```
-- LoaderUtilities.mesa
-- Last Modified by Sandman,  May 12, 1978  4:41 PM

DIRECTORY
  AltoFileDefs: FROM "altofiledefs" USING [FP],
  BcdDefs: FROM "bcddefs" USING [
    CTHandle, CTIndex, CTNull, FTIndex, FTNull, FTSelf, MTHandle, MTIndex,
    MTNull, NameString, SGHandle, SGIndex],
  ControlDefs: FROM "controldefs" USING [
    Alloc, GFT, GFTIndex, GFTNull, GlobalFrameHandle, MaxAllocSlot,
    NullGlobalFrame],
  DirectoryDefs: FROM "directorydefs" USING [EnumerateDirectory],
  FrameDefs: FROM "framedefs" USING [FrameSize],
  InlineDefs: FROM "inlinedefs" USING [BITAND],
  LoaderBcdUtilDefs: FROM "loaderbcdutildefs" USING [
    EnumerateConfigTable, EnumerateModuleTable, EnumerateSegTable],
  LoaderDefs: FROM "loaderdefs",
  LoaderUtilityDefs: FROM "loaderutilitydefs" USING [
    BcdBase, Binding, FileHandle, FileSegmentHandle, FileTable,
    FileTableObject, GlobalFrameHandle, ImportBindingLink, Relocation],
  SegmentDefs: FROM "segmentdefs" USING [
    EnumerateFileSegments, FileHandle, FileSegmentHandle, InsertFile,
    NewFileSegment, Read, ReleaseFile, VMtoFileSegment],
  StringDefs: FROM "stringdefs" USING [
    AppendString, AppendSubString, EquivalentSubStrings, SubString,
    SubStringDescriptor],
  SystemDefs: FROM "systemdefs" USING [AllocateHeapNode, FreeHeapNode];

DEFINITIONS FROM BcdDefs, LoaderUtilityDefs;

LoaderUtilities: PROGRAM
  IMPORTS DirectoryDefs, FrameDefs, LoaderBcdUtilDefs,
    SegmentDefs, StringDefs, SystemDefs
  EXPORTS LoaderDefs, LoaderUtilityDefs = PUBLIC

  BEGIN

  files: FileTable ← NIL;
  loadee: BcdBase;
  ssb: BcdDefs.NameString;
  ftb: CARDINAL;
  nfilestofind: CARDINAL ← 0;
  tableopen: BOOLEAN ← FALSE;

  SubStringDescriptor: TYPE = StringDefs.SubStringDescriptor;

  AddFileName: PUBLIC PROCEDURE [file: BcdDefs.FTIndex] =
    BEGIN
    p: FileTable;
    i, offset, length: CARDINAL;
    FOR p ← files, p.link UNTIL p = NIL DO
      IF file = p.file THEN RETURN;
      ENDLOOP;
    p ← SystemDefs.AllocateHeapNode[SIZE[FileTableObject]];
    p↑ ← [file: file, filehandle: NIL, ext: FALSE, link: files];
    files ← p;
    IF file = BcdDefs.FTSelf THEN
      BEGIN
      p.filehandle ← SegmentDefs.VMtoFileSegment[loadee].file;
      RETURN
      END;
    IF file = BcdDefs.FTNull THEN BEGIN  p.filehandle ← NIL;  RETURN  END;
    offset ← (ftb+file).name;
    length ← ssb.size[(ftb+file).name];
    FOR i IN [offset..offset+length) DO
      IF ssb.string.text[i] = '. THEN
        BEGIN  p.ext ← TRUE;  EXIT  END;
      ENDLOOP;
    nfilestofind ← nfilestofind + 1;
    RETURN;
    END;

  FindFileName: PUBLIC PROCEDURE [name: StringDefs.SubString, ext: BOOLEAN]
   RETURNS [found: BOOLEAN, file: FileTable] =
    BEGIN OPEN StringDefs;
    filename: SubStringDescriptor ← [base: @ssb.string, offset:, length:];
```

```
    FOR file ← files, file.link UNTIL file = NIL DO
      filename.offset ← (ftb+file.file).name;
      filename.length ← ssb.size[(ftb+file.file).name];
      IF LastCharIsDot[@filename] THEN name.length ← name.length + 1;
      IF ext = file.ext AND EquivalentSubStrings[@filename, name] THEN
        RETURN[TRUE, file];
      ENDLOOP;
    RETURN[FALSE, NIL];
    END;

  LastCharIsDot: PUBLIC PROCEDURE [name: StringDefs.SubString] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[name.base[name.offset+name.length-1] = '.];
    END;

  FileNotFound: PUBLIC SIGNAL [name: STRING] RETURNS [file: FileHandle] = CODE;

  LookupFileTable: PUBLIC PROCEDURE =
    BEGIN
    p: FileTable;
    ssd: StringDefs.SubStringDescriptor;
    name: STRING ← [40];
    IF nfilestofind # 0 THEN DirectoryDefs.EnumerateDirectory[CheckOne];
    FOR p ← files, p.link UNTIL p = NIL DO
      IF p.filehandle = NIL AND p.file # BcdDefs.FTNull THEN
        BEGIN
        ssd ← [base: @ssb.string, offset: (ftb+p.file).name,
          length: ssb.size[(ftb+p.file).name]];
        name.length ← 0;
        StringDefs.AppendSubString[name, @ssd];
        IF p.ext THEN StringDefs.AppendString[name, ".bcd"L];
        p.filehandle ← SIGNAL FileNotFound[name];
        END;
      ENDLOOP;
    END;

  CheckOne: PROCEDURE [fp: POINTER TO AltoFileDefs.FP, name: STRING]
   RETURNS [found: BOOLEAN] =
    BEGIN OPEN StringDefs;
    i: CARDINAL;
    dirName: SubStringDescriptor;
    bcd: SubStringDescriptor ← [base: "bcd"L, offset: 0, length: 3];
    file: LoaderUtilityDefs.FileTable;
    FOR i IN [0..name.length) DO
      IF name[i] = '. THEN
        BEGIN
        IF name.length-i # 5 THEN GOTO UseWholeName;
        dirName ← [base: name, offset: i+1, length: 3];
        IF ~EquivalentSubStrings[@dirName, @bcd] THEN GOTO UseWholeName;
        dirName.offset ← 0;  dirName.length ← i;
        GOTO HasBCDExtension;
        END;
      REPEAT
        UseWholeName => NULL;
        HasBCDExtension =>
          BEGIN
          [found, file] ← FindFileName[@dirName, FALSE];
          IF found THEN RETURN [ThisIsTheOne[fp, file]];
          END;
      ENDLOOP;
    dirName ← [base: name, offset: 0, length: name.length-1]; -- ignore dot on end
    [found, file] ← FindFileName[@dirName, TRUE];
    RETURN [IF found THEN ThisIsTheOne[fp, file] ELSE FALSE];
    END;

  ThisIsTheOne: PROCEDURE [fp: POINTER TO AltoFileDefs.FP, file: FileTable]
   RETURNS [BOOLEAN] =
    BEGIN
    file.filehandle ← SegmentDefs.InsertFile[fp, SegmentDefs.Read];
    nfilestofind ← nfilestofind - 1;
    RETURN [nfilestofind=0];
    END;

  FileHandleFromTable: PUBLIC PROCEDURE [filename: BcdDefs.FTIndex]
   RETURNS [file: SegmentDefs.FileHandle] =
    BEGIN
```

```
    p: FileTable;
    FOR p ← files, p.link UNTIL p = NIL DO
      IF p.file = filename THEN
        RETURN[p.filehandle];
      ENDLOOP;
    RETURN[NIL];
    END;

  FinalizeUtilities: PUBLIC PROCEDURE =
    BEGIN
    p: FileTable;
    FOR files ← files, p UNTIL files = NIL DO
      p ← files.link;
      IF files.file # NIL AND files.filehandle.segcount = 0 THEN
        SegmentDefs.ReleaseFile[files.filehandle];
      SystemDefs.FreeHeapNode[files];
      ENDLOOP;
    tableopen ← FALSE;
    SystemDefs.FreeHeapNode[BASE[ModuleTable]];
    END;

  InitializeUtilities: PUBLIC PROCEDURE [bcd: BcdBase] =
    BEGIN OPEN SystemDefs;
    loadee ← bcd;
    ssb ← LOOPHOLE[loadee+loadee.ssOffset];
    ftb ← LOOPHOLE[loadee+loadee.ftOffset];
    IF tableopen THEN FinalizeUtilities[];
    tableopen ← TRUE;
    ModuleTable ← DESCRIPTOR[
      AllocateHeapNode[bcd.nModules*SIZE[ModuleInfo]], bcd.nModules];
    nModulesEntered ← 0;
    END;

-- Utility Routines

  AssignControlModules: PUBLIC PROCEDURE [loadee: BcdBase, Reloc: Relocation] =
    BEGIN OPEN ControlDefs;
    ctb: CARDINAL ← LOOPHOLE[loadee+loadee.ctOffset];
    mtb: CARDINAL ← LOOPHOLE[loadee+loadee.mtOffset];
    ModuleSearch: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN OPEN ControlDefs;
      frame: GlobalFrameHandle ← GFT[Reloc[mth.gfi]].frame;
      cti: CTIndex;
      gfi: GFTIndex;
      ControlGfi: PROCEDURE [cti: CTIndex] RETURNS [GFTIndex] =
        BEGIN
        RETURN[IF cti = CTNull OR (ctb+cti).control = MTNull THEN GFTNull
          ELSE (mtb+(ctb+cti).control).gfi];
        END;
      gfi ← ControlGfi[cti ← mth.config];
      WHILE gfi = mth.gfi DO gfi ← ControlGfi[cti ← (ctb+cti).config] ENDLOOP;
      frame.global[0] ←
        (IF gfi = GFTNull THEN NullGlobalFrame ELSE GFT[Reloc[gfi]].frame);
      RETURN [FALSE];
      END;

    [] ← LoaderBcdUtilDefs.EnumerateModuleTable[loadee, ModuleSearch];
    END;

  EnterCodeFileNames: PUBLIC PROCEDURE [loadee: BcdBase] =
    BEGIN
    SegSearch: PROCEDURE [sgh: SGHandle, sgi: SGIndex] RETURNS [BOOLEAN] =
      BEGIN
      IF sgh.class = code THEN AddFileName[sgh.file];
      RETURN[FALSE];
      END;
    [] ← LoaderBcdUtilDefs.EnumerateSegTable[loadee, SegSearch];
    RETURN;
    END;

  AllocateSingleModule: PUBLIC PROCEDURE [
    loadee: BcdBase, framelinks: BOOLEAN] RETURNS [frame: POINTER] =
    BEGIN
    fsi: CARDINAL ← 0;
    i: CARDINAL;
    mth: MTHandle ← LOOPHOLE[loadee+loadee.mtOffset,CARDINAL]+FIRST[MTIndex];
```

```
    framelinks ← framelinks OR mth.links = frame OR ~mth.code.linkspace;
    IF framelinks THEN fsi ← mth.frame.length;
    fsi ← NextMultipleOfFour[fsi] + mth.framesize;
    FOR i IN [0..ControlDefs.MaxAllocSlot) DO
      IF FrameDefs.FrameSize[i] >= fsi THEN
        BEGIN fsi ← i; EXIT END;
      ENDLOOP;
    frame ← ControlDefs.Alloc[fsi];
    IF framelinks THEN
      frame ← NextMultipleOfFour[frame + mth.frame.length];
    RETURN[frame];
    END;

  NextMultipleOfFour: PROCEDURE [x: UNSPECIFIED] RETURNS [UNSPECIFIED] =
    BEGIN
    RETURN[x + InlineDefs.BITAND[-x, 3B]];
    END;

  RequiredFrameSpace: PUBLIC PROCEDURE [
    loadee: BcdBase, alloc, framelinks: BOOLEAN] RETURNS [space: CARDINAL] =
    BEGIN
    FrameSize: PROCEDURE [mth: MTHandle, mti: MTIndex] RETURNS [BOOLEAN] =
      BEGIN
      IF alloc THEN space ← NextMultipleOfFour[space+1];
      IF framelinks OR mth.links = frame OR ~mth.code.linkspace THEN
        space ← space + mth.frame.length;
      space ← NextMultipleOfFour[space] + mth.framesize;
      IF alloc AND ~framelinks AND mth.links = code AND mth.code.linkspace AND
        mth.framesize <= 4 THEN space ← space+3;  -- this tries
        -- to catch the case where a frame is alloced and framesize <= 4 so
        -- it makes it so that enough space is counted so that a small frame
        -- will fit.
      RETURN[FALSE];
      END;
    space ← 0;
    [] ← LoaderBcdUtilDefs.EnumerateModuleTable[loadee, FrameSize];
    RETURN;
    END;

  ControlModuleFrame: PUBLIC PROCEDURE [loadee: BcdBase, Reloc: Relocation]
    RETURNS [ControlDefs.GlobalFrameHandle] =
    BEGIN OPEN ControlDefs;
    mtb: CARDINAL ← LOOPHOLE[loadee + loadee.mtOffset];
    control: MTIndex ← MTNull;
    ConfigSearch: PROCEDURE [cth: CTHandle, cti: CTIndex] RETURNS [BOOLEAN] =
      BEGIN
      IF cth.config = CTNull THEN
        BEGIN
        control ← cth.control;
        RETURN [TRUE];
        END;
      RETURN [FALSE];
      END;
    [] ← LoaderBcdUtilDefs.EnumerateConfigTable[loadee, ConfigSearch];
    RETURN[IF control = MTNull THEN ControlDefs.NullGlobalFrame
      ELSE GFT[Reloc[(mtb+control).gfi]].frame];
    END;

  InitImportBinding: PUBLIC PROCEDURE [size: CARDINAL]
    RETURNS [binding: Binding] =
    BEGIN OPEN SystemDefs;
    i: CARDINAL;
    binding ←
      DESCRIPTOR[AllocateHeapNode[size*SIZE[ImportBindingLink]], size];
    FOR i IN [0..size) DO
      binding[i] ← [whichgfi: 0, body: notbound[]];
      ENDLOOP;
    END;

  FindFrameIndex: PUBLIC PROCEDURE [
    mth: MTHandle, framelinks: BOOLEAN] RETURNS [fsi: CARDINAL] =
    BEGIN
    space: CARDINAL ← 0;
    IF framelinks THEN space ← mth.frame.length;
    space ← NextMultipleOfFour[space] + mth.framesize;
    FOR fsi DECREASING IN [0..ControlDefs.MaxAllocSlot) DO
```

```
    IF space >= FrameDefs.FrameSize[fsi] THEN RETURN[fsi];
    ENDLOOP;
  RETURN[0]; -- see RequiredFrameSpace for alloced modules w/ framesize<7
  END;

ModuleInfo: TYPE = RECORD [
  mth: MTHandle,
  frame: GlobalFrameHandle,
  bound: BOOLEAN,
  sgi: SGIndex];

ModuleTable: DESCRIPTOR FOR ARRAY OF ModuleInfo;
nModulesEntered: CARDINAL;

FindCodeSegment: PUBLIC PROCEDURE [
  loadee: BcdBase, mth: MTHandle, frame: GlobalFrameHandle]
  RETURNS [seg: FileSegmentHandle] =
  BEGIN OPEN SegmentDefs;
  sgh: SGHandle ← mth.code.sgi+LOOPHOLE[loadee+loadee.sgOffset, CARDINAL];
  file: FileHandle;
  i: CARDINAL;
  pages: CARDINAL;
  FindSegment: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[s.file = file AND s.base = sgh.base AND s.pages = pages];
    END;
  FOR i IN [0..nModulesEntered) DO
    IF ModuleTable[i].mth.code.sgi = mth.code.sgi THEN
      RETURN[ModuleTable[i].frame.codesegment];
    ENDLOOP;
  file ← FileHandleFromTable[sgh.file];
  pages ← sgh.pages+sgh.extraPages;
  seg ← EnumerateFileSegments[FindSegment];
  IF seg = NIL THEN seg ← NewFileSegment[file, sgh.base, pages, Read];
  ModuleTable[nModulesEntered] ←
    ModuleInfo[mth, frame, FALSE, mth.code.sgi];
  nModulesEntered ← nModulesEntered + 1;
  RETURN
  END;

ModuleIsBound: PUBLIC PROCEDURE [mth: MTHandle] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..nModulesEntered) DO
    IF ModuleTable[i].mth = mth THEN ModuleTable[i].bound ← TRUE;
    ENDLOOP;
  RETURN
  END;

IsModuleBound: PUBLIC PROCEDURE [mth: MTHandle] RETURNS [BOOLEAN] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0..nModulesEntered) DO
    IF ModuleTable[i].mth = mth AND ModuleTable[i].bound THEN RETURN[TRUE];
    ENDLOOP;
  RETURN[FALSE];
  END;

END....
```