```
-- Resident.Mesa; edited by Sandman on Jul 25, 1978 8:41 AM

DIRECTORY
  AllocDefs: FROM "allocdefs" USING [AllocInfo, MakeSwappedIn],
  AltoDefs: FROM "altodefs" USING [PageSize],
  BootDefs: FROM "bootdefs",
  ControlDefs: FROM "controldefs" USING [
    ATPreg, AV, AVItem, ControlLink, EntryVectorItem, EPRange, FrameHandle,
    FrameVec, Free, GetReturnLink, GFT, GFTIndex, GFTItem, GlobalFrameHandle,
    LargeReturnSlot, Lreg, MainBodyIndex, MaxAllocSlot, NullFrame,
    NullGlobalFrame, OTPreg, Port, ProcDesc, SD, SetReturnFrame,
    SetReturnLink, SpecialReturnSlot, StateVector, WDCreg, XTPreg, XTSreg],
  CoreSwapDefs: FROM "coreswapdefs" USING [
    ExternalStateVector, PuntInfo, SVPointer],
  FrameDefs: FROM "framedefs",
  ImageDefs: FROM "imagedefs" USING [AbortMesa, PuntMesa],
  InlineDefs: FROM "inlinedefs" USING [BITAND, BITOR],
  KeyDefs: FROM "keydefs" USING [Keys],
  Mopcodes: FROM "mopcodes" USING [zKFCB, zPOP, zSTARTIO],
  NovaOps: FROM "novaops" USING [NovaInLd, NovaOutLd],
  NucleusDefs: FROM "nucleusdefs",
  ProcessDefs: FROM "processdefs" USING [
    ActiveWord, CV, DisableInterrupts, DisableTimeout, EnableInterrupts,
    Enter, Fork, GetPriority, ParityLevel, Priority, ReEnter, SetPriority,
    Wait, WakeupsWaiting],
  SDDefs: FROM "sddefs" USING [
    sAllocTrap, SD, sInterrupt, sIOResetBits, sProcessBreakpoint, sXferTrap],
  SegmentDefs: FROM "segmentdefs" USING [
    AddressFromPage, DataSegmentAddress, DataSegmentHandle, DefaultBase,
    DeleteDataSegment, FileSegmentHandle, NewFrameSegment, Unlock],
  TrapDefs: FROM "trapdefs" USING [
    TraceNext, TraceOff, TrapParameter, TrapStatus];

DEFINITIONS FROM AltoDefs, ControlDefs;

Resident: PROGRAM
  IMPORTS AllocDefs, ProcessDefs, SegmentDefs
  EXPORTS BootDefs, FrameDefs, NucleusDefs, TrapDefs
  SHARES ProcessDefs =

BEGIN

  -- allocation of frame space

  LargeFrameSlot: CARDINAL = 12;

  FrameSize: PUBLIC PROCEDURE [fsi: CARDINAL] RETURNS [CARDINAL] =
    BEGIN
    RETURN[IF fsi >= LENGTH[FrameVec] THEN fsi ELSE FrameVec[fsi]]
    END;

  pgft: TYPE = POINTER TO ARRAY [0..0) OF GFTItem;

  ItemPointer: TYPE = POINTER TO ControlDefs.AVItem;

  FrameSegment: TYPE = MACHINE DEPENDENT RECORD [
    segment: SegmentDefs.DataSegmentHandle,
    link: POINTER TO FrameSegment,
    size, fsi: CARDINAL];


  -- maintain a list of all new "permanent" frame segments;
  SegHeader: PUBLIC TYPE = RECORD [
    seg: SegmentDefs.DataSegmentHandle, link: pSegHeader];
  pSegHeader: PUBLIC TYPE = POINTER TO SegHeader;
  SegListHead: PUBLIC pSegHeader ← NIL;

  ExtraSpaceSize: CARDINAL = PageSize;
  ExtraSpace: ARRAY [0..ExtraSpaceSize) OF WORD;
  InitNewSpace: POINTER = LOOPHOLE[InlineDefs.BITOR[LOOPHOLE[BASE[ExtraSpace]],3]];
  InitWordsLeft: CARDINAL = BASE[ExtraSpace]+ExtraSpaceSize-InitNewSpace;

  NULLPtr: FrameHandle = LOOPHOLE[0];

  AllocTrap: PROCEDURE [otherframe: FrameHandle]
  RETURNS [myframe: FrameHandle] =
```

```
BEGIN OPEN ProcessDefs, SegmentDefs;
ATFrame: TYPE = POINTER TO FRAME [AllocTrap];
state: StateVector;
newframe: FrameHandle;
newseg: DataSegmentHandle;
eventry: POINTER TO EntryVectorItem;
i, fsize, findex: CARDINAL;
p: POINTER;
newG: GlobalFrameHandle;
NewSpacePtr: POINTER;
WordsLeft: CARDINAL ← 0;
recurring: BOOLEAN ← otherframe # NULLPtr;
alloc: BOOLEAN;
dest, tempdest: ControlLink;
gfi: GFTIndex;
ep: CARDINAL;
myframe ← REGISTER[Lreg];
state.dest ← myframe.returnlink; state.source ← 0;
state.instbyte←0;
state.stk[0] ← myframe;
state.stkptr ← 1;

ProcessDefs.DisableInterrupts[];  -- so that undo below works

DO ENABLE ANY => ImageDefs.PuntMesa[];

IF ~recurring THEN
  BEGIN
  LOOPHOLE[otherframe, ATFrame].NewSpacePtr ← InitNewSpace;
  LOOPHOLE[otherframe, ATFrame].WordsLeft ← InitWordsLeft;
  AV[SpecialReturnSlot] ← [data[0,empty]];
  END;

ProcessDefs.EnableInterrupts[];  -- guarantees one more instruction
TRANSFER WITH state;
ProcessDefs.DisableInterrupts[];

state ← STATE;
dest ← LOOPHOLE[REGISTER[ATPreg]];

SD[SDDefs.sAllocTrap] ← otherframe;
myframe.returnlink ← state.source;

tempdest ← dest;
DO
  SELECT tempdest.tag FROM
    frame =>
      BEGIN
      alloc ← TRUE;
      findex ← LOOPHOLE[tempdest, CARDINAL]/4;
      EXIT
      END;
    procedure =>
      BEGIN OPEN proc: LOOPHOLE[tempdest, ProcDesc];
      gfi ← proc.gfi; ep ← proc.ep;
      [frame: newG, epbase: findex] ← GFT[gfi];
      eventry ← @newG.code.prefix.entry[findex+ep];
      findex ← eventry.framesize;
      alloc ← FALSE;
      EXIT
      END;
    indirect => tempdest ← tempdest.link↑;
    ENDCASE => ImageDefs.PuntMesa[];
  ENDLOOP;

IF ~recurring THEN FlushLargeFrames[]
ELSE
  IF (p ← AV[SpecialReturnSlot].link) # LOOPHOLE[AVItem[data[0,empty]]] THEN
    BEGIN
    WordsLeft ← WordsLeft + (NewSpacePtr-p+1);
    NewSpacePtr ← p-1;
    AV[SpecialReturnSlot] ← [data[0,empty]];
    END;

IF findex < LargeFrameSlot THEN
  BEGIN
```

```
      fsize ← FrameVec[findex]+1;  -- includes overhead word
      THROUGH [0..1] DO
        p ← NewSpacePtr+1;
        IF fsize <= WordsLeft THEN
          BEGIN
          newframe ← p;
          (p-1)↑ ← IF recurring THEN SpecialReturnSlot ELSE findex;
          WordsLeft ← WordsLeft - fsize;
          NewSpacePtr ← NewSpacePtr + fsize;
          EXIT;
          END
        ELSE
          BEGIN
          IF recurring THEN ImageDefs.PuntMesa[];
          FOR i DECREASING IN [0..findex) DO
            IF FrameVec[i] < WordsLeft THEN
              BEGIN
              (p-1)↑ ← i;
              p↑ ← AV[i].link;
              AV[i].link ← p;
              EXIT;
              END;
            ENDLOOP;
          NewSpacePtr ←
            (p←DataSegmentAddress[newseg←NewFrameSegment[1]]) + 3;
          LOOPHOLE[p,pSegHeader]↑ ← [newseg,SegListHead];
          SegListHead ← p;
          WordsLeft ← PageSize-3;
          END;
        ENDLOOP
      END
    ELSE
      BEGIN
      fsize ← SELECT findex FROM
        > MaxAllocSlot => findex,
        = MaxAllocSlot =>
          (newG.code.codebase + CARDINAL[eventry.initialpc] - 1)↑,
        ENDCASE => FrameVec[findex];
      p ← DataSegmentAddress[newseg ←
  NewFrameSegment[(fsize + PageSize + 3)/PageSize]];
      p↑ ← newseg;
      (p+2)↑ ← fsize;
      (p+3)↑ ← LargeReturnSlot;
      newframe ← p + 4;
      END;

    IF alloc THEN
      BEGIN
      state.dest ← myframe.returnlink;
      state.stk[state.stkptr] ← newframe;
      state.stkptr ← state.stkptr+1;
      END
    ELSE
      BEGIN
      IF dest.tag # indirect THEN
        BEGIN
        state.dest ← newframe;
        newframe.accesslink ← newG;
        newframe.pc ← eventry.initialpc;
        newframe.returnlink ← myframe.returnlink;
        END
      ELSE
        BEGIN
        IF findex = MaxAllocSlot THEN ImageDefs.PuntMesa[];
        state.dest ← dest;
        newframe.accesslink ← LOOPHOLE[AV[findex].link];
        AV[findex].frame ← newframe;
        END;
      state.source ← myframe.returnlink;
      END;

    SD[SDDefs.sAllocTrap] ← myframe;

    ENDLOOP;
    END;
```

```
  FlushLargeFrames: PUBLIC PROCEDURE =
    BEGIN
    p: POINTER;
    item: ItemPointer ← @AV[LargeReturnSlot];
    WHILE item.tag = frame DO
      p ← item.frame; item.frame ← p↑;
      SegmentDefs.DeleteDataSegment[LOOPHOLE[(p-4)↑]];
      ENDLOOP;
    END;

  -- other traps

  UnboundProcedure: PUBLIC SIGNAL [dest: ControlLink] RETURNS [ControlLink] = CODE;

  UnboundProcedureTrap: PROCEDURE =
    BEGIN
    dest: ControlLink;
    state: StateVector;
    ProcessDefs.DisableInterrupts[];
    state ← STATE;
    dest ← LOOPHOLE[REGISTER[OTPreg]];
    ProcessDefs.EnableInterrupts[];
    state.source ← GetReturnLink[];
    state.dest ← SIGNAL UnboundProcedure[dest];
    RETURN WITH state
    END;

  StartFault: PUBLIC SIGNAL [dest: GlobalFrameHandle] = CODE;

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

  Start: PUBLIC PROCEDURE [dest: GlobalFrameHandle] =
    BEGIN
    state: StateVector;
    control: GlobalFrameHandle;
    state ← STATE;
    IF dest = NullGlobalFrame OR dest.started THEN
      ERROR StartFault[dest];
    IF (control ← dest.global[0]) # NullGlobalFrame
      AND ~control.started THEN Start[control];
    --SwapInCode[dest];
    --IF dest.code.prefix.fill = 1 THEN SIGNAL CodeInconsistency[dest];
    --SegmentDefs.Unlock[dest.codesegment];
    IF ~dest.started THEN
      BEGIN
      state.dest ← ControlLink[procedure[
        gfi: dest.gfi, ep: MainBodyIndex, tag: procedure]];
      state.source ← GetReturnLink[];
      dest.started ← TRUE;
      RETURN WITH state
      END
    ELSE IF state.stkptr # 0 THEN SIGNAL StartFault[dest];
    RETURN
    END;

  Restart: PUBLIC PROCEDURE [dest: GlobalFrameHandle] =
    BEGIN
    stops: BOOLEAN;
    frame: FrameHandle;
    IF dest = NullGlobalFrame THEN ERROR StartFault[dest];
    IF ~dest.started THEN Start[dest];
    SwapInCode[dest];
    stops ← dest.code.prefix.stops;
    SegmentDefs.Unlock[dest.codesegment];
    IF ~stops THEN ERROR StartFault[dest];
    IF (frame ← dest.global[0]) # NullFrame THEN
      BEGIN
      frame.returnlink ← GetReturnLink[];
      SetReturnFrame[frame];
      END;
    RETURN
    END;

  CodeTrap: PROCEDURE =
    BEGIN
    dest: ControlLink;
```

```
    state: StateVector;
    frame: GlobalFrameHandle;
    ProcessDefs.DisableInterrupts[];
    state ← STATE;
    dest ← LOOPHOLE[REGISTER[OTPreg]];
    ProcessDefs.EnableInterrupts[];
    state.dest ← dest;
    state.source ← GetReturnLink[];
    DO
      SELECT dest.tag FROM
        frame => BEGIN frame ← dest.frame.accesslink; EXIT END;
        procedure => BEGIN frame ← GFT[dest.gfi].frame; EXIT END;
        ENDCASE => dest ← dest.link↑;
      ENDLOOP;
    IF ~frame.started THEN Start[frame];
    SwapInCode[frame];
    SegmentDefs.Unlock[frame.codesegment];
    RETURN WITH state;
    END;

  SwapInCode: PUBLIC PROCEDURE [f: GlobalFrameHandle] =
    BEGIN OPEN SegmentDefs;
    seg: FileSegmentHandle;
    -- It is believed that Disabling during SwapIn is unnecessary
    -- as long as ALL interrupt code is locked.  The
    -- Swapper should have segment locks to help fix this.
    info: AllocDefs.AllocInfo = [0,easy,bottomup, initial, code, FALSE, FALSE];
    AllocDefs.MakeSwappedIn[(seg ← f.codesegment), DefaultBase, info];
    ProcessDefs.DisableInterrupts[];
    IF f.code.swappedout THEN
      BEGIN
      -- Don't call FileSegmentAddress; it's not locked!
      f.code.codebase ← AddressFromPage[seg.VMpage]+f.code.offset;
      f.code.swappedout ← FALSE;
      END;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  -- Parity Errors

  ParityError: PUBLIC SIGNAL [address: POINTER] = CODE;
  PhantomParityError: PUBLIC SIGNAL = CODE;

  ParityProcess: PROCEDURE =
    BEGIN OPEN ProcessDefs;
    p: ORDERED POINTER;
    dummy: MONITORLOCK;
    error: CONDITION;
    ww: POINTER TO MACHINE DEPENDENT RECORD [
      other: [0..77777B], parity: BOOLEAN] ←
      LOOPHOLE[ProcessDefs.WakeupsWaiting];
    POP: PROCEDURE [WORD] = MACHINE CODE BEGIN Mopcodes.zPOP END;
    [] ← Enter[@dummy];
    DisableTimeout[@error];
    CV[ParityLevel] ← @error;
    DO  -- forever
      Wait[@dummy, @error, error.timeout];
      WHILE ~ReEnter[@dummy, @error] DO NULL ENDLOOP;
      ActiveWord↑ ← 0;
      FOR p DECREASING IN [LOOPHOLE[0]..LOOPHOLE[177000B]) DO
        POP[p↑];
        IF ww.parity THEN
          BEGIN SIGNAL ParityError[p]; EXIT END;
        REPEAT FINISHED => SIGNAL PhantomParityError;
        ENDLOOP;
      ww.parity ← FALSE;
      ActiveWord↑ ← 77777B;
      ENDLOOP;
    END;

  InitParity: PROCEDURE =
    BEGIN OPEN ProcessDefs;
    save: Priority ← GetPriority[];
    SetPriority[LAST[Priority]];
    [] ← Fork[ParityProcess];
```

```
    SetPriority[save];
    RETURN
    END;

  -- Getting the Debugger

  level: INTEGER;
  StartIO: PROCEDURE [CARDINAL] = MACHINE CODE BEGIN Mopcodes.zSTARTIO END;

  CSPort: PORT RETURNS [POINTER TO CoreSwapDefs.ExternalStateVector];
  WBPort: PORT [POINTER TO CoreSwapDefs.ExternalStateVector];

  MemorySwap: PROCEDURE [ESV: POINTER TO CoreSwapDefs.ExternalStateVector] =
    BEGIN
    savewdc: UNSPECIFIED;
    xferTrapStatus: UNSPECIFIED;
    xferTrapHandler: UNSPECIFIED;
    flag: [0..2];
    DO
      ESV ← CSPort[];
      SetReturnLink[LOOPHOLE[CSPort, Port].dest];
      StartIO[SD[SDDefs.sIOResetBits]];  -- reset IO devices
      ESV.level ← level;
      xferTrapStatus ← REGISTER[XTSreg];
      xferTrapHandler ← SD[SDDefs.sXferTrap];
      SD[SDDefs.sXferTrap] ← REGISTER[Lreg];
      REGISTER[XTSreg] ← TrapDefs.TraceOff;
      savewdc ← REGISTER[WDCreg];
      flag ← NovaOps.NovaOutLd[OutLd,CoreSwapDefs.PuntInfo↑.pCoreFP,ESV];
      REGISTER[WDCreg] ← savewdc;
      SELECT flag FROM
        0 => NovaOps.NovaInLd[InLd,CoreSwapDefs.PuntInfo↑.pDebuggerFP,ESV];
        1 => level ← ESV.level;
        ENDCASE => ESV.reason ← proceed;
      REGISTER[XTSreg] ← xferTrapStatus;
      SD[SDDefs.sXferTrap] ← xferTrapHandler;
      ENDLOOP;
    END;

  Break: PROCEDURE =
    -- executed by (non-worry) BRK instruction
    BEGIN
    ProcessBreakpoint: PROCEDURE [CoreSwapDefs.SVPointer] =
      MACHINE CODE BEGIN Mopcodes.zKFCB, SDDefs.sProcessBreakpoint END;
    f: FrameHandle;
    state: StateVector;
    xferTrapStatus: TrapDefs.TrapStatus;
    state ← STATE;
    state.dest ← f ← state.source;
    state.source ← REGISTER[Lreg];
    f.pc ← [IF f.pc < 0 THEN -f.pc ELSE (1-f.pc)];
    xferTrapStatus ← REGISTER[XTSreg];
    ProcessBreakpoint[@state];
    IF xferTrapStatus.state = on THEN REGISTER[XTSreg] ← TrapDefs.TraceNext;
    RETURN WITH state
    END;

-- Worry mode breakpoints

  WorryBreaker: PROCEDURE RETURNS [FrameHandle] =
    BEGIN
    worrystate: StateVector;
    worryframe: FrameHandle;
    worryESV: CoreSwapDefs.ExternalStateVector;
    xferTrapStatus: TrapDefs.TrapStatus;

    worrystate.instbyte ← 0;
    worrystate.stkptr ← 1;
    worrystate.stk[0] ← REGISTER[Lreg];
    worrystate.dest ← ControlDefs.GetReturnLink[];
    ProcessDefs.DisableInterrupts[];
    DO
      xferTrapStatus ← REGISTER[XTSreg];
      IF xferTrapStatus.state = on THEN REGISTER[XTSreg] ← TrapDefs.TraceNext;
      ProcessDefs.EnableInterrupts[];
      TRANSFER WITH worrystate;
```

```
      ProcessDefs.DisableInterrupts[];
      worrystate ← STATE;
      worrystate.dest ← worryframe ← worrystate.source;
      worrystate.source ← REGISTER[Lreg];
      worryframe.pc ←
        [IF worryframe.pc < 0 THEN -worryframe.pc ELSE (1-worryframe.pc)];
      worryESV ← CoreSwapDefs.PuntInfo↑.puntESV;
      worryESV.state ← @worrystate;
      worryESV.reason ← worrybreak;
      DO
        WBPort[@worryESV];
        SELECT worryESV.reason FROM
          proceed => EXIT;
          kill => ImageDefs.AbortMesa[];
          showscreen =>
            UNTIL KeyDefs.Keys.Spare3 = down DO NULL ENDLOOP;
          ENDCASE;
        worryESV.reason ← return;
        ENDLOOP;
      ENDLOOP;
    END;

continueTracing: BOOLEAN;
Notify: PROCEDURE = MACHINE CODE BEGIN Mopcodes.zKFCB, SDDefs.sInterrupt END;

StartTrace: PROCEDURE [
  loc: POINTER, val: UNSPECIFIED, mask: WORD, equal: BOOLEAN] =
  BEGIN OPEN TrapDefs, ControlDefs;
  state: StateVector;
  trapParam: TrapParameter;
  status: TrapStatus;
  frame: FrameHandle;
  ep: CARDINAL;
  lval: UNSPECIFIED;

  continueTracing ← TRUE;
  state ← STATE;
  state.dest ← GetReturnLink[];
  SDDefs.SD[SDDefs.sXferTrap] ← state.source ← REGISTER[Lreg];
  ProcessDefs.DisableInterrupts[];
  DO
    lval ← InlineDefs.BITAND[loc↑, mask];
    IF (IF equal THEN val = lval ELSE val # lval) THEN Notify[];
    IF ~continueTracing THEN
      BEGIN
      ProcessDefs.EnableInterrupts[];
      RETURN WITH state;
      END
    ELSE
      BEGIN
      REGISTER[XTSreg] ← TraceNext;
      ProcessDefs.EnableInterrupts[];
      TRANSFER WITH state;
      END;
    ProcessDefs.DisableInterrupts[];
    state ← STATE;
    trapParam ← REGISTER[XTPreg];
    status ← REGISTER[XTSreg];
    REGISTER[XTSreg] ← TraceOff;
    SELECT status.reason FROM
      other => SetReturnLink[
        IF state.source = NullFrame THEN trapParam.link ELSE state.source];
      localCall =>
        BEGIN
        ep ← (trapParam.ep-2)/2;
        frame ← state.source;
        trapParam.link ← ControlLink[procedure[tag: procedure,
          gfi: frame.accesslink.gfi+ep/EPRange, ep: ep MOD EPRange]];
        SetReturnFrame[frame];
        END;
      return =>
        BEGIN
        frame ← trapParam.frame-6;
        trapParam.link ← frame.returnlink;
        SetReturnFrame[frame];
        END;
```

```
      ENDCASE;
    state.dest ← trapParam.link;
    IF status.reason = return THEN
      BEGIN Free[frame]; state.source ← NullFrame; END;
    ENDLOOP;
  END;

StopTrace: PROCEDURE =
  BEGIN
  continueTracing ← FALSE;
  RETURN
  END;

-- Main Body;

STOP;
InitParity[];


END.
```