

```
-- Fsp.Mesa Edited by Sandman on August 31, 1977 1:40 PM
```

```
DIRECTORY
```

```
AltoDefs: FROM "altodefs",
SystemDefs: FROM "systemdefs",
StringDefs: FROM "stringdefs",
FSPDefs: FROM "fspdefs";
```

```
DEFINITIONS FROM FSPDefs;
```

```
FSP: PROGRAM
```

```
IMPORTS StringDefs, SystemDefs EXPORTS FSPDefs, SystemDefs SHARES FSPDefs = PUBLIC
BEGIN -- Mesa Free Storage Package --
```

```
-- A set of procedures to manage allocation within a zone.
-- Coalescing of free nodes occurs during allocation; all
-- free nodes following a candidate node are merged before
-- any space is allocated. The logic is derived from a
-- BCPL program by E. M. McCreight and was suggested by an
-- exercise in Knuth Volume I, p. 453 #19
```

```
UsedNodeSize: PRIVATE BlockSize = SIZE [inuse NodeHeader];
FreeNodeSize: PRIVATE BlockSize = SIZE [free NodeHeader];
ZoneHeaderSize: PRIVATE BlockSize = SIZE [ZoneHeader];
```

```
ZoneTooSmall: ERROR [POINTER] = CODE;
```

```
DoNothingDeallocate: Deallocator = BEGIN NULL END;
```

```
MakeZone: PROCEDURE [base: POINTER, length: BlockSize] RETURNS [z: ZonePointer] =
BEGIN
RETURN[MakeNewZone[base, length, DoNothingDeallocate]];
END;
```

```
MakeNewZone: PROCEDURE [base: POINTER, length: BlockSize, deallocate: Deallocator]
RETURNS [z: ZonePointer] =
BEGIN
fn: FreeNodePointer;
an: NodePointer;
IF length < ZoneHeaderSize+FreeNodeSize+UsedNodeSize
THEN ERROR ZoneTooSmall[base];
z ← base;
-- set up the bulk of the zone as a large free block.
fn ← base + ZoneHeaderSize;
fn↑ ← NodeHeader[length-(ZoneHeaderSize+UsedNodeSize), free[@z.node, @z.node]];
-- set up an allocated node (the smallest possible) at the end of the block.
an ← base + (length-UsedNodeSize);
an↑ ← NodeHeader[UsedNodeSize, inuse[]];
-- set up the zone header
z↑ ← ZoneHeader[
NodeHeader[0, free[fn, fn]], fn, NIL, length, deallocate, FreeNodeSize, FALSE];
RETURN
END;
```

```
AddToZone: PROCEDURE [z: ZonePointer, base: POINTER, length: BlockSize] =
BEGIN
AddToNewZone[z, base, length, DoNothingDeallocate];
END;
```

```
AddToNewZone: PROCEDURE [z: ZonePointer, base: POINTER, length: BlockSize, deallocate: Deallocator] =
**
BEGIN
newz: ZonePointer;
firstnew, lastnew: FreeNodePointer;
ValidateZone[z];
newz ← MakeNewZone[base, length, deallocate];
-- splice the zones together
firstnew ← newz.node.fwdp; lastnew ← newz.node.backp;
z.node.backp.fwdp ← firstnew;
firstnew.backp ← z.node.backp;
lastnew.fwdp ← @z.node;
z.node.backp ← lastnew;
-- make newz head an empty list
newz.node.fwdp ← newz.node.backp ← @newz.node;
-- link newz with restofzones in z
newz.restofzone ← z.restofzone;
```

```

    z.restofzone ← newz;
    IF z.checking THEN CheckZone[z];
    RETURN
    END;

InvalidZone: ERROR [POINTER] = CODE; -- zone header looks fishy

ValidateZone: PRIVATE PROCEDURE [z: ZonePointer] =
    BEGIN
    SELECT TRUE FROM
        (z.node.length # 0),
        (LOOPHOLE[z + (z.length-UsedNodeSize),NodePointer].length # UsedNodeSize),
        (z.node.fwdp.backp # @z.node OR z.node.backp.fwdp # @z.node) =>
        ERROR InvalidZone[z];
    ENDCASE;
    RETURN
    END;

NodeLoop: ERROR [ZonePointer] = CODE;

CheckZone: PRIVATE PROCEDURE [z: ZonePointer] =
    BEGIN
    node: FreeNodePointer;
    count: INTEGER;
    ValidateZone[z];
    count ← (LAST[BlockSize]-FIRST[BlockSize])/FreeNodeSize + 1;
    node ← @z.node;
    DO
        CheckNode[z, node];
        IF (count ← count-1) < 0 THEN ERROR NodeLoop[z];
        IF (node ← node.fwdp) = @z.node THEN EXIT;
    ENDLOOP;
    RETURN
    END;

InvalidNode: ERROR [POINTER] = CODE; -- node appears damaged

CheckNode: PRIVATE PROCEDURE [z: ZonePointer, node: NodePointer] =
    BEGIN
    DO
    WITH node SELECT FROM
        inuse =>
            IF length = UsedNodeSize THEN EXIT; -- end of zone
        free =>
            BEGIN
            IF fwdp.backp # node OR backp.fwdp # node
                THEN GO TO error;
            IF length=0 AND node # @z.node THEN GO TO error;
            END;
        ENDCASE;
    node ← node + node.length;
    IF node.state # inuse THEN EXIT;
    REPEAT
        error =>
            BEGIN z.checking ← FALSE;
            ERROR InvalidNode[node+UsedNodeSize];
            END;
    ENDLOOP;
    RETURN
    END;

NoRoomInZone: SIGNAL [ZonePointer] = CODE; -- not enough space to fill a request

MakeNode: PROCEDURE [z: ZonePointer, n: BlockSize] RETURNS [POINTER] =
    BEGIN
    rover: FreeNodePointer;
    node, neighbour: NodePointer;
    nodelength, n1, t: BlockSize;
    IF z.checking THEN CheckZone[z];
    n ← MAX[n+UsedNodeSize, FreeNodeSize];
    DO
        rover ← z.rover;
        DO
            nodelength ← rover.length;
            FOR neighbour ← rover+nodelength, neighbour+n1
            DO

```

```

WITH neighbour SELECT FROM
  inuse =>
    EXIT;
  free =>
    BEGIN -- coalesce
      IF (n1 ← length) = 0 THEN EXIT; -- end of zone
      fwdp.backp ← backp; backp.fwdp ← fwdp;
      z.rover ← rover; -- in case neighbor was z.rover
      nodelength ← nodelength+n1;
    END;
  ENDCASE;
ENDLOOP;
IF nodelength ≥ n
  THEN
    BEGIN
      IF (t ← (nodelength-n)) > MAX[FreeNodeSize, z.threshold]
        THEN
          BEGIN -- split the block
            z.rover ← rover; rover.length ← t;
            node ← rover+t; nodelength ← n;
          END
        ELSE
          BEGIN
            rover.fwdp.backp ← rover.backp;
            z.rover ← rover.backp.fwdp ← rover.fwdp;
            node ← rover;
          END;
          node↑ ← NodeHeader[nodelength, inuse[]];
          RETURN [node + UsedNodeSize]
        END
      ELSE rover.length ← nodelength;
      IF (rover ← rover.fwdp) = z.rover THEN EXIT;
    ENDLOOP;
    SIGNAL NoRoomInZone[z];
  ENDLOOP; -- try again if RESUMEd from the signal
END;

FreeNode: PROCEDURE [z: ZonePointer, p: POINTER] =
  BEGIN
    node: NodePointer = p-UsedNodeSize;
    IF z.checking THEN CheckZone[z];
    WITH node SELECT FROM
      free =>
        ERROR InvalidNode[p];
      ENDCASE =>
        node↑ ← NodeHeader[node.length, free[@z.node, z.node.backp]];
    WITH n:node SELECT FROM
      free =>
        z.node.backp ← n.backp.fwdp ← @n;
      ENDCASE;
    RETURN
  END;

SplitNode: PROCEDURE [z: ZonePointer, p: POINTER, n: BlockSize] =
  BEGIN
    node: NodePointer = p-UsedNodeSize;
    lastpart: NodePointer;
    t: INTEGER;
    IF z.checking THEN CheckZone[z];
    n ← n+UsedNodeSize;
    WITH node SELECT FROM
      free =>
        ERROR InvalidNode[p];
      ENDCASE =>
        IF (t ← node.length - n) ≥ MAX[FreeNodeSize, z.threshold]
          THEN
            BEGIN
              lastpart ← node+n;
              lastpart↑ ← NodeHeader[t, inuse[]];
              FreeNode[z, lastpart+UsedNodeSize];
            END;
          RETURN
        END;

NodeSize: PROCEDURE [p: POINTER] RETURNS [BlockSize] =
  BEGIN

```

```

node: NodePointer = p-UsedNodeSize;
WITH node SELECT FROM
  free =>
    ERROR InvalidNode[p];
  ENDCASE =>
    RETURN [length-UsedNodeSize];
END;

PruneZone: PROCEDURE [z: ZonePointer] RETURNS [BOOLEAN] =
BEGIN
  didit: BOOLEAN ← FALSE;
  rest: ZonePointer;
  headzone, prev: ZonePointer ← z;
  node: NodePointer;
  n1: BlockSize;
  FOR z ← z.restofzone, rest UNTIL z = NIL DO
    rest ← z.restofzone;
    FOR node ← LOOPHOLE[z+ZoneHeaderSize, NodePointer], node+n1 DO
      WITH node SELECT FROM
        inuse =>
          BEGIN
            IF length = UsedNodeSize THEN
              BEGIN -- end of zone
                FreeZone[z];
                didit ← TRUE;
                prev.restofzone ← rest;
              END
            ELSE prev ← z;
            EXIT
          END;
        free => n1 ← length;
      ENDCASE;
    ENDOLOOP;
  ENDOLOOP;
  headzone.rover ← headzone.node.fwdp; -- reset rover incase it in a zone that was freed
  RETURN[didit];
END;

FreeZone: PROCEDURE [zone: ZonePointer] =
BEGIN
  node: NodePointer;
  n1: BlockSize;
  FOR node ← LOOPHOLE[zone+ZoneHeaderSize, NodePointer], node+n1 DO
    WITH node SELECT FROM
      inuse => EXIT; -- end of zone
      free =>
        BEGIN
          n1 ← length;
          backp.fwdp ← fwdp;
          fwdp.backp ← backp;
        END;
      ENDCASE;
    ENDOLOOP;
  zone.deallocate[zone];
END;

DestroyZone: PROCEDURE [z: ZonePointer] =
BEGIN
  rest: ZonePointer;
  IF z = TheHeap THEN RETURN;
  FOR z ← z, rest UNTIL z = NIL DO
    rest ← z.restofzone;
    z.deallocate[z];
  ENDOLOOP;
END;

-- management of the heap

TheHeap: PRIVATE ZonePointer;

HeapZone: PROCEDURE RETURNS [ZonePointer] =
BEGIN
  RETURN [TheHeap]
END;

```

```
AllocateHeapNode: PROCEDURE [nwords: CARDINAL] RETURNS [p: POINTER] =
  BEGIN OPEN SystemDefs;
  np: CARDINAL;
  p ← MakeNode[TheHeap, nwords
    !NoRoomInZone =>
    BEGIN
      np ← PagesForWords[nwords + ZoneOverhead + UsedNodeSize];
      AddToNewZone[TheHeap, AllocatePages[np], np*AltoDefs.PageSize, FreePages];
      RESUME
    END];
  RETURN
  END;

FreeHeapNode: PROCEDURE [p: POINTER] =
  BEGIN
  FreeNode[TheHeap, p]; RETURN
  END;

AllocateHeapString: PROCEDURE [nchars: CARDINAL] RETURNS [STRING] =
  BEGIN
  OPEN StringDefs;
  p: POINTER TO MACHINE DEPENDENT RECORD[      -- faked string header
    length: CARDINAL,
    maxlength: CARDINAL];
  p ← AllocateHeapNode[WordsForString[nchars]];
  p.length ← 0; p.maxlength ← nchars;
  RETURN [LOOPHOLE[p, STRING]]
  END;

FreeHeapString: PROCEDURE [s: STRING] =
  BEGIN
  FreeNode[TheHeap, s]; RETURN
  END;

PruneHeap: PROCEDURE RETURNS [BOOLEAN] =
  BEGIN
  RETURN[PruneZone[TheHeap]]
  END;

-- initialization code

TheHeap ←
  MakeNewZone[SystemDefs.AllocatePages[1], 1*AltoDefs.PageSize, SystemDefs.FreePages];

END.
```