```
-- Streams.Mesa  Edited by Sandman on August 23, 1977  10:19 PM

DIRECTORY
  AltoDefs: FROM "altodefs",
  AltoFileDefs: FROM "altofiledefs",
  BFSDefs: FROM "bfsdefs",
  DiskDefs: FROM "diskdefs",
  InlineDefs: FROM "inlinedefs",
  MiscDefs: FROM "miscdefs",
  SegmentDefs: FROM "segmentdefs",
  StreamDefs: FROM "streamdefs",
  SystemDefs: FROM "systemdefs";

DEFINITIONS FROM AltoDefs, AltoFileDefs, StreamDefs;

Streams: PROGRAM
  IMPORTS BFSDefs, MiscDefs, SegmentDefs, SystemDefs
  EXPORTS StreamDefs SHARES StreamDefs, SegmentDefs = BEGIN

  WindowSize: PageCount = 1;

  StreamError: PUBLIC SIGNAL [stream:StreamHandle, error:StreamErrorCode] = CODE;

  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;

  Model: Disk StreamObject = StreamObject [
    Reset, ReadByte, PutBack,
    WriteByte, EndOf, Destroy,
    Disk [
      FALSE, FALSE, 1, 0, NIL, 0, 0,
      Fixup, ReadError, Fixup, WriteByte,
      NIL, FALSE, FALSE, FALSE, 0, 0, , ]];

  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↑ ← Model;  stream.file ← file;
    stream.read ← InlineDefs.BITAND[access,Read]#0;
    stream.write ← InlineDefs.BITAND[access,Write]#0;
    stream.append ← InlineDefs.BITAND[access,Append]#0;
    stream.buffer ← SystemDefs.AllocatePages[WindowSize];
    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;
```

```
    LockFile[file];  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];
    RETURN
    END;

  OpenDiskStream: PUBLIC PROCEDURE [stream:StreamHandle] =
    BEGIN  fa: FA;
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.buffer=NIL THEN s.buffer ←
          SystemDefs.AllocatePages[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]
        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];
        IF s.file.segcount=0 THEN
          SegmentDefs.CloseFile[s.file];
        s.buffer ← 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 # 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,@file.fp,da,pn+1];
      SystemDefs.FreePages[buffer];
      END;
    SegmentDefs.UnlockFile[file];
    IF file.segcount=0 THEN
      SegmentDefs.ReleaseFile[file];
    SystemDefs.FreeHeapNode[stream];
    RETURN
    END;


  -- block mode transfers

  direction: TYPE = {in,out};

  -- the fast stream overflow handler; should only be called
  -- from the fast stream get, put, and endof routines.  It
  -- always supplies a new count (which may be zero, in which
  -- case get and/or put is replaced with an error routine).

  Fixup: PROCEDURE [stream:StreamHandle] =
    BEGIN  pos: CARDINAL;
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        Cleanup[@s,FALSE]; -- don't flush
        IF (pos ← Pos[@s]) >= s.char THEN
          BEGIN
          SetEnd[@s,TRUE]; -- ran into eof
          Setup[@s,s.buffer,pos,CharsPerPage];
          END;
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;


  -- Cleanup makes the disk look like the stream, unless the
  -- current page is not full and you didn't ask for a flush.

  Cleanup: PROCEDURE [s:DiskHandle, flush:BOOLEAN] =
    BEGIN  pos: CARDINAL;
    IF (pos ← Pos[s]) > s.char THEN PositionByte[s,pos];
    IF pos=CharsPerPage THEN
      -- write current page, read (maybe create) next one
      IF s.dirty THEN [] ← TransferPages[s,NIL,1,out,FALSE]
      -- donothing with current page, read next one
      ELSE [] ← TransferPages[s,NIL,1,in,TRUE]
    ELSE IF s.dirty AND flush THEN
      BEGIN
      -- write current page w/ new numChars
      [] ← TransferPages[s,NIL,0,out,TRUE];
      PositionByte[s,pos];
      END;
    RETURN
    END;

  ReadBlock: PUBLIC PROCEDURE [
    stream:StreamHandle, address:POINTER, words:CARDINAL]
    RETURNS [CARDINAL] =
    BEGIN
    done: CARDINAL ← 0;
    WITH s:stream SELECT FROM
      Disk => IF s.read THEN
```

```
      done ← TransferBlock[@s,address,words,in];
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN[done]
  END;

WriteBlock: PUBLIC PROCEDURE [
  stream:StreamHandle, address:POINTER, words:CARDINAL]
  RETURNS [CARDINAL] =
  BEGIN
  done: CARDINAL ← 0;
  WITH s:stream SELECT FROM
    Disk =>
      IF (~s.write AND ~s.append)
      OR (~s.write AND s.append AND ~EndOf[@s])
      OR (s.write AND ~s.append AND EndOf[@s])
        THEN NULL
        ELSE done ← TransferBlock[@s,address,words,out];
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN[done]
  END;

TransferBlock: PROCEDURE [
  s:DiskHandle, a:POINTER, n:CARDINAL, d:direction]
  RETURNS [CARDINAL] =
  BEGIN OPEN InlineDefs;
  np: PageCount;
  done: CARDINAL ← 0;
  left, pos, words: CARDINAL;
  IF BITAND[Pos[s],CharsPerWord-1]#0
    THEN ERROR StreamError[s,StreamPosition];
  WHILE done # n DO
    left ← n-done;
    pos ← BITSHIFT[Pos[s],-LogCharsPerWord];
    words ←
     (IF d=out AND s.append THEN PageSize
      ELSE BITSHIFT[s.char+CharsPerWord-1,-LogCharsPerWord]) - pos;
    words ← IF left > words THEN words ELSE left;
    IF words # 0 THEN
      BEGIN
      PositionByte[s,BITSHIFT[pos+words,LogCharsPerWord]];
      SELECT d FROM
        in => COPY[from:s.buffer+pos,to:a,nwords:words];
        out =>
          BEGIN
          COPY[from:a,to:s.buffer+pos,nwords:words];
          s.dirty ← TRUE;
          END;
        ENDCASE;
      END;
    IF s.char # CharsPerPage
    AND s.endof[s] AND (d=in OR ~s.append)
      THEN RETURN [done+words];
    np ← BITSHIFT[left-words,-LogPageSize];
    IF left-words # 0 THEN
      words ← words+BITSHIFT[
        TransferPages[s,a+words,np,d,FALSE],LogPageSize];
    a ← a+words;  done ← done+words;
    ENDLOOP;
  RETURN[done]
  END;

-- Transfers np pages (or fewer if the file runs out while reading/updating),
-- starting at address a and the current page of the file (the one in
-- the buffer).  It leaves the next page in the buffer, with the stream
-- set up at the first character.  Note that if writing, the next page
-- is read, not written; if the file is extended, the buffer is cleared.
-- Returns the number of pages transferred, not counting the next one
-- that was read into the buffer.  It's only legal to call TransferPages
-- when the buffer is full or empty; use TransferBlock otherwise.

-- Some special uses:
--   a=0    All transfers are into buffer (useful for positioning).
--   np=0   The current page is transfered (useful for Cleanup).
--   np=-1  Backup one page (useful for positioning).

-- The last argument is for very special uses (described below), do
```

```
-- not supply it unless you know what you are doing!  If special is
-- true, the following funny things happen, depending on direction:
--   direction=in: action is made DoNothing (np should be one)
--     Used by Cleanup to skip the current page and read next one.
--   direction=out: lastAction is replaced by WriteD, and last-
--     Bytes is replaced by the numChars from the stream (np should
--     be zero).  Used by Cleanup to flush with new buffer length.

TransferPages: PROCEDURE [
  s:DiskHandle, a:POINTER, np:PageCount, d:direction, special:BOOLEAN]
  RETURNS [PageCount] =
  BEGIN OPEN DiskDefs;
  backup: BOOLEAN;
  arg:  DiskRequest;
  i, fp, lp: PageNumber;
  dobuffer: BOOLEAN ← FALSE;
  DAs: DESCRIPTOR FOR ARRAY OF vDA;
  CAs: DESCRIPTOR FOR ARRAY OF POINTER;
  f: POINTER TO FP ← @s.file.fp;
  -- flush the buffer if the transfer won't
  IF d=in THEN
    IF s.dirty THEN Cleanup[s,TRUE]
    ELSE NULL; -- should mark written
  -- include the buffer if the transfer doesn't
  IF a # NIL AND Pos[s] = CharsPerPage THEN
    BEGIN
    -- the stream is at [page n, byte 0], but the
    -- buffer is at [page n-1, byte CharsPerPage];
    -- transfer the buffer, too, even if not dirty.
    dobuffer ← TRUE;  np ← np+1;
    a ← a-PageSize; -- fixed below
    END;
  fp ← s.page;  PositionByte[s,0];
  IF backup ← (np=-1) THEN
    BEGIN  fp ← fp-1;  np ← 0  END;
  lp ← fp+np;
  CAs ← DESCRIPTOR [
    SystemDefs.AllocateHeapNode[np+3]-(fp-1),lp+2];
  DAs ← DESCRIPTOR [
    SystemDefs.AllocateHeapNode[np+3]-(fp-1),lp+2];
  FOR i IN [fp-1..lp+1] DO
    CAs[i] ←
      IF a=NIL THEN s.buffer
      ELSE a+(i-fp)*PageSize;
    DAs[i] ← fillinDA;
    ENDLOOP;
  CAs[lp] ← s.buffer; IF dobuffer THEN CAs[fp] ← s.buffer;
  InlineDefs.COPY [
    from:@s.das,to:@DAs[IF backup THEN fp ELSE fp-1],
    nwords:IF backup THEN LENGTH[s.das]-1 ELSE LENGTH[s.das]];
  arg ← DiskRequest [
    @CAs[0],@DAs[0],fp,lp,f,FALSE,
    WriteD,ReadD,FALSE,update[BFSDefs.GetNextDA]];
  IF d=in OR (d=out AND ~special AND ~s.append) THEN
    BEGIN
    IF d=in THEN arg.action ← ReadD;
    IF special THEN arg.action ← DoNothing;
    [i,s.char] ← BFSDefs.ActOnPages[LOOPHOLE[@arg]];
    IF i#lp AND s.char>0 AND CAs[i]#s.buffer THEN
      InlineDefs.COPY[from:CAs[i],to:s.buffer,nwords:PageSize];
    END
  ELSE
    BEGIN
    arg ← DiskRequest [,,,,,,,
      IF special THEN WriteD ELSE ReadD,,
      extend[IF special THEN s.char ELSE 0]];
    [i,s.char] ← BFSDefs.WritePages[LOOPHOLE[@arg]];
    END;
  s.page ← i;
  IF s.char=0 THEN MiscDefs.Zero[s.buffer,PageSize];
  InlineDefs.COPY [
    from:@DAs[i-1],to:@s.das,nwords:LENGTH[s.das]];
  IF s.das[next]=eofDA THEN SetFileLength[s];
  SystemDefs.FreeHeapNode[BASE[CAs]+fp-1];
  SystemDefs.FreeHeapNode[BASE[DAs]+fp-1];
  Setup[s,s.buffer,0,s.char];
```

```
    SetEnd[s,s.count=0];  s.dirty ← FALSE;
    RETURN[i-fp-(IF dobuffer THEN 1 ELSE 0)]
    END;

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

  PositionPage: PROCEDURE [s:DiskHandle, p:PageNumber] =
    BEGIN  d, dp, np: PageNumber;
    Cleanup[s,TRUE]; PositionByte[s,0];
    -- should we reset first?
    SELECT s.page-p FROM
      <= 0  => NULL;
      = 1, < 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;

  PositionByte: PROCEDURE [s:DiskHandle, b:CARDINAL] =
    BEGIN OPEN s;
    pos: CARDINAL;
    IF das[next]=eofDA THEN
      BEGIN
      IF (pos ← Pos[s]) > char
      AND append AND dirty
        THEN char ← pos;
      IF b > char THEN
        IF ~append THEN b ← char
        ELSE BEGIN char ← b;  dirty ← TRUE  END;
      END;
    Setup[s,buffer,b,char];
    SetEnd[s,count=0 AND char#CharsPerPage];
    RETURN
    END;

  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;

  NormalizeIndex: PUBLIC PROCEDURE [index:StreamIndex]
    RETURNS [StreamIndex] =
    BEGIN OPEN InlineDefs;
    delta: PageNumber ← BITSHIFT[index.byte,-LogCharsPerPage];
    index.byte ← BITAND[index.byte,CharsPerPage-1];
    index.page ← index.page+delta;
    RETURN[index]
    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];
      END;
    ENDCASE => SIGNAL StreamError[@s,StreamType];
  RETURN
  END;

ModifyIndex: PUBLIC PROCEDURE [index: StreamIndex, change: INTEGER]
 RETURNS [StreamIndex] =
  BEGIN  OPEN AltoDefs;
  pages: INTEGER ← InlineDefs.BITSHIFT[ABS[change], -LogCharsPerPage];
  bytes: INTEGER ← InlineDefs.BITAND[ABS[change], CharsPerPage-1];
  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
      bytes ← index.byte - bytes;
      IF bytes < 0 THEN
        BEGIN bytes ← bytes + CharsPerPage; pages ← pages + 1 END;
      pages ← index.page - pages;
      END;
    ENDCASE;
  IF pages < 0 THEN RETURN [[0, 0]];
  RETURN [[LOOPHOLE[pages], LOOPHOLE[bytes]]];
  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
      fa ← s.file.eof;
      fa.byte ← CharsPerPage;
      Jump[@s,@fa,MaxFilePage];
      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;

SetFileLength: PROCEDURE [s:DiskHandle] =
  BEGIN OPEN s;
  fa: FA ← FA[das[current],page,char];
  SegmentDefs.UpdateFileLength[file,@fa];
```

```
    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];
    [das[last],das[next]] ← SegmentDefs.JumpToPage[@cfa,pn,buffer];
    [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]];
    RETURN
    END;

  --  procedures to test for equality of stream indexes
  EqualIndex: PUBLIC PROCEDURE[i1, i2: StreamIndex] RETURNS [BOOLEAN] =
    BEGIN
    RETURN[(i1.page = i2.page) AND (i1.byte = i2.byte)];
    END;


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

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

  --   F A S T   S T R E A M S

  -- the counts and positions should be optimized for
  -- the instruction set (as in the bcpl implementation).

  Setup: PROCEDURE [s:DiskHandle, base:POINTER, pos,end:CARDINAL] =
    BEGIN OPEN InlineDefs, s;
    mask: WORD ← -unit;
    shift: INTEGER ← unit-1;
    -- both pos and end are rounded
    pos ← BITAND[pos+shift,mask];
    end ← BITAND[end+shift,mask];
    byte ← BITAND[pos,CharsPerWord-1];
    word ← base+BITSHIFT[pos,-LogCharsPerWord];
    count ← BITSHIFT[end-pos,-shift];
    size ← end;
    RETURN
    END;

  Pos: PROCEDURE [s:DiskHandle] RETURNS [CARDINAL] =
    BEGIN OPEN s;
    RETURN [size-InlineDefs.BITSHIFT[count,unit-1]]
    END;

  SetEnd: PROCEDURE [s:DiskHandle, b:BOOLEAN] =
    BEGIN
    g: PROCEDURE [StreamHandle] RETURNS [UNSPECIFIED];
    p: PROCEDURE [StreamHandle,UNSPECIFIED];
    IF s.eof # b THEN
      BEGIN  s.eof ← b;
      g ← s.get;  s.get ← s.savedGet;  s.savedGet ← g;
      p ← s.put;  s.put ← s.savedPut;  s.savedPut ← p;
      END;
    RETURN
    END;

  bytepointer: TYPE = POINTER TO bytepair;
  bytepair: TYPE = MACHINE DEPENDENT RECORD [left,right:BYTE];

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

```
    END;

  ReadByte: PROCEDURE [stream:StreamHandle] RETURNS [item:UNSPECIFIED] =
    BEGIN  item ← 0;
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.count=0 THEN
          BEGIN s.getOverflow[@s];
          RETURN[s.get[@s]]; END;
        IF s.byte=0 THEN
          BEGIN
          item ← LOOPHOLE[s.word,bytepointer].left;
          s.byte ← 1;
          END
        ELSE
          BEGIN
          item ← LOOPHOLE[s.word,bytepointer].right;
          s.word ← s.word+1;  s.byte ← 0;
          END;
        s.count ← s.count-1;
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  ReadWord: PROCEDURE [stream:StreamHandle] RETURNS [item:UNSPECIFIED] =
    BEGIN  item ← 0;
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.count=0 THEN
          BEGIN s.getOverflow[@s];
          RETURN[s.get[@s]]; END;
        item ← s.word↑;
        s.word ← s.word+1;
        s.count ← s.count-1;
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    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;

  WriteByte: PROCEDURE [stream:StreamHandle, item:UNSPECIFIED] =
    BEGIN
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.count=0 THEN
          BEGIN s.putOverflow[@s];
          s.put[@s,item];
          RETURN; END;
        IF s.byte=0 THEN
          BEGIN
          LOOPHOLE[s.word,bytepointer].left ← item;
          s.byte ← 1;
          END
        ELSE
          BEGIN
          LOOPHOLE[s.word,bytepointer].right ← item;
          s.word ← s.word+1;  s.byte ← 0;
          END;
        s.count ← s.count-1;
        s.dirty ← TRUE;
        END;
```

```
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  WriteWord: PROCEDURE [stream:StreamHandle, item:UNSPECIFIED] =
    BEGIN
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.count=0 THEN
          BEGIN s.putOverflow[@s];
          s.put[@s,item];
          RETURN; END;
        s.word↑ ← item;
        s.word ← s.word+1;
        s.count ← s.count-1;
        s.dirty ← TRUE;
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN
    END;

  EndOf: PROCEDURE [stream:StreamHandle] RETURNS [BOOLEAN] =
    BEGIN
    WITH s:stream SELECT FROM
      Disk =>
        BEGIN
        IF s.eof THEN RETURN[TRUE];
        IF s.count#0 THEN RETURN[FALSE];
        s.getOverflow[@s]; RETURN[s.endof[@s]];
        END;
      ENDCASE => SIGNAL StreamError[@s,StreamType];
    RETURN[FALSE]
    END;

  END.
```