs.NameString;

  Notifier: TableDefs.TableNotifier =
    BEGIN
    tb ← base[treetype];
    stb ← base[sttype];
    ctb ← base[cttype];
    mtb ← base[mttype];
    etb ← base[exptype];
    itb ← base[imptype];
    sgb ← base[sgtype];
    ftb ← base[fttype];
    ntb ← base[nttype];
    ssb ← LOOPHOLE[base[sstype]];
    IF bcd # NIL THEN
      BEGIN
      bcd.ctb ← ctb;
      bcd.mtb ← mtb;
      IF ~packing THEN bcd.sgb ← sgb;
      END;
    RETURN
    END;

  bcd: POINTER TO BcdUtilDefs.BcdBases ← NIL;
  header: POINTER TO BcdDefs.BCD;

  WriteBcd: PUBLIC PROCEDURE [root: BcdTreeDefs.TreeLink] =
    BEGIN
    TableDefs.AddNotify[Notifier];
    WITH r: root SELECT FROM
      subtree =>
        BEGIN
        packing ← (tb+r.index).son2 # BcdTreeDefs.empty AND data.copycode;
        Initialize[];
        IF packing THEN
```

```
        BEGIN
        MakePackItem[(tb+r.index).son2];
        FillInSgMap[];
        END;
      CopyConfigs[];
      CopyModules[];
      WITH config: (tb+r.index).son3 SELECT FROM
        subtree =>
          BEGIN OPEN c: tb+config.index;
          BcdTreeDefs.scanlist[c.son1, CopyImport];
          BcdTreeDefs.scanlist[c.son2, CopyExport];
          END;
        ENDCASE;
      END;
    ENDCASE;
  TableOut[];
  Finalize[];
  TableDefs.DropNotify[Notifier];
  RETURN
  END;

Initialize: PROCEDURE =
  BEGIN OPEN InlineDefs, TableDefs;
  impbase, expbase, sgbase, fbase, nbase, ssbase: TableBase;
  impsize, expsize, sgsize, fsize, nsize, sssize: CARDINAL;
  nsgis: CARDINAL;
  d: CARDINAL ← SIZE[BcdUtilDefs.BcdBases]+SIZE[BCD];
  p: POINTER;
  IF data.copycode OR data.copysymbols THEN InitCodeSymbolCopy[];
  [impbase, impsize] ← TableBounds[imptype];
  [expbase, expsize] ← TableBounds[exptype];
  [sgbase, sgsize] ← TableBounds[sgtype];
  nsgis ← sgsize/SIZE[SGRecord];
  IF ~packing THEN sgsize ← 0;
  [fbase, fsize] ← TableBounds[fttype];
  [nbase, nsize] ← TableBounds[nttype];
  [ssbase, sssize] ← TableBounds[sstype];
  bcd ← p ←
    BcdHeapDefs.GetSpace[d+impsize+expsize+sgsize+fsize+nsize+sssize];
  header ← p+SIZE[BcdUtilDefs.BcdBases];
  InitHeader[header];
  d ← LOOPHOLE[p+d];
  COPY[to: LOOPHOLE[bcd.etb ← d], from: LOOPHOLE[expbase], nwords: expsize];
    d ← d + expsize;
    TrimTable[exptype,0];
  COPY[to: LOOPHOLE[bcd.itb ← d], from: LOOPHOLE[impbase], nwords: impsize];
    d ← d + impsize;
    TrimTable[imptype,0];
  COPY[to: LOOPHOLE[bcd.ftb ← d], from: LOOPHOLE[fbase], nwords: fsize];
    d ← d + fsize;
    TrimTable[fttype,0];
  COPY[to: LOOPHOLE[bcd.ntb ← d], from: LOOPHOLE[nbase], nwords: nsize];
    d ← d + nsize;
    TrimTable[nttype,0];
  COPY[to: bcd.ssb ← LOOPHOLE[d], from: LOOPHOLE[ssbase], nwords: sssize];
    d ← d + sssize;
    BcdTabDefs.BcdTabReset[];
  IF packing THEN
    BEGIN
    COPY[to: LOOPHOLE[bcd.sgb ← d], from: LOOPHOLE[sgbase], nwords: sgsize];
      d ← d + sgsize;
      TrimTable[sgtype,0];
    END;
  bcd.ctb ← TableBounds[cttype].base;
  bcd.mtb ← TableBounds[mttype].base;
  IF data.copycode OR data.copysymbols THEN
    BEGIN MapCodeSymbolFiles[]; InitCopyMap[nsgis]; END;
  IF packing THEN InitSgMap[nsgis]
  ELSE
    BEGIN
    bcd.sgb ← TableBounds[sgtype].base;
    IF ~data.copycode THEN MapSegments[code];
    IF ~data.copysymbols THEN MapSegments[symbols];
    END;
  END;
```

```
Finalize: PROCEDURE =
  BEGIN
  ReleaseCodeSymbolCopy[];
  BcdHeapDefs.FreeSpace[bcd];
  bcd ← NIL;
  FreeSgMap[];
  FreeCopyMap[];
  FreePackItems[];
  END;

InitHeader: PROCEDURE [header: POINTER TO BCD] =
  BEGIN
  MiscDefs.Zero[header, SIZE[BCD]];
  header.versionident ← BcdDefs.VersionID;
  header.version ← [time: TimeDefs.CurrentDayTime[], zapped: FALSE,
    net: data.network, host: data.host];
  header.creator ← data.binderVersion;
  header.definitions ← FALSE;
  header.source ← BcdDefs.NullName;
  RETURN
  END;

Map: TYPE = RECORD [
  fti: FTIndex,
  type: SegClass,
  filename: STRING,
  filehandle: SegmentDefs.FileHandle];

codemap, symbolmap: POINTER TO Map;

InitCodeSymbolCopy: PROCEDURE =
  BEGIN OPEN BcdFileDefs;
  setup: PROCEDURE [file: STRING, type: SegClass]
    RETURNS [p: POINTER TO Map] =
    BEGIN
    p ← BcdHeapDefs.GetSpace[SIZE[Map]];
    p.type ← type;
    p.filename ← file;
    p.filehandle ← NIL;
    IF file.length = 0 THEN p.fti ← FTSelf
    ELSE p.fti ← BcdUtilDefs.EnterFile[file];
    END;
  IF data.copycode THEN
    codemap ← setup[data.codefile,code];
  IF data.copysymbols THEN
    symbolmap ← setup[data.symbolfile,symbols];
  LookupFileTable[];
  IF data.copycode AND codemap.fti # FTSelf THEN
    codemap.filehandle ← HandleForFile[
      codemap.fti ! UnknownFile => CONTINUE];
  IF data.copysymbols AND symbolmap.fti # FTSelf THEN
    symbolmap.filehandle ← HandleForFile[
      symbolmap.fti ! UnknownFile => CONTINUE];
  END;

MapCodeSymbolFiles: PROCEDURE =
  BEGIN
  IF data.copycode THEN
    codemap.fti ← BcdUtilDefs.MergeFile[bcd,codemap.fti];
  IF data.copysymbols THEN
    symbolmap.fti ← BcdUtilDefs.MergeFile[bcd,symbolmap.fti];
  END;

ReleaseCodeSymbolCopy: PROCEDURE =
  BEGIN
  IF data.copycode THEN
    BcdHeapDefs.FreeSpace[codemap];
  IF data.copysymbols THEN
    BcdHeapDefs.FreeSpace[symbolmap];
  codemap ← symbolmap ← NIL;
  END;

BumpVersion: PROCEDURE [v: VersionStamp, n: CARDINAL] RETURNS [VersionStamp] =
  BEGIN
  old: CARDINAL = v.time.lowbits;
  IF (v.time.lowbits ← v.time.lowbits + n) < old THEN
```

```
    v.time.highbits ← v.time.highbits + 1;
  RETURN[v]
  END;

EnumerateSegments: PROCEDURE [proc: PROCEDURE[SGIndex]] =
  BEGIN OPEN TableDefs;
  sgi: SGIndex;
  sgLimit: SGIndex = LOOPHOLE[TableBounds[sgtype].size];
  FOR sgi ← FIRST[SGIndex], sgi + SIZE[SGRecord] UNTIL sgi = sgLimit DO
    proc[sgi];
    ENDLOOP;
  RETURN
  END;

EnumerateOldSegments: PROCEDURE [proc: PROCEDURE[SGIndex]] =
  BEGIN OPEN TableDefs;
  sgi, sgLimit: SGIndex;
  IF ~packing THEN BEGIN EnumerateSegments[proc]; RETURN END;
  sgLimit ← LOOPHOLE[LENGTH[sgMap]*SIZE[SGRecord]];
  FOR sgi ← FIRST[SGIndex], sgi + SIZE[SGRecord] UNTIL sgi = sgLimit DO
    proc[sgi];
    ENDLOOP;
  RETURN
  END;

InitFile: PROCEDURE [p: POINTER TO Map]
  RETURNS [s: StreamHandle, page: CARDINAL] =
  BEGIN OPEN SegmentDefs, StreamDefs;
  lh: BcdDefs.BCD;
  BcdUtilDefs.SetFileVersion[p.fti,
    BumpVersion[header.version, IF p.type = code THEN 1 ELSE 2]];
  InitHeader[@lh];
  lh.version ← (ftb+p.fti).version;
  IF p.filehandle = NIL THEN
    BEGIN
    p.filehandle ← NewFile[p.filename,Write+Append,DefaultVersion];
    END;
  s ← CreateWordStream[p.filehandle,Write+Append];
  [] ← WriteBlock[s,@lh,SIZE[BcdDefs.BCD]];
  page ← 1+(SIZE[BcdDefs.BCD]+AltoDefs.PageSize-1)/AltoDefs.PageSize;
  RETURN
  END;

-- Code Packing

packing: BOOLEAN;

PackItem: TYPE = RECORD [
  link: PackHandle,
  newsgi: SGIndex,  -- in the new table
  count: CARDINAL,
  item: ARRAY [0..0) OF MTIndex];

PackHandle: TYPE = POINTER TO PackItem;

phHead, phTail: PackHandle ← NIL;

MakePackItem: BcdTreeDefs.TreeScan =
  -- t is empty, a list of ids, or a list of lists of ids
  BEGIN OPEN BcdTreeDefs;
  ph: PackHandle;
  px: TreeXIndex;
  i, nsons: CARDINAL;
  IF t = empty THEN RETURN;
  WITH t SELECT FROM
    subtree =>
      BEGIN OPEN tt: tb+index;
      IF tt.name # list THEN error[];
      IF tt.son1.tag = subtree THEN
        BEGIN scanlist[t,MakePackItem]; RETURN END;
      px ← LOOPHOLE[index, TreeXIndex]+TreeNodeSize;
      END;
    ENDCASE;
  nsons ← listlength[t];
  ph ← BcdHeapDefs.GetSpace[SIZE[PackItem]+nsons];
  ph↑ ← [link: NIL,
```

```
    newsgi: TableDefs.Allocate[sgtype, SIZE[SGRecord]],
    count: nsons, item:];
  (sgb+ph.newsgi)↑ ← [class: code, file: codemap.fti,
    base:0, pages:0, extraPages:0];
  MiscDefs.SetBlock[@ph.item[0], MTNull, nsons];
  FOR i IN [0..nsons) DO
    WITH (tb+px+i).soni SELECT FROM
      symbol =>
        WITH (stb+index) SELECT FROM
          external =>
            WITH m: map SELECT FROM
              module =>
                BEGIN
                ph.item[i] ← m.mti;
                SetSgMap[old: (mtb+m.mti).code.sgi, new: ph.newsgi];
                END;
              ENDCASE;
          ENDCASE;
      ENDCASE;
    ENDLOOP;
  IF phTail = NIL THEN phHead ← phTail ← ph
  ELSE BEGIN phTail.link ← ph; phTail ← ph END;
  RETURN
  END;

FreePackItems: PROCEDURE =
  BEGIN
  p, next: PackHandle;
  FOR p ← phHead, next UNTIL p = NIL DO
    next ← p.link;
    BcdHeapDefs.FreeSpace[p];
    ENDLOOP;
  phHead ← phTail ← NIL;
  RETURN
  END;

PackCodeSegments: PROCEDURE [out: StreamHandle, startpage: CARDINAL]
  RETURNS [nextpage: CARDINAL] =
  BEGIN
  pi: CARDINAL;
  ph: PackHandle;
  offset, validlength: CARDINAL;
  oldsgi: SGIndex;
  mti: MTIndex;
  seg: FileSegmentHandle;
  FixUp: PROCEDURE [mti: MTIndex] RETURNS [BOOLEAN] =
    BEGIN OPEN m: mtb+mti;
    length: CARDINAL;
    IF m.code.sgi = oldsgi THEN
      BEGIN
      length ← m.code.offset+m.code.length/2;
      m.code.offset ← m.code.offset+offset;
      m.code.packed ← TRUE;
      IF length > validlength THEN validlength ← length;
      END;
    RETURN[FALSE]
    END;
  nextpage ← startpage;
  FOR ph ← phHead, ph.link UNTIL ph = NIL DO
    StreamDefs.SetIndex[out,[page:nextpage-1,byte:0]];
    offset ← 0;
    (sgb+ph.newsgi).base ← nextpage;
    FOR pi IN [0..ph.count) DO
      BEGIN OPEN SegmentDefs, module: mtb+mti;
      mti ← ph.item[pi];
      IF module.links = code AND ~module.code.linkspace THEN
        BEGIN
        offset ← AddLinksToCodeSegment[out,mti,offset,TRUE] + offset;
        GOTO ignore
        END;
      oldsgi ← module.code.sgi;
      IF (seg←SegmentForSgi[oldsgi]) = NIL THEN GOTO ignore;
      SwapIn[seg];
      IF offset MOD Alignment # 0 THEN
        BEGIN
        i: CARDINAL;
```

```
        FOR i IN [offset MOD Alignment..Alignment) DO
          out.put[out,0]; offset ← offset + 1 ENDLOOP;
        END;
      validlength ← 0; EnumerateModules[FixUp];
      [] ← StreamDefs.WriteBlock[out, FileSegmentAddress[seg],validlength];
      offset ← offset + validlength;
      Unlock[seg]; DeleteFileSegment[seg];
      EXITS ignore => NULL;
      END;
      ENDLOOP;
    nextpage ← ((sgb+ph.newsgi).pages←SystemDefs.PagesForWords[offset]) + nextpage;
    ENDLOOP;
  RETURN
  END;

-- Segment Mapping

sgMap: DESCRIPTOR FOR ARRAY OF SGIndex;
copyMap: DESCRIPTOR FOR ARRAY OF BOOLEAN;

InitCopyMap: PROCEDURE [nsgis: CARDINAL] =
  BEGIN
  copyMap ← DESCRIPTOR[BcdHeapDefs.GetSpace[nsgis], nsgis];
  MiscDefs.SetBlock[BASE[copyMap],FALSE,nsgis];
  END;

FreeCopyMap: PROCEDURE =
  BEGIN
  IF data.copycode OR data.copysymbols THEN
    BcdHeapDefs.FreeSpace[BASE[copyMap]];
  RETURN
  END;

Copied: PROCEDURE [sgi: SGIndex] RETURNS [BOOLEAN] =
  BEGIN
  i: CARDINAL = LOOPHOLE[sgi,CARDINAL]/SIZE[SGRecord];
  RETURN[copyMap[i]]
  END;

SetCopied: PROCEDURE [sgi: SGIndex] =
  BEGIN
  i: CARDINAL = LOOPHOLE[sgi,CARDINAL]/SIZE[SGRecord];
  copyMap[i] ← TRUE;
  RETURN
  END;

InitSgMap: PROCEDURE [nsgis: CARDINAL] =
  BEGIN
  sgMap ← DESCRIPTOR[BcdHeapDefs.GetSpace[nsgis], nsgis];
  MiscDefs.SetBlock[BASE[sgMap],SGNull,nsgis];
  END;

FreeSgMap: PROCEDURE =
  BEGIN
  IF packing THEN BcdHeapDefs.FreeSpace[BASE[sgMap]];
  RETURN
  END;

SetSgMap: PROCEDURE [old, new: SGIndex] =
  BEGIN
  i: CARDINAL = LOOPHOLE[old,CARDINAL]/SIZE[SGRecord];
  IF packing AND old # SGNull THEN sgMap[i] ← new;
  RETURN
  END;

ReadSgMap: PROCEDURE [old: SGIndex] RETURNS [SGIndex] =
  BEGIN
  i: CARDINAL = LOOPHOLE[old,CARDINAL]/SIZE[SGRecord];
  RETURN[IF ~packing OR old = SGNull THEN old ELSE sgMap[i]]
  END;

FillInSgMap: PROCEDURE =
  BEGIN -- called only when packing => copycode = TRUE
  i: CARDINAL;
  oldsgi, newsgi: SGIndex;
  FOR i IN [0..LENGTH[sgMap]) DO
```

```
    IF sgMap[i] = SGNull THEN
      BEGIN
      oldsgi ← LOOPHOLE[i*SIZE[SGRecord]];
      newsgi ← TableDefs.Allocate[sgtype, SIZE[SGRecord]];
      (sgb+newsgi)↑ ← (bcd.sgb+oldsgi)↑;
      (sgb+newsgi).file ←
        (IF (sgb+newsgi).class = symbols THEN
          (IF data.copysymbols THEN symbolmap.fti
          ELSE BcdUtilDefs.MergeFile[bcd, (sgb+newsgi).file])
        ELSE codemap.fti);
      sgMap[i] ← newsgi;
      END;
    ENDLOOP;
  RETURN
  END;

FixAllSgis: PROCEDURE =
  BEGIN -- replace all sgis with ReadSgMap[sgi]
  FixOne: PROCEDURE [mti: MTIndex] RETURNS [BOOLEAN] =
    BEGIN OPEN m: mtb+mti;
    m.code.sgi ← ReadSgMap[m.code.sgi];
    m.sseg ← ReadSgMap[m.sseg];
    RETURN[FALSE]
    END;
  EnumerateModules[FixOne];
  RETURN
  END;

SegmentForSgi: PROCEDURE [sgi: SGIndex] RETURNS [s: FileSegmentHandle] =
  BEGIN OPEN SegmentDefs, seg: bcd.sgb+sgi;
  s ← NIL;
  IF Copied[sgi] OR seg.file = FTNull THEN RETURN;
  s ← NewFileSegment[
    BcdFileDefs.HandleForFile[seg.file
     ! BcdFileDefs.UnknownFile => CONTINUE],
       seg.base,seg.pages+seg.extraPages,Read];
  IF s = NIL THEN
    BEGIN
    BcdErrorDefs.ErrorNameBase[
      error, "cannot be opened"L,
      (bcd.ftb+seg.file).name, bcd.ssb];
    header.versionident ← 0;
    END;
  SetCopied[sgi];
  RETURN
  END;

-- Code Links

AddLinksToCodeSegment: PROCEDURE
  [out: StreamHandle, mti: MTIndex, offset: CARDINAL, packed: BOOLEAN]
  RETURNS [CARDINAL] =
  BEGIN OPEN SegmentDefs;
  sgi: SGIndex ← (mtb+mti).code.sgi;
  codelength: CARDINAL ← (mtb+mti).code.length/2;
  i, linkspace: CARDINAL;
  s: FileSegmentHandle;
  prefixwords: CARDINAL ← 0;
  FixOffset: PROCEDURE [mti: MTIndex] RETURNS[BOOLEAN] =
    BEGIN OPEN c: (mtb+mti).code;
    IF c.sgi = sgi THEN
      BEGIN
      c.linkspace ← TRUE;
      c.offset ← c.offset+offset;
      c.packed ← packed;
      END;
    RETURN[FALSE]
    END;
  IF (s←SegmentForSgi[sgi]) = NIL THEN RETURN[0];
  linkspace ← (mtb+mti).frame.length;
  IF offset = 0 AND linkspace # 0 THEN
    BEGIN
    prefixwords ← 1;
    out.put[out, linkspace + Alignment - (linkspace MOD Alignment)];
    offset ← offset+1;
    END;
```

```
    IF (offset+linkspace) MOD Alignment # 0 THEN
      linkspace ← linkspace + Alignment - ((offset+linkspace) MOD Alignment);
    offset ← offset+linkspace;
    EnumerateModules[FixOffset];
    FOR i IN [0..linkspace) DO out.put[out,0] ENDLOOP;
    SwapIn[s];
    [] ← StreamDefs.WriteBlock[out,FileSegmentAddress[s],codelength];
    Unlock[s]; DeleteFileSegment[s];
    RETURN[prefixwords+linkspace+codelength]
    END;

  MoveCodeSegments: PROCEDURE =
    BEGIN
    out: StreamHandle;
    nextpage: CARDINAL;
    AddLinks: PROCEDURE [mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN OPEN m: mtb+mti;
      nwords, npages: CARDINAL;
      newsgi: SGIndex;
      IF m.links = code AND ~m.code.linkspace THEN
        BEGIN
        StreamDefs.SetIndex[out,[page:nextpage-1,byte:0]];
        nwords ← AddLinksToCodeSegment[out, mti, 0, FALSE];
        npages ← SystemDefs.PagesForWords[nwords];
        newsgi ← ReadSgMap[m.code.sgi];
        (sgb+newsgi).file ← codemap.fti;
        (sgb+newsgi).base ← nextpage;
        (sgb+newsgi).pages ← npages;
        nextpage ← nextpage + npages;
        END;
      RETURN[FALSE]
      END;
    MoveOne: PROCEDURE [oldsgi: SGIndex] =
      BEGIN OPEN SegmentDefs, seg: bcd.sgb+oldsgi;
      s: FileSegmentHandle;
      newsgi: SGIndex;
      IF seg.class = code AND (s←SegmentForSgi[oldsgi]) # NIL THEN
        BEGIN
        newsgi ← ReadSgMap[oldsgi];
        (sgb+newsgi).file ← codemap.fti;
        (sgb+newsgi).base ← nextpage;
        SwapIn[s];
        StreamDefs.SetIndex[out,[page:nextpage-1,byte:0]];
        [] ← StreamDefs.WriteBlock[out,
          FileSegmentAddress[s],s.pages*AltoDefs.PageSize];
        nextpage ← nextpage + s.pages;
        Unlock[s]; DeleteFileSegment[s];
        END;
      RETURN
      END;
    IF codemap.fti = FTSelf THEN
      BEGIN out ← bcdStream; nextpage ← nextBcdPage END
    ELSE [out, nextpage] ← InitFile[codemap];
    nextpage ← PackCodeSegments[out,nextpage];
    EnumerateModules[AddLinks];
    EnumerateOldSegments[MoveOne];
    IF codemap.fti = FTSelf THEN nextBcdPage ← nextpage
    ELSE out.destroy[out];
    RETURN
    END;

  MoveSymbolSegments: PROCEDURE =
    BEGIN
    out: StreamHandle;
    nextpage: CARDINAL;
    MoveOne: PROCEDURE [oldsgi: SGIndex] =
      BEGIN OPEN SegmentDefs, seg: bcd.sgb+oldsgi;
      s: FileSegmentHandle;
      newsgi: SGIndex;
      IF seg.class = symbols AND (s←SegmentForSgi[oldsgi]) # NIL THEN
        BEGIN
        newsgi ← ReadSgMap[oldsgi];
        (sgb+newsgi).file ← symbolmap.fti;
        (sgb+newsgi).base ← nextpage;
        StreamDefs.SetIndex[out,[page:nextpage-1,byte:0]];
        IF data.compress THEN
```

```
        BEGIN
        s.pages ← (sgb+newsgi).pages;
        (sgb+newsgi).extraPages ← 0;
        (sgb+newsgi).pages ← CompressModule[oldsgi, s, out];
        nextpage ← nextpage + (sgb+newsgi).pages;
        END
      ELSE
        BEGIN
        SwapIn[s];
        [] ← StreamDefs.WriteBlock[out,
          FileSegmentAddress[s],s.pages*AltoDefs.PageSize];
        nextpage ← nextpage + s.pages;
        Unlock[s]; DeleteFileSegment[s];
        END;
      END;
    RETURN
    END;
  IF symbolmap.fti = FTSelf THEN
    BEGIN out ← bcdStream; nextpage ← nextBcdPage END
  ELSE [out, nextpage] ← InitFile[symbolmap];
  EnumerateOldSegments[MoveOne];
  IF symbolmap.fti = FTSelf THEN nextBcdPage ← nextpage
  ELSE out.destroy[out];
  RETURN
  END;

CompressModule: PROCEDURE [sgi: SGIndex, seg: FileSegmentHandle, stream: StreamHandle]
  RETURNS [pages: CARDINAL] =
  BEGIN OPEN StringDefs;
  smti: MTIndex;
  n: NameRecord;
  mname: STRING;
  ss: SubStringDescriptor;
  Find: PROCEDURE [mti: MTIndex] RETURNS[BOOLEAN] =
    BEGIN
    IF (mtb+mti).sseg = sgi THEN BEGIN smti ← mti; RETURN[TRUE] END;
    RETURN[FALSE]
    END;
  EnumerateModules[Find];
  n ← (mtb+smti).name;
  mname ← BcdHeapDefs.GetString[ssb.size[n]];
  ss ← [base: @ssb.string, offset: n, length: ssb.size[n]];
  AppendSubString[mname,@ss];
  pages ← SymbolCompressorDefs.CompressSymbols[mname, seg, stream];
  BcdHeapDefs.FreeString[mname];
  RETURN
  END;

CopyName: PROCEDURE [olditem, newitem: Namee] =
  BEGIN OPEN TableDefs;
  nti, newnti: NTIndex;
  newnti ← Allocate[nttype, SIZE[NTRecord]];
  FOR nti ← FIRST[NTIndex], nti+SIZE[NTRecord] DO
    OPEN old: bcd.ntb+nti;
    IF old.item = olditem THEN
      BEGIN OPEN new: ntb+newnti;
      new.item ← newitem;
      new.name ← BcdUtilDefs.MapName[bcd, old.name];
      RETURN;
      END;
    ENDLOOP;
  END;

CopyConfigs: PROCEDURE =
  BEGIN OPEN TableDefs;
  -- configs are already copied, only map names and files
  cti: CTIndex;
  ctLimit: CTIndex = LOOPHOLE[TableBounds[cttype].size];
  FOR cti ← FIRST[CTIndex], cti+SIZE[CTRecord] UNTIL cti = ctLimit DO
    OPEN c: ctb+cti;
    header.nConfigs ← header.nConfigs + 1;
    c.name ← BcdUtilDefs.MapName[bcd, c.name];
    c.file ← BcdUtilDefs.MergeFile[bcd, c.file];
    IF c.namedinstance THEN CopyName[[config[cti]], [config[cti]]];
    ENDLOOP;
  RETURN
```

```
    END;

  EnumerateModules: PROCEDURE [p: PROCEDURE[MTIndex] RETURNS[BOOLEAN]] =
    BEGIN OPEN TableDefs;
    mti: MTIndex;
    mtLimit: MTIndex = LOOPHOLE[TableBounds[mttype].size];
    FOR mti ← FIRST[MTIndex], mti+SIZE[MTRecord]+(mtb+mti).frame.length
      UNTIL mti = mtLimit DO
      IF p[mti] THEN EXIT;
      ENDLOOP;
    RETURN
    END;

  CopyModules: PROCEDURE =
    BEGIN OPEN TableDefs;
    -- modules are already copied, only map names and files
    MapOne: PROCEDURE[mti: MTIndex] RETURNS[BOOLEAN] =
      BEGIN OPEN m: mtb+mti;
      header.nModules ← header.nModules + 1;
      m.name ← BcdUtilDefs.MapName[bcd, m.name];
      m.file ← BcdUtilDefs.MergeFile[bcd, m.file];
      IF m.namedinstance THEN CopyName[[module[mti]], [module[mti]]];
      RETURN[FALSE]
      END;
    EnumerateModules[MapOne];
    RETURN
    END;

  MapSegments: PROCEDURE[type: SegClass] =
    BEGIN
    CopySegment: PROCEDURE [sgi: SGIndex] =
      BEGIN OPEN s: sgb+sgi;
      IF s.class = type THEN
        s.file ← BcdUtilDefs.MergeFile[bcd, s.file];
      RETURN
      END;
    EnumerateSegments[CopySegment];
    RETURN
    END;

  CopyImport: BcdTreeDefs.TreeScan =
    BEGIN OPEN BcdTabDefs;
    sti: STIndex ← STNull;
    olditi, iti: IMPIndex;
    name: HTIndex ← HTNull;
    WITH t SELECT FROM
      symbol => BEGIN sti ← index; olditi ← (stb+sti).impi END;
      subtree =>
        IF (tb+index).name = item THEN
          BEGIN
          WITH s1:(tb+index).son1 SELECT FROM
            symbol =>
              BEGIN OPEN s: (stb+s1.index);
              --name ← s.hti;
              sti ← s1.index;
              olditi ← s.impi;
              END;
            ENDCASE;
          IF (tb+index).son2 # empty THEN WITH s2:(tb+index).son2 SELECT FROM
            symbol =>
              BEGIN OPEN s: (stb+s2.index);
              sti ← s2.index;
              olditi ← s.impi;
              END;
            ENDCASE;
          END;
      ENDCASE => error[];
    IF sti = STNull OR olditi = IMPNull THEN RETURN;
    iti ← BcdUtilDefs.EnterImport[bcd, olditi, name];
    (itb+iti).file ← BcdUtilDefs.MergeFile[bcd, (itb+iti).file];
    IF header.firstdummy = 0 THEN header.firstdummy ← (itb+iti).gfi;
    IF name = HTNull AND (itb+iti).namedinstance THEN
      CopyName[olditem: [import[olditi]], newitem: [import[iti]]];
    header.nImports ← header.nImports + 1;
    header.nDummies ← header.nDummies + (itb+iti).ngfi;
    RETURN
```

```
    END;

  CopyExport: BcdTreeDefs.TreeScan =
    BEGIN OPEN BcdTabDefs;
    sti: STIndex ← STNull;
    neweti: EXPIndex;
    oldeti: EXPIndex ← EXPNull;
    name: HTIndex ← HTNull;
    WITH t SELECT FROM
      symbol => sti ← index;
      subtree =>
        IF (tb+index).name = item THEN
          BEGIN
          WITH s1:(tb+index).son1 SELECT FROM
            symbol =>
              BEGIN
              --name ← (stb+s1.index).hti;
              sti ← s1.index;
              IF (tb+index).son2 # empty THEN WITH s:stb+s1.index SELECT FROM
                external =>
                  WITH m:s.map SELECT FROM
                    interface => oldeti ← m.expi;
                    ENDCASE => error[];
                ENDCASE => error[];
              END;
            ENDCASE => error[];
          IF (tb+index).son2 # empty THEN WITH s2:(tb+index).son2 SELECT FROM
            symbol => sti ← s2.index;
            ENDCASE => error[];
          END;
      ENDCASE => error[];
    WITH s:stb+sti SELECT FROM
      external =>
        WITH m:s.map SELECT FROM
          interface =>
            BEGIN OPEN new: etb+neweti;
            IF oldeti = EXPNull THEN oldeti ← m.expi;
            neweti ← BcdUtilDefs.EnterExport[bcd, oldeti, name];
            InlineDefs.COPY[
              from: @(bcd.etb+oldeti).links,
              to: @new.links,
              nwords: new.size];
            new.file ← BcdUtilDefs.MergeFile[bcd, new.file];
            IF name = HTNull AND new.namedinstance THEN
              CopyName[olditem: [export[oldeti]], newitem: [export[neweti]]];
            END;
          module => [] ← NewExportForModule[m.mti, name];
          ENDCASE => RETURN;
      ENDCASE => RETURN;
    header.nExports ← header.nExports + 1;
    RETURN
    END;

  NewExportForModule: PROCEDURE [mti: MTIndex, name: BcdTabDefs.HTIndex] RETURNS [eti: EXPIndex] =
    BEGIN OPEN TableDefs, BcdTabDefs;
    nti: NTIndex;
    eti ← Allocate[exptype, SIZE[EXPRecord]+1];
    (etb+eti)↑ ← [
      name: (mtb+mti).name,
      size: 1,
      port: module,
      namedinstance: name # HTNull,
      file: (mtb+mti).file,
      links: ];
    (etb+eti).links[0] ← [procedure[gfi: (mtb+mti).gfi, ep: 0, tag: frame]];
    IF name # HTNull THEN
      BEGIN
      nti ← Allocate[nttype,SIZE[NTRecord]];
      (ntb+nti)↑ ← [
        name: BcdUtilDefs.NameForHti[name],
        item: [module[mti]]];
      END;
    RETURN
    END;
```

```
-- Bcd Output Routines

bcdStream: StreamHandle;
nextBcdPage: CARDINAL;

WriteBcdWords: PROCEDURE [addr: POINTER, n: CARDINAL] =
  BEGIN  OPEN StreamDefs;
  [] ← WriteBlock[bcdStream, addr, n];
  RETURN
  END;

WriteSubTable: PROCEDURE [table: TableDefs.TableSelector] =
  BEGIN  OPEN TableDefs;
  base: TableBase;
  size: CARDINAL;
  [base, size] ← TableBounds[table];
  WriteBcdWords[LOOPHOLE[base], size];
  RETURN
  END;

TableOut: PROCEDURE =
  BEGIN OPEN TableDefs;
  d, s: CARDINAL;
  savenextpage: CARDINAL;
  OpenOutputFile[];
  BEGIN OPEN header;
    IF firstdummy = 0 THEN firstdummy ← BcdUtilDefs.GetGfi[0];
    d ← SIZE[BCD];
    ssOffset ← d;  d ← d + (ssLimit ← TableBounds[sstype].size);
    ctOffset ← d;  d ← d + (s ← TableBounds[cttype].size);
    ctLimit ← LOOPHOLE[s, CTIndex];
    mtOffset ← d;  d ← d + (s ← TableBounds[mttype].size);
    mtLimit ← LOOPHOLE[s, MTIndex];
    impOffset ← d;  d ← d + (s ← TableBounds[imptype].size);
    impLimit ← LOOPHOLE[s, IMPIndex];
    expOffset ← d;  d ← d + (s ← TableBounds[exptype].size);
    expLimit ← LOOPHOLE[s, EXPIndex];
    sgOffset ← d;  d ← d + (s ← TableBounds[sgtype].size);
    sgLimit ← LOOPHOLE[s, SGIndex];
    ftOffset ← d;  d ← d + (s ← TableBounds[fttype].size);
    ftLimit ← LOOPHOLE[s, FTIndex];
    ntOffset ← d;  d ← d + (s ← TableBounds[nttype].size);
    ntLimit ← LOOPHOLE[s, NTIndex];
    nPages ← (d+AltoDefs.PageSize-1)/AltoDefs.PageSize;
  END;
  savenextpage ← nextBcdPage ← header.nPages+1;
  IF data.copycode THEN MoveCodeSegments[];
  IF data.copysymbols THEN MoveSymbolSegments[];
  IF packing THEN FixAllSgis[];
  bcdStream.reset[bcdStream];
  WriteBcdWords[header, SIZE[BCD]];
  WriteSubTable[sstype];
  WriteSubTable[cttype];
  WriteSubTable[mttype];
  WriteSubTable[imptype];
  WriteSubTable[exptype];
  WriteSubTable[sgtype];
  WriteSubTable[fttype];
  WriteSubTable[nttype];
  IF nextBcdPage # savenextpage THEN
    StreamDefs.SetIndex[bcdStream,[page:nextBcdPage-1, byte:0]];
  CloseOutputFile[];
  IF ~data.errors THEN
    BEGIN OPEN IODefs, header;
    WriteDecimal[nConfigs]; WriteString[" configs, "L];
    WriteDecimal[nModules]; WriteString[" modules, "L];
    WriteDecimal[nImports]; WriteString[" imports, "L];
    WriteDecimal[nExports]; WriteString[" exports, "L];
    WriteDecimal[nPages]; WriteString[" pages, "L];
    WriteDecimal[BcdUtilDefs.TimeSince[data.starttime]]; WriteLine[" seconds"L];
    END;
  RETURN
  END;

OpenOutputFile: PROCEDURE =
  BEGIN OPEN StreamDefs, SegmentDefs;
```

```
    file: FileHandle ← NIL;
    file ← BcdFileDefs.HandleForFile[data.outputfti
      ! BcdFileDefs.UnknownFile => CONTINUE];
    IF file = NIL THEN
      file ← NewFile[data.outputfile, Write+Append, DefaultVersion];
    bcdStream ← CreateWordStream[file, Write+Append];
    RETURN
    END;

  CloseOutputFile: PROCEDURE =
    BEGIN
    bcdStream.destroy[bcdStream];
    bcdStream ← NIL;
    RETURN
    END;


  END...
```