```
-- DebugNub.mesa; edited by Sandman August 15, 1978  2:40 PM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [BYTE],
  BFSDefs: FROM "bfsdefs" USING [MakeCFP],
  BootDefs: FROM "bootdefs" USING [GetSystemTable],
  ControlDefs: FROM "controldefs" USING [
    ControlLink, FieldDescriptor, FrameHandle, GetReturnFrame, GetReturnLink,
    GlobalFrameHandle, Lreg, NullFrame, SD, SetReturnFrame, SetReturnLink,
    StateVector, SVPointer],
  CoreSwapDefs: FROM "coreswapdefs" USING [
    BBHandle, callDP, DebugParameter, ExternalStateVector, PuntInfo,
    PuntTable, startDP, SVPointer, SwapReason, UBBPointer, UserBreakBlock],
  DiskDefs: FROM "diskdefs" USING [RealDA],
  FrameDefs: FROM "framedefs" USING [LockCode, UnlockCode, UnNew],
  ImageDefs: FROM "imagedefs" USING [
    AbortMesa, AddCleanupProcedure, AddFileRequest, CleanupItem, CleanupMask,
    CleanupProcedure, FileRequest, PuntMesa, StopMesa, UserCleanupProc],
  KeyDefs: FROM "keydefs" USING [Keys],
  LoadStateDefs: FROM "loadstatedefs" USING [GetLoadState],
  Mopcodes: FROM "mopcodes" USING [zRFS],
  NovaOps: FROM "novaops" USING [NovaJSR],
  NucleusDefs: FROM "nucleusdefs" USING [Wart],
  ProcessDefs: FROM "processdefs" USING [
    Aborted, DisableInterrupts, EnableInterrupts],
  SDDefs: FROM "sddefs" USING [sBreakBlock, sBreakBlockSize, sCallDebugger,
    sCoreSwap, sFirstFree, sInterrupt, sProcessBreakpoint, sUncaughtSignal],
  SegmentDefs: FROM "segmentdefs" USING [
    DeleteFileSegment, FileHandle, FileSegmentHandle, GetFileSegmentDA,
    LockFile, NewFileSegment, Read, ReleaseFile];

DEFINITIONS FROM CoreSwapDefs;

DebugNub: PROGRAM [user: PROGRAM]
IMPORTS BFSDefs, BootDefs, DiskDefs, FrameDefs, ImageDefs, LoadStateDefs,
  NucleusDefs, ProcessDefs, SegmentDefs
EXPORTS CoreSwapDefs SHARES BootDefs, SegmentDefs, ControlDefs =

BEGIN

FrameHandle: TYPE = ControlDefs.FrameHandle;
SVPointer: TYPE = ControlDefs.SVPointer;

ProcessBreakpoint: PROCEDURE [s: SVPointer] =
  BEGIN -- called by BRK trap handler in resident code
  inst: AltoDefs.BYTE;
  swap: BOOLEAN;
  IF ~Swappable THEN BEGIN SwatBreak[s]; RETURN END;
  [inst, swap] ← DoBreakpoint[s];
  IF swap THEN
    BEGIN
    FrameDefs.LockCode[s.dest];
    CoreSwap[breakpoint, s];
    FrameDefs.UnlockCode[s.dest];
    END
  ELSE s.instbyte ← inst;  --replant the instruction and go on
  RETURN
  END;

DoBreakpoint: PROCEDURE [s: SVPointer] RETURNS [AltoDefs.BYTE, BOOLEAN] =
  BEGIN OPEN ControlDefs;
  ubb: CoreSwapDefs.UBBPointer;
  bba: BBHandle = SD[SDDefs.sBreakBlock];
  i: CARDINAL;
  l: FrameHandle ← s.dest;
  FOR i IN [0..bba.length) DO
    ubb ← @bba.blocks[i];
    IF ubb.frame = l.accesslink AND ubb.pc = l.pc THEN
      IF TrueCondition[ubb, l] THEN EXIT ELSE RETURN[ubb.inst, FALSE];
    ENDLOOP;
  RETURN[0, TRUE];
  END;

TrueCondition: PROCEDURE [ubb: CoreSwapDefs.UBBPointer, base: POINTER]
  RETURNS [BOOLEAN] =
  BEGIN --decide whether to take the breakpoint
  fd: ControlDefs.FieldDescriptor;
  locL, locR: POINTER;
  left, right: UNSPECIFIED;
  IF ubb.counterL THEN
    RETURN[(ubb.ptrL ← ubb.ptrL + 1) = ubb.ptrR];
  locL ← IF ubb.localL THEN base+LOOPHOLE[ubb.ptrL, CARDINAL] ELSE ubb.ptrL;
  fd ← [offset: 0, posn: ubb.posnL, size: ubb.sizeL];
  left ← ReadField[locL, fd];
  IF ~ubb.immediateR THEN
    BEGIN
    fd ← [offset: 0, posn: ubb.posnR, size: ubb.sizeR];
    locR ← IF ubb.localR THEN base+LOOPHOLE[ubb.ptrR, CARDINAL] ELSE ubb.ptrR;
    right ← ReadField[locR, fd];
    END
  ELSE right ← ubb.ptrR;
  RETURN[SELECT ubb.relation FROM
    lt  => left < right,
    gt  => left > right,
    eq  => left = right,
    ne  => left # right,
    le  => left <= right,
    ge  => left >= right,
    ENDCASE => FALSE]
  END;

ReadField: PROCEDURE [POINTER, ControlDefs.FieldDescriptor]
  RETURNS[UNSPECIFIED] = MACHINE CODE BEGIN Mopcodes.zRFS END;

NumberBlocks: CARDINAL = 5;

InitBreakBlocks: PROCEDURE =
  BEGIN OPEN ControlDefs;
  sd: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← SD;
  sd[SDDefs.sBreakBlock] ← @sd[SDDefs.sFirstFree];
  sd[SDDefs.sBreakBlockSize] ← SIZE[UserBreakBlock]*NumberBlocks+1;
  sd[SDDefs.sFirstFree] ← 0;
  RETURN
  END;

SwatBreak: PROCEDURE [s: CoreSwapDefs.SVPointer] =
  BEGIN OPEN ControlDefs, NovaOps;
  break: RECORD[a,b: WORD];
  break ← [77400B, 1400B];
  s.instbyte ← NovaJSR[JSR, @break, 0];
  RETURN
  END;

Interrupt: PROCEDURE =
  BEGIN -- called by BRK trap handler in resident code
  state: ControlDefs.StateVector;
  state ← STATE;
  state.dest ← REGISTER[ControlDefs.Lreg];
  CoreSwap[breakpoint, @state];
  END;

Catcher: PROCEDURE [msg, signal: UNSPECIFIED, frame: FrameHandle] =
  BEGIN
  OPEN ControlDefs;
  SignallerGF: GlobalFrameHandle;
  state: StateVector;
  f: FrameHandle;
  state.stk[0] ← msg;
  state.stk[1] ← signal;
  state.stkptr ← 0;
  -- the call stack below here is: Signaller, [Signaller,] offender
  f ← GetReturnFrame[];
  SignallerGF ← f.accesslink;
  state.dest ← f ← f.returnlink.frame;
  IF f.accesslink = SignallerGF THEN state.dest ← f.returnlink;
  IF ~Swappable THEN BEGIN SwatBreak[@state]; RETURN END;
  BEGIN
    CoreSwap[uncaughtsignal, @state ! CAbort => GOTO abort];
  EXITS
    abort =>
      IF signal = ProcessDefs.Aborted THEN
        BEGIN
  BackStop[frame];
  ERROR KillThisTurkey;
        END
      ELSE SIGNAL ProcessDefs.Aborted;
  END;
  RETURN
  END;

BackStop: PROCEDURE [root: FrameHandle] =
  BEGIN OPEN ControlDefs;
  endProcess: ControlLink ← root.returnlink;
  caller: PROCEDURE = LOOPHOLE[GetReturnLink[]];
  root.returnlink ← LOOPHOLE[REGISTER[Lreg]];
  SetReturnFrame[NullFrame];
  caller[! KillThisTurkey => CONTINUE];
  SetReturnLink[endProcess];
  RETURN
  END;

KillThisTurkey: SIGNAL = CODE;

-- The core swapper

Quit: SIGNAL = CODE;
CantSwap: PUBLIC SIGNAL = CODE;
CAbort: PUBLIC SIGNAL = CODE;
DoSwap: PORT [POINTER TO ExternalStateVector];

parmstring: STRING ← [40];

CoreSwap: PUBLIC PROCEDURE [why: SwapReason, sp: SVPointer] =
  BEGIN OPEN NovaOps;
    loadstate: SegmentDefs.FileSegmentHandle;
    e: ExternalStateVector;
    DP: DebugParameter;
    decode: PROCEDURE RETURNS [BOOLEAN] =
      BEGIN OPEN ControlDefs; -- decode the SwapReason
      f: GlobalFrameHandle;
      lsv: StateVector;
      SELECT e.reason FROM
        proceed, resume => RETURN[TRUE];
        call =>
          BEGIN
          lsv ← LOOPHOLE[e.parameter, callDP].sv;
          lsv.source ← REGISTER[Lreg];
          TRANSFER WITH lsv;
          lsv ← STATE;
          LOOPHOLE[e.parameter, callDP].sv ← lsv;
          why ← return;
          END;
        start =>
          BEGIN
          f ← LOOPHOLE[e.parameter, startDP].frame;
          IF ~f.started THEN START LOOPHOLE[f, PROGRAM] ELSE RESTART f;
          why ← return;
          END;
        quit => SIGNAL Quit;
        kill => ImageDefs.AbortMesa[];
        showscreen =>
          BEGIN
          UNTIL KeyDefs.Keys.Spare3 = down DO NULL ENDLOOP;
          why ← return;
          END;
        ENDCASE =>
          BEGIN
          RETURN [TRUE];
          END;
      RETURN [FALSE]
      END;

    -- Body of CoreSwap

    IF ~Swappable THEN SIGNAL CantSwap;

    e.state ← sp;
    e.drumFile ← MesaCoreFH;
```

```
DP.string ← parmstring;
e.parameter ← @DP;
e.tables ← BootDefs.GetSystemTable[];
e.extension.loadstate ← loadstate ← LoadStateDefs.GetLoadState[];
e.loadstateCFA.fp ← loadstate.file.fp;
e.loadstateCFA.fa ← [page: loadstate.base, byte: 0,
  da: SegmentDefs.GetFileSegmentDA[loadstate]];
e.lspages ← loadstate.pages;
e.fill ← [0,0,0];

DO
  e.reason ← why;
  ImageDefs.UserCleanupProc[OutLd ! ANY => CONTINUE];
  ProcessDefs.DisableInterrupts[];
  DoSwap[@e];
  ProcessDefs.EnableInterrupts[];
  ImageDefs.UserCleanupProc[InLd];
    BEGIN
    IF decode[
      ! CAbort => IF e.level>0 THEN BEGIN why ← return; CONTINUE END;
        Quit => GOTO abort] THEN EXIT
    EXITS abort => SIGNAL CAbort;
    END;
  ENDLOOP;

RETURN
END;
```

```
-- initialization
DebuggerFileRequest: short ImageDefs.FileRequest ← ImageDefs.FileRequest [
  file: NIL, access: SegmentDefs.Read, link:,
  body: short[fill:, name:"MesaDebugger."]];

CoreFileRequest: short ImageDefs.FileRequest ← ImageDefs.FileRequest [
  file: NIL, access: SegmentDefs.Read, link:,
  body: short[fill:, name:"MesaCore."]];

SwatFileRequest: short ImageDefs.FileRequest ← ImageDefs.FileRequest [
  file: NIL, access: SegmentDefs.Read, link:,
  body: short[fill:, name:"Swatee."]];

Swappable: BOOLEAN;

puntData: PuntTable;
MesaCoreFH: SegmentDefs.FileHandle ← NIL;

FindFiles: PROCEDURE =
  BEGIN OPEN ControlDefs;
  f: SegmentDefs.FileHandle;
  s: SegmentDefs.FileSegmentHandle;

  puntData.puntESV.reason ← punt;
  puntData.puntESV.tables ← BootDefs.GetSystemTable[];
  puntData.puntESV.extension.loadstate ← s ← LoadStateDefs.GetLoadState[];
  puntData.puntESV.loadstateCFA.fp ← s.file.fp;
  puntData.puntESV.loadstateCFA.fa ← [page: s.base,
    byte: 0, da: SegmentDefs.GetFileSegmentDA[s]];
  puntData.puntESV.lspages ← s.pages;
  puntData.pDebuggerFP ← puntData.pCoreFP ← LOOPHOLE[0];

  Swappable ← TRUE;
  IF (f ← CoreFileRequest.file) = NIL THEN
    IF (f ← SwatFileRequest.file) = NIL THEN
      Swappable ← FALSE
    ELSE NULL
  ELSE IF SwatFileRequest.file # NIL THEN
    SegmentDefs.ReleaseFile[SwatFileRequest.file];
  IF Swappable THEN
    BEGIN OPEN DiskDefs, SegmentDefs;
    ENABLE ANY => GOTO bad;

    LockFile[puntData.puntESV.drumFile ← MesaCoreFH ← f];
    s ← NewFileSegment[f,1,1,Read];
    BFSDefs.MakeCFP[@puntData.coreFP,@f.fp];
    puntData.coreFP.leaderDA ← LOOPHOLE[RealDA[GetFileSegmentDA[s]]];
    puntData.pCoreFP ← @puntData.coreFP;
    DeleteFileSegment[s];

    IF (f ← DebuggerFileRequest.file) = NIL THEN GOTO bad;
    s ← NewFileSegment[f,1,1,Read];
    BFSDefs.MakeCFP[@puntData.debuggerFP,@f.fp];
    puntData.debuggerFP.leaderDA ← LOOPHOLE[RealDA[GetFileSegmentDA[s]]];
    puntData.pDebuggerFP ← @puntData.debuggerFP;
    DeleteFileSegment[s];
    PuntInfo↑ ← @puntData;
    EXITS
      bad => Swappable ← FALSE;
    END;
  puntData.puntESV.drumFile ← MesaCoreFH;
  RETURN
  END;

RequestFiles: PROCEDURE =
  BEGIN
  DebuggerFileRequest.file ← NIL;
  CoreFileRequest.file ← NIL;
  SwatFileRequest.file ← NIL;
  ImageDefs.AddFileRequest[@DebuggerFileRequest];
  ImageDefs.AddFileRequest[@CoreFileRequest];
  ImageDefs.AddFileRequest[@SwatFileRequest];
  END;

CleanupItem: ImageDefs.CleanupItem ← [link:, proc: CleanupNub,
  mask: ImageDefs.CleanupMask[Save] + ImageDefs.CleanupMask[Restore]];
```

```
CleanupNub: ImageDefs.CleanupProcedure =
  BEGIN
  SELECT why FROM
    Save => RequestFiles[];
    Restore => FindFiles[];
    ENDCASE;
  END;

CallDebugger: PROCEDURE [s: STRING] =
  BEGIN -- user's entry point to debugger
  state: ControlDefs.StateVector;
  state ← STATE;
  state.stk[0] ← s;
  state.stkptr ← 1;
  state.dest ← ControlDefs.GetReturnLink[];
  CoreSwap[explicitcall, @state];
  RETURN
  END;

SetSD: PROCEDURE =
  BEGIN OPEN SDDefs;
  sd: POINTER TO ARRAY [0..0) OF UNSPECIFIED ← ControlDefs.SD;
  sd[sProcessBreakpoint] ← ProcessBreakpoint;
  sd[sUncaughtSignal] ← Catcher;
  sd[sInterrupt] ← Interrupt;
  sd[sCallDebugger] ← CallDebugger;
  END;

-- Main body

P: TYPE = MACHINE DEPENDENT RECORD [in, out: UNSPECIFIED]; -- PORT

LOOPHOLE[DoSwap,P] ← [in: 0, out: ControlDefs.SD[SDDefs.sCoreSwap]];
RequestFiles[];

START user;

STOP;

BEGIN
  ENABLE ANY => ImageDefs.PuntMesa;
FindFiles[];
InitBreakBlocks[];
SetSD[];
FrameDefs.UnNew[LOOPHOLE[NucleusDefs.Wart]];
END;

ImageDefs.AddCleanupProcedure[@CleanupItem];

RESTART user [! ProcessDefs.Aborted, CAbort => CONTINUE];

ImageDefs.StopMesa[];

END...
```