```
-- NonResident.mesa; edited by Sandman, Jul 25, 1978 8:26 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [BYTE],
  ControlDefs: FROM "controldefs" USING [
    Alloc, ControlLink, EPRange, FrameCodeBase, FrameHandle, Free,
    GetReturnFrame, GetReturnLink, GFT, GFTIndex, GFTItem, GlobalFrameHandle,
    InstWord, Lreg, MainBodyIndex, MaxAllocSlot, NullEpBase, NullFrame,
    NullGlobalFrame, Port, PortHandle, SD, StateVector],
  CoreSwapDefs: FROM "coreswapdefs",
  FrameDefs: FROM "framedefs" USING [FrameSize, SwapInCode],
  ImageDefs: FROM "imagedefs",
  InlineDefs: FROM "inlinedefs" USING [
    BITAND, BITNOT, BITSHIFT, BITXOR, COPY, DIVMOD, LDIVMOD, LongCARDINAL,
    LongDiv, LongDivMod, LongMult],
  LoadStateDefs: FROM "loadstatedefs" USING [
    ConfigIndex, ConfigNull, EnterGfi, InputLoadState, MapRealToConfig,
    ReleaseLoadState],
  MiscDefs: FROM "miscdefs",
  Mopcodes: FROM "mopcodes" USING [zDADD, zDCOMP, zDSUB, zINC, zPORTI],
  NucleusDefs: FROM "nucleusdefs",
  ProcessDefs: FROM "processdefs" USING [DisableInterrupts, EnableInterrupts],
  Resident: FROM "resident" USING [
    AllocTrap, Break, CodeTrap, CSPort, level, MemorySwap, Restart, Start,
    UnboundProcedureTrap, WBPort, WorryBreaker],
  SDDefs: FROM "sddefs" USING [
    sAllocTrap, sAlternateBreak, sBLTE, sBLTEC, sBreak, sBYTBLTE, sBYTBLTEC,
    sControlFault, sCopy, sCoreSwap, SD, sDivSS, sError, sGFTLength,
    sIOResetBits, sLongDiv, sLongDivMod, sLongMod, sLongMul, sRestart,
    sStackError, sStart, sStringInit, sSwapTrap, sUnbound, sUnNew],
  SegmentDefs: FROM "segmentdefs" USING [
    AddressFromPage, DeleteFileSegment, FileSegmentHandle, SwapError, Unlock],
  TrapDefs: FROM "trapdefs" USING [UnboundProcedure];

DEFINITIONS FROM ControlDefs;

NonResident: PROGRAM
  IMPORTS FrameDefs, LoadStateDefs, ResidentPtr: Resident, SegmentDefs, TrapDefs
  EXPORTS FrameDefs, ImageDefs, InlineDefs, MiscDefs, NucleusDefs, TrapDefs, CoreSwapDefs
  SHARES ControlDefs, ImageDefs, Resident = BEGIN


  -- Global Frame Table management

  gftrover: CARDINAL ← 0;  -- okay to start at 0 because incremented before used

  NoGlobalFrameSlots: PUBLIC SIGNAL [CARDINAL] = CODE;

  EnumerateGlobalFrames: PUBLIC PROCEDURE [
    proc: PROCEDURE [GlobalFrameHandle] RETURNS [BOOLEAN]]
    RETURNS [GlobalFrameHandle] =
    BEGIN
    i: GFTIndex;
    frame: GlobalFrameHandle;
    gft: POINTER TO ARRAY [0..0) OF GFTItem ← GFT;
    FOR i IN [0..SD[SDDefs.sGFTLength]) DO
      frame ← gft[i].frame;
      IF frame # NullGlobalFrame AND gft[i].epbase = 0
      AND proc[frame] THEN RETURN[frame];
      ENDLOOP;
    RETURN[NullGlobalFrame]
    END;

  EnterGlobalFrame: PUBLIC PROCEDURE [frame: GlobalFrameHandle, nslots: CARDINAL]
    RETURNS [entryindex: GFTIndex] =
    BEGIN
    gft: POINTER TO ARRAY [0..0) OF GFTItem ← GFT;
    i, imax, n, epoffset: CARDINAL;
    i ← gftrover;  imax ← SD[SDDefs.sGFTLength] - nslots;  n ← 0;
    DO
      IF (i ← IF i>=imax THEN 1 ELSE i+1) = gftrover
        THEN SIGNAL NoGlobalFrameSlots[nslots];
      IF gft[i].frame # NullGlobalFrame THEN n ← 0
      ELSE IF gft[i].epbase = NullEpBase THEN n ← 0
      ELSE IF (n ← n+1) = nslots THEN EXIT;
      ENDLOOP;
```

```
    entryindex ← (gftrover←i)-nslots+1;  epoffset ← 0;
    FOR i IN [entryindex..gftrover] DO
      gft[i] ← GFTItem[frame, epoffset];
      epoffset ← epoffset + EPRange;
      ENDLOOP;
    RETURN
    END;

  RemoveGlobalFrame: PUBLIC PROCEDURE [frame: GlobalFrameHandle] =
    BEGIN
    gft: POINTER TO ARRAY [0..0) OF GFTItem ← GFT;
    sd: POINTER TO ARRAY [0..0) OF CARDINAL ← SD;
    i: CARDINAL;
    FOR i ← frame.gfi, i+1
    WHILE i<sd[SDDefs.sGFTLength] AND gft[i].frame=frame DO
      gft[i] ← IF frame.copied THEN
        GFTItem[NullGlobalFrame,0] ELSE GFTItem[NullGlobalFrame,NullEpBase];
      ENDLOOP;
    RETURN
    END;

  -- Traps

  StackError: PUBLIC ERROR [FrameHandle] = CODE;

  StackErrorTrap: PROCEDURE =
    BEGIN
    state: StateVector;
    foo: BOOLEAN;
    state ← STATE;
    foo ← TRUE;
    IF foo THEN ERROR StackError[GetReturnFrame[]];
    END;

  NullPort: PortHandle = LOOPHOLE[0];

  PortFault: PUBLIC ERROR = CODE;
  LinkageFault: PUBLIC ERROR = CODE;
  ControlFault: PUBLIC SIGNAL [source: FrameHandle] RETURNS [ControlLink] = CODE;
  PORTI: PROCEDURE = MACHINE CODE BEGIN Mopcodes.zPORTI END;


  ControlFaultTrap: PROCEDURE =
    BEGIN
    errorStart, savedState: StateVector;
    p, q: PortHandle;
    sourceFrame, self: FrameHandle;
    savedState ← STATE;
    self ← REGISTER[Lreg];
    IF PortCall[self.returnlink] THEN
      BEGIN
      p ← self.returnlink.port;
      q ← p.dest.port;
      sourceFrame ← p.frame;
      IF q = NullPort THEN
        errorStart.stk[0] ← LinkageFault
      ELSE
        BEGIN
        q↑ ← Port[links[NullFrame,[indirect[port[p]]]]];
        errorStart.stk[0] ← PortFault;
        END;
      errorStart.stk[1] ← 0;
      errorStart.instbyte ← 0;
      errorStart.stkptr ← 2;
      errorStart.source ← sourceFrame.returnlink;
      errorStart.dest ← SD[SDDefs.sError];
      IF savedState.stkptr = 0 THEN
        RETURN WITH errorStart  --  RESPONDING port
      ELSE
        BEGIN
        p.frame ← self;
        TRANSFER WITH errorStart;
        PORTI;
        p.frame ← sourceFrame;
        savedState.source ← p;
        savedState.dest ← p.dest;
```

```
      RETURN WITH savedState;
      END;
    END
  ELSE
    BEGIN
    savedState.source ← self.returnlink;
    savedState.dest ← SIGNAL ControlFault[savedState.source];
    RETURN WITH savedState
    END;
  END;

PortCall: PROCEDURE [source: ControlLink] RETURNS [BOOLEAN] =
  BEGIN
  portcall: BOOLEAN ← FALSE;
  WHILE source.tag = indirect DO
    source ← source.link↑;
    ENDLOOP;
  IF source.tag = frame AND
    FrameDefs.ReturnByte[source.frame,0] = Mopcodes.zPORTI THEN
      portcall ← TRUE;
  RETURN[portcall]
  END;

ReturnByte: PUBLIC PROCEDURE [frame: FrameHandle, byteoffset: INTEGER]
  RETURNS [byte: AltoDefs.BYTE] =
  BEGIN
  OPEN SegmentDefs;
  g: GlobalFrameHandle = frame.accesslink;
  iw: POINTER TO InstWord;
  bytePC: CARDINAL = byteoffset + (IF frame.pc<0
    THEN 2*(-frame.pc)+1 ELSE 2*frame.pc);
  FrameDefs.SwapInCode[g];
  iw ← g.code.codebase + bytePC/2;
  byte ← IF bytePC MOD 2 # 0 THEN iw.oddbyte ELSE iw.evenbyte;
  Unlock[g.codesegment];
  RETURN
  END;

-- Frame manipulation

InvalidGlobalFrame: PUBLIC SIGNAL [frame: GlobalFrameHandle] = CODE;

ValidateGlobalFrame: PUBLIC PROCEDURE [g: GlobalFrameHandle] =
  BEGIN
  IF ~ValidGlobalFrame[g] THEN SIGNAL InvalidGlobalFrame[g];
  END;

ValidGlobalFrame: PROCEDURE [g: GlobalFrameHandle]
  RETURNS[BOOLEAN] =
  BEGIN
  RETURN[LOOPHOLE[g, ControlLink].tag = frame AND GFT[g.gfi].frame = g]
  END;

GlobalFrame: PUBLIC PROCEDURE [link: UNSPECIFIED]
  RETURNS [GlobalFrameHandle] =
  BEGIN OPEN l: LOOPHOLE[link, ControlLink];
  DO SELECT l.tag FROM
    frame =>
      BEGIN
      IF link = 0 THEN RETURN[NullGlobalFrame];
      IF ValidGlobalFrame[link] THEN RETURN[link];
      IF ValidGlobalFrame[l.frame.accesslink] THEN
        RETURN[l.frame.accesslink];
      RETURN[NullGlobalFrame]
      END;
    procedure => RETURN[GFT[l.gfi].frame];
    indirect => link ← l.link↑;
    unbound => link ← SIGNAL TrapDefs.UnboundProcedure[link];
    ENDCASE ENDLOOP
  END;

Copy: PROCEDURE [old: GlobalFrameHandle] RETURNS [new: GlobalFrameHandle] =
  BEGIN
  linkspace: CARDINAL ← 0;
  FrameDefs.SwapInCode[old];
  [new, linkspace] ← AllocGlobalFrame[old];
```

```
    new ← new + linkspace;
    new↑ ← [gfi:, unused: 0, alloced: TRUE, shared: TRUE, copied: TRUE,
      started: FALSE, trapxfers: FALSE, codelinks: old.codelinks,
      code:, codesegment: old.codesegment, global:];
    new.gfi ← FrameDefs.EnterGlobalFrame[new, old.code.prefix.ngfi];
    new.code.offset ←
      old.code.codebase - SegmentDefs.AddressFromPage[old.codesegment.VMpage];
    new.code.swappedout ← TRUE;
    new.global[0] ← NullGlobalFrame;
    old.shared ← TRUE;
    IF linkspace # 0 THEN InlineDefs.COPY[
      from: old-linkspace, to: new-linkspace, nwords: linkspace];
    SegmentDefs.Unlock[old.codesegment];
    RETURN
    END;

  MakeFsi: PUBLIC PROCEDURE [words: CARDINAL] RETURNS [fsi: CARDINAL] =
    BEGIN
    FOR fsi IN [0..MaxAllocSlot) DO
      IF FrameDefs.FrameSize[fsi] >= words THEN RETURN;
      ENDLOOP;
    RETURN[words]
    END;

  AllocGlobalFrame: PROCEDURE [old: GlobalFrameHandle]
    RETURNS [frame: GlobalFrameHandle, linkspace: CARDINAL] =
    BEGIN OPEN cp: old.code.prefix;
    pbody: POINTER;
    size: CARDINAL;
    pbody ← old.code.codebase+CARDINAL[cp.entry[MainBodyIndex].initialpc];
    size ← IF cp.entry[MainBodyIndex].framesize = MaxAllocSlot THEN (pbody-2)↑
      ELSE (pbody-1)↑;
    linkspace ← IF ~old.codelinks THEN
      cp.nlinks + InlineDefs.BITAND[-LOOPHOLE[cp.nlinks, INTEGER], 3B]
      ELSE 0;
    frame ← Alloc[MakeFsi[FrameDefs.FrameSize[size]+linkspace]];
    RETURN
    END;

  UnNew: PROCEDURE [frame: GlobalFrameHandle] =
    BEGIN
    cseg: SegmentDefs.FileSegmentHandle ← frame.codesegment;
    sharer: GlobalFrameHandle ← NullGlobalFrame;
    original: GlobalFrameHandle ← NullGlobalFrame;
    copy: GlobalFrameHandle ← NullGlobalFrame;
    fcb: FrameCodeBase;
    nothers: CARDINAL ← 0;
    nlinks: CARDINAL;
    RemoveAllTraces: PROCEDURE [f: GlobalFrameHandle] RETURNS [BOOLEAN] =
      BEGIN
      IF f#frame THEN
        BEGIN
        IF f.global[0] = frame AND ~f.started THEN f.global[0] ← NullFrame;
        IF f.codesegment = cseg THEN
          BEGIN
          nothers ← nothers + 1;  sharer ← f;
          ProcessDefs.DisableInterrupts[];
          IF (IF f.code.swappedout THEN f.code = fcb
              ELSE f.code.codebase = frame.code.codebase) THEN
            IF f.copied THEN copy ← f ELSE original ← f;
          ProcessDefs.EnableInterrupts[];
          END;
        END;
      RETURN[FALSE];
      END;
    ValidateGlobalFrame[frame];
    FrameDefs.SwapInCode[frame];
    nlinks ← frame.code.prefix.nlinks;
    fcb.offset ← frame.code.codebase - SegmentDefs.AddressFromPage[cseg.VMpage];
    fcb.swappedout ← TRUE;
    [] ← FrameDefs.EnumerateGlobalFrames[RemoveAllTraces];
    SegmentDefs.Unlock[cseg];
    IF original = NullGlobalFrame AND ~frame.copied AND copy # NullGlobalFrame THEN
      BEGIN OPEN LoadStateDefs;
      config: ConfigIndex;
      cgfi: GFTIndex;
```

```
    copy.copied ← FALSE;
    [] ← InputLoadState[];
    [cgfi: cgfi, config: config] ← MapRealToConfig[frame.gfi];
    EnterGfi[cgfi: 0, rgfi: frame.gfi, config: ConfigNull];
    EnterGfi[cgfi: cgfi, rgfi: copy.gfi, config: config];
    ReleaseLoadState[];
    END;
  IF frame.shared THEN
    BEGIN
    IF nothers = 1 THEN sharer.shared ← FALSE
    END
  ELSE
    BEGIN OPEN SegmentDefs;
    DeleteFileSegment[cseg ! SwapError => CONTINUE];
    END;
  FrameDefs.RemoveGlobalFrame[frame];
  IF frame.alloced THEN
    BEGIN
    Align: PROCEDURE [POINTER, WORD] RETURNS [POINTER] =
      LOOPHOLE[InlineDefs.BITAND];
    IF frame.codelinks THEN Free[frame]
    ELSE Free[Align[frame - nlinks, 177774B]]
    END;
  END;


-- unimplemented instructions

BlockEqual: PROCEDURE [p1: POINTER, n: CARDINAL, p2: POINTER]
  RETURNS [BOOLEAN] =
  BEGIN
  i: CARDINAL;
  FOR i IN [0 .. n) DO
    IF (p1+i)↑ # (p2+i)↑ THEN RETURN[FALSE]; ENDLOOP;
  RETURN[TRUE]
  END;

PPA: TYPE = POINTER TO PACKED ARRAY [0..0) OF AltoDefs.BYTE;

ByteBlockEqual: PROCEDURE [p1: PPA, n: CARDINAL, p2: PPA]
  RETURNS [BOOLEAN] =
  BEGIN
  RETURN[BlockEqual[p1: p1, p2: p2, n: n/2] AND p1[n-1] = p2[n-1]]
  END;

BlockEqualCode: PROCEDURE [p1: POINTER, n: CARDINAL, offset: CARDINAL]
  RETURNS [result: BOOLEAN] =
  BEGIN
  frame: GlobalFrameHandle = ControlDefs.GetReturnFrame[].accesslink;
  FrameDefs.SwapInCode[frame];
  result ← BlockEqual[p1: p1, n: n, p2: frame.code.codebase + offset];
  SegmentDefs.Unlock[frame.codesegment];
  RETURN
  END;

ByteBlockEqualCode: PROCEDURE [p1: POINTER, n: CARDINAL, offset: CARDINAL]
  RETURNS [result: BOOLEAN] =
  BEGIN
  frame: GlobalFrameHandle = ControlDefs.GetReturnFrame[].accesslink;
  FrameDefs.SwapInCode[frame];
  result ← ByteBlockEqual[p1: p1, n: n, p2: frame.code.codebase + offset];
  SegmentDefs.Unlock[frame.codesegment];
  RETURN
  END;


-- data shuffling

StringInit: PROCEDURE [coffset, n: CARDINAL, reloc, dest: POINTER] =
  BEGIN OPEN ControlDefs;
  g: GlobalFrameHandle = GetReturnFrame[].accesslink;
  i: CARDINAL;
  FrameDefs.SwapInCode[g];
  InlineDefs.COPY [
    from:g.code.codebase+coffset, to:dest, nwords:n];
```

```
    FOR i IN [0..n) DO
      (dest+i)↑ ← (dest+i)↑ + reloc;
      ENDLOOP;
    SegmentDefs.Unlock[g.codesegment];
    RETURN
    END;

  -- long, signed and mixed mode arithmetic

  DIVMOD: PROCEDURE [n,d: CARDINAL] RETURNS [QR] =
    LOOPHOLE[InlineDefs.DIVMOD];
  LDIVMOD: PROCEDURE [nlow,nhigh,d: CARDINAL] RETURNS [QR] =
    LOOPHOLE[InlineDefs.LDIVMOD];
  QR: TYPE = RECORD [q, r: INTEGER];
  PQR: TYPE = POINTER TO QR;

  LongSignDivide: PROCEDURE [numhigh: INTEGER, pqr: PQR] =
    BEGIN
    negnum,negden: BOOLEAN ← FALSE;
    IF negden ← (pqr.r < 0) THEN pqr.r ← -pqr.r;
    IF negnum ← (numhigh < 0) THEN
      BEGIN
      IF pqr.q = 0 THEN numhigh ← -numhigh
      ELSE BEGIN pqr.q ← -pqr.q; numhigh ← InlineDefs.BITNOT[numhigh] END;
      END;
    pqr↑ ← LDIVMOD[nlow: pqr.q, nhigh: numhigh, d: pqr.r];
    -- following assumes TRUE = 1; FALSE = 0
    IF InlineDefs.BITXOR[LOOPHOLE[negnum],LOOPHOLE[negden]] # 0 THEN
      pqr.q ← -pqr.q;
    IF negnum THEN pqr.r ← -pqr.r;
    RETURN
    END;

  DivSS: PROCEDURE =
    BEGIN
    state: ControlDefs.StateVector;
    p: PQR;
    t: CARDINAL;
    state ← STATE;
    state.stkptr ← t ← state.stkptr-1;
    state.dest ← ControlDefs.GetReturnLink[];
    p ← @state.stk[t-1];
    LongSignDivide[numhigh: (IF p.q<0 THEN -1 ELSE 0), pqr: p];
    RETURN WITH state
    END;

  LongCARDINAL: TYPE = InlineDefs.LongCARDINAL;
  DAdd: PROCEDURE [a,b: LongCARDINAL] RETURNS [LongCARDINAL] =
    MACHINE CODE BEGIN Mopcodes.zDADD END;
  DSub: PROCEDURE [a,b: LongCARDINAL] RETURNS [LongCARDINAL] =
    MACHINE CODE BEGIN Mopcodes.zDSUB END;
  DCompare: PROCEDURE [a,b: LongCARDINAL] RETURNS [{less, equal, greater}] =
    MACHINE CODE BEGIN Mopcodes.zDCOMP; Mopcodes.zINC END;

  DDivMod: PROCEDURE [num, den: LongCARDINAL]
    RETURNS [quotient, remainder: LongCARDINAL] =
    BEGIN OPEN InlineDefs;
    negNum, negDen: BOOLEAN ← FALSE;
    qq: CARDINAL;
    count: [0..31);
    lTemp: LongCARDINAL;
    IF LOOPHOLE[num.highbits, INTEGER] < 0 THEN
      BEGIN negNum ← TRUE; num ← DSub[[0,0],num]; END;
    IF LOOPHOLE[den.highbits, INTEGER] < 0 THEN
      BEGIN negDen ← TRUE; den ← DSub[[0,0],den]; END;
    IF den.highbits = 0 THEN
      BEGIN
      [quotient.highbits, qq] ←
        LongDivMod[[lowbits:num.highbits,highbits:0],den.lowbits];
      [quotient.lowbits, remainder.lowbits] ←
        LongDivMod[[lowbits:num.lowbits,highbits:qq],den.lowbits];
      remainder.highbits ← 0;
      END
    ELSE
      BEGIN
      count ← 0;
```

```
    quotient.highbits ← 0;
    lTemp ← den;
    WHILE lTemp.highbits # 0 DO -- normalize
      lTemp.lowbits ←
        BITSHIFT[lTemp.lowbits,-1] + BITSHIFT[lTemp.highbits,15];
      lTemp.highbits ← BITSHIFT[lTemp.highbits,-1];
      count ← count + 1;
      ENDLOOP;
    qq ← LongDiv[num,lTemp.lowbits]; -- trial quotient
    qq ← BITSHIFT[qq,-count];
    lTemp ← LongMult[den.lowbits,qq]; -- multiply by trial quotient
    lTemp.highbits ← lTemp.highbits + den.highbits*qq;
    UNTIL DCompare[lTemp, num] # greater DO
      -- decrease quotient until product is small enough
      lTemp ← DSub[lTemp,den];
      qq ← qq - 1;
      ENDLOOP;
    quotient.lowbits ← qq;
    remainder ← DSub[num,lTemp];
    END;
  IF BITXOR[LOOPHOLE[negNum],LOOPHOLE[negDen]] # 0 THEN
    quotient ← DSub[[0,0],quotient];
  IF negNum THEN remainder ← DSub[[0,0],remainder];
  RETURN
  END;

DDiv: PROCEDURE [a,b: LongCARDINAL] RETURNS [LongCARDINAL] =
  BEGIN OPEN InlineDefs;
  RETURN[DDivMod[a,b].quotient]
  END;

DMod: PROCEDURE [a,b: LongCARDINAL] RETURNS [r: LongCARDINAL] =
  BEGIN OPEN InlineDefs;
  [remainder: r] ← DDivMod[a,b];
  RETURN
  END;

DMultiply: PROCEDURE [a,b: LongCARDINAL]
  RETURNS [product: LongCARDINAL] =
  BEGIN OPEN InlineDefs;
  product ← LongMult[a.lowbits, b.lowbits];
  product.highbits ←
    product.highbits + a.lowbits*b.highbits + a.highbits*b.lowbits;
  RETURN
  END;

GetLevel: PUBLIC PROCEDURE RETURNS [INTEGER] =
  BEGIN RETURN[ResidentPtr.level] END;

SetLevel: PUBLIC PROCEDURE [l: INTEGER] = BEGIN ResidentPtr.level ← l; END;

Init: PROCEDURE =
  BEGIN OPEN SDDefs;
  sd: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
  resident: POINTER TO FRAME [Resident] ← ResidentPtr;
  sd[sStackError] ← StackErrorTrap;
  sd[sControlFault] ← ControlFaultTrap;
  sd[sBLTE] ← BlockEqual;
  sd[sBYTBLTE] ← ByteBlockEqual;
  sd[sBLTEC] ← BlockEqualCode;
  sd[sBYTBLTEC] ← ByteBlockEqualCode;
  sd[sStringInit] ← StringInit;
  sd[sDivSS] ← DivSS;
  sd[sLongMul] ← DMultiply;
  sd[sLongDivMod] ← DDivMod;
  sd[sLongMod] ← DMod;
  sd[sLongDiv] ← DDiv;
  sd[sCopy] ← Copy;
  sd[sUnNew] ← UnNew;

  BEGIN OPEN resident;
  sd[sAllocTrap] ← AllocTrap[AllocTrap[NullFrame]];
  sd[sSwapTrap] ← CodeTrap;
  sd[sUnbound] ← UnboundProcedureTrap;
  sd[sStart] ← Start;
  sd[sRestart] ← Restart;
```

```
    sd[sBreak] ← Break;
    sd[sAlternateBreak] ← WorryBreaker[];
    sd[sIOResetBits] ← 3;
    LOOPHOLE[CSPort,Port].in ← MemorySwap;
    LOOPHOLE[CSPort,Port].out ← @WBPort;
    sd[sCoreSwap] ← LOOPHOLE[WBPort,Port].out ← @CSPort;
    WBPort[NIL];
    level ← -1;
    END;
    END;


-- Main Body;

Init[];

END...
```