```
-- Swapper.Mesa  Edited by Sandman on Jul 12, 1978 4:05 PM

DIRECTORY
  AllocDefs: FROM "allocdefs" USING [
    AllocHandle, AllocInfo, AllocObject, DefaultDataSegmentInfo,
    DefaultFileSegmentInfo, DefaultFrameSegmentInfo, DefaultTableSegmentInfo,
    PageState, SwappingProcedure, SwapStrategy],
  AltoDefs: FROM "altodefs" USING [
    BYTE, MaxVMPage, PageCount, PageNumber, PageSize],
  AltoFileDefs: FROM "altofiledefs" USING [eofDA, vDC],
  BootDefs: FROM "bootdefs" USING [
    BusyPage, FreePage, PageMap, PositionSeg, SystemTable, SystemTableHandle,
    Table, TableHandle],
  ControlDefs: FROM "controldefs" USING [
    CSegPrefix, GFT, GlobalFrameHandle, NullGlobalFrame],
  DiskDefs: FROM "diskdefs" USING [DiskRequest, SwapPages],
  FrameDefs: FROM "framedefs" USING [FlushLargeFrames, ValidateGlobalFrame],
  InlineDefs: FROM "inlinedefs" USING [BITAND],
  NucleusDefs: FROM "nucleusdefs",
  ProcessDefs: FROM "processdefs" USING [DisableInterrupts, EnableInterrupts],
  SDDefs: FROM "sddefs" USING [SD, sGFTLength],
  SegmentDefs: FROM "segmentdefs" USING [
    DataSegmentHandle, DefaultBase, FileSegmentHandle, FrobHandle,
    FrobLink, FrobNull, InvalidSegmentSize, MaxLocks, MaxRefs, Object,
    ObjectHandle, ObjectType, OpenFile, PageNumber, RemoteSegCommand,
    SegmentHandle];

DEFINITIONS FROM SegmentDefs;

Swapper: PROGRAM [ffvmp, lfvmp: AltoDefs.PageNumber]
  IMPORTS BootDefs, DiskDefs, FrameDefs, SegmentDefs
  EXPORTS AllocDefs, BootDefs, FrameDefs, NucleusDefs, SegmentDefs
  SHARES SegmentDefs = BEGIN

  AllocInfo: TYPE = AllocDefs.AllocInfo;
  AllocHandle: TYPE = AllocDefs.AllocHandle;
  PageState: TYPE = AllocDefs.PageState;
  SwappingProcedure: TYPE = AllocDefs.SwappingProcedure;
  SwapStrategy: TYPE = AllocDefs.SwapStrategy;
  PageCount: TYPE = AltoDefs.PageCount;
  PageNumber: TYPE = AltoDefs.PageNumber;
  TableHandle: TYPE = BootDefs.TableHandle;
  SegmentHandle: TYPE = SegmentDefs.SegmentHandle;
  DataSegmentHandle: TYPE = SegmentDefs.DataSegmentHandle;
  FileSegmentHandle: TYPE = SegmentDefs.FileSegmentHandle;
  MaxVMPage: PageNumber = AltoDefs.MaxVMPage;
  PageSize: CARDINAL = AltoDefs.PageSize;

  PAGEDISP: TYPE = MACHINE DEPENDENT RECORD [
    page: [0..MaxVMPage],
    disp: [0..PageSize)];

  PageFromAddress: PUBLIC PROCEDURE [a:POINTER] RETURNS [PageNumber] =
    BEGIN
    RETURN[LOOPHOLE[a,PAGEDISP].page]
    END;

  AddressFromPage: PUBLIC PROCEDURE [p:PageNumber] RETURNS [POINTER] =
    BEGIN
    RETURN[LOOPHOLE[PAGEDISP[p,0]]]
    END;

  PagePointer: PUBLIC PROCEDURE [a:POINTER] RETURNS [POINTER] =
    BEGIN
    LOOPHOLE[a,PAGEDISP].disp ← 0;
    RETURN[a]
    END;


  -- Data Segments

  DefaultBase: PageNumber = SegmentDefs.DefaultBase;

  NewDataSegment: PUBLIC PROCEDURE [base:PageNumber, pages:PageCount]
    RETURNS [seg:DataSegmentHandle] =
    BEGIN
```

```
    RETURN[MakeDataSegment[base, pages, AllocDefs.DefaultDataSegmentInfo]]
    END;

  NewFrameSegment: PUBLIC PROCEDURE [pages:PageCount]
    RETURNS [seg:DataSegmentHandle] =
    BEGIN
    RETURN[MakeDataSegment[DefaultBase, pages, AllocDefs.DefaultFrameSegmentInfo]]
    END;

  MakeDataSegment: PUBLIC PROCEDURE [
    base: PageNumber, pages: PageCount, info: AllocInfo]
    RETURNS [seg: DataSegmentHandle] =
    BEGIN
    IF pages ~IN (0..MaxVMPage+1] THEN ERROR InvalidSegmentSize[pages];
    seg ← AllocateDataSegment[];
    seg↑ ← [busy:, body: segment[data[VMpage:, pages:, unused:]]];
    base ← alloc.alloc[base, pages, seg, info
      ! UNWIND => LiberateDataSegment[seg]];
    seg↑ ← [busy: FALSE,
      body: segment[data[VMpage: base, pages: pages, unused: 0]]];
    alloc.update[base, pages, inuse, seg];
    RETURN
    END;

  BootDataSegment: PUBLIC PROCEDURE [base:PageNumber, pages:PageCount]
    RETURNS [seg:DataSegmentHandle] =
    BEGIN OPEN AllocDefs;
    i: PageNumber;
    FOR i IN [base..base+pages) DO
      IF alloc.status[i].status # busy THEN ERROR;
      ENDLOOP;
    seg ← AllocateDataSegment[];
    seg↑ ← [busy: FALSE,
      body: segment[data[VMpage: base, pages: pages, unused: 0]]];
    alloc.update[base, pages, inuse, seg];
    RETURN
    END;

  DeleteDataSegment: PUBLIC PROCEDURE [seg:DataSegmentHandle] =
    BEGIN
    base: PageNumber; pages: PageCount;
    ValidateDataSegment[seg];
    base ← seg.VMpage; pages ← seg.pages;
    alloc.update[base, pages, busy, NIL];
    LiberateDataSegment[seg];
    alloc.update[base, pages, free, NIL];
    RETURN
    END;

  DataSegmentAddress: PUBLIC PROCEDURE [seg:DataSegmentHandle] RETURNS [POINTER] =
    BEGIN
    RETURN[LOOPHOLE[PAGEDISP[seg.VMpage,0]]]
    END;
```

```
-- Swapping Segments

SwapError: PUBLIC SIGNAL [seg:FileSegmentHandle] = CODE;

MakeSwappedIn: PUBLIC PROCEDURE [
  seg: FileSegmentHandle, base: PageNumber, info: AllocInfo] =
  BEGIN
  vmpage: PageNumber;
  alreadyIn: BOOLEAN ← FALSE;
  IF seg.lock = MaxLocks THEN ERROR SwapError[seg];
  ValidateObject[seg];
  ProcessDefs.DisableInterrupts[];
  IF ~seg.swappedin THEN
    BEGIN
    ProcessDefs.EnableInterrupts[];
    IF seg.file.swapcount = MaxRefs THEN SIGNAL SwapError[seg];
    IF ~seg.file.open THEN OpenFile[seg.file];
    vmpage ← alloc.alloc[base, seg.pages, seg, info];
    ProcessDefs.DisableInterrupts[];
    IF ~seg.swappedin THEN
      BEGIN
      ProcessDefs.EnableInterrupts[];
      BEGIN ENABLE UNWIND => alloc.update[vmpage, seg.pages, free, NIL];
        seg.VMpage ← vmpage;
        WITH s: seg SELECT FROM
          disk =>
            IF (s.hint.page # s.base OR s.hint.da = AltoFileDefs.eofDA)
            AND BootDefs.PositionSeg[@s,TRUE] AND s.pages = 1
              THEN NULL ELSE MapVM[@s, ReadD];
          remote => s.proc[@s, remoteRead];
          ENDCASE;
        END;
      ProcessDefs.DisableInterrupts[];
      seg.file.swapcount ← seg.file.swapcount+1;
      alloc.update[vmpage, seg.pages, inuse, seg];
      seg.swappedin ← TRUE;
      END
    ELSE alreadyIn ← TRUE;
    END;
  seg.lock ← seg.lock+1;
  IF alreadyIn THEN alloc.update[vmpage, seg.pages, free, NIL];
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

SwapIn: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN
  MakeSwappedIn[seg, DefaultBase, AllocDefs.DefaultFileSegmentInfo];
  RETURN
  END;

Unlock: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN OPEN seg;
  IF lock = 0 THEN ERROR SwapError[seg];
  ValidateObject[seg];
  ProcessDefs.DisableInterrupts[];
  lock ← lock-1;
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

SwapUp: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN OPEN seg;
  ValidateObject[seg];
  IF swappedin AND write THEN
    BEGIN
    WITH s: seg SELECT FROM
      disk =>
        BEGIN
        IF s.hint.page # base OR s.hint.da = AltoFileDefs.eofDA THEN
          [] ← BootDefs.PositionSeg[@s,FALSE];
        MapVM[@s, WriteD];
        END;
      remote => s.proc[@s, remoteWrite];
      ENDCASE;
    END;
```

```
    RETURN
    END;

  SwapOut: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
    BEGIN OPEN seg;
    temp: PageNumber;
    ValidateObject[seg];
    SwapUp[seg];
    ProcessDefs.DisableInterrupts[];
    IF ~swappedin THEN BEGIN ProcessDefs.EnableInterrupts[]; RETURN END;
    IF lock # 0 THEN
      BEGIN ProcessDefs.EnableInterrupts[]; ERROR SwapError[seg] END;
    busy ← TRUE;
    alloc.update[temp ← VMpage, pages, busy, NIL];
    swappedin ← FALSE;
    busy ← FALSE;
    VMpage ← 0;
    file.swapcount ← file.swapcount-1;
    ProcessDefs.EnableInterrupts[];
    -- IF swapcount = 0 THEN CloseFile[]; ???
    alloc.update[temp, pages, free, NIL];
    RETURN
    END;

  remoteRead: RemoteSegCommand = 0;
  remoteWrite: RemoteSegCommand = 1;

  SegmentFault: PUBLIC SIGNAL [seg:FileSegmentHandle, pages:PageCount] = CODE;

  MapVM: PUBLIC PROCEDURE [seg:FileSegmentHandle, dc: AltoFileDefs.vDC] =
    BEGIN OPEN seg;
    page: PageNumber;  byte: CARDINAL;  temp: PageCount;
    arg: swap DiskDefs.DiskRequest;
    WITH s: seg SELECT FROM
      disk =>
        BEGIN
        arg ← DiskDefs.DiskRequest[AddressFromPage[s.VMpage], @s.hint.da,
          s.base, s.base+s.pages-1, @s.file.fp, FALSE, dc, dc, FALSE,
          swap[NIL]];
        IF s.hint.page # s.base THEN ERROR SwapError[@s];
        [page,byte] ← DiskDefs.SwapPages[@arg];
        temp ← page-base+(IF byte=0 THEN 0 ELSE 1);
        IF temp=0 THEN ERROR SegmentFault[@s,0];
        IF temp # pages THEN
          BEGIN SIGNAL SegmentFault[@s,temp];
          alloc.update[s.VMpage+temp, s.pages-temp, free, NIL];
          s.pages ← temp;
          END;
        END;
      remote => ERROR SwapError[@s];
      ENDCASE;
    RETURN
    END;
```

```
  -- Code Swapping and Swap Strategies

  trySwapInProgress: BOOLEAN ← FALSE;

  TrySwapping: SwappingProcedure =
    BEGIN
    did: BOOLEAN;
    sp, next: POINTER TO SwapStrategy;
    ProcessDefs.DisableInterrupts[];
    IF trySwapInProgress THEN
      BEGIN
      ProcessDefs.EnableInterrupts[];
      RETURN[TryCodeSwapping[needed, info, seg]];
      END;
    trySwapInProgress ← TRUE;
    ProcessDefs.EnableInterrupts[];
    did ← TRUE;
    FOR sp ← StrategyList, next UNTIL sp = NIL DO
      next ← sp.link;
      IF sp.proc[needed, info, seg] THEN EXIT;
      REPEAT FINISHED => did ← FALSE;
      ENDLOOP;
    trySwapInProgress ← FALSE;
    RETURN[did]
    END;

  CantSwap: PUBLIC SwappingProcedure =
    BEGIN
    RETURN[FALSE]
    END;

  pageRover: AltoDefs.PageNumber ← 0;

  TryCodeSwapping: PUBLIC SwappingProcedure =
    BEGIN OPEN ControlDefs;
    foundHole: BOOLEAN ← FALSE;
    pass: {first, second, quit} ← first;
    okay: BOOLEAN;
    base, page: AltoDefs.PageNumber;
    segment: SegmentHandle;
    status: AllocDefs.PageState;
    p: POINTER TO CSegPrefix;
    n, inc: PageCount;
    page ← n ← 0;
    ProcessDefs.DisableInterrupts[];
    segment ← alloc.status[pageRover].seg;
    IF segment # NIL THEN
      WITH s: segment SELECT FROM
        data => pageRover ← s.VMpage;
        file => pageRover ← s.VMpage;
        ENDCASE;
    base ← pageRover;
    DO -- until we've looked at them all twice
      [segment, status] ← alloc.status[pageRover];
      okay ← FALSE;
      SELECT status FROM
        inuse =>
          WITH s: segment SELECT FROM
            data => inc ← s.pages;
            file =>
              BEGIN
              IF s.lock = 0 AND ~s.write THEN
                BEGIN
                IF s.class = code THEN
                  BEGIN
                  p ← AddressFromPage[s.VMpage];
                  IF p.swapinfo > 1 THEN p ← p + p.swapinfo;
                  IF p.swapinfo = 0 THEN okay ← TRUE ELSE p.swapinfo ← 0;
                  END
                ELSE okay ← TRUE;
                END;
              inc ← s.pages;
              END;
            ENDCASE;
        ENDCASE =>
          BEGIN
```

```
        IF status = free THEN okay ← TRUE;
        inc ← 1;
        END;
      IF ~okay THEN
        BEGIN page ← n ← 0; IF pass = quit THEN EXIT; END
      ELSE
        BEGIN
        IF page = 0 THEN page ← pageRover;
        IF (n ← n+inc) >= needed THEN
          BEGIN foundHole ← TRUE; EXIT END;
        END;
      IF (pageRover ← pageRover+inc) > MaxVMPage THEN
        IF pass = quit THEN EXIT ELSE pageRover ← page ← n ← 0;
      IF pageRover = base THEN pass ← IF pass = first THEN second ELSE quit;
      ENDLOOP;
    base ← page + n;
    WHILE page < base DO
      segment ← alloc.status[page].seg;
      IF segment # NIL THEN
        WITH s: segment SELECT FROM
          data =>
            page ← s.VMpage + s.pages;
          file =>
            BEGIN
            page ← s.VMpage;
            IF s.lock = 0 THEN
              BEGIN
              IF s.class = code THEN UpdateCodebases[@s];
              IF ~s.write THEN SwapOutUnlocked[@s];
              alloc.update[page, s.pages, free, NIL];
              END;
            page ← page + s.pages;
            END;
          ENDCASE
        ELSE page ← page + 1;
      ENDLOOP;
    ProcessDefs.EnableInterrupts[];
    RETURN[foundHole]
    END;

  SwapOutCode: PUBLIC PROCEDURE [f:ControlDefs.GlobalFrameHandle] =
    BEGIN OPEN SegmentDefs, ControlDefs;
    cseg: SegmentDefs.FileSegmentHandle = f.codesegment;
    FrameDefs.ValidateGlobalFrame[f];
    ProcessDefs.DisableInterrupts[];
    IF cseg.swappedin THEN
      BEGIN
      SwapIn[cseg]; -- lock it so it won't go away
      UpdateCodebases[cseg];
      Unlock[cseg]; SwapOut[cseg];
      END;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  LastResort: SwapStrategy ← SwapStrategy[NIL,TryCodeSwapping];
  StrategyList: POINTER TO SwapStrategy ← @LastResort;

  AddSwapStrategy: PUBLIC PROCEDURE [strategy:POINTER TO SwapStrategy] =
    BEGIN
    sp: POINTER TO SwapStrategy;
    ProcessDefs.DisableInterrupts[];
    FOR sp ← StrategyList, sp.link
    UNTIL sp = NIL DO
      IF sp = strategy THEN RETURN;
      ENDLOOP;
    strategy.link ← StrategyList;
    StrategyList ← strategy;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  RemoveSwapStrategy: PUBLIC PROCEDURE [strategy:POINTER TO SwapStrategy] =
    BEGIN
    sp: POINTER TO SwapStrategy;
    prev: POINTER TO SwapStrategy ← NIL;
```

```
    ProcessDefs.DisableInterrupts[];
    FOR sp ← StrategyList, sp.link UNTIL sp = NIL DO
      IF sp = strategy THEN
        BEGIN
        IF prev = NIL
          THEN StrategyList ← sp.link
          ELSE prev.link ← sp.link;
        EXIT END;
      prev ← sp;
      ENDLOOP;
    ProcessDefs.EnableInterrupts[];
    strategy.link ← NIL;
    RETURN
    END;
```

```
  -- Memory Allocator

  PageMap: BootDefs.PageMap;
  FreePage: SegmentHandle = BootDefs.FreePage;
  BusyPage: SegmentHandle = BootDefs.BusyPage;

-- *** The framesize of PageAvailable is depended on below ***

  PageAvailable: PROCEDURE [page: PageNumber, info: AllocInfo]
    RETURNS [available: BOOLEAN] =
    BEGIN
    seg: SegmentHandle;
    ProcessDefs.DisableInterrupts[];
    seg ← PageMap[page];
    available ← FALSE;
    IF seg = FreePage THEN available ← TRUE
    ELSE IF seg # BusyPage THEN
      WITH s: seg SELECT FROM
        file =>
          IF (info.effort = hard OR info.swapunlocked) AND
            s.lock = 0 AND ~s.write AND ~s.busy THEN available ← TRUE;
        ENDCASE;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

-- *** PageStatus' framesize must be <= PageAvailable's ***

  PageStatus: PROCEDURE [page: PageNumber]
    RETURNS [seg: SegmentHandle, status: PageState] =
    BEGIN
    ProcessDefs.DisableInterrupts[];
    seg ← PageMap[page];
    SELECT seg FROM
      BusyPage => BEGIN status ← busy; seg ← NIL END;
      FreePage => BEGIN status ← free; seg ← NIL END;
      ENDCASE =>
        IF seg.busy THEN BEGIN status ← free; seg ← NIL; END
        ELSE status ← inuse;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  InsufficientVM: PUBLIC SIGNAL [needed: PageCount] = CODE;
  VMnotFree: PUBLIC SIGNAL [base: PageNumber, pages: PageCount] = CODE;

  AllocVM: PROCEDURE [base: PageNumber, pages: PageCount,
    seg: SegmentHandle, info: AllocInfo]
    RETURNS [PageNumber] =
    BEGIN
    tempseg: SegmentHandle;
    n: CARDINAL;
    direction: INTEGER;
    vm: PageNumber;
    FrameDefs.FlushLargeFrames[];
    IF base # DefaultBase THEN
      DO -- repeat if requested VM not free
      ProcessDefs.DisableInterrupts[];
      FOR vm IN [base.. base+pages) DO
        IF ~alloc.avail[base, info] THEN EXIT;
        REPEAT
          FINISHED => GOTO found;
        ENDLOOP;
      ProcessDefs.EnableInterrupts[];
      SIGNAL VMnotFree[base, pages];
      REPEAT
        found => NULL
      ENDLOOP
    ELSE
      DO -- repeat if insufficient VM
      ProcessDefs.DisableInterrupts[];
      n ← 0;  -- count of contiguous free pages
      IF info.direction = bottomup THEN
        BEGIN  direction ← 1;  base ← ffvmp;  END
      ELSE
        BEGIN  direction ← -1;  base ← lfvmp;  END;
```

```
      WHILE base IN [ffvmp..lfvmp] DO
        IF ~alloc.avail[base, info] THEN n ← 0
        ELSE IF (n ← n+1) = pages THEN
          BEGIN IF direction>0 THEN base ← base-n+1; GOTO foundHole END;
        base ← base+direction
        ENDLOOP;
      ProcessDefs.EnableInterrupts[];
      IF info.class = frame OR info.class = table
        OR ~TrySwapping[pages, info, seg] THEN SIGNAL InsufficientVM[pages];
      REPEAT
        foundHole => NULL;
      ENDLOOP;
    FOR vm IN [base..base+pages) DO
      tempseg ← alloc.status[vm].seg;
      IF tempseg # NIL THEN
        WITH s: tempseg SELECT FROM
          file =>
            BEGIN
            alloc.update[s.VMpage, s.pages, free, NIL];
            IF s.class = code THEN UpdateCodebases[@s];
            SwapOutUnlocked[@s];
            END;
          ENDCASE;
      ENDLOOP;
    alloc.update[base, pages, busy, seg];
    ProcessDefs.EnableInterrupts[];
    RETURN[base]
    END;

-- *** UpdateVM's framesize must be <= PageAvailable's ***

  UpdateVM: PROCEDURE [base: PageNumber, pages: PageCount, status: PageState,
    seg: SegmentHandle] =
    BEGIN
    IF status = free THEN
      BEGIN
      ProcessDefs.DisableInterrupts[];
      ffvmp ← MIN[ffvmp,base];
      lfvmp ← MAX[lfvmp,base+pages-1];
      ProcessDefs.EnableInterrupts[];
      END;
    seg ← SELECT status FROM
      free => FreePage,
      busy => BusyPage,
      ENDCASE => IF seg = NIL THEN BusyPage ELSE seg;
    ProcessDefs.DisableInterrupts[];
    FOR base IN [base..base+pages) DO PageMap[base] ← seg; ENDLOOP;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

-- *** SwapOutUnlocked's framesize must be <= PageAvailable's ***

  SwapOutUnlocked: PROCEDURE [seg: FileSegmentHandle] =
    BEGIN
    ProcessDefs.DisableInterrupts[];
    seg.swappedin ← FALSE;
    seg.VMpage ← 0;
    seg.file.swapcount ← seg.file.swapcount-1;
    ProcessDefs.EnableInterrupts[];
    END;

-- *** UpdateCodebases's framesize must be <= PageAvailable's ***

  UpdateCodebases: PROCEDURE [seg: FileSegmentHandle] =
    BEGIN OPEN ControlDefs;
    lastUser, f: ControlDefs.GlobalFrameHandle;
    nUsers, i: CARDINAL;
    epbase: CARDINAL;
    ProcessDefs.DisableInterrupts[];
    nUsers ← 0;
    FOR i IN [1..SDDefs.SD[SDDefs.sGFTLength]) DO
      [frame: f, epbase: epbase] ← GFT[i];
      IF f # NullGlobalFrame AND epbase = 0 AND f.codesegment = seg THEN
        BEGIN
        IF ~f.code.swappedout THEN
```

```
        BEGIN
        f.code.codebase ← f.code.codebase - AltoDefs.PageSize*seg.VMpage;
        f.code.swappedout ← TRUE;
        END;
      IF ~f.shared THEN EXIT;
      nUsers ← nUsers+1;
      lastUser ← f;
      END;
    REPEAT
      FINISHED => IF nUsers = 1 THEN lastUser.shared ← FALSE;
    ENDLOOP;
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

SetAllocationObject: PUBLIC PROCEDURE [new: AllocHandle]
  RETURNS [old: AllocHandle] =
  BEGIN
  ProcessDefs.DisableInterrupts[];
  old ← alloc;
  alloc ← new;
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

GetAllocationObject: PUBLIC PROCEDURE RETURNS [old: AllocHandle] =
  BEGIN RETURN[alloc] END;
```

```
-- Primative Object Allocation

ObjectSeal: AltoDefs.BYTE = 21B;

InvalidObject: PUBLIC SIGNAL [object: POINTER] = CODE;

AllocateObject: PUBLIC PROCEDURE [size: CARDINAL] RETURNS [ObjectHandle] =
  BEGIN OPEN BootDefs;
  frob: FrobLink;
  frobject: FrobHandle;
  table: TableHandle;
  base, length: CARDINAL;
  ProcessDefs.DisableInterrupts[];
  FOR table ← systemTable.table, table.link UNTIL table = NIL DO
    IF table.free.fwdp # FIRST[FrobLink] THEN
      BEGIN
      base ← LOOPHOLE[table, CARDINAL];
      FOR frob ← table.free.fwdp, frobject.fwdp UNTIL frob = FIRST[FrobLink] DO
        frobject ← base+frob;
        length ← frobject.size;
        UNTIL frob+length > FrobNull DO
          WITH n: LOOPHOLE[frobject+length, ObjectHandle] SELECT FROM
            free =>  -- coalesce nodes
              BEGIN
              (base+n.fwdp).backp ← n.backp;
              (base+n.backp).fwdp ← n.fwdp;
              length ← length + n.size;
              END;
            ENDCASE => EXIT;
          ENDLOOP;
        SELECT length FROM
          = size =>
            BEGIN
            (base+frobject.fwdp).backp ← frobject.backp;
            (base+frobject.backp).fwdp ← frobject.fwdp;
            table.free.size ← table.free.size + size;
            ProcessDefs.EnableInterrupts[];
            RETURN[frobject];
            END;
          > size =>
            BEGIN
            frobject.size ← length - size;
            table.free.size ← table.free.size + size;
            ProcessDefs.EnableInterrupts[];
            RETURN[frobject+length-size];
            END;
          ENDCASE => frobject.size ← length;
        ENDLOOP;
      END;
    ENDLOOP;
  table ← AllocateTable[! UNWIND => ProcessDefs.EnableInterrupts[]];
  frob ← table.free.fwdp;
  frobject ← LOOPHOLE[table, CARDINAL]+frob;
  frobject.size ← frobject.size - size;
  table.free.size ← table.free.size + size;
  ProcessDefs.EnableInterrupts[];
  RETURN[frobject+frobject.size];
  END;

LiberateObject: PUBLIC PROCEDURE [object: ObjectHandle] =
  BEGIN
  table: TableHandle ← SegmentDefs.PagePointer[object];
  size, base: CARDINAL;
  frob: FrobLink ← LOOPHOLE[InlineDefs.BITAND[LOOPHOLF[object], 377B]];
  base ← LOOPHOLE[table];
  ValidateObject[object];
  size ← WITH o: object SELECT FROM
    segment => SELECT o.type FROM
      data => SIZE[data segment Object],
      ENDCASE => SIZE[file segment Object],
    file => SIZE[file Object],
    ENDCASE => SIZE[length Object];
  ProcessDefs.DisableInterrupts[];
  IF (table.free.size ← table.free.size - size) = 0 THEN
    LiberateTable[table ! UNWIND => ProcessDefs.EnableInterrupts[]]
  ELSE
```

```
      BEGIN
      object↑ ← Object[FALSE, free[seal: ObjectSeal, size: size,
        fwdp: table.free.fwdp, backp: FIRST[FrobLink]]];
      (base+table.free.fwdp).backp ← frob;
      table.free.fwdp ← frob;
      END;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  AllocateTable: PROCEDURE RETURNS [newTable: TableHandle] =
    BEGIN OPEN BootDefs, SegmentDefs;
    frob: FrobLink = LOOPHOLE[SIZE[Table]];
    base: CARDINAL;
    page: PageNumber;
    page ←
      alloc.alloc[DefaultBase, 1, NIL, AllocDefs.DefaultTableSegmentInfo];
    newTable ← AddressFromPage[page];
    newTable↑ ←
      Table[[FALSE, free[ObjectSeal,0, frob, frob]],systemTable.table,NIL];
    base ← LOOPHOLE[newTable];
    (base+frob)↑ ← [FALSE, free[ObjectSeal, AltoDefs.PageSize-SIZE[Table],
      FIRST[FrobLink], FIRST[FrobLink]]];
    systemTable.table ← newTable;
    systemTable.table.seg ← BootDataSegment[page, 1];
    RETURN
    END;

  LiberateTable: PROCEDURE [table:TableHandle] =
    BEGIN
    current: TableHandle;
    prev: TableHandle ← NIL;
    FOR current ← systemTable.table, current.link UNTIL current = NIL DO
      IF current = table THEN
        BEGIN
        IF prev = NIL
          THEN systemTable.table ← current.link
          ELSE prev.link ← current.link;
        -- oops: this had better not recur!
        DeleteDataSegment[current.seg];
        RETURN
        END;
      prev ← current;
      ENDLOOP;
    ERROR InvalidObject[table];
    END;

  ValidateObject: PUBLIC PROCEDURE [object:ObjectHandle] =
    BEGIN
    t: TableHandle;
    table: TableHandle = PagePointer[object];
    BEGIN
    IF object = NIL OR InlineDefs.BITAND[LOOPHOLE[object, CARDINAL], 1] = 1 OR
      object.tag = free THEN GOTO invalid;
    IF table.free.seal # ObjectSeal THEN GOTO invalid;
    FOR t ← systemTable.table, t.link UNTIL t = NIL DO
      IF t = table THEN EXIT;
      REPEAT
        FINISHED => GOTO invalid;
      ENDLOOP;
    EXITS
      invalid => ERROR InvalidObject[object];
    END;
    RETURN
    END;

  EnumerateObjects: PUBLIC PROCEDURE [type: ObjectType,
    proc:PROCEDURE [ObjectHandle] RETURNS [BOOLEAN]]
    RETURNS [object: ObjectHandle] =
    BEGIN
    i, j: CARDINAL;
    table: TableHandle;
    FOR table ← systemTable.table, table.link UNTIL table = NIL DO
      j ← i ← SIZE[BootDefs.Table];
      FOR object ← @table.free + i, object + i UNTIL j >= AltoDefs.PageSize DO
        i ← WITH obj:object SELECT FROM
```

```
        segment => SELECT obj.type FROM
          data => SIZE[data segment Object],
          ENDCASE => SIZE[file segment Object],
        file => SIZE[file Object],
        free => obj.size,
        ENDCASE => SIZE[length Object];
      j ← j + i;
      IF object.tag = type AND proc[object] THEN RETURN[object];
      ENDLOOP;
    ENDLOOP;
  RETURN[NIL]
  END;
```

```
-- Managing Data Segment Objects

AllocateDataSegment: PROCEDURE RETURNS [DataSegmentHandle] =
  BEGIN
  RETURN[LOOPHOLE[AllocateObject[SIZE[data segment Object]]]];
  END;

ValidateDataSegment: PROCEDURE [DataSegmentHandle] = LOOPHOLE[ValidateObject];
LiberateDataSegment: PROCEDURE [DataSegmentHandle] = LOOPHOLE[LiberateObject];

GetSystemTable: PUBLIC PROCEDURE RETURNS [BootDefs.SystemTableHandle] =
  BEGIN
  RETURN[@systemTable];
  END;


-- Main body

systemTable: BootDefs.SystemTable ← BootDefs.SystemTable[@PageMap, NIL];

systemObject: AllocDefs.AllocObject ← [
  PageAvailable, PageStatus, UpdateVM, AllocVM];

alloc: AllocHandle ← @systemObject;

Init: PROCEDURE =
  BEGIN
  i: [0..MaxVMPage];
  FOR i IN [0..MaxVMPage] DO
    PageMap[i] ← IF i IN [ffvmp..lfvmp] THEN FreePage ELSE BusyPage;
    ENDLOOP;
  END;

Init[];

END.....
```