```
-- StreamsB.Mesa  Edited by Sandman on Jul 24, 1978 9:40 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [
    CharsPerPage, MaxFilePage, PageCount, PageNumber],
  AltoFileDefs: FROM "altofiledefs" USING [CFA, eofDA, FA, fillinDA, vDA],
  BFSDefs: FROM "bfsdefs" USING [DeletePages],
  InlineDefs: FROM "inlinedefs" USING [BITAND],
  SegmentDefs: FROM "segmentdefs" USING [
    Append, CloseFile, DefaultAccess, DefaultVersion, FileHandle,
    GetFileLength, InsertFileLength, JumpToPage, LockFile, NewFile, OpenFile,
    Read, ReleaseFile, SetFileAccess, UnlockFile, UpdateFileLength, Write],
  StreamDefs: FROM "streamdefs" USING [
    AccessOptions, DiskHandle, StreamHandle, StreamIndex, StreamObject],
  StreamsA: FROM "streamsa" USING [
    Cleanup, EndOf, Fixup, Pos, PositionByte, ReadByte, ReadWord, StreamError,
    TransferPages, WriteByte, WriteWord],
  SystemDefs: FROM "systemdefs" USING [
    AllocateHeapNode, AllocateResidentPages, FreeHeapNode, FreePages];

DEFINITIONS FROM AltoDefs, AltoFileDefs, StreamDefs;

StreamsB: PROGRAM
  IMPORTS BFSDefs, SegmentDefs, SystemDefs, StreamsA
  EXPORTS StreamDefs SHARES StreamsA, StreamDefs, SegmentDefs = BEGIN

  OPEN StreamsA;

  WindowSize: PageCount = 1;

  NewByteStream: PUBLIC PROCEDURE [name: STRING, access:AccessOptions]
    RETURNS [DiskHandle] =
    BEGIN OPEN SegmentDefs;
    RETURN[Create[NewFile[name, access, DefaultVersion],bytes,access]]
    END;

  NewWordStream: PUBLIC PROCEDURE [name: STRING, access:AccessOptions]
    RETURNS [DiskHandle] =
    BEGIN OPEN SegmentDefs;
    RETURN[Create[NewFile[name, access, DefaultVersion],words,access]]
    END;

  CreateByteStream: PUBLIC PROCEDURE [file:SegmentDefs.FileHandle, access: AccessOptions]
    RETURNS [DiskHandle] = BEGIN
    RETURN[Create[file,bytes,access]]
    END;

  CreateWordStream: PUBLIC PROCEDURE [file:SegmentDefs.FileHandle, access: AccessOptions]
    RETURNS [DiskHandle] = BEGIN
    RETURN[Create[file,words,access]]
    END;

  Create: PROCEDURE [file:SegmentDefs.FileHandle, units:{bytes,words}, access: AccessOptions]
    RETURNS [stream: DiskHandle] =
    BEGIN OPEN SegmentDefs;
    fa: FA ← FA[eofDA,0,0];
    IF access = DefaultAccess THEN access ← Read;
    SetFileAccess[file,access];
    stream ← SystemDefs.AllocateHeapNode[SIZE[Disk StreamObject]];
    stream↑ ← StreamObject[
      reset: Reset, get: ReadByte, putback: PutBack,
      put: WriteByte, endof: EndOf, destroy: Destroy,
      body: Disk[
        eof: FALSE, dirty: FALSE, unit: 1, index: 0, size: 0,
        getOverflow: Fixup, savedGet: ReadError, putOverflow: Fixup, savedPut: WriteByte,
        file:, read:, write:, append:, page: 0, char: 0, buffer:, das:]];
    stream.file ← file;
    stream.read ← InlineDefs.BITAND[access,Read]#0;
    stream.write ← InlineDefs.BITAND[access,Write]#0;
    stream.append ← InlineDefs.BITAND[access,Append]#0;
    IF units=words THEN
      BEGIN OPEN stream;
      get ← ReadWord;  unit ← 2;
      put ← savedPut ← WriteWord;
      END;
    IF ~stream.read THEN stream.get ← ReadError;
```

```
    SELECT InlineDefs.BITAND[access,Write+Append] FROM
      0 => stream.put ← stream.savedPut ← WriteError;
      Write => stream.savedPut ← WriteError;
      Append => stream.put ← WriteError;
      ENDCASE;
    stream.buffer.word ← SystemDefs.AllocateResidentPages[WindowSize];
    BEGIN ENABLE UNWIND => SystemDefs.FreePages[stream.buffer.word];
      LockFile[file];  InsertFileLength[file, @fa];  OpenFile[file];
      stream.das[last] ← stream.das[next] ← fillinDA;
      stream.das[current] ← file.fp.leaderDA;
      IF access = Append
        THEN [] ← FileLength[stream]
        ELSE Jump[stream,@fa,1];
    END;
    RETURN
    END;

  OpenDiskStream: PUBLIC PROCEDURE [stream:StreamHandle] =
    BEGIN  fa: FA;
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.buffer.word=NIL THEN s.buffer.word ←
          SystemDefs.AllocateResidentPages[WindowSize];
        fa ← FA[s.das[current],s.page,Pos[@s]];
        SegmentDefs.OpenFile[s.file];
        JumpToFA[@s,@fa];
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  CleanupDiskStream: PUBLIC PROCEDURE [stream:StreamHandle] =
    BEGIN
    WITH s:stream SELECT FROM
      Disk => Cleanup[@s,TRUE];
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  Reset: PROCEDURE [stream:StreamHandle] =
    BEGIN  fa: FA;
    WITH s:stream SELECT FROM
      Disk =>
        IF s.page = 1 THEN PositionByte[@s,0,FALSE]
        ELSE BEGIN fa ← FA[eofDA,0,0]; Jump[@s,@fa,1]; END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  CloseDiskStream: PUBLIC PROCEDURE [stream:StreamHandle] =
    BEGIN
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        Cleanup[@s,TRUE];
        SystemDefs.FreePages[s.buffer.word];
        IF s.file.segcount=0 THEN
          SegmentDefs.CloseFile[s.file];
        s.buffer.word ← NIL;
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  TruncateDiskStream: PUBLIC PROCEDURE [stream:StreamHandle] =
    BEGIN
    WITH s:stream SELECT FROM
      Disk => Kill[@s,s.write];
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  Destroy: PROCEDURE [stream:StreamHandle] =
    BEGIN
    WITH s:stream SELECT FROM
```

```
    Disk => Kill[@s,~s.read];
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN
  END;

Kill: PROCEDURE [stream:DiskHandle, trunc:BOOLEAN] =
  BEGIN OPEN stream;
  da: vDA;  pn: PageNumber;
  IF buffer.word # NIL THEN
    BEGIN da ← eofDA;
    IF trunc AND GetIndex[stream] # StreamIndex[0,0] THEN
      BEGIN -- truncate the file
      -- this is not a separate procedure because it
      -- leaves the stream buffer in an awful state.
      pn ← page;  da ← das[next];  das[next] ← eofDA;
      IF char # Pos[stream] THEN
        BEGIN char ← Pos[stream]; dirty ← TRUE END;
      END;
    IF dirty THEN Cleanup[stream,TRUE];
    IF da # eofDA THEN
      BFSDefs.DeletePages[buffer.word,@file.fp,da,pn+1];
    SystemDefs.FreePages[buffer.word];
    END;
  SegmentDefs.UnlockFile[file];
  IF file.segcount=0 THEN
    SegmentDefs.ReleaseFile[file];
  SystemDefs.FreeHeapNode[stream];
  RETURN
  END;

Jump: PROCEDURE [s:DiskHandle, fa:POINTER TO FA, pn:PageNumber] =
  BEGIN OPEN s;
  cfa: CFA ← CFA[file.fp,fa↑];
  IF dirty THEN Cleanup[s,TRUE];  PositionByte[s,0,FALSE];
  [das[last],das[next]] ← SegmentDefs.JumpToPage[@cfa,pn,buffer.word];
  [das[current],page,char] ← cfa.fa;
  IF das[next]=eofDA THEN SegmentDefs.UpdateFileLength[file,@cfa.fa];
  PositionByte[s,IF page#pn THEN char ELSE MIN[char,fa.byte],FALSE];
  RETURN
  END;

ReadError: PROCEDURE [s:StreamHandle] RETURNS [UNSPECIFIED] =
  BEGIN
  SIGNAL StreamError[s,StreamAccess];
  RETURN[0]
  END;

PutBack: PROCEDURE [stream:StreamHandle, item:UNSPECIFIED] =
  BEGIN
  SIGNAL StreamError[stream,StreamOperation];
  RETURN
  END;

WriteError: PROCEDURE [stream:StreamHandle, item:UNSPECIFIED] =
  BEGIN
  SIGNAL StreamError[stream,StreamAccess];
  RETURN
  END;

bite: INTEGER = 60;  -- don't use too much heap

PositionPage: PROCEDURE [s:DiskHandle, p:PageNumber] =
  BEGIN
  d, dp: INTEGER;
  np: PageCount;
  Cleanup[s,TRUE]; PositionByte[s,0,FALSE];
  -- should we reset first?
  SELECT INTEGER[s.page-p] FROM
    <= 0  => NULL;
    = 1, < INTEGER[s.page/10]  => NULL;
    ENDCASE => Reset[s];
  WHILE (d ← p-s.page)#0 DO
    dp ← IF d < 0 THEN -1 ELSE MIN[d,bite];
    np ← TransferPages[s,NIL,dp,in,FALSE];
    IF dp > 0 AND np # dp THEN EXIT;
    REPEAT FINISHED => RETURN;
```

```
    ENDLOOP;
  IF ~s.append THEN ERROR StreamError[s,StreamAccess];
  -- extend the file (the first transfer flushes the buffer)
  IF s.char > 0 THEN [] ← TransferPages[s,NIL,1,out,FALSE];
  WHILE (d ← p-s.page)#0 DO
    [] ← TransferPages[s,NIL,MIN[d,bite],out,FALSE];
    ENDLOOP;
  RETURN
  END;

-- StreamIndex Manipulation

GetIndex: PUBLIC PROCEDURE [stream:StreamHandle]
  RETURNS [StreamIndex] = BEGIN
  WITH s:stream SELECT FROM
    Disk =>
      BEGIN
      -- make sure we're not at end of page
      Cleanup[@s,FALSE]; -- don't flush
      RETURN[StreamIndex[s.page-1,Pos[@s]]];
      END;
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN[StreamIndex[0,0]]
  END;

SetIndex: PUBLIC PROCEDURE [stream:StreamHandle, index:StreamIndex] =
  BEGIN
  WITH s:stream SELECT FROM
    Disk =>
      BEGIN
      index ← NormalizeIndex[index];
      IF index.page+1 # s.page
        THEN PositionPage[@s,index.page+1];
      PositionByte[@s,index.byte,FALSE];
      END;
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN
  END;

NormalizeIndex: PUBLIC PROCEDURE [index:StreamIndex]
  RETURNS [StreamIndex] =
  BEGIN
  IF index.byte >= CharsPerPage THEN
    BEGIN
    index.page ← index.page + index.byte/CharsPerPage;
    index.byte ← index.byte MOD CharsPerPage;
    END;
  RETURN[index]
  END;

ModifyIndex: PUBLIC PROCEDURE [index: StreamIndex, change: INTEGER]
  RETURNS [StreamIndex] =
  BEGIN  OPEN AltoDefs;
  delta: CARDINAL = ABS[change];
  pages: CARDINAL ← delta/CharsPerPage;
  bytes: CARDINAL ← delta MOD CharsPerPage;
  index ← NormalizeIndex[index];
  SELECT change FROM
    > 0 =>
      BEGIN
      bytes ← index.byte + bytes;
      IF bytes >= CharsPerPage THEN
        BEGIN bytes ← bytes - CharsPerPage; pages ← pages + 1 END;
      pages ← index.page + pages;
      END;
    = 0 => RETURN [index];
    < 0 =>
      BEGIN
      IF bytes <= index.byte THEN bytes ← index.byte - bytes
      ELSE BEGIN bytes ← index.byte+CharsPerPage-bytes; pages ← pages+1 END;
      IF pages <= index.page THEN pages ← index.page - pages
      ELSE RETURN[[0, 0]];
      END;
    ENDCASE;
  RETURN [[pages, bytes]];
  END;
```

```
-- procedures to test for equality of stream indexes
EqualIndex: PUBLIC PROCEDURE[i1, i2: StreamIndex] RETURNS [BOOLEAN] =
  BEGIN
  i1 ← NormalizeIndex[i1]; i2 ← NormalizeIndex[i2];
  RETURN[i1 = i2];
  END;

GrEqualIndex: PUBLIC PROCEDURE[i1, i2: StreamIndex] RETURNS [BOOLEAN] =
  BEGIN
  RETURN[EqualIndex[i1,i2] OR GrIndex[i1,i2]];
  END;

GrIndex: PUBLIC PROCEDURE[i1, i2: StreamIndex] RETURNS [BOOLEAN] =
  BEGIN
  i1 ← NormalizeIndex[i1]; i2 ← NormalizeIndex[i2];
  RETURN[i1.page > i2.page OR (i1.page = i2.page AND
    i1.byte > i2.byte)];
  END;

GetFA: PUBLIC PROCEDURE [stream:StreamHandle, fa:POINTER TO FA] =
  BEGIN
  WITH s:stream SELECT FROM
    Disk =>
      BEGIN
      -- make sure not at end of a page
      Cleanup[@s,FALSE];  -- don't flush
      fa↑ ← FA[s.das[current],s.page,Pos[@s]];
      END;
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN
  END;

FileLength: PUBLIC PROCEDURE [stream:StreamHandle]
  RETURNS [StreamIndex] =
  BEGIN fa: FA;
  WITH s:stream SELECT FROM
    Disk =>
      BEGIN
      SegmentDefs.GetFileLength[s.file, @fa];
      Jump[@s,@fa,MaxFilePage];
      SegmentDefs.UpdateFileLength[s.file, @fa];
      RETURN[GetIndex[@s]];
      END;
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN[StreamIndex[0,0]]
  END;

JumpToFA: PUBLIC PROCEDURE [stream:StreamHandle, fa:POINTER TO FA] =
  BEGIN
  WITH s:stream SELECT FROM
    Disk =>
      BEGIN Jump[@s,fa,fa.page];
      IF fa.page # s.page OR fa.byte # Pos[@s] THEN
        SIGNAL StreamError[@s,StreamEnd];
      END;
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN
  END;

END..
```