```
-- BcdLoad.mesa; edited by Johnsson on August 30, 1978  10:14 PM

DIRECTORY
  BcdDefs: FROM "bcddefs",
  BcdBindDefs: FROM "bcdbinddefs",
  BcdControlDefs: FROM "bcdcontroldefs",
  BcdErrorDefs: FROM "bcderrordefs",
  BcdFileDefs: FROM "bcdfiledefs",
  BcdHeapDefs: FROM "bcdheapdefs",
  BcdTabDefs: FROM "bcdtabdefs",
  BcdTreeDefs: FROM "bcdtreedefs",
  BcdUtilDefs: FROM "bcdutildefs",
  MiscDefs: FROM "miscdefs",
  SegmentDefs: FROM "segmentdefs",
  StringDefs: FROM "stringdefs",
  TableDefs: FROM "tabledefs";

DEFINITIONS FROM BcdTreeDefs, BcdDefs, BcdTabDefs;

BcdLoad: PROGRAM [data: BcdControlDefs.BinderData]
  IMPORTS BcdErrorDefs, BcdFileDefs, BcdHeapDefs, TableDefs, BcdTreeDefs, BcdUtilDefs, MiscDefs, Segmen
**tDefs
  EXPORTS BcdBindDefs, BcdControlDefs =
  BEGIN

  CTIndex: TYPE = BcdDefs.CTIndex; CTNull: CTIndex = BcdDefs.CTNull;
  MTIndex: TYPE = BcdDefs.MTIndex; MTNull: MTIndex = BcdDefs.MTNull;
  IMPIndex: TYPE = BcdDefs.IMPIndex; IMPNull: IMPIndex = BcdDefs.IMPNull;
  EXPIndex: TYPE = BcdDefs.EXPIndex; EXPNull: EXPIndex = BcdDefs.EXPNull;
  FTIndex: TYPE = BcdDefs.FTIndex; FTNull: FTIndex = BcdDefs.FTNull;
  HTIndex: TYPE = BcdTabDefs.HTIndex; HTNull: HTIndex = BcdTabDefs.HTNull;
  STIndex: TYPE = BcdTabDefs.STIndex; STNull: STIndex = BcdTabDefs.STNull;
  CXIndex: TYPE = BcdTabDefs.CXIndex; CXNull: CXIndex = BcdTabDefs.CXNull;

  LoadError: PUBLIC SIGNAL = CODE;

  currentCx, loadCx: CXIndex;
  loadTree: BcdTreeDefs.TreeIndex;
  loadExpi: EXPIndex;
  packSti: STIndex;
  currentCti: CTIndex;
  currentOp: InterfaceOp;
  tb, stb, cxb: TableDefs.TableBase;

  localBases: BcdUtilDefs.BcdBases;
  limits: BcdUtilDefs.BcdLimits;
  bcd: BcdUtilDefs.BcdBasePtr;

  Notifier: TableDefs.TableNotifier =
    BEGIN OPEN localBases;
    tb ← base[treetype];
    stb ← base[sttype];
    ctb ← base[cttype];
    cxb ← base[cxtype];
    mtb ← base[mttype];
    etb ← base[exptype];
    itb ← base[imptype];
    ftb ← base[fttype];
    ntb ← base[nttype];
    ssb ← LOOPHOLE[base[sstype]];
    RETURN
    END;

  FileMapItem: TYPE = RECORD [old, new: FTIndex];
  InterfaceOp: TYPE = {plus, then};
  ExportAssigner: TYPE = PROCEDURE;

  error: PROCEDURE = BEGIN SIGNAL LoadError END;


  LoadRoot: PUBLIC PROCEDURE [root: TreeLink] =
    BEGIN
    TableDefs.AddNotify[Notifier];
    bcd ← @localBases;
    loadExpi ← EXPNull;
    currentCti ← CTNull;
    currentOp ← plus;
    currentParameters ← empty;
    ProcessExports ← VerifyExports;
    relocationHead ← NIL;
    loadTree ← nullTreeIndex;
    loadCx ← CXNull;
    WITH root SELECT FROM
      subtree =>
        BEGIN OPEN tb+index;
        SELECT name FROM
          source =>
            BEGIN
            packSti ← FindPackSti[son2];
            WITH son3 SELECT FROM
              subtree => LoadLocalConfig[index, outer, HTNull];
              ENDCASE => error[];
            END;
          ENDCASE => error[];
        END;
      ENDCASE => error[];
    TableDefs.DropNotify[Notifier];
    RETURN
    END;

  FindPackSti: PROCEDURE [t: TreeLink] RETURNS [STIndex] =
    BEGIN
    IF t = empty THEN RETURN[STNull];
    WITH t SELECT FROM
      symbol => RETURN[index];
      subtree => RETURN[FindPackSti[(tb+index).son1]];
      ENDCASE => error[];
    END;

  currentParameters: TreeLink;

  BodyWalk: TreeScan =
    BEGIN
    saveIndex: CARDINAL = data.textIndex;
    WITH t SELECT FROM
      symbol => LoadSti[index, HTNull];
      subtree =>
        BEGIN
        data.textIndex ← (tb+index).sourceindex;
        SELECT (tb+index).name FROM
          list => scanlist[t, BodyWalk];
          item => LoadItem[t];
          config => NULL;
          assign => LoadAssign[index];
          plus, then => LoadExpression[t];
          module =>
            BEGIN
            currentParameters ← (tb+index).son2;
            LoadItem[(tb+index).son1];
            END;
          ENDCASE => error[];
        END;
      ENDCASE => error[];
    data.textIndex ← saveIndex;
    RETURN
    END;

  LoadLocalConfig: PROCEDURE [index: TreeIndex, level: BcdBindDefs.RelocationType, name: HTIndex] =
    BEGIN OPEN t:tb+index, newct: localBases.ctb+currentCti;
    saveCx: CXIndex = currentCx;
    saveCti: CTIndex = currentCti;
    saveLhs: TreeLink = lhs;
    saveAssigner: ExportAssigner = ProcessExports;
    saveName: NameRecord = data.currentname;
    saveIndex: CARDINAL = data.textIndex;
    firstimport: IMPIndex = LOOPHOLE[TableDefs.TableBounds[imptype].size];
    localRel: POINTER TO BcdRelocations;
    currentCx ← BcdUtilDefs.ContextForTree[t.son4];
    currentCti ← TableDefs.Allocate[cttype,SIZE[CTRecord]];
    lhs ← empty;
    ProcessExports ← NormalExports;
    data.currentname ← newct.name ← NameForLink[t.son4];
    data.textIndex ← t.sourceindex;
    IF name = HTNull THEN newct.namedinstance ← FALSE
    ELSE
      BEGIN
      newct.namedinstance ← TRUE;
      BcdUtilDefs.CreateInstanceName[name, [config[currentCti]]];
      END;
    newct.file ← FTSelf;
    newct.config ← saveCti;
    AllocateRelocations[level];
    localRel ← rel;
    localRel.parentcx ← saveCx;
    BodyWalk[t.son5];
    ProcessExports ← saveAssigner;
    lhs ← saveLhs;
    newct.control ← IF t.son3 = empty THEN MTNull ELSE ControlModuleForLink[t.son3];
    loadTree ← index;
    loadCx ← currentCx;
    currentCx ← saveCx;
    ProcessExports[];
    currentCx ← loadCx;
    localRel.import ← TableDefs.TableBounds[imptype].size;
    localRel.dummygfi ← BcdUtilDefs.GetDummyGfi[0];
    ProcessLocalImports[firstimport];
    localRel.importLimit ← LOOPHOLE[TableDefs.TableBounds[imptype].size];
    loadTree ← nullTreeIndex;
    loadCx ← CXNull;
    currentCti ← saveCti;
    currentCx ← saveCx;
    data.currentname ← saveName;
    data.textIndex ← saveIndex;
    END;

  ControlModuleForLink: PROCEDURE [t: TreeLink] RETURNS [MTIndex] =
    BEGIN
    gfi: GFTIndex;
    FindModule: PROCEDURE [mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN RETURN[(localBases.mtb+mti).gfi = gfi] END;
    WITH t SELECT FROM
      symbol =>
        BEGIN
        WITH s:stb+index SELECT FROM
          external =>
            WITH m:s.map SELECT FROM
              module => RETURN[m.mti];
              interface =>
                IF (localBases.etb+m.expi).port = module THEN
                  BEGIN
                  gfi ← (localBases.etb+m.expi).links[0].gfi;
                  limits.mt ← LOOPHOLE[TableDefs.TableBounds[mttype].size];
                  RETURN[EnumerateModules[FindModule]];
                  END
                ELSE GOTO notvalid;
              ENDCASE => GOTO notvalid;
          ENDCASE => GOTO notvalid;
        EXITS notvalid =>
          BcdErrorDefs.ErrorHti[error, "is not valid as a CONTROL module"L, (stb+index).hti];
        END;
      ENDCASE => error[];
    RETURN[MTNull]
    END;

  NameForLink: PROCEDURE [t: TreeLink] RETURNS [NameRecord] =
    BEGIN
    WITH t SELECT FROM
      symbol => RETURN[BcdUtilDefs.NameForSti[index]];
      ENDCASE => error[];
    END;

  NotLoadable: PROCEDURE [sti: STIndex] =
    BEGIN
    BcdErrorDefs.ErrorSti[error,
      "is not loadable (probably needs ""[]"")"L, sti];
    RETURN
    END;
```

```
LoadSti: PROCEDURE [sti: STIndex, name: HTIndex] =
  BEGIN
  BEGIN
  ENABLE BcdErrorDefs.GetSti => RESUME[sti];
  WITH s: stb+sti SELECT FROM
    external =>
      WITH p:s SELECT FROM
        file => s.map ← Load[sti, name];
        instance => s.map ← Load[p.sti, name];
        ENDCASE => error[];
    local => LoadLocalConfig[s.info, inner, name];
    ENDCASE => NotLoadable[sti];
  END;
  END;

FileForSti: PROCEDURE [sti: STIndex] RETURNS [FTIndex] =
  BEGIN
  IF sti = STNull THEN RETURN[FTNull];
  WITH s: stb+sti SELECT FROM
    unknown => RETURN[FTNull];
    external =>
      WITH p:s SELECT FROM
        file => RETURN[p.fti];
        instance => RETURN[FileForSti[p.sti]];
        ENDCASE => error[];
    ENDCASE => error[];
  END;

FileForPortableItem: PROCEDURE [p: PortableItem] RETURNS [FTIndex] =
  BEGIN
  WITH p SELECT FROM
    interface => RETURN[MapFile[(bcd.etb+expi).file]];
    module => RETURN[MapFile[(bcd.mtb+mti).file]];
    ENDCASE => error[];
  END;

DeclarePortableItem: PROCEDURE [sti: STIndex, p: PortableItem] =
  BEGIN
  WITH p SELECT FROM
    interface => DeclareInterface[sti, expi];
    module => DeclareModule[sti, mti];
    ENDCASE => error[];
  END;

DeclareInterface: PROCEDURE [sti: STIndex, eti: EXPIndex] =
  BEGIN
  fti: FTIndex ← MapFile[(bcd.etb+eti).file];
  WITH s:(stb+sti) SELECT FROM
    external =>
      BEGIN
      s.map ← [interface[EXPNull]];
      WITH p:s SELECT FROM
        instance =>
          IF p.sti = STNull THEN s.pointer ← file[fti]
          ELSE DeclareInterface[p.sti, eti];
        file => p.fti ← fti;
        ENDCASE => error[];
      END;
    unknown =>
      (stb+sti).body ←
        external[pointer: file[fti], map:[interface[EXPNull]]];
    ENDCASE => error[];
  END;

DeclareModule: PROCEDURE [sti: STIndex, mti: MTIndex] =
  BEGIN
  fti: FTIndex;
  WITH s:(stb+sti) SELECT FROM
    external =>
      BEGIN
      s.map ← [module[MTNull]];
      WITH p:s SELECT FROM
        instance => DeclareModule[p.sti, mti];
        file => p.fti ← MapFile[(bcd.mtb+mti).file];
        ENDCASE => error[];
      END;
    unknown =>
      BEGIN
      fti ← MapFile[(bcd.mtb+mti).file];
      (stb+sti).body ←
        external[pointer: file[fti], map:[module[MTNull]]];
      END;
    ENDCASE => error[];
  END;

currentCodeLinks: BOOLEAN;

LoadItem: PROCEDURE [t: TreeLink] =
  BEGIN
  sti: STIndex;
  WITH link: t SELECT FROM
    subtree =>
      BEGIN OPEN i: (tb+link.index);
      IF i.name # item THEN error[];
      WITH s1: i.son1 SELECT FROM
        symbol =>
          BEGIN
          sti ← s1.index;
          currentCodeLinks ← i.codelinks;
          LoadSti[sti, IF i.son2 = empty THEN HTNull ELSE (stb+sti).hti];
          END;
        ENDCASE => error[];
      END;
    ENDCASE => error[];
  END;

BcdRelocations: TYPE = BcdBindDefs.BcdRelocations;

relocationHead: POINTER TO BcdRelocations;
rel: POINTER TO BcdRelocations;

fileMap: DESCRIPTOR FOR ARRAY OF FTIndex;

MapFile: PROCEDURE [fti: FTIndex] RETURNS [FTIndex] =
  BEGIN
  fileIndex: CARDINAL;
  IF bcd = @localBases THEN RETURN[fti];
  IF fti = FTSelf THEN RETURN[bcdFile]
  ELSE IF fti = FTNull THEN RETURN[FTNull];
  fileIndex ← LOOPHOLE[fti,CARDINAL]/SIZE[FTRecord];
  IF fileMap[fileIndex] = FTNull THEN
    fileMap[fileIndex] ← BcdUtilDefs.MergeFile[bcd, fti];
  RETURN[fileMap[fileIndex]]
  END;

AllocateRelocations: PROCEDURE [type: BcdBindDefs.RelocationType] =
  BEGIN
  p: POINTER TO BcdRelocations ← BcdHeapDefs.GetSpace[SIZE[BcdRelocations]];
  MiscDefs.Zero[p,SIZE[BcdRelocations]];
  p.link ← NIL;
  IF relocationHead = NIL THEN relocationHead ← rel ← p
  ELSE BEGIN rel.link ← p; rel ← p END;
  IF (rel.type ← type) = file THEN
    BEGIN
    rel.firstgfi ← rel.lastgfi ← BcdUtilDefs.GetGfi[0];
    rel.dummygfi ← BcdUtilDefs.GetDummyGfi[0];
    rel.import ← TableDefs.TableBounds[imptype].size;
      rel.importLimit ← LOOPHOLE[rel.import];
    rel.module ← TableDefs.TableBounds[mttype].size;
    rel.config ← TableDefs.TableBounds[cttype].size;
    rel.parentcx ← CXNull;
    END
  ELSE
    BEGIN
    rel.originalfirstdummy ← 1;
    END;
  rel.context ← currentCx;
  rel.textIndex ← data.textIndex;
  rel.parameters ← currentParameters;
  currentParameters ← empty;
  RETURN
  END;
  GetRelocationHead: PUBLIC PROCEDURE RETURNS [POINTER TO BcdRelocations] =
    BEGIN
    RETURN[relocationHead]
    END;

  ProcessExports: ExportAssigner;

  Load: PROCEDURE [sti: STIndex, name: HTIndex]
    RETURNS [map: BcdTabDefs.STMap] =
    BEGIN
    cantopen: STRING ← "cannot be opened"L;
    fti: FTIndex = FileForSti[sti];
    i, nFiles: CARDINAL;
    BEGIN
    IF fti = FTNull THEN
      BEGIN
      NotLoadable[SIGNAL BcdErrorDefs.GetSti];
      GOTO return
      END;
    IF fti = data.outputfti THEN
      BcdErrorDefs.Error[error, "Output file referenced as input"L];
    LoadBcd[fti ! BcdFileDefs.UnknownFile =>
      BEGIN
      BcdErrorDefs.ErrorFile[error, cantopen, fti];
      GOTO return
      END];
    CheckInternalName[sti];
    EXITS return => RETURN [[unknown[]]];
    END;
    nFiles ← LOOPHOLE[limits.ft,CARDINAL]/SIZE[FTRecord];
    fileMap ← DESCRIPTOR[BcdHeapDefs.GetSpace[nFiles], nFiles];
    FOR i IN [0..LENGTH[fileMap]) DO fileMap[i] ← FTNull ENDLOOP;
    IF limits.ct # FIRST[CTIndex] THEN
      BEGIN
      map ← [config[LOOPHOLE[TableDefs.TableBounds[cttype].size]]];
      LoadConfigs[name];
      name ← HTNull
      END
    ELSE map ← [module[LOOPHOLE[TableDefs.TableBounds[mttype].size]]];
    LoadModules[name];
    ProcessExports[];
    ProcessImports[];
    rel.lastgfi ← BcdUtilDefs.GetGfi[0]-1;
    rel.importLimit ← LOOPHOLE[TableDefs.TableBounds[imptype].size];
    UnloadBcd[];
    BcdHeapDefs.FreeSpace[BASE[fileMap]];
    END;

  CheckInternalName: PROCEDURE [sti: STIndex] =
    BEGIN
    iname: NameRecord =
      IF limits.ct = FIRST[CTIndex] THEN (bcd.mtb+FIRST[MTIndex]).name
      ELSE (bcd.ctb+FIRST[CTIndex]).name;
    ihti: HTIndex = BcdUtilDefs.HtiForName[bcd, iname];
    IF ihti # (stb+sti).hti THEN
      BcdErrorDefs.ErrorSti[error, "does not name a module or configuration"L, sti];
    RETURN
    END;

  bcdSegment: SegmentDefs.FileSegmentHandle;
  bcdFile: FTIndex;

  LoadBcd: PROCEDURE [fti: FTIndex] =
    BEGIN OPEN SegmentDefs;
    pages: CARDINAL;
    bHeader: POINTER TO BCD;
    SwapIn[bcdSegment ← NewFileSegment[BcdFileDefs.HandleForFile[fti],
      1, 1, Read]];
    bHeader ← FileSegmentAddress[bcdSegment];
    IF bHeader.versionident # BcdDefs.VersionID OR bHeader.definitions THEN
      BEGIN
      Unlock[bcdSegment];
      DeleteFileSegment[bcdSegment];
      bcdSegment ← NIL;
      ERROR BcdFileDefs.UnknownFile[fti];
      END;
    IF (pages←bHeader.nPages) # 1 THEN
      BEGIN
      Unlock[bcdSegment];
      MoveFileSegment[bcdSegment,bcdSegment.base,pages];
      SwapIn[bcdSegment];
      bHeader ← FileSegmentAddress[bcdSegment];
      END;
    bcdFile ← fti;
    BcdUtilDefs.SetFileVersion[fti,bHeader.version];
    bcd ← BcdHeapDefs.GetSpace[SIZE[BcdUtilDefs.BcdBases]];
    bcd↑ ← [
      ctb: LOOPHOLE[bHeader+bHeader.ctOffset],
      mtb: LOOPHOLE[bHeader+bHeader.mtOffset],
      etb: LOOPHOLE[bHeader+bHeader.expOffset],
      itb: LOOPHOLE[bHeader+bHeader.impOffset],
      sgb: LOOPHOLE[bHeader+bHeader.sgOffset],
      ftb: LOOPHOLE[bHeader+bHeader.ftOffset],
      ssb: LOOPHOLE[bHeader+bHeader.ssOffset],
      ntb: LOOPHOLE[bHeader+bHeader.ntOffset]];
    limits ← [
      ct: bHeader.ctLimit,
      sg: bHeader.sgLimit,
      ft: bHeader.ftLimit,
      mt: bHeader.mtLimit,
      et: bHeader.expLimit,
      it: bHeader.impLimit,
      nt: bHeader.ntLimit];
    AllocateRelocations[file];
    rel.originalfirstdummy ← bHeader.firstdummy;
    RETURN
    END;

  UnloadBcd: PROCEDURE =
    BEGIN OPEN SegmentDefs;
    file: FileHandle = bcdSegment.file;
    Unlock[bcdSegment];
    DeleteFileSegment[bcdSegment];
    bcdSegment ← NIL;
    BcdHeapDefs.FreeSpace[bcd];
    bcd ← @localBases;
    RETURN
    END;

  EnumerateConfigurations: PROCEDURE [proc: PROCEDURE [CTIndex]] =
    BEGIN
    cti: CTIndex;
    cti ← FIRST[CTIndex];
    UNTIL cti = limits.ct DO
      proc[cti];
      cti ← cti +  SIZE[CTRecord];
      ENDLOOP;
    RETURN
    END;

  LoadConfigs: PROCEDURE [name: HTIndex] =
    BEGIN
    LoadOne: PROCEDURE [cti: CTIndex] =
      BEGIN
      newcti: CTIndex ← BcdUtilDefs.EnterConfig[bcd, cti, name];
        BEGIN OPEN new: localBases.ctb+newcti;
        name ← HTNull;
        IF new.config = CTNull THEN new.config ← currentCti
        ELSE new.config ← new.config + rel.config;
        new.file ← MapFile[new.file];
        IF new.control # MTNull THEN
          new.control ← new.control + rel.module;
        END;
      END;
    EnumerateConfigurations[LoadOne];
    RETURN
    END;

  EnumerateModules: PROCEDURE [proc: PROCEDURE [MTIndex] RETURNS [BOOLEAN]]
  RETURNS [mti: MTIndex] =
    BEGIN
    mti ← FIRST[MTIndex];
    UNTIL mti = limits.mt DO
      IF proc[mti] THEN RETURN;
      mti ← mti + SIZE[MTRecord] + (bcd.mtb+mti).frame.length;
      ENDLOOP;
    RETURN[MTNull]
    END;

  CheckPacking: PROCEDURE [mti: MTIndex] =
    BEGIN
    sti: STIndex;
    name: NameRecord = (localBases.mtb+mti).name;
    FOR sti ← packSti, (stb+sti).link UNTIL sti = STNull DO
      IF BcdUtilDefs.NameForSti[sti] = name THEN
        BEGIN
        (stb+sti).body ← external[
          map:[module[mti]], pointer: file[(localBases.mtb+mti).file]];
        EXIT;
        END;
      ENDLOOP;
    RETURN
    END;

  MapSegment: PROCEDURE [oldsgi: SGIndex] RETURNS [SGIndex] =
    BEGIN
    seg: SGRecord ← (bcd.sgb+oldsgi)↑;
    seg.file ← MapFile[seg.file];
    RETURN[BcdUtilDefs.EnterSegment[seg]]
    END;

  LoadModules: PROCEDURE [name: HTIndex] =
    BEGIN
    LoadOne: PROCEDURE [mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN
      newmti: MTIndex ← BcdUtilDefs.EnterModule[bcd, mti, name];
        BEGIN OPEN new: localBases.mtb+newmti;
        name ← HTNull;
        IF new.config = CTNull THEN new.config ← currentCti
        ELSE new.config ← new.config + rel.config;
        new.gfi ← BcdUtilDefs.GetGfi[new.ngfi];
        new.file ← MapFile[new.file];
        new.code.sgi ← MapSegment[new.code.sgi];
        new.sseg ← MapSegment[new.sseg];
        CheckPacking[newmti];
        IF currentCodeLinks THEN new.links ← code;
        END;
      data.nModules ← data.nModules + 1;
      RETURN[FALSE]
      END;
     [] ← EnumerateModules[LoadOne];
    RETURN
    END;

  ProcessImports: PROCEDURE =
    BEGIN
    newimpi, impi: IMPIndex;
    sti: STIndex;
    [impi,sti] ← FirstImport[];
    UNTIL impi = IMPNull DO
      OPEN new: localBases.itb+newimpi;
      newimpi ← BcdUtilDefs.EnterImport[bcd, impi, HTNull];
      new.file ← MapFile[new.file];
      [] ← BcdUtilDefs.GetDummyGfi[new.ngfi];
      [impi,sti] ← NextImport[impi, sti];
      ENDLOOP;
    RETURN
    END;

  nextLocalGfi: CARDINAL;

  GetLocalGfi: PROCEDURE [n: CARDINAL] RETURNS [gfi: GFTIndex] =
    BEGIN
    gfi ← nextLocalGfi;
    nextLocalGfi ← nextLocalGfi + n;
    [] ← BcdUtilDefs.GetDummyGfi[n];
    END;
```

```
ProcessLocalImports: PROCEDURE [start: IMPIndex] =
  BEGIN
  newimpi, impi: IMPIndex;
  sti: STIndex;
  CantImport: PROCEDURE =
    BEGIN
    BcdErrorDefs.ErrorSti[error, "Cannot be IMPORTed"L, sti];
    END;
  nextLocalGfi ← 1;
  [impi,sti] ← FirstImport[];
  UNTIL sti = STNull DO
    OPEN new: localBases.itb+newimpi;
    WITH s:stb+sti SELECT FROM
      unknown => DeclareImportByName[sti, start];
      external =>
        WITH m:s.map SELECT FROM
          interface => DeclareImport[sti, m.expi];
          unknown => DeclareImportByName[sti, start];
          ENDCASE => error[];
      ENDCASE => error[];
    [impi,sti] ← NextImport[impi, sti];
    ENDLOOP;
  RETURN
  END;

DeclareImportByName: PROCEDURE [sti: STIndex, start: IMPIndex] =
  BEGIN
  name: NameRecord;
  impi: IMPIndex;
  maxngfi: [1..4] ← 1;
  firstimpi: IMPIndex ← IMPNull;
  impLimit: IMPIndex = LOOPHOLE[TableDefs.TableBounds[imptype].size];
  WITH s:stb+sti SELECT FROM
    external =>
      WITH p:s SELECT FROM
        file => name ← BcdUtilDefs.NameForSti[sti];
        instance => name ← BcdUtilDefs.NameForSti[p.sti];
        ENDCASE => error[];
    unknown => name ← BcdUtilDefs.NameForSti[sti];
    ENDCASE => error[];
  FOR impi ← start, impi+SIZE[IMPRecord] UNTIL impi = impLimit DO
    IF (localBases.itb+impi).name = name THEN
      BEGIN
      IF firstimpi = IMPNull THEN firstimpi ← impi;
      maxngfi ← MAX[maxngfi, (localBases.itb+impi).ngfi];
      END;
    ENDLOOP;
  IF firstimpi = IMPNull THEN
    BEGIN
    BcdErrorDefs.ErrorName[warning, "is not IMPORTed by any modules"L,name];
    (stb+sti).imported ← FALSE;
    RETURN
    END;
  (stb+sti).impi ← impi ←
     BcdUtilDefs.EnterImport[@localBases, firstimpi, HTNull];
  (localBases.itb+impi).ngfi ← maxngfi;
  (localBases.itb+impi).gfi ← GetLocalGfi[maxngfi];
  WITH s:stb+sti SELECT FROM
    unknown => (stb+sti).body ← external[
      map:[unknown[]],
      pointer:file[(localBases.itb+impi).file]];
    ENDCASE;
  END;

DeclareImport: PROCEDURE [sti: STIndex, expi: EXPIndex] =
  BEGIN OPEN localBases, exp: localBases.etb+expi;
  impi: IMPIndex ← TableDefs.Allocate[imptype, SIZE[IMPRecord]];
  (itb+impi)↑ ← [
    port: interface,
    namedinstance: FALSE,
    file: exp.file,
    ngfi: (exp.size+EPLimit-1)/EPLimit,
    name:,
    gfi:];
  (itb+impi).name ← BcdUtilDefs.NameForSti[sti];
```

```
    (itb+impi).gfi ← GetLocalGfi[(itb+impi).ngfi];
    (stb+sti).impi ← impi;
    WITH s:stb+sti SELECT FROM
      unknown => (stb+sti).body ← external[
        map:[unknown[]],
        pointer:file[exp.file]];
      ENDCASE;
    RETURN
    END;

  Lookup: PROCEDURE [hti: HTIndex] RETURNS [sti: STIndex] =
    BEGIN
    last: STIndex;
    IF hti = HTNull THEN RETURN[STNull];
    FOR sti ← (cxb+currentCx).link, (stb+sti).link UNTIL sti = STNull DO
      IF (stb+sti).hti = hti THEN RETURN;
      last ← sti;
      ENDLOOP;
    sti ← BcdUtilDefs.NewSemanticEntry[hti];
    (stb+sti).hti ← hti;
    (stb+last).link ← sti;
    RETURN
    END;

  FirstImport: PROCEDURE RETURNS [IMPIndex, STIndex] =
    BEGIN OPEN localBases;
    sti: STIndex;
    IF loadCx = CXNull THEN
      RETURN[
        IF limits.it = FIRST[IMPIndex] THEN IMPNull ELSE FIRST[IMPIndex],
        STNull];
    FOR sti ← (cxb+loadCx).link, (stb+sti).link UNTIL sti = STNull DO
      IF (stb+sti).imported THEN RETURN[IMPNull,sti];
      ENDLOOP;
    RETURN[IMPNull,STNull]
    END;

  NextImport: PROCEDURE [impi: IMPIndex, sti: STIndex]
    RETURNS [IMPIndex, STIndex] =
    BEGIN OPEN localBases;
    IF loadCx = CXNull THEN
      BEGIN
      IF impi = IMPNull THEN RETURN [impi, sti];
      impi ← impi + SIZE[IMPRecord];
      IF impi = limits.it THEN impi ← IMPNull;
      RETURN[impi, STNull];
      END;
    IF sti = STNull THEN RETURN [impi, sti];
    UNTIL (sti ← (stb+sti).link) = STNull DO
      IF (stb+sti).imported THEN RETURN[IMPNull,sti]
      ENDLOOP;
    RETURN[IMPNull,STNull]
    END;

  PortableItem: TYPE = RECORD [
    SELECT type: * FROM
      interface => [expi: EXPIndex],
      module => [mti: MTIndex],
      unknown => [name: HTIndex],
      null => [fill: TableDefs.TableIndex],
      ENDCASE];

  PortNull: PortableItem = [null[EXPNull]];

  HtiForPortable: PROCEDURE [p: PortableItem] RETURNS [HTIndex] =
    BEGIN OPEN BcdUtilDefs;
    WITH p SELECT FROM
      interface => RETURN[HtiForName[bcd, (bcd.etb+expi).name]];
      module => RETURN[HtiForName[bcd, (bcd.mtb+mti).name]];
      ENDCASE;
    RETURN[HTNull]
    END;

  EnumerateExports: PROCEDURE [proc: PROCEDURE [PortableItem]]
    RETURNS [PortableItem] =
    BEGIN OPEN localBases;
    eti: EXPIndex;
    PassItOn: TreeScan =
      BEGIN
      sti: STIndex;
      WITH t SELECT FROM
        symbol => sti ← index;
        subtree => WITH (tb+index).son1 SELECT FROM
          symbol => sti ← index;
          ENDCASE => error[];
        ENDCASE => error[];
      IF ~(stb+sti).exported THEN RETURN;
      WITH s:stb+sti SELECT FROM
        external =>
          WITH m:s.map SELECT FROM
            interface => proc[[interface[m.expi]]];
            module => proc[[module[m.mti]]];
            ENDCASE => proc[[unknown[s.hti]]];
        ENDCASE => proc[[unknown[s.hti]]];
      END;

    SELECT TRUE FROM
      (loadExpi # EXPNull) => proc[[interface[loadExpi]]];
      (loadTree = nullTreeIndex) =>
        FOR eti ← FIRST[EXPIndex], eti+SIZE[EXPRecord]+(bcd.etb+eti).size
          UNTIL eti = limits.et DO
          proc[[interface[eti]]];
          ENDLOOP;
      ENDCASE => scanlist[(tb+loadTree).son2, PassItOn];
    RETURN[PortNull]
    END;

  VerifyExports: ExportAssigner =
    BEGIN
    ExportOne: PROCEDURE [p: PortableItem] =
      BEGIN
      WITH p SELECT FROM
        unknown =>
          BEGIN
          BcdErrorDefs.ErrorHti[warning, "is not EXPORTed by any modules"L, name];
          RETURN;
          END;
        ENDCASE;
      END;
    [] ← EnumerateExports[ExportOne];
    RETURN
    END;

  NormalExports: ExportAssigner =
    BEGIN
    ExportOne: PROCEDURE [p: PortableItem] =
      BEGIN
      CombineExport[
        Lookup[HtiForPortable[p]], p, currentOp];
      END;
    [] ← EnumerateExports[ExportOne];
    RETURN
    END;

  lhs: TreeLink;

  AssignedExports: ExportAssigner =
    BEGIN
    port: TYPE = MACHINE DEPENDENT RECORD[in,out: UNSPECIFIED];
    left: PORT [TreeLink] RETURNS [TreeLink];
    right: PORT RETURNS [PortableItem];
    t: TreeLink;
    p: PortableItem;
    LOOPHOLE[left,port].out ← updatelist;
    LOOPHOLE[right,port].out ← EnumerateExports;
    t ← LOOPHOLE[left,PORT[TreeLink,POINTER] RETURNS [TreeLink]][lhs, @left];
    p ← LOOPHOLE[right,PORT[POINTER] RETURNS [PortableItem]][@right];
    DO
      WITH t SELECT FROM
        symbol => CombineExport[index, p, currentOp];
        subtree =>
          BEGIN OPEN tb+index;
        IF name # item THEN error[];
        WITH son1 SELECT FROM
          symbol => CombineExport[index, p, currentOp];
          ENDCASE => error[];
        END;
      ENDCASE => error[];
    t ← left[t];
    p ← right[];
    IF t = lhs THEN
      BEGIN
      IF p = PortNull THEN EXIT;
      BcdErrorDefs.Error[error, "Too many exports in right hand side of assignment"L];
      UNTIL p = PortNull DO p ← right[] ENDLOOP;
      EXIT
      END;
    IF p = PortNull THEN
      BEGIN
      BcdErrorDefs.Error[error, "Too few exports in right hand side of assignment"L];
      UNTIL t = lhs DO t ← left[t] ENDLOOP;
      EXIT
      END;
    ENDLOOP;
  RETURN
  END;

LoadAssign: PROCEDURE [t: TreeIndex] =
  BEGIN
  saveAssigner: ExportAssigner ← ProcessExports;
  ProcessExports ← AssignedExports;
  lhs ← (tb+t).son1;
  LoadExpression[(tb+t).son2];
  ProcessExports ← saveAssigner;
  END;

NewExport: PROCEDURE [expi: EXPIndex] RETURNS [newexpi: EXPIndex] =
  BEGIN
  OPEN new: localBases.etb+newexpi;
  newexpi ← BcdUtilDefs.EnterExport[bcd, expi, HTNull];
  new.file ← MapFile[new.file];
  END;

CombineExport: PROCEDURE [sti: STIndex, p: PortableItem, op: InterfaceOp] =
  BEGIN
  target: FTIndex ← FileForSti[sti];
  WITH p SELECT FROM
    unknown =>
      BEGIN
      BcdErrorDefs.ErrorHti[warning, "is not EXPORTed by any modules"L, name];
      RETURN;
      END;
    ENDCASE;
  IF target = FTNull THEN DeclarePortableItem[sti,p]
  ELSE IF FileForPortableItem[p] # target THEN
    BcdErrorDefs.Error2Files[error, "cannot be exported as"L, FileForPortableItem[p],target];
  WITH p SELECT FROM
    interface => CombineInterface[sti, expi, op];
    module => CombineModule[sti, mti, op];
    ENDCASE;
  RETURN
  END;

CombineModule: PROCEDURE [sti: STIndex, mti: MTIndex, op: InterfaceOp] =
  BEGIN
  WITH s:(stb+sti) SELECT FROM
    external =>
      WITH m:s.map SELECT FROM
        module =>
          IF m.mti = MTNull THEN
            BEGIN m.mti ← mti; RETURN END
          ELSE IF op = plus THEN
            BcdErrorDefs.ErrorModule[warning, "is a duplicate export"L,m.mti];
        unknown =>
          s.map ← [module[BcdUtilDefs.EnterModule[bcd, mti, HTNull]]];
        ENDCASE => error[];
    ENDCASE => error[];
  RETURN
```

```
    END;


  CombineInterface: PROCEDURE [sti: STIndex, eti: EXPIndex, op: InterfaceOp] =
    BEGIN
    i: CARDINAL;
    neweti: EXPIndex;
    WITH s:(stb+sti) SELECT FROM
      external =>
        WITH m:s.map SELECT FROM
          interface =>
            BEGIN
            IF m.expi = EXPNull THEN m.expi ← NewExport[eti];
            neweti ← m.expi;
            END;
          unknown =>
            BEGIN
            neweti ← NewExport[eti];
            s.map ← [interface[neweti]];
            END;
          ENDCASE => error[];
      ENDCASE => error[];
    BEGIN OPEN old: bcd.etb+eti, new: localBases.etb+neweti;
    FOR i IN [0..old.size) DO
      IF old.links[i] # NullLink THEN
        BEGIN
        IF new.links[i] = NullLink THEN
          new.links[i] ← RelocateExportLink[old.links[i]]
        ELSE IF op=plus THEN
          BcdErrorDefs.ErrorItem[warning, "is a duplicate export"L, i
            ! BcdErrorDefs.GetInterface => RESUME[neweti]];
        END;
      ENDLOOP;
    END;
    RETURN
    END;


  RelocateExportLink: PROCEDURE [cl: ControlLink] RETURNS [ControlLink] =
    BEGIN
    IF loadExpi = EXPNull AND loadCx = CXNull THEN
      cl.gfi ← cl.gfi + rel.firstgfi-1;
    RETURN[cl]
    END;

  xLoadSti: PROCEDURE [sti: STIndex] =
    BEGIN
    WITH s: stb+sti SELECT FROM
      external =>
        WITH m:s.map SELECT FROM
          interface =>
            BEGIN
            IF m.expi = EXPNull THEN error[];
            loadExpi ← m.expi;
            ProcessExports[];
            loadExpi ← EXPNull;
            END;
          ENDCASE => LoadSti[sti,HTNull];
      ENDCASE => LoadSti[sti,HTNull];
    END;

  xLoadItem: PROCEDURE [t: TreeLink] =
    BEGIN
    WITH link: t SELECT FROM
      subtree =>
        BEGIN OPEN i: (tb+link.index);
        IF i.name # item THEN error[];
        WITH s1: i.son1 SELECT FROM
          symbol =>
            BEGIN
            WITH s: stb+s1.index SELECT FROM
              external =>
                WITH m:s.map SELECT FROM
                  interface =>
                    BEGIN
                    xLoadSti[s1.index];
                  RETURN
                  END;
                ENDCASE;
            ENDCASE;
          LoadSti[s1.index, (stb+s1.index).hti];
          END;
        ENDCASE => error[];
      END;
    ENDCASE => error[];
  END;

LoadExpression: PROCEDURE [exp: TreeLink] =
  BEGIN
  WITH exp SELECT FROM
    symbol => xLoadSti[index];
    subtree =>
      SELECT (tb+index).name FROM
        item => xLoadItem[exp];
        module =>
          BEGIN
          currentParameters ← (tb+index).son2;
          LoadItem[(tb+index).son1];
          END;
        plus, then =>
          BEGIN OPEN tb+index;
          LoadExpression[son1];
          IF name = then THEN currentOp ← then;
          LoadExpression[son2];
          currentOp ← plus;
          END;
        ENDCASE => error[];
    ENDCASE => error[];
  RETURN
  END;


END...
```