```
*****************************************************************************
*                                                                           *
*                          Memory Module                                    *
*  This module handles all of the memory management functions except        *
*    Garbage collection.                                                    *
*                                                                           *
*****************************************************************************
        name    Memory

#include "../include/defOS.i"
#include "../include/parameters.i"
#include "../include/traceCalls.i"
#include "../include/exceptions.i"
#include "../include/registerDefinitions.i"

        lib     objdefs/MemoryStructs.def
        lib     objdefs/MemoryVars.def
        lib     objdefs/GCVars.def
        if      UniFlex
        lib     sysdef
        endif


*       freeSearch                                              */
*  This routine searches a table of free areas and              */
* returns the first which is large enough to satisfy            */
* the request. If none is found an attempt to sbrk              */
* more memory is made. Tables are used instead of links         */
* so we dont thrash V-mem searching. This routine is            */
* passed the table parameters which are updated. This           */
* implies call by name which C doesn't really do. Zero          */
* lenght hunks are left in the table but may be later           */
* removed by free. The hunks in the table are in order          */
* by memory location.                                           */
* converted into a macro to improve effiency in asmbly         */
* The parameters are passed in regs, fail return is to          */
* drop out the bottom, suscess goto macro parm.                 */
*
* freeSearch( bp, cp, fc)
*     register freemem *bp ;
*     register freeblk *cp ;
*     register int      fc ;
*     {
*
*     forever
*       {
*       while ( fc )
*           {
*           if ( cp->freesize >= size )
*               {
*               val = cp->freeaddr ;
*               cp->freeaddr += size ;
*               cp->freesize -= size ;
*               return ( val ) ;
*               }
```

```
*           ++cp ;
*           --fc ;
*           }
*       bp = bp->freenxt ;
*       if ( bp == NULL ) return ( NULL ) ;
*       cp = bp->freetbl ;
*       fc = bp->freecnt ;
*       }
*     }

#define freeSearchM(success) \
11      tst.w   fc                      ; check for end of current block. \
        bmi.s   13f                     ; End of current block ? \
12      cmp.l   freesize(cp),size       ; is this hunk big enough ? \
        ble.s   14f                     ; yes, go get it. \
        add.l   #sizeof_freeblk,cp      ; skip to next hunk.\
        dbra    fc,12b                  ; go look at it.\
13      move.l  freenxt(bp),bp          ; get next free block \
        move.l  bp,tmp1                 ; test it. \
        beq.s   15f                     ; if NULL fail \
        lea     freetbl(bp),cp          ; get pointer to first free hunk.\
        move.w  freecnt(bp),fc          ; number of entries in block to fc \
        bra.s   11b                     ; go search it.\
14      move.l  freeaddr(cp),val        ; get addr of hunk we found.\
        add.l   size,val                ; get end of part we used.\
        move.l  val,freeaddr(cp)        ; store it back.\
        sub.l   size,val                ; back up pointer to front.\
        sub.l   size,freesize(cp)       ; reduce size by amount we took.\
        bra.s   success                 ; and return value.\
*                                       ; failure drop out.\
15      ds      0


*       alloc                                           */
*  This routine allocates a chunk of memory             */
* from the bulk memory gotten from the system           */
* by sbrk. It has two areas it works from.              */
* one is for relatively small blocks which are          */
* for indirect because of becomes. They are             */
* short lived since the next scavenge of their          */
* grade will merge the data back with the base          */
* object. The other is for data which is really*/
* large. The size is rounded to the nearest             */
* long                                                  */
*
* alloc(size)
*     int size ;
*     {
*     static int      initialize = TRUE ;
*     static freemem *bbp, *sbp ;             /* block pointers       */
*     static freeblk *bcp, *scp ;             /* current entry pointer*/
*     static int      bfc,  sfc ;             /* free count           */
*     char *val ;
*
*
*     if ( initialize )
```

```
*         {
*       bbp = bigfree ;
*       bcp = bigfree->freetbl ;
*       bfc = bigfree->freecnt ;
*       sbp = smallfree ;
*       scp = smallfree->freetbl ;
*       sfc = smallfree->freecnt ;
*       initialize = FALSE ;
*       }
*
*     size = size + 3 & ~3 ;
*     if ( size > SMALLSIZE )
*       {
*         val = freeSearch (bbp, bcp, bfc ) ;
*       if ( val == NULL )
*           {
*           bbp = bigfree ;
*           bcp = bigfree->freetbl ;
*           bfc = bigfree->freecnt ;
*           val = freeSearch (bbp, bcp, bfc ) ;
*           }
*       }
*     else
*       {
*         val = freeSearch (sbp, scp, sfc ) ;
*       if ( val == NULL )
*           {
*           sbp = smallfree ;
*           scp = smallfree->freetbl ;
*           sfc = smallfree->freecnt ;
*           val = freeSearch (sbp, scp, sfc ) ;
*           }
*       }
*
*     if ( val == NULL )
*       {
*       /* should compact memory here */
*       return (NULL) ;
*       }
*     return ( val ) ;
*     }

#define size d0
#define val a0
#define bp a1
#define cp a2
#define fc d1
#define tmp1 d2
#define AMregs a1/a2/d0/d1/d2

        global  alloc
alloc
        movem.l AMregs,-(sp)            ; save the working regs.
*       initialization is done in another routine in this module.
        add.l   #3,size                 ; round parm up to nearest
        and.l   #!3,size                ; long word.
```

```
        if      DEBUG
        bne.s   90f                     ; test for zero length.
        errtrap("Zero len. Alloc")      ; why.
90      ds.w    0                       ; keep on truckin.
        endif
        cmp.l   #SMALLSIZE,size         ; is it suppose to come
        blt     4f                      ; out of the "small" area.
        bclr    #0,bflg                 ; have frees been done ?
        bne.s   1f                      ; yes, start at beginning.
        move.l  bbp:w,bp                ; load the parameters
        move.l  bcp:w,cp                ; to freesearch from the
        move.w  bfc:w,fc                ; big area parms.
        freeSearchM(3f)                 ; go find it boy.
1       move.l  bigfree:w,bp            ; noop, no luck try again from
        lea     freetbl(bp),cp          ; from the begining.
        move.w  freecnt(bp),fc          ;
        freeSearchM(3f)                 ; now find me some memory.
*       Not enough on free list try to sbrk some more memory.
        if      UniFlex
        move.l  memend:w,val            ; get the current top.
        add.l   size,val                ; where this chunk will end.
        cmp.l   memlimit:w,val          ; will it fit ?
        bgt.s   2f                      ; no, return null.
        move.l  val,memend:w            ; bump pointer to end.
        sub.l   size,val                ; get back beginning.
        endif
        if      UTek
        move.l  size,-(sp)              ; set up parm
        jsr     _sbrk                   ; call for memory
        add.l   #4,sp                   ; pop parameter
        move.l  d0,val                  ; move in result
        endif
        movem.l (sp)+,AMregs            ; restore regs.
        rts                             ; and quit.
*       Failure couldn't get enough memory anywhere.
2       move.l  #0,val                  ; return NULL
        movem.l (sp)+,AMregs            ; restore regs.
        rts                             ; to caller.
*       Success for chunk from big space, save where it was found.
3       move.l  bp,bbp:w                ; save block
        move.l  cp,bcp:w                ; place in block
        move.w  fc,bfc:w                ; and whats left in block.
        movem.l (sp)+,AMregs            ; restore regs.
        rts                             ; to caller.
4       bclr    #0,sflg                 ; doe a free sinc last call ?
        bne.s   24f                     ; yes, start at beginning.
        move.l  sbp:w,bp                ; load the parameters
        move.l  scp:w,cp                ; to freesearch from the
        move.w  sfcx:w,fc               ; small area parms.
        freeSearchM(5f)                 ; go find it boy.
24      move.l  smallfree:w,bp          ; noop, no luck try again from
        lea     freetbl(bp),cp          ; from the begining.
        move.w  freecnt(bp),fc          ;
        freeSearchM(5f)                 ; now find me some memory.
        bra     1b                      ; no, try in the big space.
*       Success found chunk in small space save where we found it.
```

```
5       move.l  bp,sbp:w                ; save block
        move.l  cp,scp:w                ; place in block
        move.w  fc,sfcx:w               ; and whats left in block.
        movem.l (sp)+,AMregs            ; restore regs.
        rts                             ; to caller.

#undef size
#undef val
#undef bp
#undef cp
#undef fc
#undef tmpl
#undef AMregs

*       freeMem                                         */
*  This routine adds the memory passed of size          */
* passed to the free list. It rounds the size           */
* up to 4 bytes. The memory address is used to          */
* decide if the area is in the small or big             */
* hunks. It collects adjacent frees together            */
* when it can. If it cant it moves all the              */
* following hunks up as far as necessary.               */
*
* freeMem(mem, size)
*     char *mem ;
*     int   size ;
*     {
*     freemem *bp, *lbp ;
*     register freeblk *cp ;
*     char *tmpaddr, *tmpend, swapaddr ;
*     int tmpsize, swapsize ;
*
*     size = size + 3 & ~3 ;
*     tmpaddr = mem ;
*     tmpsize = size ;
*     tmpend = mem + size ;
*     if ( mem > smallarea &&
*        mem < smallAreaEnd )
*       {
*       bp = smallfree ;
*       }
*     else
*       {
*       bp = bigfree ;
*       }
*
*     while ( bp != NULL )
*       {
*       cp = bp->freetbl ;
*       fc = bp->freecnt ;
*       while ( fc )
*           {
*           if ( cp->freeaddr == tmpend )
*               { /* this hunk is the front of another free hunk */
*               cp->freeaddr = tmpaddr ;
*               cp->freesize += tmpsize ;
```

```
*               return ;
*               }
*           if ( cp->freeaddr + cp->freesize == tmpaddr )
*               { /* this hunk follows a free hunk */
*               cp->freesize += tmpsize ;
*               return ;
*               }
*           if ( cp->freeaddr < tmpaddr )
*               { /* this hunk is below the current hunk. */
*               break ;
*               }
*           --fc ;
*           }
*       while ( fc )
*           {
*           swapaddr = cp->freeaddr ;
*           swapsize = cp->freesize ;
*           cp->freeaddr = tmpaddr  ;
*           cp->freesize = tmpsize  ;
*           if ( swapsize == 0 ) return ; /* found a zero length hunk   */
*                                         /* just delete it             */
*           tmpaddr = swapaddr ;
*           tmpsize = swapsize ;
*           --fc ;
*           }
*       }
*       lbp = bp ;
*       bp = bp->freenxt ;
*       }
*     if ( lbp->freecnt < freelast )
*       {
*       lbp->freecnt += 1 ;
*       lbp->freetbl[lbp->freecnt].freeaddr = tmpaddr ;
*       lbp->freetbl[lbp->freecnt].freesize = tmpsize ;
*       }
*     else
*       {
*       bp = alloc(sizeof(freeblk)) ;
*       if ( bp == NULL )
*           {
*           bp = tmpaddr ;
*           tmpaddr  += sizeof(freemem) ;
*           freesize -= sizeof(freemem) ;
*           }
*       bp->freecnt = 1 ;
*       bp->freetbl.freeaddr = tmpaddr ;
*       bp->freetbl.freesize = tmpsize ;
*         lbp->freenxt = bp ;
*       }
*     return ;
*     }

#define size d1
#define swapsize d1
#define tmpsize d2
#define tmpend d3
```

```
#define fc d4
#define tmp0 d0
#define tmp1 d0
#define tmp2 d0
#define tmp3 d0
#define tmp4 d0
#define mem a1
#define swapaddr a1
#define tmpaddr a2
#define bp a0
#define lbp a3
#define ncp a3
#define cp a4

#define FMregs d0/d1/d2/d3/d4/a0/a1/a2/a3/a4

        global  freeMem
freeMem
        movem.l FMregs,-(sp)            ; save working regs.
        add.l   #3,size                 ; round size up to 4
        and.l   #!3,size                ;
        if      DEBUG
        bne.s   90f                     ; check free size.
        errtrap("zero length free.")    ; if zero bitch
90      ds.w    0                       ; next instr.
        endif
        move.l  mem,tmpaddr             ; copy the memory address,
        move.l  size,tmpsize            ; block size
        move.l  mem,tmpend              ; and block end address
        add.l   size,tmpend             ; to scratch registers.
        cmp.l   smallAreaEnd:w,mem      ; check beginning and end.
        blt.s   8f                      ; yes, use small area
1       move.l  bigfree:w,bp            ; get the big area free list.
        move.b  #-1,bflg:w              ; show a free has been done
2       lea     freetbl:w(bp),cp        ; get address of first discripter.
        move.w  freecnt(bp),fc          ; get number of entries in this block
        bmi.s   4f                      ; no entries dont search.
3       move.l  freeaddr(cp),tmp0       ; get a free mem address.
        if      DEBUG
        cmp.l   tmpend,tmp0             ; is it all  below the free entry ?
        bge.s   91f                     ; yes then don't bitch.
        add.l   freesize(cp),tmp0       ; how about all above ?
        cmp.l   tmpaddr,tmp0            ;
        ble.s   91f                     ; yse, then is A-OK.
        tst.l   freesize(cp)            ; ok to overlap zero length free
        beq.s   91f                     ;
        errtrap("overlapping frees.")   ; oops, somebody blew it allen.
91      ds.w    0
        endif
        move.l  freeaddr(cp),tmp0       ; get a free mem address.
        cmp.l   tmpend,tmp0             ; is it at end of new hunk ?
        beq.s   5f                      ; yes, hook em together.
        add.l   freesize(cp),tmp0       ; calc end of free entry.
        cmp.l   tmpaddr,tmp0            ; is this hunk at end of entry ?
        beq.s   6f                      ; yes, hook em together.
        bgt.s   10f                     ; are we past the entry. ?
```

```
        add.l   #sizeof_freeblk,cp      ; bump entry pointer
        dbra    fc,3b                   ; next entry ?
4       move.l  bp,lbp                  ; save block address.
        move.l  freenxt(bp),bp          ; get next block.
        move.l  bp,tmp0                 ; test entry.
        bne.s   2b                      ; loop back.
        bra.s   12f                     ; else store at end.

5       move.l  tmpaddr,freeaddr(cp)    ; update entry beginning.
        add.l   tmpsize,freesize(cp)    ; update entry size.
        movem.l (sp)+,FMregs            ; restore registers.
        rts                             ; and home again higady-jig.

8       move.l  smallfree:w,bp          ; get small area free list.
        move.b  #-1,sflg:w              ; show a free has been done
        bra.s   2b                      ; to play with.

6       add.l   freesize(cp),tmpsize    ; update entry size.
        move.l  tmpsize,freesize(cp)    ; and put it back.
        move.l  freeaddr(cp),tmpend     ; calulate the new end of block.
        add.l   tmpsize,tmpend          ;
        move.l  cp,ncp                  ; where to start looking for next block
16      sub.w   #1,fc                   ; if we're not to the end.
        bmi.s   7f                      ;
        add.l   #sizeof_freeblk,ncp     ; bump the pointer.
        move.l  freesize(ncp),tmp4      ; get it's size.
        beq.s   16b                     ; if unused keep skipping.
        cmp.l   freeaddr(ncp),tmpend    ; do they link up?
        bne.s   7f                      ; no, quit.
        clr.l   freesize(ncp)           ; yes, make this one unused.
        add.l   tmp4,freesize(cp)       ; and add it's length to previous block
7       movem.l (sp)+,FMregs            ; restore registers.
        rts                             ; and home again higady-jig.

9       lea     freetbl(bp),cp          ; get address of first discripter.
        move.w  freecnt(bp),fc          ; get number of entries in this block
        bmi.s   11f                     ; no entries dont search.
10      move.l  freeaddr(cp),swapaddr   ; copy out entry stuff
        move.l  freesize(cp),swapsize   ;
        move.l  tmpaddr,freeaddr(cp)    ; and swap in the new entry.
        move.l  tmpsize,freesize(cp)    ;
        move.l  swapsize,tmpsize        ; now swap the old entry over.
        beq.s   7b                      ; if its not zero length.
        move.l  swapaddr,tmpaddr        ;
        add.l   #sizeof_freeblk,cp      ; bump entry pointer.
        dbra    fc,10b                  ; and go back for more.
11      move.l  bp,lbp                  ; save this block address.
        move.l  freenxt(bp),bp          ; get next free block.
        move.l  bp,tmp0                 ; test it.
        bne.s   9b                      ; if not NULL bubble up this one.

12      move.w  freecnt(lbp),tmp1       ; is the last block full
        cmp.w   #freelast-1,tmp1        ; compare with max count.
        bge.s   13f                     ; yes, allocate a new one.
        add.w   #1,tmp1                 ; bump count
        move.w  tmp1,freecnt(lbp)       ; and store it.
```

```
        move.l  tmpaddr,(freetbl+freeaddr,lbp,tmp1*sizeof_freeblk)
        move.l  tmpsize,(freetbl+freesize,lbp,tmp1*sizeof_freeblk)
        movem.l (sp)+,FMregs            ; restore regs
        rts                             ; and return.

13      move.l  #sizeof_freemem,d0      ; parm to
        jsr     alloc                   ; get another free block.
        move.l  bp,tmp2                 ; test return
        bne.s   14f                     ; O.K. we're done.
        move.l  tmpaddr,bp              ; try allocating it in the
        move.l  #sizeof_freemem,tmp3    ; space just freed.
        sub.l   tmp3,tmpsize            ; take size off freed size.
        bmi.s   15f                     ; oops wouldn't fit.
        add.l   tmp3,tmpaddr            ; bump start address.
14      move.l  bp,freenxt(lbp)         ; store address of this block.
        clr.l   freenxt(bp)             ; Null to new blocks successor.
        clr.w   freecnt(bp)             ; One to new blocks count. (-1 drba)
        move.l  tmpaddr,freetbl+freeaddr(bp) ; put in the new entry.
        move.l  tmpsize,freetbl+freesize(bp)
        movem.l (sp)+,FMregs            ; restore regs
        rts                             ; and return.

15      exception (No_Big_Memory)
        movem.l (sp)+,FMregs            ; restore regs
        rts                             ; and return.


#undef size
#undef swapsize
#undef tmpsize
#undef tmpend
#undef fc
#undef tmp0
#undef tmp1
#undef tmp2
#undef tmp3
#undef tmp4
#undef mem
#undef swapaddr
#undef tmpaddr
#undef bp
#undef lbp
#undef cp
#undef ncp
#undef FMregs


*       packFree                        */
*   A routine to pack active elements   */
* of the free list into The first few   */
* free blocks. It removes zero length   */
* free entries and merges any adjacent  */
* entries which were in different free  */
* blocks. (this should only occur at    */
* the first entry.)                     */
```

```
*packFree(freeList)
*    freemem *freeList ;
*    {
*    freemem *storeblk ;
*    freeblk *store ;
*    int      storecnt ;
*    freemem *fromblk ;
*    freeblk *from ;
*    int      fromcnt ;
*    freemem *delete, *nxtdelete ;
*
*    if ( freeList == NULL ) return ;
*    storeblk = freeList ;
*    storecnt = 0 ;
*    store    = &storeblk->freetbl[-1] ;
*    fromblk = freelist ;
*    fromcnt = freeblk->freecnt ;
*    from    = fromblk->freetbl ;
*    while ( true )
*        {
*       for ( ; fromcnt > 0 ; --fromcnt )
*           {
*           if ( ++from->freesize == 0 ) continue ;
*           ++storecnt ;
*           ++store ;
*           if ( storecnt > freelast )
*               {
*               storeblk->freecnt = freelast ;
*               storeblk = storeblk->freenxt ;
*               storecnt = 1 ;
*               store = storeblk->freetbl ;
*               }
*           if ( store != free )
*               {
*               store->freeaddr = from->freeaddr ;
*               store->freesize = from->freesize ;
*               }
*           ++from ;
*           }
*       fromblk = fromblk->freenxt ;
*       if ( fromblk == NULL ) break ;
*       from = fromblk->freetbl ;
*       fromcnt = from->freecnt ;
*       if ( store->freeaddr + store+freesize == from->freeaddr )
*           {
*           store->freesize += from->freesize ;
*           ++from ;
*           --fromcnt ;
*           }
*       }
*    storeblk->freecnt = storecnt ;
*    delete = storeblk->freenxt ;
*    storeblk->freenxt = NULL ;
*    while ( delete != NULL )
*        {
*       nxtdelete = delete->freenxt ;
```

```
*       free(delete, sizeof(freemem) ) ;
*       delete = nxtdelete ;
*       }
*    return ;
*    }
*

#define freelist a1
#define storeblk a0
#define fromblk a1
#define store a2
#define from a3
#define storecnt d1
#define fromcnt d2
#define delete a1
#define nxtdelete d0
#define freeBlkSize d3
#define tmp1 d0
#define tmp2 d0
#define tmp3 d0
#define tmp4 d0
#define tmp5 d1
#define PFregs d0/d1/d2/d3/a0/a1/a2/a3

        global  packFree

packFree
        rts                                 ; stub out till debugged.
        movem.l PFregs,-(sp)                ; save a few regs.
        move.l  freelist,tmp1               ; test parm.
        beq     7f                          ; quit if no frees.
        move.l  #freelast,freeBlkSize       ; a constant.
        move.l  freelist,storeblk           ; get block storing in.
        lea     freetbl-sizeof_freeblk(storeblk),store
        move.l  #-1,storecnt                ; clear number stored here.
        lea     freetbl(fromblk),from       ; get pointer to first active
        move.w  freecnt(fromblk),fromcnt ; get number there are.
        bmi.s   4f                          ; if empty next input block
1       tst.l   freesize(from)              ; is this of size zero ?
        beq.s   3f                          ; yes, dont copy it.
        add.l   #sizeof_freeblk,store       ; bump pointer of where to store.
        add.l   #1,storecnt                 ; bump count of things stored.
        cmp.l   freeBlkSize,storecnt        ; did we overflow a block ?
        blt.s   2f                          ; no, store in this block.
        move.w  freeBlkSize,freecnt(storeblk) ; save number store in this hunk
        move.l  freenxt(storeblk),storeblk ; chain forward to next block.
        clr.l   storecnt                    ; number stored.
        lea     freetbl(storeblk),store ; where to store this one.
2       cmp.l   store,from                  ; are they the same ?
        beq.s   3f                          ; yes, skip the copy.
        move.l  freeaddr(from),freeaddr(store)
        move.l  freesize(from),freesize(store)
3       add.l   #sizeof_freeblk,from        ; bump pointer of where to get.
        dbra    fromcnt,1b                  ; go back if more.
4       move.l  freenxt(fromblk),tmp2       ; get next block of frees
        beq.s   5f                          ; if none then nearly done.
```

```
        move.l  tmp2,fromblk            ; make addressable.
        move.w  freecnt(fromblk),fromcnt ; get number of entries in blk
        bmi.s   4b                      ; none in here try next block.
        lea     freetbl(fromblk),from   ; get first entry in block.
        move.l  freeaddr(store),tmp3    ; get the end addr of the
        add.l   freesize(store),tmp3    ; last free in last block.
        cmp.l   freeaddr(from),tmp3     ; is it next to this one ?
        bne.s   1b                      ; no, dont merge.
        move.l  freesize(from),tmp4     ; yes, add length of
        add.l   tmp4,freesize(store)    ; this to last one.
        sub.l   #1,fromcnt              ; less one for this entry.
        bmi.s   4b                      ; if last one next block.
        add.l   #sizeof_freeblk,from    ; else skip over this free.
        bra.s   1b                      ; and we are done.
5       move.w  storecnt,freecnt(storeblk) ; save number stored in this blk.
        move.l  freenxt(storeblk),nxtdelete ; address of next block.
        beq.s   7f                      ; if zero no next one.
        clr.l   freenxt(storeblk)       ; else break chain.
6       move.l  nxtdelete,delete        ; get parm
        move.l  freenxt(delete),nxtdelete ; get next free block in advance.
        move.l  #freesize,tmp5          ; other parm.
        jsr     freeMem                 ; go free unused block.
        tst.l   nxtdelete               ; is there another ?
        bne.s   6b                      ; yes fre it too.
7       movem.l (sp)+,PFregs            ; restore regs.
        rts

#undef freelist
#undef storeblk
#undef freeblk
#undef store
#undef free
#undef storecnt
#undef freecnt
#undef delete
#undef nxtdelete
#undef tmp1
#undef tmp2
#undef tmp3
#undef tmp4
#undef tmp5
#undef  PFregs

*       getBot                                          */
*    Get a BOT element from the BOT free list           */
* and return it to the caller. Allocate a new           */
* hunk of free list if the list is empty the            */
* various fielsd (especially the list chain)            */
* should be zeroed in debuging versions.                */
*
* getBot()
*     {
*     bote *wkbot ;
*     int i ;
*
*     if ( botfree == NULL )
```

```
*       {
*       wkbot = alloc(1024 * sizeof(bote) ) ;
*       if ( wkbot != NULL )
*           {
*           for ( i = 0 ; i < 1024 ; ++i )
*               {
*               wkbot->botnxt = botfree ;
*               botfree = wkbot ;
*               wkbot += 1 ;
*               }
*           else
*               {
*               signal(noBigMem) ;
*               botfree = robot ;
*               }
*           }
*       }
*     wkbot = botfree ;
*     botfree = botfree->botnxt ;
*     return (wkbot) ;
*     }

#define i d0
#define wkbot a0
#define lastbot a1

        global  getBot
getBot
        move.l  i,-(sp)                 ; save working regs.
        move.l  botfree,wkbot           ; get a copy of the
        move.l  wkbot,i                 ; next free and test it.
        beq.s   1f                      ; if not zero O.K.
        move.l  botnxt(wkbot),botfree   ; unchain from the free list.
        if      DEBUG
        clr.l   botnxt(wkbot)           ; clear pointer to free list.
        clr.l   botsize(wkbot)          ; clear the size
        clr.l   botref(wkbot)           ; and back pointer to base obj.
        clr.l   botdat(wkbot)           ; clear pointer to data area.
        endif
        move.l  (sp)+,i                 ; restore reg values.
        rts                             ; and home to mama.

1       move.l  lastbot,-(sp)           ; else save another work reg.
        move.l  #1024*sizeof_bote,i     ; and get a new block
        jsr     alloc                   ; of free bot elements.
        move.l  wkbot,i                 ; did we have memory ?
        beq.s   4f                      ; no, go use reserve tank.
        move.w  #1024-1,i               ; else format the block.
        move.l  #0,lastbot                    ; link to last is NULL
2       move.l  lastbot,botnxt(wkbot)   ; save link.
        move.l  wkbot,lastbot           ; save address of last
        add.w   #sizeof_bote,wkbot      ; bump table entry pointer
        dbra    i,2b                    ; and if not done go for more.
        move.l  lastbot,wkbot           ; get return value.
3       move.l  botnxt(wkbot),botfree   ; unchain from the free list.
        if      DEBUG
```

```
        clr.l   botnxt(wkbot)           ; clear pointer to free list.
        clr.l   botsize(wkbot)          ; clear the size
        clr.l   botref(wkbot)           ; and back pointer to base obj.
        clr.l   botdat(wkbot)           ; clear pointer to data area.
        endif
        move.l  (sp)+,lastbot           ; restore the
        move.l  (sp)+,i                 ; working regs.
        rts                             ; ant thats all there is.

4       exception (No_Big_Memory)               ; bitch and
        move.l  robot,wkbot             ; use the reserve tank.
        bra     3b                      ; now regular processing.

#undef i
#undef lastbot
#undef wkbot


*       initMem                                         */
* Initialize the global memory variables.               */
* This routine first sets up the small-big              */
* object area rounding up so the memory is on           */
* a 4K boundry. Then it allocates the mamory            */
* for the grades and fills in the school array          */
* It then locks grade zero into memory. */
* Next it allocates the small frelist and puts          */
* one free area of the whole small area in it.          */
* Then comes the big free list with no free             */
* free hunks. finally a few variables are               */
* set and the module returns.                           */
*
* initMem(theEnd)
*     char *theEnd ;
*     {
*     register int i ;
*
*     smallArea = theEnd ;
*     theEnd += SMALLAREASIZE ;
*     smallAreaEnd =
*     theEnd = theEnd + 4095 & ~4096 ;
*     for ( i = 0 ; i < MAXGRADE ; ++i )
*         {
*       size = gp[i].size * 1024 ;
*       maxage = gp[i].itern - 1 ;
*
*       space = theEnd ;
*       theEnd += size ;
*       school[i].active = space ;
*       school[i].grdend = space + 4 ;
*       school[i].grdmax = theEnd - EXTRASPACE ;
*       remset = theEnd - sizeof(remembered) ;
*       remset.remobj = 0 ;
*       *space =
*       school[i].grdrememm = remset ;
*       school[i].bigobjs = NULL ;
*       school[i].oldAge = maxAge + (i << 4) ;
```

```
*       school[i].inactive = theEnd ;
*       *theEnd = theEnd + size ;
*       theEnd += size ;
*       }
*     memman( 1, school[0].active, school[0].active->spclast ) ;
*     memman( 1, school[0].inactive, school[0].inactive->spclast ) ;
*     smallfree = theEnd ;
*     smallfree->freenxt = NULL
*     smallfree->freecnt = 1 ;
*     smallfree->freetbl->freeaddr = smallArea ;
*     smallfree->freetbl->freesize = SMALLAREASIZE ;
*     sbp = smallfree ;
*     scp = smallfree->freetbl ;
*     sfcx = 1 ;
*     bigfree =
*     theEnd += sizeof(freemem) ;
*     bigfree->freenxt = NULL ;
*     bigfree->freecnt = 0 ;
*     bbp = bigfree ;
*     bcp = bigfree->freetbl ;
*     bfc = 0 ;
*     theEnd += sizeof(freemem) ;
*     botfree = NULL ;
*     mustCompact = -1 ;
*     memend = theEnd ;
*     /* memlimit is set by the guy who does the original sbrk. */
*     return ;
*     }

#define space a0
#define tmp3 a0
#define gp a1
#define tmp2 a1
#define remset a2
#define tmp0 d0
#define tmp1 d1
#define i d2
#define tmp5 d2
#define size d3
#define tmp4 d3
#define theEnd d6

#define IMregs d0/d1/d2/d3/d4/a0/a1/a2

pagernd equ 4095                        ; page size - 1

        extern  gradParm
        global  initMem

        if      UTek

        extern  _brk,_sbrk

brk     move.l  d0,-(sp)                ; save ret reg
        move.l  theEnd,-(sp)            ; push parm
        jsr     _brk                    ; go do it.
```

```
        add.l   #4,sp                   ; pop parm
        move.l  (sp)+,d0                ; pop saved reg
        rts                             ; and we're done.
        endif

initMem
        movem.l IMregs,-(sp)            ; save scratch regs.
        move.l  theEnd,smallArea:w      ; set beginning of small block area.
        add.l   #SMALLAREASIZE+pagernd,theEnd ;
        and.l   #!pagernd,theEnd        ; round up to even page boundry.
        move.l  theEnd,smallAreaEnd:w   ; and store end.
        clr.l   i                       ; loop counter.
        bset    #31,theEnd              ; make these addresses look like oops.
        move.l  #gradParm,gp            ; pointer to parms for grades.
1       move.l  (gp)+,size              ; get size of this grade (in K).
        asl.l   #5,size                 ; convert to bytes.
        asl.l   #5,size                 ; convert to bytes.
        add.l   #pagernd,size           ; then round up to an even page
        and.l   #!pagernd,size          ; to make faster access.
        move.l  theEnd,space            ; get start of this grade active.
        add.l   size,theEnd             ; and the end.
        if      UTek
        bsr     brk                     ; map the memory.
        endif
        move.l  space,(school_active:w,,i*4) ; store beginning.
        move.l  theEnd,(space)+         ; and the end.
        move.l  space,(school_grdend:w,,i*4) ; store first available.
        move.l  theEnd,remset           ; start off the remembered set.
        clr.l   tmp0                    ; get a zero.
        move.l  tmp0,-(remset)          ; and zero out the last remembered set entr
        move.l  remset,(school_grdremem:w,,i*4) ; and save it's address.
        sub.l   #EXTRASPACE,remset      ; save a block for slop
        move.l  remset,(school_grdmax:w,,i*4) ; and save as upper limit.
        move.l  tmp0,(school_bigobjs:w,,i*4) ; null big object set.
        move.w  i,tmp1                  ; get grade number.
        lsl.w   #4,tmp1                 ; shift to grade field.
        add.l   (gp)+,tmp1              ; add number of iterations.
        sub.w   #1,tmp1                 ; decrement by one for dbra
        move.b  tmp1,(school_oldAge:w,,i*1) ; and store it as first age.
        move.l  theEnd,(school_inactive:w,,i*4) ; store start of inactive set.
        move.l  theEnd,space            ; and a copy
        add.l   size,theEnd             ; calculate the end of this space.
        if      UTek
        bsr     brk                     ; map the memory.
        endif
        move.l  theEnd,(space)          ; store it.
        add.w   #1,i                    ; next grade.
        cmp.w   #MAXGRADE,i             ; are we done?
        ble.s   1b                      ; no, next grade.
        if      UniFlex
*       lock grade 0 spaces into real memory.
        move.l  school_active:w,tmp2    ; get active space for grade zero.
        move.l  (tmp2),-(sp)            ; push the end,
        move.l  tmp2,-(sp)              ; the beginning,
        move.l  #1,-(sp)                ; lock to real code,
        move.w  #memman,-(sp)           ; and mem man code onto stack.
```

```
        move.l  sp,a0                   ; get the stack pointer.
        sys     indx                    ; and do it.
        add.w   #14,sp                  ; pop the stuff off the stack.
        move.l  school_inactive:w,tmp2  ; now for the grade zero inactive.
        move.l  (tmp2),-(sp)            ; push the end,
        move.l  tmp2,-(sp)              ; the beginning,
        move.l  #1,-(sp)                ; lock to real code,
        move.w  #memman,-(sp)           ; and mem man code onto stack.
        move.l  sp,a0                   ; get the stack pointer.
        sys     indx                    ; and do it.
        add.w   #14,sp                  ; pop the stuff off the stack.
        endif
        bclr    #31,theEnd              ; make these plain old addresses.
        move.l  theEnd,smallfree:w      ; area for the smaller free list.
        move.l  theEnd,sbp:w            ; current list to look in.
        clr.l   tmp0                    ; a zero to clear stuff with.
        move.l  #1,tmp1                 ; a 1 to init stuff with.
        move.l  theEnd,tmp3             ; address register copy.
        add.l   #sizeof_freemem,theEnd  ; skip over block.
        if      UTek
        bsr     brk                     ; map the memory.
        endif
        move.l  tmp0,(tmp3)+            ; next free block.
        move.w  tmp0,(tmp3)+            ; current count.
        move.w  tmp0,sfcx:w             ; and in look up parms
        move.w  tmp0,(tmp3)+            ; filler to get to dblword.
        move.l  tmp3,scp:w              ; addr to lookup parms.
        move.l  smallArea:w,tmp4        ; area start
        move.l  smallAreaEnd:w,tmp5     ; get the small end,
        sub.l   tmp4,tmp5               ; calculate the size.
        move.l  tmp5,(tmp3)+            ; and store in first free hunk.
        move.l  tmp4,(tmp3)+            ; addr of first hunk address.
        move.b  tmp1,sflg:w             ; set flag to start at beginning.
        move.l  theEnd,bigfree:w        ; make a big  free list.
        move.l  theEnd,bbp:w            ; and the palce to start looking.
        move.l  theEnd,tmp3             ; make addressable,
        add.l   #sizeof_freemem,theEnd  ; skip over block.
        if      UTek
        bsr     brk                     ; map the memory.
        endif
        move.l  tmp0,(tmp3)+            ; NULL next block
        neg.l   tmp1                    ; get a minus one.
        move.w  tmp1,(tmp3)+            ; zero entries.
        move.w  tmp1,bfc:w              ; also in current count.
        move.w  tmp0,(tmp3)+            ; zero entries.
        move.l  tmp3,bcp:w              ; current hunk address.
        move.l  tmp0,(tmp3)+            ; set address and
        move.l  tmp0,(tmp3)+            ; hunk size to zip.
        move.b  tmp1,bflg:w             ; set setup flag.
        if      UTek
        move.l  #4*4*8+4,-(sp)          ; size of reserve bot hunks
        jsr     _sbrk                   ; get the memory.
        add.l   #4,sp                   ; pop parm
        clr.l   tmp0                    ; reset the zero.
        endif
        move.l  theEnd,tmp3             ; and as address.
```

```
        move.l  tmp0,(tmp3)+            ; NULL next pointer
        move.l  tmp0,(tmp3)+            ; empty bot entry.
        move.l  tmp0,(tmp3)+            ; empty bot entry.
        move.l  tmp3,theEnd             ; pointer to this bot.
        move.l  tmp0,(tmp3)+            ; empty bot entry.
        move.w  #6,i                    ; count of how many to make.
2       move.l  theEnd,(tmp3)+          ; pointer to the last one.
        move.l  tmp0,(tmp3)+            ; empty bot entry.
        move.l  tmp0,(tmp3)+            ; empty bot entry.
        move.l  tmp3,theEnd             ; pointer to this bot.
        move.l  tmp0,(tmp3)+            ; empty bot entry.
        dbra    i,2b                    ; go back for more ?
        move.l  theEnd,robot            ; pointer to some reserve bots.
        add.l   #4,theEnd               ; up the pointer.
        move.w  tmp1,mustCompact:w      ; nothing to compact.
        if      UniFlex
        move.l  theEnd,memend:w         ; save next working storage.
        endif
        movem.l (sp)+,IMregs            ; restore registers.
        rts                             ; and we are done.

#undef space
#undef tmp3
#undef gp
#undef tmp2
#undef remset
#undef tmp0
#undef tmp1
#undef tmp4
#undef tmp5
#undef i
#undef size
#undef theEnd
#undef IMregs
```