```
-- Miscellaneous.Mesa  Edited by Sandman on October 11, 1977  9:45 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  BcdDefs: FROM "bcddefs",
  ControlDefs: FROM "controldefs",
  FrameDefs: FROM "framedefs",
  ImageDefs: FROM "imagedefs",
  InlineDefs: FROM "inlinedefs",
  MiscDefs: FROM "miscdefs",
  Mopcodes: FROM "mopcodes",
  OsStaticDefs: FROM "osstaticdefs",
  ProcessDefs: FROM "processdefs",
  SegmentDefs: FROM "segmentdefs",
  TrapDefs: FROM "trapdefs";

DEFINITIONS FROM ControlDefs;

Miscellaneous: PROGRAM
  IMPORTS FrameDefs, SegmentDefs
  EXPORTS FrameDefs, ImageDefs, InlineDefs, MiscDefs, TrapDefs
  SHARES ControlDefs, ImageDefs = BEGIN

  PORTI: MACHINE CODE = INLINE [Mopcodes.zPORTI];

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

  NoGlobalFrameSlots: PUBLIC SIGNAL [CARDINAL] = CODE;

  EnterGlobalFrame: PUBLIC PROCEDURE [frame: GlobalFrameHandle, nslots: CARDINAL] RETURNS [entryindex:
**GFTIndex] =
    BEGIN
    gft: POINTER TO ARRAY [0..0) OF GFTItem = REGISTER[GFTreg];
    sd: POINTER TO ARRAY [0..0) OF CARDINAL = REGISTER[SDreg];
    i, imax, n, epoffset: CARDINAL;
    i ← gftrover;  imax ← sd[sGFTLength] - nslots;  n ← 0;
    DO
      IF (i ← IF i>=imax THEN 1 ELSE i+1) = gftrover
        THEN SIGNAL NoGlobalFrameSlots[nslots];
      IF gft↑[i].frame # NULLFrame
        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 = REGISTER[GFTreg];
    sd: POINTER TO ARRAY [0..0) OF CARDINAL = REGISTER[SDreg];
    i: CARDINAL;
    FOR i ← frame.gftindex.gftindex, i+1
    WHILE i<sd[sGFTlength] AND gft↑[i].frame=frame DO
      gft↑[i] ← GFTItem[NULLFrame,NULLEpBase];
      ENDLOOP;
    RETURN
    END;

  ReleaseFrame: PUBLIC PROCEDURE [frame:GlobalFrameHandle] =
    BEGIN
    updatelinks: PROCEDURE [f:GlobalFrameHandle]
      RETURNS [BOOLEAN] = BEGIN
      IF f # frame THEN
        BEGIN
        IF f.pc = 0 AND (f+globalbase).accesslink = frame THEN
          (f+globalbase).accesslink ← NULLFrame;
        IF f.ownerlink = frame THEN
          f.ownerlink ← frame.ownerlink;
        IF f.bindentry = frame THEN
```

```
        f.bindentry ← frame.bindlink;
      IF f.bindlink = frame THEN
        f.bindlink ← frame.bindlink;
      END;
    RETURN[FALSE]
    END;
  [] ← FrameDefs.EnumerateGlobalFrames[updatelinks];
  RemoveGlobalFrame[frame];
  Free[frame];
  RETURN
  END;

DeletedFrame: PUBLIC PROCEDURE [gfi: GFTIndex] RETURNS [BOOLEAN] =
  BEGIN
  gft: POINTER TO ARRAY [0..0) OF GFTItem = REGISTER[GFTreg];
  RETURN[gft[gfi] = [frame: NULLFrame, epbase: NULLEpBase]];
  END;

LockCode: PUBLIC PROCEDURE [link: UNSPECIFIED] =
  BEGIN
  FrameDefs.SwapInCode[FrameDefs.GlobalFrame[link]];
  RETURN
  END;

UnlockCode: PUBLIC PROCEDURE [link: UNSPECIFIED] =
  BEGIN
  SegmentDefs.Unlock[FrameDefs.GlobalFrame[link].codesegment];
  RETURN
  END;

CodeSegment: PUBLIC PROCEDURE [frame:FrameHandle]
  RETURNS [codeseg: SegmentDefs.FileSegmentHandle] =
  BEGIN
  codeseg ← frame.accesslink.codesegment;
  IF codeseg # NIL AND codeseg.class # code THEN ERROR;
  RETURN
  END;

StackError: PUBLIC ERROR [FrameHandle] = CODE;

StackErrorTrap: PROCEDURE =
  BEGIN
  state: StateVector;
  state ← STATE;
  ERROR StackError[GetReturnFrame[]];
  END;

NullPort: PortHandle = LOOPHOLE[0];

PortFault: PUBLIC ERROR = CODE;
LinkageFault: PUBLIC ERROR = CODE;
ControlFault: PUBLIC SIGNAL [source: FrameHandle] RETURNS [ControlLink] = CODE;

ControlFaultTrap: PROCEDURE =
  BEGIN
  errorStart, savedState: StateVector;
  p, q: PortHandle;
  sourceFrame, self: FrameHandle;
  savedState ← STATE;
  self ← REGISTER[Lreg];
  IF PortCall[self.returnlink] THEN
    BEGIN
    p ← LOOPHOLE[self.returnlink];
    WITH pdest:p SELECT plink FROM
      plink => q ← pdest.port;
      ENDCASE;
    WITH pf:p.pendingFrame SELECT frame FROM
      frame => sourceFrame ← pf.frameLink;
      ENDCASE;
    IF q = NullPort THEN
      errorStart.stk[0] ← LinkageFault
    ELSE
      BEGIN
      q↑ ← Port[TrapLink,plink[p]];
      errorStart.stk[0] ← PortFault;
      END;
```

```
      errorStart.stk[1] ← 0;
      errorStart.instbyte ← 0;
      errorStart.stkptr ← 2;
      errorStart.Y ← sourceFrame.returnlink;
      errorStart.X ←(REGISTER[SDreg]+sError)↑;
      IF savedState.stkptr = 0 THEN
        RETURN WITH errorStart  --  RESPONDING port
      ELSE
        BEGIN
        p.pendingFrame ← ControlLink[frame[self]];
        TRANSFER WITH errorStart;
        PORTI;
        p.pendingFrame ← ControlLink[frame[sourceFrame]];
        savedState.Y ← p;
        WITH dp:p SELECT plink FROM
          plink => savedState.X ← dp.port;
          ENDCASE;
        RETURN WITH savedState;
        END;
      END
    ELSE
      BEGIN
      savedState.Y ← self.returnlink;
      savedState.X ← SIGNAL ControlFault[savedState.Y];
      RETURN WITH savedState
      END;
    END;

  PortCall: PROCEDURE [source: ControlLink] RETURNS [BOOLEAN] =
    BEGIN
    portcall: BOOLEAN ← FALSE;
    WITH cLink: source SELECT representation FROM
      representation =>
        BEGIN
        WHILE cLink.type = indirecttag DO
          source ← LOOPHOLE[cLink,indirect ControlLink].indirectLink↑;
          ENDLOOP;
        IF cLink.type = frametag THEN
          IF FrameDefs.ReturnByte[LOOPHOLE[cLink,frame ControlLink].frameLink,0] = Mopcodes.zPORTI
            THEN portcall ← TRUE;
        END;
      ENDCASE;
    RETURN[portcall]
    END;

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

  UnboundProcedureTrap: PROCEDURE [dest: UnboundDesc] =
    BEGIN
    state: StateVector;
    state ← STATE;
    state.Y ← GetReturnLink[];
    state.X ← SIGNAL UnboundProcedure[dest];
    RETURN WITH state
    END;

  Copy: PROCEDURE [oldframe: GlobalFrameHandle] RETURNS [newframe: GlobalFrameHandle] =
    BEGIN
    gft: POINTER TO ARRAY [0..1) OF GFTItem = REGISTER[GFTreg];
    codeseg: SegmentDefs.FileSegmentHandle ← oldframe.codesegment;
    cp: POINTER TO CsegPrefix;
    gfti: GFTIndex;
    oldgfti: GFTIndex ← oldframe.gftindex.gftindex;
    i, size: CARDINAL ← 0;
    procvar: POINTER TO ProcDesc;
    FrameDefs.LockCode[oldframe];
    [newframe, size, cp] ← AllocGlobalFrame[codeseg];
    -- initialize control fields
    InlineDefs.COPY[from: oldframe, to: newframe, nwords: SIZE[global FrameBase]];
    gfti ← InitializeGlobalFrame[newframe, cp];
    -- initialize global proc vars
    procvar ← LOOPHOLE[newframe+cp.linkbase];
    InlineDefs.COPY[from: oldframe+cp.linkbase, to: procvar, nwords: cp.nlinks];
    THROUGH [0..cp.nlinks) DO
      IF procvar.gftindex IN [oldgfti .. oldgfti+cp.ngfi)
        THEN procvar.gftindex ← procvar.gftindex-oldgfti+gfti;
```

```
      procvar ← procvar+1;
      ENDLOOP;
    FrameDefs.UnlockCode[oldframe];
    RETURN
    END;

  AllocGlobalFrame: PROCEDURE [cseg: SegmentDefs.FileSegmentHandle] RETURNS
    [frame: GlobalFrameHandle, size: CARDINAL, cp: POINTER TO CsegPrefix] =
    BEGIN OPEN SegmentDefs;
    codebase: POINTER;
    cp ← codebase ← FileSegmentAddress[cseg];
    size ← cp.EntryVector[MainBodyIndex].framesize;
    IF size = maxallocslot
      THEN size ← (codebase+cp.EntryVector[MainBodyIndex].initialpc-1)↑;
    frame ← ControlDefs.Alloc[size];
    RETURN
    END;

  InitializeGlobalFrame: PROCEDURE [frame: GlobalFrameHandle, cp: POINTER TO CsegPrefix]
    RETURNS [gfti: GFTIndex] =
    BEGIN
    frame.accesslink ← frame;
    (frame+globalbase).accesslink ← NULLFrame;
    frame.pc ← ControlDefs.WordPC[0];
    frame.returnlink ← ControlLink[frame[NULLFrame]];
    frame.codebase ← LOOPHOLE[1];
    gfti ← FrameDefs.EnterGlobalFrame[frame, cp.ngfi];
    frame.gftindex ← ProcDesc[gftindex: gfti, epoffset: 0, tag: 0];
    frame.ownerlink ← NULLFrame;
    frame.bindentry ← frame;
    RETURN
    END;

  UnNew: PROCEDURE [frame: GlobalFrameHandle, freeframe: BOOLEAN] =
    BEGIN
    alone: BOOLEAN ← TRUE;
    cseg: SegmentDefs.FileSegmentHandle ← frame.codesegment;
    sseg: SegmentDefs.FileSegmentHandle ← frame.symbolsegment;
    RemoveAllTraces: PROCEDURE [f: GlobalFrameHandle] RETURNS [BOOLEAN] =
      BEGIN
      othercseg: SegmentDefs.FileSegmentHandle ← f.codesegment;
      IF f#frame THEN
        BEGIN
        IF cseg=othercseg THEN alone ← FALSE;
        IF f.ownerlink = frame THEN f.ownerlink ← frame.ownerlink;
        IF f.bindentry = frame THEN f.bindentry ← frame.bindlink;
        IF f.bindlink = frame THEN f.bindlink ← frame.bindlink;
        IF (f+globalbase).accesslink = frame THEN
          (f+globalbase).accesslink ← NULLFrame;
        END;
      RETURN[FALSE];
      END;
    [] ← FrameDefs.EnumerateGlobalFrames[RemoveAllTraces];
    IF alone THEN
      BEGIN OPEN SegmentDefs;
      DeleteFileSegment[cseg ! SwapError => CONTINUE];
      IF sseg#NIL THEN DeleteFileSegment[sseg ! SwapError => CONTINUE];
      END;
    FrameDefs.RemoveGlobalFrame[frame];
    IF freeframe THEN ControlDefs.Free[frame];
    END;

  -- data shuffling

  SetBlock: PUBLIC PROCEDURE [p:POINTER, v:UNSPECIFIED, l:CARDINAL] =
    BEGIN
    IF l=0 THEN RETURN;  p↑ ← v;
    InlineDefs.COPY[from:p, to:p+1, nwords:l-1];
    END;

  READ: PUBLIC PROCEDURE [a: UNSPECIFIED] RETURNS [UNSPECIFIED] =
    BEGIN RETURN [MEMORY[a]] END;

  WRITE: PUBLIC PROCEDURE [a, v: UNSPECIFIED]=
    BEGIN MEMORY[a] ← v; RETURN END;
```

```
  StringInit: PROCEDURE [coffset, n, offset: CARDINAL] =
    BEGIN OPEN ControlDefs;
    l: FrameHandle = GetReturnFrame[];
    g: GlobalFrameHandle = l.accesslink;
    p: POINTER = l + offset;
    i: CARDINAL;
    FrameDefs.LockCode[g];
    InlineDefs.COPY [
      from:g.codebase+coffset, to:p, nwords:n];
    FOR i IN [0..n) DO
      (p+i)↑ ← (p+i)↑ + g;
      ENDLOOP;
    FrameDefs.UnlockCode[g];
    RETURN
    END;

  -- procedure lists

  UserCleanupList: POINTER TO ImageDefs.CleanupItem ← NIL;

  AddCleanupProcedure: PUBLIC PROCEDURE [item: POINTER TO ImageDefs.CleanupItem] =
    BEGIN
    ProcessDefs.DisableInterrupts[];
    RemoveCleanupProcedure[item];
    item.link ← UserCleanupList;
    UserCleanupList ← item;
    ProcessDefs.EnableInterrupts[];
    END;

  RemoveCleanupProcedure: PUBLIC PROCEDURE [item: POINTER TO ImageDefs.CleanupItem] =
    BEGIN
    prev, this: POINTER TO ImageDefs.CleanupItem;
    IF UserCleanupList = NIL THEN RETURN;
    ProcessDefs.DisableInterrupts[];
    prev ← this ← UserCleanupList;
    IF this = item THEN UserCleanupList ← this.link
    ELSE UNTIL (this ← this.link) = NIL DO
      IF this = item THEN
        BEGIN prev.link ← this.link; EXIT END;
      prev ← this;
      ENDLOOP;
    ProcessDefs.EnableInterrupts[];
    END;

  UserCleanupProc: PUBLIC ImageDefs.CleanupProcedure =
    BEGIN -- all interrupts off if why = finish or abort
    this, next: POINTER TO ImageDefs.CleanupItem;
    this ← UserCleanupList;
    UserCleanupList ← NIL;
    WHILE this # NIL DO
      next ← this.link;
      this.proc[why ! ANY => IF why <= Abort THEN CONTINUE];
      AddCleanupProcedure[this];
      this ← next;
      ENDLOOP;
    SELECT why FROM
      Finish => ImageDefs.StopMesa[];
      Abort => ImageDefs.AbortMesa[];
      ENDCASE;
    END;

  -- Image Version

  ImageVersion: PUBLIC PROCEDURE RETURNS [version: BcdDefs.VersionStamp] =
    BEGIN OPEN ControlDefs, SegmentDefs;
    sd: POINTER TO ARRAY [0..0) OF ControlLink = REGISTER[SDreg];
    imagefile: FileHandle ← FrameDefs.GlobalFrame[sd[sCsegSwappedOut]].codesegment.file;
    headerseg: FileSegmentHandle ← NewFileSegment[imagefile, 1, 1, Read];
    image: POINTER TO ImageDefs.ImageHeader;
    SwapIn[headerseg];
    image ← FileSegmentAddress[headerseg];
    IF image.versionident # ImageDefs.VersionID THEN ERROR;
    version ← image.version;
    Unlock[headerseg];
    DeleteFileSegment[headerseg];
    RETURN
```

```
    END;

  -- signed and mixed mode division

  DIVMOD: MACHINE CODE [n,d: CARDINAL] RETURNS [QR] = INLINE [Mopcodes.zDIV];
  LDIVMOD: MACHINE CODE [nlow,nhigh,d: CARDINAL] RETURNS [QR] = INLINE [Mopcodes.zLDIV];
  QR: TYPE = RECORD [q, r: INTEGER];
  PQR: TYPE = POINTER TO QR;

--   DivSU: PROCEDURE =
--     BEGIN
--     neg: BOOLEAN;
--     state: ControlDefs.StateVector;
--     p: PQR;
--     state ← STATE;
--     state.X ← ControlDefs.GetReturnLink[];
--     p ← @state.stk[state.stkptr-2];
--     IF neg ← (p.q < 0) THEN p.q ← -p.q;
--     p↑ ← DIVMOD[p.q,p.r];
--     IF neg THEN
--       BEGIN
--       p.q ← -p.q;
--       p.r ← -p.r;
--       END;
--     RETURN WITH state
--     END;

  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;

--   DivUS: PROCEDURE =
--     BEGIN
--     state: ControlDefs.StateVector;
--     p: PQR;
--     state ← STATE;
--     state.X ← ControlDefs.GetReturnLink[];
--     p ← @state.stk[state.stkptr-2];
--     LongSignDivide[numhigh: 0, pqr: p];
--     RETURN WITH state
--     END;

  DivSS: PROCEDURE =
    BEGIN
    state: ControlDefs.StateVector;
    p: PQR;
    state ← STATE;
    state.X ← ControlDefs.GetReturnLink[];
    p ← @state.stk[state.stkptr-2];
    LongSignDivide[numhigh: (IF p.q<0 THEN -1 ELSE 0), pqr: p];
    RETURN WITH state
    END;

  -- Unsigned Compare

  USC: PUBLIC PROCEDURE [a1, a2: WORD] RETURNS [INTEGER] =
    BEGIN
    RETURN [SELECT LOOPHOLE[a1, CARDINAL] FROM
      > LOOPHOLE[a2, CARDINAL] => 1,
      < LOOPHOLE[a2, CARDINAL] => -1,
      ENDCASE => 0];
    END;
```

```
  -- Get Network Number

wordsPerPup: INTEGER = 280;
Byte: TYPE = [0..255];

PupHeader:TYPE= MACHINE DEPENDENT RECORD [
   eDest, eSource: Byte,
   eWord2, pupLength: INTEGER,
   transportControl, pupType: Byte,
   pupID1, pupID2: INTEGER,
   destNet, destHost: Byte,
   destSocket1, destSocket2: INTEGER,
   sourceNet, sourceHost: Byte,
   sourceSocket1, sourceSocket2: INTEGER,
   xSum:CARDINAL];

Pup:TYPE= MACHINE DEPENDENT RECORD [
   head:PupHeader,
   junk:ARRAY [0..300] OF WORD];

EthernetDeviceBlock: TYPE = MACHINE DEPENDENT RECORD [
    EPLocMicrocodeStatus, EPLocHardwareStatus: Byte,
    EBLocInterruptBit: WORD,
    EELocInputFinishCount: INTEGER,
    ELLocCollisionMagic: WORD,
    EILocInputCount: INTEGER,
    EILocInputPointer: POINTER,
    EOLocOutputCount: INTEGER,
    EOLocOutputPointer: POINTER];

-- StartIO is Mesa bytecode used to control Ethernet interface
StartIO: MACHINE CODE [WORD] = INLINE [Mopcodes.zSTARTIO];
outputCommand: WORD = 1;
inputCommand: WORD = 2;
resetCommand: WORD = 3;

timer: POINTER TO INTEGER = LOOPHOLE[430B];

GetNetworkNumber: PUBLIC PROCEDURE RETURNS[CARDINAL] =
  BEGIN
  myHost: Byte ← OsStaticDefs.OsStatics.SerialNumber;
  then: INTEGER ← timer↑;
  now: INTEGER;
  device: POINTER TO EthernetDeviceBlock ← LOOPHOLE[600B];
  xpup: Pup;
  pup: POINTER TO Pup = @xpup;
  gatewayRequest:PupHeader ← [
     eDest: 0,                 eSource: myHost,
     eWord2: 1000B,            pupLength: 22,
     transportControl: 0,      pupType: 200B,
     pupID1:,                  pupID2:,
     destNet: 0,               destHost: 0,
     destSocket1: 0,           destSocket2: 2,
     sourceNet: 0,             sourceHost: myHost,
     sourceSocket1: 0,         sourceSocket2: 2,
     xSum: 177777B];
  device.EBLocInterruptBit ← 0;
  StartIO[resetCommand];
  device↑ ← EthernetDeviceBlock[
      EPLocMicrocodeStatus: 0,
      EPLocHardwareStatus: 0,
      EBLocInterruptBit: 0,
      EELocInputFinishCount: 0,
      ELLocCollisionMagic: 0,
      EILocInputCount: 0,
      EILocInputPointer: pup,
      EOLocOutputCount: 13,
      EOLocOutputPointer: @gatewayRequest];
  StartIO[outputCommand];
  THROUGH [0..2) DO
    DO
      IF device.EPLocHardwareStatus#0 THEN
        BEGIN
        IF device.EPLocMicrocodeStatus = 0
          AND pup.head.eWord2 = 1000B
          AND wordsPerPup+2-device.EELocInputFinishCount > 13
```

```
          AND pup.head.destSocket1 = 0
          AND pup.head.destSocket2 = 2
          AND pup.head.pupType = 201B
          THEN RETURN[pup.head.sourceNet];
        device↑ ← EthernetDeviceBlock[
            EPLocMicrocodeStatus: 0,
            EPLocHardwareStatus: 0,
            EBLocInterruptBit: 0 ,
            EELocInputFinishCount: 0,
            ELLocCollisionMagic: 0,
            EILocInputCount: wordsPerPup+2,
            EILocInputPointer: pup,
            EOLocOutputCount: 0,
            EOLocOutputPointer: NIL];
        StartIO[inputCommand];
        END;
      now ← timer↑;
      IF now-then > 14 THEN EXIT;
      ENDLOOP;
    ENDLOOP;
  RETURN[0];
  END;


  Init: PROCEDURE =
    BEGIN
    sd: POINTER TO ARRAY [0..0) OF UNSPECIFIED =
      REGISTER[SDreg];
    sd[sGoingAway] ← UserCleanupProc;
    sd[sUnbound] ← UnboundProcedureTrap;
    sd[sStackError] ← StackErrorTrap;
    sd[sControlFault] ← ControlFaultTrap;
    sd[sStringInit] ← StringInit;
    sd[sInPortInit] ← 0;
    sd[sOutPortInit] ← 0;
    sd[sDivSS] ← DivSS;
--    sd[sDivSU] ← DivSU;
--    sd[sDivUS] ← DivUS;
    sd[sCopy] ← Copy;
    sd[sUnNew] ← UnNew;
    END;


  -- Main Body;

  Init[];

  END...
```