```
-- file FilePack.Mesa
-- last modified by Satterthwaite, July 5, 1978  11:29 AM

DIRECTORY
  AltoFileDefs: FROM "altofiledefs",
  BcdDefs: FROM "bcddefs",
  CopierDefs: FROM "copierdefs",
  DirectoryDefs: FROM "directorydefs",
  SegmentDefs: FROM "segmentdefs",
  StringDefs: FROM "stringdefs",
  SymbolTableDefs: FROM "symboltabledefs",
  SymDefs: FROM "symdefs",
  SymTabDefs: FROM "symtabdefs",
  SystemDefs: FROM "systemdefs",
  TableDefs: FROM "tabledefs";

FilePack: PROGRAM
    IMPORTS
        DirectoryDefs, SegmentDefs, StringDefs, SymbolTableDefs, SymTabDefs,
        SystemDefs, TableDefs
    EXPORTS CopierDefs =
  BEGIN
  OPEN SymDefs;

 -- tables defining the current symbol table

  ctxb: TableDefs.TableBase;            -- context table
  mdb: TableDefs.TableBase;             -- module directory base
  bb: TableDefs.TableBase;              -- body table

  FilePackNotify: TableDefs.TableNotifier =
    BEGIN  -- called whenever the main symbol table is repacked
    OPEN TableDefs;
    ctxb ← base[ctxtype];  mdb ← base[mdtype];
    bb ← base[bodytype];
    RETURN
    END;


  SubString: TYPE = StringDefs.SubString;
  SubStringDescriptor: TYPE = StringDefs.SubStringDescriptor;


 -- included module accounting

  VersionStamp: TYPE = BcdDefs.VersionStamp;

  FileProblem: PUBLIC SIGNAL [HTIndex] RETURNS [BOOLEAN]= CODE;
  FileVersionMix: PUBLIC SIGNAL [HTIndex] = CODE;

  AnyVersion: VersionStamp = [zapped:FALSE, net:0, host:0, time:[0,0]];

  EqStamps: PROCEDURE [v1, v2: POINTER TO VersionStamp] RETURNS [BOOLEAN] =
    BEGIN
    RETURN [v1.time = v2.time AND v1.net = v2.net AND v1.host = v2.host]
    END;

  ZappedStamp: PROCEDURE [v: POINTER TO VersionStamp] RETURNS [BOOLEAN] =
    BEGIN
    RETURN [v.zapped]
    END;


  EnterFile: PUBLIC PROCEDURE [name: STRING] RETURNS [HTIndex] =
    BEGIN
    desc: SubStringDescriptor ← [base:name, offset:0, length:name.length];
    mdi: MDIndex = FindMdEntry[@desc, AnyVersion];
    RETURN [(mdb+mdi).mdhti]
    END;

  NormalizeFileName: PROCEDURE [name: SubString] RETURNS [hti: HTIndex] =
    BEGIN
    i, j, n: CARDINAL;
    char: CHARACTER;
    dot: BOOLEAN ← FALSE;
    s: STRING ← SystemDefs.AllocateHeapString[name.length+(".bcd").length+1];
```

```
    desc: SubStringDescriptor;
    i ← name.offset;  n ← i + name.length;
    IF name.base[i] = '<
      THEN
        FOR i IN (i .. n)
          DO
          IF name.base[i-1] = '> THEN EXIT;
          REPEAT
            FINISHED => i ← name.offset;
          ENDLOOP;
    FOR j IN [i .. n)
      DO
      char ← name.base[j];
      SELECT char FROM
        IN ['A..'Z] =>  char ← LOOPHOLE[LOOPHOLE[char, CARDINAL] + 40B];
        '. =>  dot ← TRUE;
        ';, '! =>  EXIT;
        ENDCASE;
      StringDefs.AppendChar[s, char];
      ENDLOOP;
    IF ~dot THEN StringDefs.AppendString[s, ".bcd"];
    IF s[s.length-1] # '. THEN StringDefs.AppendChar[s, '.];
    desc ← SubStringDescriptor[base:s, offset:0, length: s.length];
    hti ← SymTabDefs.EnterString[@desc];
    SystemDefs.FreeHeapString[s];  RETURN
    END;

  HtiToMdi: PUBLIC PROCEDURE [hti: HTIndex] RETURNS [mdi: MDIndex] =
    BEGIN
    limit: MDIndex = LOOPHOLE[TableDefs.TableBounds[mdtype].size];
    FOR mdi ← FIRST[MDIndex], mdi + SIZE[MDRecord] UNTIL mdi = limit
      DO
      IF hti = (mdb+mdi).mdhti THEN RETURN;
      ENDLOOP;
    RETURN [MDNull]
    END;

  FindMdEntry: PUBLIC PROCEDURE [name: SubString, version: VersionStamp] RETURNS [mdi: MDIndex] =
    BEGIN
    hti: HTIndex = NormalizeFileName[name];
    limit: MDIndex = LOOPHOLE[TableDefs.TableBounds[mdtype].size];
    duplicate: BOOLEAN ← FALSE;
    FOR mdi ← FIRST[MDIndex], mdi + SIZE[MDRecord] UNTIL mdi = limit
      DO
      IF hti = (mdb+mdi).mdhti
        THEN
          BEGIN
          IF EqStamps[@(mdb+mdi).mdStamp, @version] THEN RETURN;
          OpenSymbols[mdi !FileProblem => RESUME [FALSE]];
          IF EqStamps[@(mdb+mdi).mdStamp, @version]
           OR (OpenedSymbols[mdi] AND ZappedStamp[@(mdb+mdi).mdStamp])
            THEN RETURN;
          IF (mdb+mdi).mdStamp # AnyVersion THEN duplicate ← TRUE;
          END;
      ENDLOOP;
    IF duplicate THEN SIGNAL FileVersionMix[hti];
    mdi ← TableDefs.Allocate[mdtype, SIZE[MDRecord]];
    (mdb+mdi)↑ ← MDRecord[
        mdhti: hti,
        mdctx: includedCTXNull,
        mdshared: FALSE,
        mdExported: FALSE,
        mdStamp: version,
        mdFile: nullFileIndex];
    RETURN
    END;


  GetSymbolTable: PUBLIC PROCEDURE [mdi: MDIndex] RETURNS [SymbolTableDefs.SymbolTableBase] =
    BEGIN
    index: FileIndex;
    OpenSymbols[mdi];
    index ← (mdb+mdi).mdFile;
    RETURN [IF fileTable[index].file # NIL
      THEN SymbolTableDefs.AcquireSymbolTable[fileTable[index].table]
      ELSE NIL]
```

```
    END;

  FreeSymbolTable: PUBLIC PROCEDURE [base: SymbolTableDefs.SymbolTableBase] =
    BEGIN
    SymbolTableDefs.ReleaseSymbolTable[base];
    RETURN
    END;


  -- low-level file manipulation

  SymbolTableHandle: TYPE = SymbolTableDefs.SymbolTableHandle;
  NULLTableHandle: SymbolTableHandle = SymbolTableDefs.NULLTableHandle;

  FileRecord: TYPE = RECORD[
    file: SegmentDefs.FileHandle,
    table: SymbolTableHandle];

  NullFileRecord: FileRecord = FileRecord[NIL, NULLTableHandle];

  fileTable: DESCRIPTOR FOR ARRAY OF FileRecord;
  lastFile: INTEGER;


  -- file table management

  FilePackInit: PUBLIC PROCEDURE [self: STRING, version: VersionStamp] =
    BEGIN
    OPEN SystemDefs;
    desc: SubStringDescriptor ← [base:self, offset:0, length:self.length];
    TableDefs.AddNotify[FilePackNotify];
    IF FindMdEntry[@desc, version] # FIRST[MDIndex] THEN ERROR;
    RETURN
    END;

  CreateFileTable: PUBLIC PROCEDURE [size: CARDINAL] =
    BEGIN  OPEN SystemDefs;
    i: CARDINAL;
    fileTable ← DESCRIPTOR [AllocateHeapNode[size*SIZE[FileRecord]], size];
    FOR i IN [0..size) DO fileTable[i] ← NullFileRecord ENDLOOP;
    lastFile ← -1;  RETURN
    END;

  ExpandFileTable: PROCEDURE =
    BEGIN  OPEN SystemDefs;
    table: DESCRIPTOR FOR ARRAY OF FileRecord;
    i: CARDINAL;
    size: CARDINAL = LENGTH[fileTable] + 2;
    table ← DESCRIPTOR [AllocateHeapNode[size*SIZE[FileRecord]], size];
    FOR i IN [0..LENGTH[fileTable]) DO table[i] ← fileTable[i] ENDLOOP;
    FOR i IN [LENGTH[fileTable]..size) DO table[i] ← NullFileRecord ENDLOOP;
    IF LENGTH[fileTable] > 0 THEN FreeHeapNode[BASE[fileTable]];
    fileTable ← table;
    RETURN
    END;

  FilePackReset: PUBLIC PROCEDURE =
    BEGIN
    i: INTEGER;
    SymbolTableDefs.SetSymbolCacheSize[0];
    FOR i IN [0..lastFile]
      DO
      ENABLE ANY => CONTINUE;
      SELECT TRUE FROM
        (fileTable[i].table # NULLTableHandle) =>
          SegmentDefs.DeleteFileSegment[
            SymbolTableDefs.SegmentForTable[fileTable[i].table]];
        (fileTable[i].file # NIL) =>
          SegmentDefs.ReleaseFile[fileTable[i].file];
        ENDCASE;
      ENDLOOP;
    SystemDefs.FreeHeapNode[BASE[fileTable]];
    TableDefs.DropNotify[FilePackNotify];
    RETURN
    END;
```

```
-- file setup

OwnFile: PUBLIC SIGNAL [file: SegmentDefs.FileHandle] = CODE;

LocateTables: PUBLIC PROCEDURE [ntables: CARDINAL] =
  BEGIN
  n: CARDINAL ← ntables;
  limit: MDIndex = LOOPHOLE[TableDefs.TableBounds[mdtype].size];

  CheckFile: PROCEDURE [fp: POINTER TO AltoFileDefs.FP, s: STRING] RETURNS [BOOLEAN] =
    BEGIN
    d1: SubStringDescriptor ← [base:s, offset:0, length:s.length];
    s1: SubString = @d1;
    d2: SubStringDescriptor;
    s2: SubString = @d2;
    mdi: MDIndex;
    FOR mdi ← FIRST[MDIndex], mdi+SIZE[MDRecord] UNTIL mdi = limit
      DO
      SymTabDefs.SubStringForHash[s2, (mdb+mdi).mdhti];
      IF StringDefs.EquivalentSubStrings[s1, s2]
        THEN
          BEGIN
          (mdb+mdi).mdFile ← lastFile ← lastFile+1;
          fileTable[lastFile] ← FileRecord[
              file: SegmentDefs.InsertFile[fp, SegmentDefs.Read],
              table: NULLTableHandle];
          IF mdi = OwnMdi THEN SIGNAL OwnFile[fileTable[lastFile].file];
          n ← n-1;
          EXIT
          END;
      ENDLOOP;
    RETURN [n = 0]
    END;

  DirectoryDefs.EnumerateDirectory[CheckFile];
  RETURN
  END;


MakeFileTableEntry: PUBLIC PROCEDURE
    [file: SegmentDefs.FileHandle, table: SymbolTableHandle]
    RETURNS [FileIndex] =
  BEGIN
  lastFile ← lastFile + 1;
  UNTIL lastFile < LENGTH[fileTable] DO ExpandFileTable[] ENDLOOP;
  fileTable[lastFile] ← FileRecord[file: file, table: table];
  RETURN [lastFile]
  END;

FindFile: PROCEDURE [hti: HTIndex] RETURNS [newFile: FileIndex] =
  BEGIN
  desc: SubStringDescriptor;
  s: SubString = @desc;
  name: STRING;
  SymTabDefs.SubStringForHash[s, hti];
  newFile ← lastFile + 1;
  UNTIL newFile < LENGTH[fileTable] DO ExpandFileTable[] ENDLOOP;
  fileTable[newFile] ← NullFileRecord;
  name ← SystemDefs.AllocateHeapString[s.length];
  StringDefs.AppendSubString[name, s];
    BEGIN
    OPEN SegmentDefs;
    fileTable[newFile].file ← NewFile[name, Read, OldFileOnly
      !FileProblem => NULL;
       ANY =>
        IF SIGNAL FileProblem[hti] THEN CONTINUE ELSE GO TO noEntry];
    lastFile ← newFile;
    EXITS
      noEntry => newFile ← nullFileIndex;
    END;
  SystemDefs.FreeHeapString[name];
  RETURN
  END;
```

```
OpenSymbols: PROCEDURE [mdi: MDIndex] =
  BEGIN
  index: FileIndex;
  bcdPages: CARDINAL;
  symbolSeg, headerSeg: SegmentDefs.FileSegmentHandle ← NIL;

  DeleteHeader: PROCEDURE =
    BEGIN OPEN SegmentDefs;
    IF headerSeg # NIL
      THEN
        BEGIN  Unlock[headerSeg];  DeleteFileSegment[headerSeg];
        headerSeg ← NIL;
        END;
    RETURN
    END;

  bcd: POINTER TO BcdDefs.BCD;
  mtRoot: POINTER TO BcdDefs.MTRecord;
  sgBase: CARDINAL;
  sSeg: BcdDefs.SGIndex;
  IF (mdb+mdi).mdFile = nullFileIndex THEN
    (mdb+mdi).mdFile ← FindFile[(mdb+mdi).mdhti];
  index ← (mdb+mdi).mdFile;
  IF index # nullFileIndex AND fileTable[index].table = NULLTableHandle
   AND fileTable[index].file # NIL
    THEN
      BEGIN  OPEN SegmentDefs;
      ENABLE
        BEGIN
        UNWIND => NULL;
        ANY => GO TO badFile
        END;
      bcdPages ← 1;
      headerSeg ← NewFileSegment[fileTable[index].file, 1, bcdPages, Read];
        DO
        SwapIn[headerSeg];  bcd ← FileSegmentAddress[headerSeg];
        IF bcdPages = bcd.nPages THEN EXIT;
        bcdPages ← bcd.nPages;
        Unlock[headerSeg];  SwapOut[headerSeg];
        MoveFileSegment[headerSeg, 1, bcdPages];
        ENDLOOP;
      IF bcd.versionident # BcdDefs.VersionID
        OR bcd.nConfigs # 0 OR bcd.nModules # 1 THEN GO TO badFile;
      mtRoot ← LOOPHOLE[bcd + bcd.mtOffset];
      sgBase ← LOOPHOLE[bcd + bcd.sgOffset];  sSeg ← mtRoot.sseg;
      IF sSeg = BcdDefs.SGNull
       OR (sgBase+sSeg).pages = 0 OR (sgBase+sSeg).file # BcdDefs.FTSelf
        THEN GO TO badFile;
      IF (mdb+mdi).mdStamp # AnyVersion
       AND ~EqStamps[@(mdb+mdi).mdStamp, @bcd.version]
        AND ~ZappedStamp[@bcd.version] THEN GO TO badFile;
      (mdb+mdi).mdStamp ← bcd.version;
      symbolSeg ← NewFileSegment[
                        headerSeg.file,
                        (sgBase+sSeg).base, (sgBase+sSeg).pages,
                        Read];
      symbolSeg.class ← other;
      fileTable[index].table ← SymbolTableDefs.TableForSegment[symbolSeg];
      DeleteHeader[];
      EXITS
        badFile =>
          IF SIGNAL FileProblem[(mdb+mdi).mdhti]
            THEN
              BEGIN
              IF headerSeg # NIL
                THEN  DeleteHeader[]
                ELSE  SegmentDefs.ReleaseFile[fileTable[index].file
                  !SegmentDefs.FileError => CONTINUE];        -- if shared
              fileTable[index].file ← NIL;
              END
            ELSE  DeleteHeader[];
      END;
  RETURN
  END;

OpenedSymbols: PROCEDURE [mdi: MDIndex] RETURNS [BOOLEAN] =
```

```
    BEGIN
    index: FileIndex = (mdb+mdi).mdFile;
    RETURN [index # nullFileIndex
      AND fileTable[index].table # NULLTableHandle]
    END;


  TableForModule: PUBLIC PROCEDURE [mdi: MDIndex] RETURNS [SymbolTableHandle] =
    BEGIN
    RETURN[fileTable[(mdb+mdi).mdFile].table]
    END;

  END.
```