```
        name    forAllPointersIn

        lib     objdefs/Objects.def
        lib     objdefs/Context.def
        lib     objdefs/MemoryStructs.def

        data

#define isRemoteWordOffset isRemote_registerBit
#define isRemoteByteOffset isRemote_bit
#define isContext isContext_registerBit

*       forAllPointersIn                                *
*   A routine to scan an object and find each           *
* oop in it. It then calls a routine and passes         *
* the address and contents of the oop.                  *
*                                                       *
* Registers:                                            *
*   totally unused:                                     *
*                d0, d1, d5, d6, a3, a4, a5             *
*    these may be used to pass info to/from the         *
*       process routine.
*   used but restored on exit:                          *
*               d2, d3, d4, a1, a2                      *
*   input parameters:                                   *
*               a0 = oop of object to be scanned        *
*               a6 = routine to process oops            *
*   output to process routine:                          *
*               a0 = oop of object being scanned        *
*               a1 = pointer to field in object         *
*               a2 = contents of field at a2            *

#define len d2
#define hdr2 d3
#define K64 d3
#define tmp1 d4
#define objptr a0
#define refptr a1
#define botp a1
#define contents a2
#define doit a6
#define FApregs d2/d3/d4/a1/a2
#define contextBaseSize (Context_temp0_offset/4)-4

        global  forAllPointersIn

forAllPointersIn
        movem.l FApregs,-(sp)
        move.w  objectSizeOffset(objptr),len    ; get objects length.
        cmp.w   #12,len                         ; just for drill chk len
        blt.s   6f                              ; too small not a real object.
*       process class always a pointer (unless not an object.)
        lea     objectClassOffset(objptr),refptr ; address of class.
        move.l  (refptr),tmp1                   ; the class oop.
        bpl.s   6f                              ; unreal object so dont process
```

```
        move.l  tmp1,contents                   ; move over to addr.
        jsr     (doit)                          ; go process it.
*       now check object flags and get the length if it is pointers.
        move.l  objectHdr2Offset(objptr),hdr2   ; get flags.
        bftst   hdr2{13:3}                      ; does it contain oops?
        bgt.s   6f                              ; no, done already.
        add.w   #4,refptr                       ; bump pointer into obj.
        lsr.w   #2,len                          ; convert size to oops.
        sub.w   #4,len                          ; three for header, 1 for dbra
        bmi.s   6f                              ; no oops so done.
        btst    #isContext,hdr2                 ; is it a context ?
        beq.s   2f                              ; no, use len as is.
        move.l  Context_stackp_offset(objptr),len ; get context SP
        bpl.s   1f                              ; if int O.K.
        clr.l   len                             ; assume nil = 0.
1       add.l   #contextBaseSize,len            ; add in size of fixed stuff.
2       move.l  (refptr),tmp1                   ; get a field from the obj.
        bpl.s   3f                              ; if small int do nothing.
        move.l  tmp1,contents                   ; else make address.
        jsr     (doit)                          ; process the oop.
3       add.w   #4,refptr                       ; bump addr of field.
        dbra    len,2b                          ; go back for more?
        btst    #isRemoteWordOffset,hdr2        ; is there a remote part ?
        beq.s   6f                              ; no, we're done.
        move.l  -(refptr),botp                  ; get address of bot entry.
        move.l  botsize(botp),len               ; get the length.
        beq.s   6f                              ; it's the nullbot, we're done.
        lsr.l   #2,len                          ; convert to longs.
        sub.l   #1,len                          ; one for the dbra.
        move.l  botdat(botp),refptr             ; get data address.
        move.l  #$10000,K64                     ; get constant for dcr hi word
4       move.l  (refptr),tmp1                   ; get a field from obj.
        bpl.s   5f                              ; if small int do nothing.
        move.l  tmp1,contents                   ; else make address.
        jsr     (doit)                          ; process the oop.
5       add.w   #4,refptr                       ; bump addr of field.
        dbra    len,4b                          ; go back for more?
        sub.l   K64,len                         ; do a borrow from high word.
        bge.s   4b                              ; done yet ?
6       movem.l (sp)+,FAPregs                   ; restore regs.
        rts

#undef tmp1
#undef len
#undef hdr2
#undef K64
#undef contents
#undef refptr
#undef botp
#undef objptr
#undef doit
#undef FAPregs
#undef contextBaseSize
```

```
        name    GC

#include "../include/defOS.i"
#include "../include/parameters.i"
#include "../include/idioms.i"
#include "../include/traceCalls.i"
        lib     objdefs/MemoryStructs.def
        lib     objdefs/GCVars.def
        lib     objdefs/MemoryVars.def
        lib     objdefs/Objects.def
        lib     objdefs/Context.def
        if      UniFlex
        lib     sysdef
        endif
        if      UTek
        lib     objdefs/syscall.def
        lib     objdefs/syscall.mac
        lib     objdefs/memmap.def
        endif
#include "../include/registerDefinitions.i"
#include "../include/exceptions.i"
#include "../include/knownOops.i"
#include "../include/Memory.i"

#define isContext isContext_registerBit

#define ageField 4:4
#define gradeField 1:3

        extern  setGCcursor
        extern  restoreCursor
        extern  packFree
        extern  gradParm
        extern  forAllPointersIn
        extern  compactGradeFlag

        if      STATS
        extern  rec_copied,rec_grad,rec_held,rec_merged,rec_msize,rec_cntxt
        extern  rec_rssize,rec_rsptrs,rec_rscopy,rec_rsctx
        extern  rec_botcnt,rec_botmem,rec_botfree,rec_botFM
        extern  collect_stat
        extern  grade_stat1,grade_stat2
        extern  rem_stat1,rem_stat2
        extern  bot_stat1,bot_stat2
        endif

*       ---- Global all labels so debug will disassemble them
*               remove for production.
        global  addref
        global  copyAndForwardObject
        global  scavengeSpaceStartingAt
        global  salvageRememberedObjects
        global  copyRememberedObject
        global  salvageInterpObjects
        global  updateRemembered
```

```
        global  updateBot
        global  compactGrade

        data

*
*       addref                                          */
*  This routine checks the reference at refptr          */
* in object objptr for being a pointer from            */
* older to younger objects. If it is then the          */
* reference (and its object) are saved in the          */
* remembered set for the grade of the pointee          */
* object. The target object is also saved for          */
* checking that the value has not been replaced        */
* If the memory space overflows then a garbage         */
* collection is signaled.                              */
*
* addref(objptr, tgtptr)
*    object  *objptr ;          /* object to be remembered              */
*    object  *tgtptr ;          /* target of pointer in object          */
*    {
*    int    tgtgrade ;          /* grade of referenced object.          */
*
*    tgtgrade = tgtptr->objgrade ;
*    if (objptr->objrem[tgtgrade] ) return ;
*    remptr = school[tgtgrade]->grdremem ;
*    *remptr-- = objptr ;
*    school[tgtgrade]->grdremem = remptr ;
*
*    school[tgtgrade]->grdmax  -= sizeof(remembered) ;
*    if ( school[tgtgrade]->grdend > school[tgtgrade]->grdmax &&
*         tgtgrade > mustCompact )
*       {
*       mustCompact =  tgtgrade ;
*       }
*    return ;
*    }
*

#define objptr  a0
#define room    a1
#define target  a2
#define prset   a3
#define tgtgrade d0
#define tmpd    d1
#define ARSregs tgtgrade/tmpd/room/prset

addref
        movem.l ARSregs,-(sp)           ; save used regs.
        bfextu  (target){gradeField},tgtgrade ; get grade of target.
        bclr    tgtgrade,objectRemFlagsOffset(objptr) ; show no need to rem
        beq.s   1f                      ; if already shown then done
        move.l  tgtgrade,tmpd           ; a copy of the grade
        lsl.w   #2,tmpd                 ; convert to word offset
        move.w  tmpd,room               ; and store as an address.
        move.l  school_grdremem:w(room),prset ; and get the pointer.
```

```
        move.l  objptr,-(prset)          ; save object doing pointing.
        move.l  prset,school_grdrememem:w(room) ; and update memory
        move.l  school_grdmax:w(room),tmpd ; get old max
        sub.l   #sizeof_remembered,tmpd ; get new max value
        move.l  tmpd,school_grdmax:w(room) ; update memory.
        cmp.l   school_grdend:w(room),tmpd ; did it over flow ?
        bgt.s   1f                       ; no, we can quit.
        cmp.w   mustCompact,tgtgrade     ; doing this grade ?
        ble.s   1f                       ; yes,  no need
        move.w  tgtgrade,mustCompact:w   ; no, well we are now.
1       movem.l (sp)+,ARSregs            ; pop saved regs.
        rts                              ; and quit.

#undef  objptr
#undef  target
#undef  prset
#undef  room
#undef  tgtgrade



*       copyAndForwardObject                                    */
*   This routine copys an object to the proper new              */
* space and leaves a forwarding pointer. It selects             */
* the new space depending on the age in the current             */
* grade. It then copies the object to that space in a           */
* manner depending on the object. If the object is a            */
* context it only copies up to the stack pointer and            */
* nulls the rest so defunct popped objects wont be              */
* copied later. If the object is a small indirect(from          */
* becomes) it is made non-indirect. Otherwise all of            */
* the object is copied. Finally the receiving space             */
* memory pointers are adjusted and checked for space            */
* overflow. The caller should check that it is not a            */
* small integer.                                                */
*
* copyAndForwardObject(oldLocation)
*     object *oldLocation ;
*     {
*     grade   *newRoom ;
*     object  *newLocation ;
*     int      i ;
*     bote    *botp ;
*
*     if ( oldLocation->isForwarded ) return ;
*     if ( oldLocation->objage-- )
*       { /* stay in current grade but age up */
*       newRoom = toRoom ;
*       }
*     else
*       {
*       if ( gradRoom->grdend > gradRoom->grdmax )
*           { /* Held back cause next grade is full, dont age. */
*           newRoom = toRoom ;
*           ++oldLocation->objage ;
*           }
```

```
*       else
*           { /* graduate and set time in grade to the max. */
*           newRoom = gradRoom ;
*           oldLocation->objage = gradRoom->oldAge ;
*           if ( oldLocation->objgrade < MAXGRADE ) ++oldLocation->objgrade ;
*           }
*       }
*
*     newLocation = newRoom->grdend ;
*     i = oldLocation->objsize ;
*
*     if ( oldLocation->isContext )
*       {
*       newLocation->objsize     = oldLocation->objsize ;
*       newLocation->objgrade    = oldLocation->objgrade ;
*       newLocation->objage      = oldLocation->objage ;
*       newLocation->objhash     = oldLocation->objhash ;
*       newLocation->isForwarded = 0 ;
*       newLocation->isContext   = oldLocation->isContext ;
*       newLocation->isPointers  = oldLocation->isPointers ;
*       newLocation->isIndirect  = oldLocation->isIndirect ;
*       newLocation->class       = oldLocation->class ;
*         for ( i = 0 ; i < CSPval(oldLocation) ; ++i )
*           {
*           newLocation->contents[i] = oldLocation->contents[i] ;
*           }
* #ifdef DEBUG
*         for ( ; i < oldLocation->objsize ; ++i )
*           {
*           newLocation->contents[i] = 0 ;
*           }
* #endif
*       }
*     else if ( oldLocation->isIndirect &&
*               BOTpnt(oldLocation)->botsize < SMALLSIZE &&
*               BOTpnt(oldLocation)->botsize != 0  )
*       {
*       botp = BOTpnt(oldLocation) ;
*       newLocation->objsize     = botp->botsize + sizeof(object) ;
*       newLocation->objgrade    = oldLocation->objgrade ;
*       newLocation->objage      = oldLocation->objage ;
*       newLocation->objhash     = oldLocation->objhash ;
*       newLocation->isForwarded = 0 ;
*       newLocation->isContext   = oldLocation->isContext ;
*       newLocation->isPointers  = oldLocation->isPointers ;
*       newLocation->isIndirect  = 0 ;
*       newLocation->class       = oldLocation->class ;
*       for ( --i ; i > 0 ; -- i )
*           {
*           newLocation.contents[i] = oldLocation.contents[i] ;
*           }
*       for ( i = 0 ; i < botp->botsize ; ++i )
*           {
*           newLocation->contents[i] = botp->botdat[i] ;
*           }
*       }
```

```
*     else
*       {
*       newLocation->objsize     = oldLocation->objsize ;
*       newLocation->objgrade    = oldLocation->objgrade ;
*       newLocation->objage      = oldLocation->objage ;
*       newLocation->objhash     = oldLocation->objhash ;
*       newLocation->isForwarded = 0 ;
*       newLocation->isContext   = oldLocation->isContext ;
*       newLocation->isPointers  = oldLocation->isPointers ;
*       newLocation->isIndirect  = oldLocation->isIndirect ;
*       newLocation->class       = oldLocation->class ;
*         for (  ; i ; --i )
*           {
*           newLocation->contents[i] = oldLocation->contents[i] ;
*           }
*       }
*     oldLocation->forwardingPointer = newLocation ;
*     oldLocation->isForwarded = TRUE ;
*     newRoom->grdend += newLocation->objsize ;
*     if ( newRoom->grdend > newRoom->grdmax  &&
*        newLocation->objgrade < mustCompact )
*       {
*       mustCompact = newLocation->objgrade ;
*       }
*     return ;
*     }

#define botp a0
#define newLocation a1
#define oldLocation a2
#define newRoom a3
#define i d0
#define hdr1 d1
#define hdr2 d1
#define tmpsize d2
#define tmpend d2
#define tmpzero d2
#define tmpage d3
#define tmpgrade d3
#define fwdPtr d4
#define newEnd d5
#define xtdGrade d5
#define CAFOregs d0/d1/d2/d4/d5/a0/a1/a3

12      move.l  (oldLocation),oldLocation       ; return new address.
        move.l  (sp)+,tmpage                    ; pop scratch reg.
        move    #$0005,ccr                      ; show sombody moved it.
        rts                                     ; and we're done.

13      move.l  (sp)+,tmpage                    ; pop scratch reg.
        move    #$0004,ccr                      ; show not moved.
        rts                                     ; and quit.

copyAndForwardObject
        move.l  tmpage,-(sp)                    ; save one reg.
        move.b  (oldLocation),tmpage            ; is it already forwarded ?
```

```
        bmi.s   12b                                     ; yes just return new pointer.
        and.b   #Object_grade_byteMask,tmpage           ; mask off all but the grade.
        cmp.b   tmpage,xtdGrade                         ; is it the hot grade ?
        bne.s   13b                                     ; no, ret not moved.
        movem.l CAFOregs,-(sp)                          ; save working regs.
        move.l  (oldLocation),hdr1                      ; get header word 1.
        bftst.l hdr1{ageField}                          ; is it aged out ?
        beq.s   1f                                      ; yes, see if graduate
        tst.w   mustGraduate:w                          ; is grade full of young stuff?
        beq     51f                                     ; yes graduate this one then.
        bfextu  (oldLocation){gradeField},tmpgrade ;get the current grade number.
        cmp.w   #MAXGRADE,tmpgrade                      ; is it in oldest grade ?
        beq     52f                                     ; yes, leave it here.
        move.w  gradRoom:w,newRoom                      ; assume will graduate.
        move.l  school_grdend(,newRoom.w*4),tmpend ; check remaining
        cmp.l   school_grdmax(,newRoom.w*4),tmpend ; room in grad space
        bgt.s   52f                                     ; it's full so leave here.
        if      STATS
        add.l   #1,rec_grad                             ; show another one graduated.
        endif
        bset    tmpgrade,objectRemFlagsOffset(oldLocation) ; and rem bit.
        move.b  school_oldAge:w(,newRoom.w),tmpage ; get new grade byte
        bfins.l tmpage,hdr1{0:8}                        ; reset grade
        bra.s   2f

1       bfextu  (oldLocation){gradeField},tmpgrade ;get the current grade number.
        cmp.w   #MAXGRADE,tmpgrade                      ; is it in oldest grade ?
        beq.s   61f                                     ; yes, leave it here.
        move.w  gradRoom:w,newRoom                      ; assume will graduate.
        move.l  school_grdend:w(,newRoom.w*4),tmpend ; check remaining
        cmp.l   school_grdmax:w(,newRoom.w*4),tmpend ; room in grad space
        bgt.s   55f                                     ; it's full so leave here.
        if      STATS
        add.l   #1,rec_grad                             ; show another one graduated.
        endif
        bset    tmpgrade,objectRemFlagsOffset(oldLocation) ; and rem bit.
        move.b  school_oldAge:w(,newRoom.w),tmpage ; get grade byte
        bfins.l tmpage,hdr1{0:8}                        ; reset grade
        bra.s   2f


55      move.w  toRoom:w,newRoom                        ; hold over in this grade.
        if      STATS
        add.l   #1,rec_held                             ; show another one graduated.
        endif
        bra.s   2f

61      tst.w   mustGraduate:w                          ; is grade full of young stuff?
        bne.s   53f                                     ; yes graduate this one then.
54      clr.w   mustGraduate:w                          ; next grade full cant grad
53      move.w  toRoom:w,newRoom                        ; leave in this grade.
        if      STATS
        add.l   #1,rec_copied                           ; show another one graduated.
        endif
        move.b  school_oldAge:w(,newRoom.w),tmpage ; get new grade byte
        bfins.l tmpage,hdr1{0:8}                        ; reset grade
```

```
        bra.s   2f

52      clr.w   mustGraduate:w                  ; next grade full cannot grad
51      move.w  toRoom:w,newRoom                ; leave in this grade.
        sub.l   #$01000000,hdr1                 ; age it a little
        if      STATS
        add.l   #1,rec_copied                   ; show another one graduated.
        endif

2       add.l   newRoom,newRoom                 ; and shift the room number
        add.l   newRoom,newRoom                 ; by 4 for indexing.
*       We have decided the room to put the object in and set it's
*         new age and grade field.
3       move.l  school_grdend:w(newRoom),newLocation ; where to put it.
        clr.l   i                               ; zap hi word.
        move.w  hdr1,i                          ; get objects current size
        add.w   #3,i                            ; rounded up
        move.l  i,newEnd                        ; saved for later.
        lsr.w   #2,i                            ; converted to words,
        sub.w   #3,i                            ; less 1 for dbra and two hdrs.
        move.l  newLocation,fwdPtr              ; save forwarding ptr.
        move.l  hdr1,(newLocation)+             ; move first word.
        move.l  fwdPtr,(oldLocation)+           ; and set fwding pointer.
*       have moved the first word special will move the rest of the
*          object as a block. length is in i, if it's right.
        move.l  (oldLocation)+,hdr2             ; get second header and test
        btst    #isContext,hdr2                 ;
        beq.s   6f                              ; not context, try indirect.
*       move a context only to stack pointer, zero the restif debugging.
*       get the size of the active context stuff,
*           -2 cause pointer has been bumped (and -1 for dbra).
        if      STATS
        add.l   #1,rec_cntxt                    ; count number of contexts copied.
        endif
        move.l  Context_stackp_offset-8(oldLocation),tmpsize
        bmi     14f                             ; object? set to zero.
        add.l   #(Context_temp0_offset)/4-3,tmpsize ; basic part of context.
13
        if      GCdebug
        sub.w   tmpsize,i                       ; reduce i by this amount.
        endif
        move.l  hdr2,(newLocation)+             ; put in header word.
4       move.l  (oldLocation)+,(newLocation)+   ; copy a word.
        dbra    tmpsize,4b                      ; as many time as necessary.
        if      GCdebug
        sub.w   #1,i                            ; for the dbra
        bmi.s   11f                             ; already done ?
        clr.l   tmpzero                         ; no, get a zero.
5       move.l  tmpzero,(newLocation)+          ; store a zero.
        dbra    i,5b                            ; as many time as you want.
        endif
        bra.s   11f                             ; and we're all but done.
*       Come here to test for remote part which is small.
6       btst    #isRemote_registerBit,hdr2      ; is it indirect ?
        beq.s   9f                              ; no normal object.
        move.l  (oldLocation,i.w*4),botp        ; get pointer to BOT
```

```
        tst.b   botdat(botp)                    ; is perm remoted ?
        bmi.s   9f                              ; yes, do normal obj stuff.
        move.l  botsize(botp),tmpsize           ; get size of object.
        beq.s   9f                              ; if zero or
        cmp.l   #SMALLSIZE,tmpsize              ; too large then dont
        bpl.s   9f                              ; combine with base object.
*       we are to combine the object and it's remote part.
        if      STATS
        add.l   #1,rec_merged                   ; show another object merged
        add.l   tmpsize,rec_msize               ; and how big it was.
        endif
        bclr    #isRemote_registerBit,hdr2      ; clear remote bit.
        sub.w   #1,i                            ; forget BOT pointer
        sub.w   #4,tmpsize                      ; subtract size of BOT ptr.
        add.w   tmpsize,-2(newLocation)         ; add to the objects size.
        add.l   tmpsize,newEnd                  ; and end of new obj.
        move.l  hdr2,(newLocation)+             ; insert second header word.
7       move.l  (oldLocation)+,(newLocation)+   ; move a word of the object.
        dbra    i,7b                            ; however often is needed.
        move.l  tmpsize,i                       ; get the size of the big part.
        add.w   #3,i                            ; round up to a word
        lsr.w   #2,i                            ; convert to word count.
*       already one less than the data.
        move.l  botdat(botp),oldLocation        ; get address of data.
8       move.l  (oldLocation)+,(newLocation)+   ; and move words undil
        dbra    i,8b                            ; the count runs out.
        bra.s   11f                             ; quit.
*       move a normal run-of-the-mill object.
9       move.l  hdr2,(newLocation)+             ; move second header word.
10      move.l  (oldLocation)+,(newLocation)+   ; by words for
        dbra    i,10b                           ; the count in i.
*       now reset end of to space and return value.
11      move.l  fwdPtr,oldLocation              ; set up return value.
        and.w   #!3,newEnd                      ; round down to words
        add.l   oldLocation,newEnd              ; calc the obj end
        move.l  newEnd,school_grdend:w(newRoom) ; new end
        cmp.l   school_grdmax:w(newRoom),newEnd ; did grade overflow ?
        blo.s   12f                             ; no, we're in fat city
        move.w  #-1,mustGraduate:w              ; yes, show we gott move.
12      movem.l (sp)+,CAFOregs                  ; restore regs.
        move.l  (sp)+,tmpage                    ; pop scratch reg.
        move    #$0001,ccr                      ; show it was moved.
        rts                                     ; and we're done.

14      move.l  #(Context_temp0_offset)/4-3,tmpsize ; basic part of context.
        bra     13b                             ; only.

#undef oldLocation
#undef newLocation
#undef botp
#undef i
#undef hdr1
#undef hdr2
#undef tmpage
#undef tmpgrade
#undef tmpsize
```

```
#undef tmpend
#undef fwdPtr
#undef tmpzero
#undef newRoom
#undef newEnd
#undef xtdGrade
#undef CAFOregs


*       scavengeSpaceStartingAt                                 */
*  This routine does a scan of the objects that have            */
* been moved starting at currentObject. It copies any           */
* un-copied objects from this grade that they refer to.*/
* It does this by scanning each object and copies               */
* objects at which it points that are in grade "grade"          */
* It always copies the class, then if the object is             */
* not pointers it quits. Otherwise it copies all stuff          */
* pointed in the contents of the object to the end or           */
* for contexts through the stack pointer.                       */
*
*
* scavengeSpaceStartingAt(grade, currentObject)
*    int     grade ;
*    object *currentObject ;
*    {
*    object **start ;
*    object **end ;
*
*    for(  ;
*        currentObject < school[grade].grdend ;
*        currentObject += ((currentObject->objsize + 3) & ~3) )
*        {
*
*        if ( currentObject->objsize == 1 ) continue ;
*
*        if ( currentObject->class->objgrade == grade)
*           {
*           copyAndForwardObject(currentObject->class) ;
*           currentObject->class = currentObject->class->forwardingPointer ;
*           addref(currentObject, currentObject->class ;
*           }
*
*        if ( !currentObject->isPointers) continue ;
*
*        start = &currentObject->contents[1] ;
*        if ( currentObject->isContext )
*           {
*           end = &(currentObject->contents[CSPval(currentObject)]) ;
*           }
*        else if ( currentObject->isIndirect )
*           {
*           end = &currentObject->contents[currentObject->objsize-1] ;
*           }
*        else
*           {
*           end = &currentObject->contents[currentObject->objsize] ;
```

```
*           }
*
*        while ( start <= end )
*           {
*           if ((*start)->objgrade == grade )
*               {
*               copyAndForwardObject(start) ;
*               *start = (*start)->forwardingPointer ;
*               addref(currentObject, start ) ;
*               }
*           }
*
*        if ( currentObject->isIndirect )
*           {
*           start = BOTpnt(currentObject)->botdat ;
*           end = start + (BOTpnt(currentObject)->botsize ) ;
*           }
*
*        while ( start <= end )
*           {
*           if ((*start)->objgrade == grade )
*               {
*               copyAndForwardObject((*start) ) ;
*               *start = (*start)->forwardingPointer ;
*               addref(currentObject, start ) ;
*               }
*           }
*       }
*    return ;
*    }

#define currentObject a0
#define start a1
#define botp start
#define contents a2
#define grade d0
#define tmp1 d1
#define objSize d2
#define end d3
#define hdr2 d4
#define K64 d4
#define tgtGrade d5
#define scanOlder d6
#define SSSAregs a0/a1/a2/a3/a4/d1/d2/d3/d4/d5/d6
#define contextBaseSize (Context_temp0_offset/4)-4

scavengeSpaceStartingAt
        cmp.l   school_grdend:w(,grade.w*4),currentObject ; is any to scan ?
        beq     7f                      ; no return.
        movem.l SSSAregs,-(sp)          ; yes save some regs.
        move.w  toRoom:w,tgtGrade       ; Get the "hot" grade.
        cmp.w   tgtGrade,grade          ; if scanned grade is younger than
        sgt     scanOlder               ; the from grade dont need to addref
        lsl.w   #4,tgtGrade             ; shifted for copyAndForwardObject
1       move.w  objectSizeOffset(currentObject),objSize ; get objects size.
        lea     objectClassOffset(currentObject),start  ; get class address.
```

```
        move.l  (start),contents        ; get class oop
        jsr     copyAndForwardObject    ; copy the class
        bcc.s   2f                      ; was it really copied ?
        move.l  contents,(start)        ; plug in forwarded address.
        tst.b   scanOlder               ; scanning grad space ?
        beq.s   2f                      ; no, then dont need to addrefs
        jsr     addref                  ; add this to the rem set if necessary
2       add.w   #3,objSize              ; round up the
        and.w   #!3,objSize             ; size of the object.
        move.l  objectHdr2Offset(currentObject),hdr2 ; get the flags word.
        bftst   hdr2{13:3}              ; is it pointer indexable ?
        bgt.s   5f                      ; if not then skip rest of object
        add.w   #4,start                ; bump the pointer to the data.
        move.w  objSize,end             ; get count of pointers
        lsr.w   #2,end                  ; converted to words.
        sub.w   #4,end                  ; 3 for hdr, 1 for dbra
        bmi.s   5f                      ; nothing left, huh!
        btst    #isContext_registerBit,hdr2 ; is it a context ?
        beq.s   3f                      ; no, go scan all of it.
        move.l  Context_stackp_offset(currentObject),end
        add.l   #contextBaseSize,end    ; else calculate active part
3       bsr.s   10f                     ; scan the object.
        btst    #isRemote_registerBit,hdr2 ; is it remote ?
        beq.s   5f                      ; no, scan next object.
        move.l  -(start),botp           ; else get pointer to big object
        move.l  botsize(botp),end       ; get size of remote part.
        sub.l   #1,end                  ; for the dbra.
        bmi.s   5f                      ; it was zero length anyway.
        lsr.l   #2,end                  ; convert to word count.
        move.l  botdat(botp),start      ; get start of remote part.
        move.l  #$10000,K64             ; get a 64 k constant.
4       bsr.s   10f                     ; scan of the big area
        sub.l   K64,end                 ; pick up the borrow.
        bpl.s   4b                      ; still more.
5       add.w   objSize,currentObject   ; bump pointer to object to scan.
        cmp.l   school_grdend:w(,grade.w*4),currentObject ; done ?
        bmi     1b                      ; no, get this one too.
        movem.l (sp)+,SSSAregs          ; restore registers.
7       rts                             ; and quit.

10      tst.b   scanOlder               ; looking at a younger grade
        beq.s   13f                     ; no, gotta add to rem. set.
11      move.l  (start),contents        ; get the oop
        move.l  contents,tmp1           ; test it.
        bpl.s   12f                     ; if small int no work to do.
        jsr     copyAndForwardObject    ; copy the object in.
        bcc.s   12f                     ; if not really copied.
        move.l  contents,(start)        ; install new pointer.
        jsr     addref                  ; if necessary add to ref table.
12      add.w   #4,start                ; bump object scan pointer.
        dbra    end,11b                 ; loop till all pointers done.
        rts                             ; back to main pgm.

13      move.l  (start),contents        ; get the oop
        move.l  contents,tmp1           ; test it for real object.
        bpl.s   14f                     ; if small int no work to do.
```

```
        jsr     copyAndForwardObject    ; copy the object in.
        bcc.s   14f                     ; if not really copied.
        move.l  contents,(start)        ; install forwarding pointer.
14      add.w   #4,start                ; and bump pointer.
        dbra    end,13b                 ; loop till all pointers done.
        rts                             ; back to main pgm

#undef currentObject
#undef start
#undef botp
#undef contents
#undef grade
#undef tgtGrade
#undef tmp1
#undef objSize
#undef hdr2
#undef contextBaseSize
#undef K64
#undef end
#undef scanOlder
#undef SSSAregs


*       salvageRememberedObjects                                */
*  This routine scans all the objects in a remembered           */
* set and copys any objects to which it refers to the           */
* toSpace or gradSpace as appropriate. It uses                  */
* copyAndForwardObject to do the copying. It then               */
* updates the references to those objects. It checks            */
* the target to see that it hasn't already been moved           */
* and hasn't been superseded. The end of the                    */
* remembered set is marked by a zero entry. Remembered          */
* set entries with a zero remptr mean the whole object          */
* must be scanned. that code has not been added yet.            */
*
* copyRememberedObject(objptr, refptr, tgtptr )
*    object  *objptr ;
*    object **refptr ;
*    object  *tgtptr ;
*    {
*    copyAndForwardObject( tgtptr ) ;
*    if ( !(tgtptr->isForwarded) return ;
*    tgtptr = *refptr = tgtptr->forwardingPointer ;
*    if ( copyFlags[newGrade] == 0 )
*       {
*       copyFlags[newGrade] = 1 ;
*       *school[newGrade].grdremem-- = objptr ;
*       }
*    return ;
*    }
*
* salvageRememberedObjects( grade, memorys, remSize)
*    int grade ;
*    remembered *memorys ;
*    int remSize ;
*    {
```

```
*    object  *objptr ;
*
*    school[grade+1].grdmax += SMALLSIZE - remSize ;
*
*    for ( ; *memorys ; ++memorys )
*         {
*         objptr = memorys->remobj ;
*         if ( !isptr(objptr) ) continue ;
*         copyFlags = objptr->objrem ;
*         copyfalgs[grade] = 0 ;
*         forAllPointersIn(objptr, copyRememberedObject) ;
*         objptr->objrem = copyFlags ;
*         ++school[grade+1].grdmax ;
*         }
*    school[grade+1].grdmax -= SMALLSIZE ;
*    return
*    }

#define copyFlags d0
#define remSize d0
#define newGrade d1
#define tmp2 d2
#define xg d5
#define grade d6
#define objptr a0
#define refptr a1
#define tgtptr a2
#define memorys a3
#define gradeLimit a4
#define tmp1 a5
#define whafa a6
#define SROregs d0/d1/d2/d5/a0/a1/a2/a3/a4/a5/a6

copyRememberedObject
        if      STATS
        add.l   #1,rec_rsptrs
        endif
        bsr     copyAndForwardObject    ; copy the damn thing.
        bcc.s   1f                      ; if no copy done.
        if      STATS
        add.l   #1,rec_rscopy
        endif
        move.l  tgtptr,(refptr)         ; update referencing obj.
        bfextu  (tgtptr){gradeField},newGrade ; get new room.
        bclr    newGrade,copyFlags      ; show remembered
        beq.s   1f                      ; already rem'ed, then done.
        move.l  school_grdremem:w(newGrade.w*4),tmp1 ; get rem ptr.
        move.l  objptr,-(tmp1)          ; save object.
        move.l  tmp1,school_grdremem:w(newGrade.w*4) ; save new rem ptr
        sub.l   #sizeof_remembered,school_grdmax:w(newGrade.w*4) ; decrease pointer
1       rts

salvageRememberedObjects
        movem.l SROregs,-(sp)           ; save rused regs.
        if      STATS
        jsr     rem_stat1
```

```
        endif
        move.w  grade,xg                ; get extended
        lsl.w   #4,xg                   ; grade for cmp.
        lea     school_grdmax+4(,grade.w*4),gradeLimit
        sub.l   remSize,(gradeLimit)    ; space for grad remembereds
        add.l   #SMALLSIZE+sizeof_remembered,(gradeLimit) ; at least one grad.
        lea     copyRememberedObject,whafa ; routine address.
1       add.l   #sizeof_remembered,(gradeLimit) ; one less
        move.l  (memorys)+,tmp2         ; get and test next obj.
        beq.s   2f                      ; ifd at end quit.
        bpl.s   1b                      ; if old ignore.
        if      STATS
        add.l   #1,rec_rssize
        endif
        move.l  tmp2,objptr             ; make object into address.
        if      STATS
        btst    #isContext,objectFlagsOffset(objptr)
        beq.s   10f
        add.l   #1,rec_rsctx
10      ds.w    0
        endif
        move.b  objectRemFlagsOffset(objptr),copyFlags
        bset    grade,copyFlags         ; no rem for this grade yet.
        jsr     forAllPointersIn        ; go do em.
        move.b  copyFlags,objectRemFlagsOffset(objptr)
        bra.s   1b                      ; next remembered.
2       sub.l   #SMALLSIZE+sizeof_remembered,(gradeLimit) ; take off slop space.
        if      STATS
        jsr     rem_stat2
        endif
        movem.l (sp)+,SROregs           ; restore work regs.
        rts

#undef  copyflags
#undef  newGrade
#undef  tmp1
#undef  tmp2
#undef  xg
#undef  grade
#undef  objptr
#undef  refptr
#undef  tgtptr
#undef  gradeLimit
#undef  memorys
#undef  whafa
#undef  SROregs



*       salvageInterpObjects                                    */
*  This routine checks any object the interpreter               */
* knows about and scavenges them as necessary.                  */
* this includes the display bitmap, known objects like          */
* nil, true, ..., active semaphores, and the context            */
* stack.                                                        */
*
```

```
* salvageInterpObjects(grade)
*    int grade ;
*    {
*    extern object **objectVars, **LastVar ;
*    extern object **knownOops, **knownOopEnd ;
*    register object **ip ;
*
*    for ( ip = objectVars ; ip <= LastVar ; ++ip )
*        {
*       if ( (**ip)->objgrade == grade )
*            {
*            copyAndForwardObject(*ip) ;
*            *ip = **ip->forwardingPointer ;
*            }
*       }
*
*    for ( ip = knownOops ; ip <= knownOopEnd ; ++ip )
*        {
*       if ( (**ip)->objgrade == grade )
*            {
*            copyAndForwardObject(*ip) ;
*            *ip = **ip->forwardingPointer ;
*            }
*       }
*    for ( ip = CSP-1 ; ip > ContextStack ; --ip )
*       {
*       if ( (**ip)->objgrade == grade )
*            {
*            copyAndForwardObject(*ip) ;
*            *ip = **ip->forwardingPointer ;
*            }
*       }
*    return ;
*    }

#define ip a1
#define atip a2
#define tmp1 d0
#define lc d1
#define xg d5
#define grade d6
#define SIOregs d0/d1/d5/d6/a1/a2

        extern  objectVars,objectVarCount
        extern  knownOops
        extern  valueStackEmptyBase

salvageInterpObjects
        movem.l SIOregs,-(sp)           ; save all working regs.
        move.w  grade,xg                ; get working copy of grade
        lsl.w   #4,xg                   ; shifted to position in object byte.
        move.l  #knownOops,ip           ; table of fixed objects.
        move.l  #knownOopCount-1,lc     ; and the count.
1       move.l  (ip)+,tmp1              ; get object pointer.
        bpl.s   2f                      ; if not object skip it.
        move.l  tmp1,atip               ; make object addressable.
```

```
        jsr     copyAndForwardObject    ; move object, get new address,
        bcc.s   2f                      ; if not move dont forward.
        move.l  atip,-4(ip)             ; and replace original pointer.
2       dbra    lc,1b                   ; if not done, next object.
        move.l  #objectVars,ip          ; get starting address of interp vars.
        move.l  #objectVarCount,lc      ; and times to loop.
3       move.l  (ip)+,tmp1              ; get object pointer.
        bpl.s   4f                      ; if not object skip it.
        move.l  tmp1,atip               ; make object addressable.
        jsr     copyAndForwardObject    ; move object, get new address,
        bcc.s   4f                      ; if not copyed dont update pointer.
        move.l  atip,-4(ip)             ; and replace original pointer.
4       dbra    lc,3b                   ; if not done, next object.
        move.l  #valueStackEmptyBase,ip ; start of context stack.
        move.l  CSP:w,lc                ; end of stack.
        sub.l   ip,lc                   ; number of elements.
        lsr.l   #2,lc                   ; convert to word count.
        sub.l   #1,lc                   ; for dbra
        bmi.s   7f                      ; if stack is empty.
5       move.l  (ip)+,tmp1              ; get object pointer.
        bpl.s   6f                      ; if not object skip it.
        move.l  tmp1,atip               ; make object addressable.
        jsr     copyAndForwardObject    ; move object, get new address,
        bcc.s   6f                      ; if not copied then dont forward.
        move.l  atip,-4(ip)             ; and replace original pointer.
6       dbra    lc,5b                   ; if not done, next object.
7       movem.l (sp)+,SIOregs           ; restore regs.
        rts                             ; home to caller.

#undef atip
#undef tmp1
#undef lc
#undef xg
#undef grade
#undef ip
#undef SIOregs

*       updateRemembered                                        */
*  This routine scans a remembered set and updates              */
* any pointer which points to a forwarded object. If           */
* the object is indirect it assumes the reference              */
* hasn't moved. If the object isn't indirect it assumes*/
* the offset to the reference hasn't changed. A zero           */
* entry marks the end of the set. A null target means          */
* this entry is dead.                                          */
*
* updateRemembered(grade)
*    int grade ;
*    {
*    remembered *memorys ;
*    object     *daObj ;
*    int         i ;
*
*    for ( i = grade-1 ; i >0 0 ; --i )
*       {
*       for ( memorys = school[i].grdremem; *memorys ; ++memorys )
```

```
*           {
*           daOb j = memorys->remobj ;
*           if ( !isPtr(daObj)) continue ;
*           if ( daObj->isForwarded )
*               {
*               memorys->remobj = daObj->forwardingPointer ;
*               }
*           else
*               {
*               if ( daObj->objgrade == grade )
*                   {
*                   memorys->remobj = 1 ;
*                   }
*               }
*           }
*       }
*   return ;
*   }
*

#define i d0
#define tmp0 d1
#define tmp1 d1
#define tmp2 d1
#define xg d5
#define grade d6
#define memorys a0
#define daObj a1
#define URregs d0/d1/d5/a0/a1

updateRemembered
        movem.l URregs,-(sp)            ; save local registers.
        move.w  grade,xg                ; build an extended
        lsl.w   #4,xg                   ; grade for cmp use.
        move.w  grade,i                 ; startin at grade
1       sub.w   #1,i                    ; minus one.
        bmi.s   4f                      ; if at bottom quit.
        move.l  school_grdremem:w(,i.w*4),memorys ;
2       move.l  (memorys)+,tmp0 ; get a rem element.
        beq.s   1b                      ; if last next grade.
        bpl.s   2b                      ; no good ignore.
        move.l  tmp0,daObj              ; make into an address
        move.l  (daObj),tmp1            ; get fwding ptr.
        bpl.s   3f                      ; izzi forwarded ?
        move.l  tmp1,-4(memorys)        ; yes, use new addr.
        bra.s   2b                      ; nex rem ptr.
3       move.l  #Object_grade_byteMask,tmp2 ; get masked off grade
        and.b   (daObj),tmp2            ; of the object.
        cmp.b   xg,tmp2                 ; is it in hot grade?
        bne.s   2b                      ; no, skip it.
        move.l  #1,-4(memorys)          ; mark as obsolete.
        bra.s   2b                      ; next rem pointer.

4       movem.l (sp)+,URregs            ; restore working regs.
        rts
```

```
#undef  i
#undef tmp0
#undef  tmp1
#undef  tmp2
#undef  xg
#undef  grade
#undef  memorys
#undef  daObj
#undef  URregs

*       updateBot                                       */
* This routine scans the list of big objects    */
* which are in grade n. If the base object has  */
* not been forwarded, it is assubmed to be      */
* dead so the big object is deleted and the     */
* bot entry is put on the free list. If it has  */
* been forwarded, the back pointer is updated   */
* and the grade is checked to see if it which   */
* which list it needs to be put on. The code    */
* which is commented out is for future use if   */
* the interpreter needs to invalidate a bot     */
* entry without going to the trouble of un-     */
* linking it for the list.                      */
*
* updateBot(grade)
*    int grade ;
*    {
*    register bote *nextBot, *thisBot, *newList, *nextList, *freeList ;
*
*    nextBot = school[grade].bigobjs ;
*    newList = NULL ;
*    nextList = school[grade+1].bigobjs ;
*    freeList = botfree ;
*    while ( (thisBot = nextBot ) != NULL)
*       {
*       nextBot = thisBot->botnxt ;
*       if ( thisBot->botref == NULL ) continue ;
*       if ( thisBot->botref->isForwarded )
*           {
*           thisBot->botref = thisBot->botref->forwardingPointer ;
*           if ( thisBot->botref->objgrad != grade )
*               {
*               thisBot->botnxt = nextList ;
*               nextList = thisBot ;
*               }
*           else
*               {
*               thisBot->botnxt = newList ;
*               newList = thisBot ;
*               }
*           }
*        else
*           {
*           free(thisBot->botdat) ;
```

```
* #ifdef DEBUG
*           thisBot->botsize = 0 ;
*           thisBot->botref =
*           thisBot->botdat = NULL ;
* #endif
*           thisBot->botnxt = freeList ;
*           freeList = thisBot ;
*           }
*       }
*    school[grade].bigobjs = newList ;
*    school[grade+1].bigobjs = nextList ;
*    botfree = freeList ;
*    return ;
*    }

#define xg d0
#define tmp1 d1
#define tmp2 d1
#define tmp3 d1
#define FMsize d1
#define nextBot d2
#define grade d6
#define gba a0
#define FMaddr a1
#define newList a2
#define nextList a3
#define freeList a4
#define thisBot a5
#define thisRef a6
#define UBregs d0/d1/d2/a0/a1/a2/a3/a4/a5/a6

        extern  freeMem

updateBot
        movem.l UBregs,-(sp)            ; save working regs.
        if      STATS
        jsr     bot_stat1               ; clear BOT stat counters.
        endif
        lea     school_bigobjs(,grade.w*4),gba ; where big list is
        move.w  grade,xg                ; get a shifted copy of
        lsl.w   #4,xg                   ; the grade for comparing.
        move.l  (gba),nextBot           ; get list of bots in grade.
        move.l  #0,newList              ; null list out to start.
        move.l  4(gba),nextList         ; list from next grade.
        move.l  botfree:w,freeList      ; list of discards.
1       tst.l   nextBot                 ; are we done yet mommy?
        beq     5f                      ; yes, save lists.
        move.l  nextBot,thisBot         ; make current bot.
        move.l  botnxt(thisBot),nextBot ; get address for next loop.
        move.l  botref(thisBot),tmp1    ; is ref zero ?
        beq     4f                      ; yes bot no longer valid.
        move.l  tmp1,thisRef            ; make working copy of obj addr.
        move.l  (thisRef),tmp2          ; is it forwarded ?
        bpl.s   3f                      ; no, discard data and bot.
        move.l  tmp2,thisRef            ; get new obj address.
        btst    #isRemote_bit,objectFlagsOffset(thisRef) ; is it still remote ?
```

```
        beq.s   3f                      ; no, throw away data.
        move.l  thisRef,botref(thisBot) ; else update bot.
        if      STATS
        add.l   #1,rec_botcnt           ; count this bot
        move.l  botsize(thisBot),tmp3   ; get it's size
        add.l   tmp3,rec_botmem         ; and accumulate it.
        endif
        move.l  #Object_grade_byteMask,tmp3 ; get the objects grade now.
        and.b   (thisRef),tmp3          ;
        cmp.b   tmp3,xg                 ; is it the same as before ?
        bne.s   2f                      ; no, go graduate bot.
        move.l  newList,botnxt(thisBot) ; link new list to this bot
        move.l  thisBot,newList         ; put bot at front of list.
        bra.s   1b                      ; next bot on old list.

2
        if      GCdebug
        sub.b   #16,tmp3                ; check  where it graduated to,
        cmp.b   tmp3,xg                 ; should be grade + 1!
        beq.s   90f                     ; else trap out,
        errtrap("BOT graduated to wrong grade.")
90      ds.w    0
        endif
        move.l  nextList,botnxt(thisBot) ; add this bot to the head
        move.l  thisBot,nextList        ; of the bot list for the next grade.
        bra.s   1b                      ; next bot on old list.

3       move.l  botdat(thisBot),FMaddr  ; set up parms to
        move.l  botsize(thisBot),FMsize ; free up the data area
        if      STATS
        add.l   #1,rec_botfree          ; count this bot
        add.l   FMsize,rec_botFM        ; and accumulate the size.
        endif
        jsr     freeMem                 ; discribed by this bot.
4
        if      GCdebug
        move.l  #-1,botsize(thisBot)    ; debug - Zap free bot entry.
        clr.l   botref(thisBot)         ; debug - Zap free bot entry.
        move.l  #-1,botdat(thisBot)     ; debug - Zap free bot entry.
        endif
        move.l  freeList,botnxt(thisBot) ; add this bot to head of
        move.l  thisBot,freeList        ; of the free bots list.
        bra     1b

5       move.l  newList,(gba)+          ; store new bot list for this grade,
        move.l  nextList,(gba)          ; updated list for next grade,
        move.l  freeList,botfree:w      ; and all the new free ones.
        if      STATS
        jsr     bot_stat2               ; write stats on BOT's
        endif
        movem.l (sp)+,UBregs            ; restore regs
        rts                             ; and we're done at last.


#undef xg
#undef tmp1
```

```
#undef tmp2
#undef tmp3
#undef FMsize
#undef nextBot
#undef grade
#undef gba
#undef FMaddr
#undef newList
#undef nextList
#undef freeList
#undef thisBot
#undef thisRef
#undef UBregs

*       compactGrade                                    */
*  This routine moves all the currently                 */
* active objects in grade to a new space.               */
* it then makes the currently active space the          */
* empty one for this grade. This has the effect         */
* of throwing away all the unused objects.              */
* The routine first finds the spaces to which           */
* objects are to be copied. This will depend on         */
* whether this is the oldest grade. It then             */
* copies all stuff refered to from the context          */
* stack. Next it copies all the stuff refered           */
* to by older objects. Then it loops alternatly         */
* copying all objects refered to by the stuff           */
* moved to the "to" and "grad" spaces. (if              */
* these are the same this is half a loop.)              */
* Finally it updates all pointers in younger            */
* objects that refered to objects in this               */
* grade. Oh Yes, It swaps the active and empty          */
* spaces for this grade.                                */
*
* compactGrade(grade)
*    int        grade ;
*    {
*    int                i ;
*    space      *toSpace ;              /* space to copy to.            */
*    object     *toSpaceScanned ;       /* pointer to unused to space   */
*    object     *gradSpaceScanned ;     /* pointer to unused grad space */
*    int          gradFlag ;            /* flag that gradroom is toroom */
*    remembered *RememberedSet;         /* remembered set for current grade */
*    int          remSetSize ;          /* the size of the RememberedSet*/
*    remembered *memorys ;
*    object *daobject ;
*
*    toRoom = &school[grade] ;
*    RememberedSet = toRoom->grdremem ;
*    remSetSize = toRoom->active->spclast - RememberedSet ;
*
*    toSpace = toRoom->inactive ;
*    toRoom->inactive = toRoom->active ;
*    toRoom->active = toSpace ;
*    toRoom->grdend = toSpace + sizeof(char *) ;
*    toSpaceScanned = toRoom->grdend ;
```

```
*     toRoom->grdremem = toSpace->spclast - sizeof(remembered) ;
*     toRoom->grdremem->remobj = 0 ;      /* mark end of remembered set */
*     toRoom->grdmax = toRoom->grdremem - EXTRASPACE ;
*     if ( grade == MAXGRADE )
*        {
*        gradRoom = toRoom ;
*        gradFlag = FALSE ;
*        gradSpaceScanned = gradRoom->grdend ;
*        }
*     else
*        {
*        gradRoom = school[grade+1] ;
*        gradFlag = TRUE ;
*        gradSpaceScanned = gradRoom->grdend ;
*        }
*
*     /* starter objects */
*     salvageRememberedObjects(grade, RememberedSet, remSetSize) ;
*     salvageInterpObjects(grade) ;
*     /* all the stuff refered to by younger objects */
*     for ( i = grade-1 ; i >= 0 ; --i )
*        {
*        scavengeSpaceStartingAt(grade, school[i].active+sizeof(char *)) ;
*        }
*     /* now the transitive closure of the sets. */
*     do  {
*        scavengeSpaceStartingAt(grade, toSpaceScanned ) ;
*        toSpaceScanned = toRoom->grdend ;
*        if ( gradFlag ) /* if max grade there is no grad space */
*            {
*            scavengeSpaceStartingAt(grade, gradSpaceScanned ) ;
*            gradSpaceScanned = gradRoom->grdend ;
*            }
*        } while ( toSpaceScanned != toRoom->grdend ) ;
*
*     /* every thing is moved, now see if anything overflowed.   */
*     if ( toRoom->grdend > toRoom->grdmax )
*        {
*        mustCompact = grade ;
*        }
*     if ( gradRoom->grdend > gradRoom->grdmax &&
*         gradRoom != toRoom )
*        {
*        mustCompact = grade+1 ;
*        }
*     /* Update remembered set elements which pooint to this grade */
*     updateRemembered(grade) ;
*     /* update BOT back pointers and free garbage remotes       */
*     updateBot(grade) ;
*
*     return ;
*     }

#define remSetSize d0
#define i d0
#define SSSAgrade d0
```

```
#define gradSpaceScanned d1
#define gradFlag d2
#define grade d6
#define toRoom d6
#define SSSAstart a0
#define toSpace a2
#define toSpaceScanned a2
#define rememberedSet a3
#define tmp1 a5
#define tmp2 a5
#define aGradRoom a5
#define aRoom a6
#define CGregs d0/d1/d2/d5/d6/a0/a2/a3/a5/a6

compactGrade
        movem.l CGregs,-(sp)            ; save working regs.
        if      STATS
        jsr     grade_stat1
        endif
        jsr     setGCcursor             ; set the cursor.
        move.w  grade,aRoom             ; get an address version
        add.w   grade,aRoom             ; of grade shifted by 4
        add.w   aRoom,aRoom             ; by adding to itself twice.
        if      GCdebug
        jmp     90f(PC,grade.w*2)       ; so I can break point on a grade
        global  gradeBreak
gradeBreak equ  *
90      bra.s   91f                     ; grade 0
        bra.s   91f                     ; grade 1
        bra.s   91f                     ; grade 2
        bra.s   91f                     ; grade 3
        bra.s   91f                     ; grade 4
        bra.s   91f                     ; grade 5
        bra.s   91f                     ; grade 6
        nop                             ; grade 7
91      add.l   #1,GCDB_collectedGrade:w(aRoom) ; mark number of sweeps
        move.l  GCDB_collectionsDone:w,GCDB_whenGrade:w(aRoom) ; which sweep
        cmp.b   GCDB_chkAft:w,grade     ; do we need to check ?
        blo.s   92f                     ; no.
        move.b  grade,GCDB_chkQ         ; tellem yes.
92
#define tmp0 d1
        move.l  school_grdremem:w(aRoom),tmp0
        cmp.l   school_grdend:w(aRoom),tmp0
        bgt.s   93f
        errtrap("Grade overflow")
93      ds.w    0
#undef tmp0
        endif
        move.l  school_grdremem:w(aRoom),rememberedSet
        move.l  school_inactive:w(aRoom),toSpace ; get where to copy stuff
        move.l  school_active:w(aRoom),tmp1 ; and where it is now.
        move.l  (tmp1),remSetSize       ; calc used remset elements.
        sub.l   rememberedSet,remSetSize
        move.l  tmp1,school_inactive:w(aRoom) ; exchange spaces.
        move.l  toSpace,school_active:w(aRoom)
```

```
        move.l  (toSpace)+,tmp2          ; get end of new space.
        move.l  toSpace,school_grdend:w(aRoom) ; and store first unused.
        clr.l   -(tmp2)                  ; zero out the last remset
        move.l  tmp2,school_grdremem:w(aRoom) ; and save next spot.
        sub.w   #EXTRASPACE,tmp2         ; add a safty net
        move.l  tmp2,school_grdmax:w(aRoom) ; and store as upper limit.
        bfins   grade,grade{0:16}        ; copy over grade to upper word.
        move.l  aRoom,aGradRoom          ; copy of toRoom index.
        clr.l   gradFlag                 ; zip flag says no grad space.
        cmp.w   #MAXGRADE,grade          ; is there really ?
        beq.s   1f                       ; yes, start copying.
        add.w   #1,grade                 ; bump to n't grade.
        swap    grade                    ; put in high word.
        add.l   #4,aGradRoom             ; and bump index.
        not.l   gradFlag                 ; reset flag.
1       move.l  grade,gradRoom:w         ; store grad and to rooms.
        move.l  school_grdend:w(aGradRoom),gradSpaceScanned ;
*       copy stuff refered to from outside the grade.
        jsr     salvageInterpObjects     ;  copy stuff the interpreter knows
        jsr     salvageRememberedObjects ; and copy the stuff refered to later.
*       now copy stuff refered to by lower grades.
        move.w  grade,i                  ; get the grade we are moving.
        sub.w   #1,i                     ; go down by one.
        bmi.s   3f                       ; if nursury (grade 0) then done.
2       move.l  school_active:w(i.w*4),SSSAstart ; get start of space.
        add.l   #4,SSSAstart             ; skip length word.
        jsr     scavengeSpaceStartingAt ; copy stuff the grade refers to.
        dbra    i,2b                     ; next lower grade.
*       now calculate transitive closure of stuff already copied.
3       move.l  toSpaceScanned,SSSAstart ; copy objects refered to by
        move.w  grade,SSSAgrade          ; get grade number.
        jsr     scavengeSpaceStartingAt ; already copied objects.
        move.l  school_grdend:w(aRoom),toSpaceScanned ; remember how much
        tst.b   gradFlag                 ; if no grad space
        beq.s   4f                       ; then dont scan it, and we are done.
        move.l  gradSpaceScanned,SSSAstart ; now copy objects refered to by
        move.w  gradRoom:w,SSSAgrade     ; the grad grade number.
        jsr     scavengeSpaceStartingAt ; by already graduated objects.
        move.l  school_grdend:w(aGradRoom),gradSpaceScanned ;
        cmp.l   school_grdend:w(aRoom),toSpaceScanned ; any more ?
        bne.s   3b                       ; yes round again.

*       all objects copied see if either the baker or grad space overflowed.
4       cmp.l   school_grdmax:w(aRoom),toSpaceScanned ; need to compact again ?
        blo.s   5f                       ; no, check grad space.
        move.w  grade,mustCompact:w      ; reset next grade to compact.
5       cmp.l   school_grdmax:w(aGradRoom),gradSpaceScanned ; now grad space?
        blo.s   6f                       ; no it's O.K.
        move.w  gradRoom:w,mustCompact:w ; bump up the next grade to compact.
6       tst.w   grade                    ; are there any younger grades ?
        beq.s   7f                       ; no dont try to update them then.
*       now update addresses in remembered sets for all younger grades.
        jsr     updateRemembered         ; routine to do it.
*       now throw away BOT objects no longer referenced.
7       jsr     updateBot                ; got a nifty routine to do this.
*       throw away the inactive space.
```

```
        if      UniFlex
        cmp.w   #1,grade                ; if the grade is very low
        ble.s   8f                      ; just leave the space
        move.l  school_inactive:w(aRoom),a0
        move.l  (a0)+,-(sp)             ; push end address on stack
        move.l  a0,-(sp)                ; and the beginning
        move.l  #5+32,-(sp)             ; the code to throw away
        move.w  #memman,-(sp)           ; and the syscall
        move.l  sp,a0                   ; get parm block addr
        sys     indx                    ; and do it indirect
        add.l   #14,sp                  ; throw away parms now
        endif
        if      UTek
        cmp.w   #1,grade                ; if the grade is very low
        ble.s   8f                      ; just leave the space
        move.l  school_inactive:w(aRoom),a0
        move.l  (a0),d1                 ; get end address
        sub.l   a0,d1                   ; convert to length
        syscall mremap,a0,a0,d1,M_PREVIOUS|M_RELEASE
        add.l   a0,d1                   ; re calculate the end
        move.l  d1,(a0)                 ; store back in new page.
        endif
*       keep track of the highest grade compacted.
8       cmp.w   maxCompacted:w,grade    ; is this one higher ?
        bls.s   9f                      ; no, dont save.
        move.w  grade,maxCompacted:w    ; else save this grade.
*       and then we were done.
9
        if      STATS
        jsr     grade_stat2
        endif
        movem.l (sp)+,CGregs            ; done, restore working regs.
        rts                             ; and return.

#undef remSetSize
#undef i
#undef gradSpaceScanned
#undef gradFlag
#undef tmp3
#undef grade
#undef toRoom
#undef rememberedSet
#undef aGradRoom
#undef SSSAstart
#undef SSSAgrade
#undef toSpace
#undef toSpaceScanned
#undef tmp1
#undef tmp2
#undef aRoom
#undef CGregs

*       salvageMemory                                   */
*  This routine is called when the memory               */
* allocator notices it is nearly out of space           */
* in the allocation area (grade-0 active).              */
```

```
* It bungles around and copies objects until    */
* either it has eliminated some, it has         */
* graduated them to an unfilled grade, or       */
* worst case it gets more memory in the oldest  */
* grade. mustCompact is reset by compactGrade   */
* and its subroutines as necessary.             */
*
* salvageMemory()
*    {
*    int  compacting, reallocmax ;
*
*    if ( mustCompact < 0 ) mustCompact = 0 ;
*    reallocmax = FALSE ;
*    for ( compacting = 0 ; compacting < mustCompact ; ++compacting )
*       {
*       compactGrade(compacting) ;
*       }
*    while ( mustCompact >= 0 )
*       {
*       compacting = mustCompact ;
*       mustCompact -= 1 ;
*       compactGrade(compacting) ;
*       if ( (mustCompact == MAXGRADE) &&
*            (compacting  == MAXGRADE) )        /* oldest filled up */
*           {
*           signal(LOWMEMORY) ;
*           realloc ( school[MAXGRADE].inactive,
*                     school[MAXGRADE].active->spclast -
*                     school[MAXGRADE].active +
*                     school[MAXGRADE].active->spclast -
*                     school[MAXGRADE].grdend ) ;
*           reallocmax = TRUE ;
*           }
*       }
*    if (reallocmax )
*           realloc ( school[MAXGRADE].inactive,
*                     school[MAXGRADE].active->spclast -
*                     school[MAXGRADE].active ) ;
*    }
*

#define reallocmax d2
#define compacting d6
#define compactTo d1
#define tmp1 d3
#define tmp2 d3
#define FMaddr a1
#define FMsize d1
#define GMsize d0
#define GMaddr a0
#define SMregs d0/d1/d2/d3/d6/a1/a2


        global  salvageMemory
        if      GCdebug
        extern  checkMemory
```

```
        extern  GCDB_collectionsDone,GCDB_collectedGrade,GCDB_whenGrade
        extern  GCDB_chkA,GCDB_chkB,GCDB_chkAft,GCDB_chkQ
        endif

salvageMemory
*       dont bother to                  ; save working regs.
        move.l  a6,CSP:w                ; save Context SP for later.
        if      GCdebug
        add.l   #1,GCDB_collectionsDone:w       ;
        tst.b   GCDB_chkB:w
        beq.s   91f
        jsr     checkMemory
91
        endif

        if      STATS
        jsr     collect_stat
        endif

        clr.w   maxCompacted:w          ; set the largest compacted grade.
        and.w   #7,d0                   ; mask grade which needs squishing.
        cmp.w   mustCompact:w,d0        ; is it larger than what we know?
        bmi.s   1f                      ; no, use old must compact value.
        move.w  d0,mustCompact:w        ; else reset mustCompact value.
1       clr.l   reallocmax              ; reset max grade full flag.
        clr.l   compacting              ; first grade to compact.
        move.w  mustCompact:w,compactTo ; how far til we're done.
        sub.w   #1,mustCompact:w        ; lower where we do next.
2       jsr     compactGrade            ; no scavenge this one.
        add.w   #1,compacting           ; and now the next higer one.
        cmp.w   compacting,compactTo    ; are we done already ?
        bge.s   2b                      ; and check again.
        bra.s   51f                     ; go get target grade.

        cmp.b   #MAXGRADE,compactTo     ; are we doing the max grade.
        bne.s   5f                      ; no, go check on next one.
3       move.w  mustCompact:w,compacting ; get next grade to compact
        cmp.b   #MAXGRADE,compacting    ; do we redo the max one
        bne.s   4f                      ; no, go do next one.
*       if we get here oldest grade filled up
        move.l  #-1,reallocmax          ; set max grade full flag.
        bsr     realloc                 ; go get a new space.
4       sub.w   #1,mustCompact:w        ; decrement the grade.
        move.l  school_grdend:w(,compacting.w*4),tmp2 ; dont compact if
        cmp.l   school_grdmax:w(,compacting.w*4),tmp2 ; some is left to
        blo.s   5f                      ; work with.
        jsr     compactGrade            ; go smash it.
        cmp.b   #MAXGRADE,compacting    ; are we doing the max grade.
        beq.s   3b                      ; no, go do next one.

5       clr.w   mustGraduate:w          ; clear forced grad flag.
        cmp.w   mustCompact:w,compacting ; if we gotta re-do a grade
        bne.s   51f                     ; signal that we must force
        move.w  #-1,mustGraduate:w      ; objects out of the grade.
51      move.w  mustCompact:w,compacting ; get grade to smash
        bpl.s   4b                      ; aha, we're done.
```

```
        tst.b   reallocmax              ; need to reallocate maxgrade ?
        beq.s   6f                      ; no, that's good.
        exception(low_mem)
        bsr.s   realloc                 ; go get a new space.
6       tst.b   bflg:w                  ; were big areas freed ?
        beq.s   7f                      ; no,
        move.b  #1,bflg:w               ; clear flag and
        move.l  bigfree:w,a1            ; pack the big list.
        jsr     packFree                ;
7       tst.b   sflg:w                  ; were little remote areas freed ?
        beq.s   8f                      ; no,
        move.b  #1,sflg:w               ; clear flag and
        move.l  smallfree:w,a1          ; pack the small list.
        jsr     packFree                ;
8       move.l  #-1,gradRoom:w          ; show not compacting anything.
        if      GCdebug
        tst.b   GCDB_chkA:w
        beq.s   92f
        jsr     checkMemory
92
        tst.b   GCDB_chkQ:w
        beq.s   93f
        jsr     checkMemory
93      clr.b   GCDB_chkQ
        endif
        jsr     restoreCursor           ; go back to normal cursor.
        move.w  maxCompacted:w,d0       ; return oldest for Allen.
        rts

realloc move.l  school_inactive+4*MAXGRADE:w,FMaddr
        move.l  (FMaddr),FMsize         ; get addr of space
        sub.l   FMaddr,FMsize           ; and calulate it's size.
        add.l   #$80000000,FMaddr       ; clear the oop bit.
        jsr     freeMem                 ; go free inactive space.
        move.l  gradParm+(7*8),GMsize   ; then the size of
        lsl.l   #5,GMsize               ; grade 6
        lsl.l   #5,GMsize               ; converted to K
        add.l   FMsize,GMsize           ; more than what we had.
        jsr     alloc                   ; go get it boy.
        move.l  GMaddr,tmp1             ; test return value.
        bne.s   90f                     ; if zero its baaad.
        errtrap("Out of memory in garbage collection.")
90      add.l   #$80000000,GMaddr       ; set the oop bit.
        add.l   GMaddr,GMsize           ; calulate end addr.
        move.l  GMsize,(GMaddr)         ; stor as first word.
        move.l  GMaddr,school_inactive+MAXGRADE*4:w
        rts

#undef reallocmax
#undef compacting
#undef tmp1
#undef tmp2
#undef FMaddr
#undef FMsize
#undef GMsize
#undef GMaddr
```

```
#undef SMregs
```

```
#include "../include/idioms.i"
#include "../include/registerDefinitions.i"
#include "../include/knownOops.i"
#include "../include/nextBytecode.i"
#include "../include/traceCalls.i"
#include "../include/exceptions.i"
#include "../include/contextStack.i"
        lib     objdefs/MemoryStructs.def
        lib     objdefs/Objects.def

        data

        extern  flushMessageCaches,purgeContextStack,salvageMemory,cacheSafeCounts
        extern  newActiveContext,nextException,initSpecialSends
        extern  compactGradeFlag
        global  GCexception

**** an attempt was just made to execute a byte code but a Garbage collection
**** /*exception*/ was signaled.  We need to stabalize everything, espically
**** the context stack so that the collector will be happy and so we can
**** restablish the state after the GC.

GCexception
        jsr     PAGE0(purgeContextStack)        if not, make it so

        lea     valueStackEmptyBase,a6_CSP
        move.l  d6_AC,(a6_CSP)+         save ACoop on stack
        bfffo   PAGE0(compactGradeFlag){0:8},d0  offset 0 corresponds to grade 7
        neg.l   d0
        addq.l  #7,d0                   ;7-offset is highest grade to compact
        clr.b   PAGE0(compactGradeFlag)
        jsr     salvageMemory
        bfclr   PAGE0(pending_exceptions){GC_request:1}
        cmp.b   #MAXGRADE,d0
        bne.s   10f
        move.l  d0,-(sp)                ;save highest processed grade
        jsr     initSpecialSends
        move.l  (sp)+,d0
10
*** check if we need to flush the message lookup cache
        moveq           #0,d1
        lea             PAGE0(cacheSafeCounts),a1
12 subq.l       #1,(a1)+        ;this grade cycled enough to svoid flushing cache?
        bpl.s           doflush ;positive count means flush required
        addq.l  #1,d1           ;increment grade number
        cmp.l           d0,d1           ;exceed highest processed grade?
        ble.s           12b             ;if not, iterate
*       reached highest grade, all counts were negative
        bra.s           noflush ;so, no flush needed
doflush
        jsr     flushMessageCaches
noflush

        move.l  OOP(nil),d5_nilOop
        move.l  valueStack_offStackAC,d6_AC
        move.l  PAGE0(ctlStackEmptyBase),sp     ;restore control stack
```

```
        jmp     PAGE0(newActiveContext)


        global  stackPurge_handler
stackPurge_handler
        jsr     PAGE0(purgeContextStack)        if not, make it so
        jmp     PAGE0(newActiveContext) need to setup execution state
```

```
        name    GCstats

#include "../include/defOS.i"
#include "../include/parameters.i"
        opt     exp

        if      STATS

        lib     objdefs/MemoryStructs.def
        lib     objdefs/GCVars.def
        lib     objdefs/Objects.def

        if      UniFlex
        lib     sysdef
        endif

        if      UTek
        lib     objdefs/syscall.def
        lib     objdefs/syscall.mac
        endif


        data

        global  _stat_init,collect_stat
        global  grade_stat1,grade_stat2
        global  rem_stat1,rem_stat2
        global  bot_stat1,bot_stat2

*       Initialize and open file
_stat_init
        link    a6,#-4
        if      UniFlex
        sys     create,fileName,$01B
        bcs.s   1f
        endif
        if      UTek
        syscall open,#fileName,#$0201,#$01b6
        bvs.s   1f
        endif
        move.l  d0,stat_file
1       unlk    a6
        rts

*       write record showing salvageMemory on grade - x
collect_stat
        move.l  d0,-(sp)
        move.l  d0,rec_request
        move.l  stat_file,d0
        bmi.s   1f
        if      UniFlex
        sys     write,rec_1,rec_1_len
        endif
        if      UTek
        syscall write,#rec_1,#rec_1_len
        endif
```

```
1       move.l  (sp)+,d0
        tst.l   initFlag
        bne.s   2f
        jmp     init_counts
2       rts

*       start of stats for collectGrade shows init size and zeros counts


#define grade d6
#define size d0
#define GS1regs grade/size

grade_stat1
        movem.l GS1regs,-(sp)           ; save some working regs.
        move.l  school_grdend:w(,grade.w*4),size
        sub.l   school_active:w(,grade.w*4),size
        move.l  size,rec_sgsize
        add.w   #1,grade
        move.l  school_grdend:w(,grade.w*4),size
        sub.l   school_active:w(,grade.w*4),size
        move.l  size,rec_snsize
        clr.l   rec_egsize
        clr.l   rec_ensize
        clr.l   rec_copied
        clr.l   rec_grad
        clr.l   rec_held
        clr.l   rec_merged
        clr.l   rec_msize
        clr.l   rec_cntxt
        movem.l (sp)+,GS1regs
        rts

#undef grade
#undef size
#undef GS1regs



*       CollectGrade show final size and counts of objects copied,
*       graduated or held over. All written to file.

#define grade d6
#define size d0
#define GS2regs grade/size

grade_stat2
        movem.l GS2regs,-(sp)
        move.l  grade,rec_grade
        move.l  school_grdend:w(,grade.w*4),size
        sub.l   school_active:w(,grade.w*4),size
        move.l  size,rec_egsize
        add.w   #1,grade
        move.l  school_grdend:w(,grade.w*4),size
        sub.l   school_active:w(,grade.w*4),size
        move.l  size,rec_ensize
```

```
        move.l  stat_file,d0
        bmi.s   1f
        if      UniFlex
        sys     write,rec_2,rec_2_len
        endif
        if      UTek
        syscall write,#rec_2,#rec_2_len
        endif
        clr.l   rec_grade               Debug stuff
        clr.l   rec_sgsize              clear all fields.
        clr.l   rec_snsize
        clr.l   rec_egsize
        clr.l   rec_ensize
        clr.l   rec_copied
        clr.l   rec_grad
        clr.l   rec_held
        clr.l   rec_merged
        clr.l   rec_msize
1       movem.l (sp)+,GS2regs
        rts

#undef grade
#undef size
#undef GS2regs

*       Remembered set statistics initialization.

#define room d6
#define RS1regs room

rem_stat1
        movem.l RS1regs,-(sp)
        clr.l   rec_rset
        move.w  room,rec_rset+2
        clr.l   rec_rssize
        clr.l   rec_rsptrs
        clr.l   rec_rscopy
        clr.l   rec_rsctx
        movem.l (sp)+,RS1regs
        rts

#undef room
#undef RS1regs


#define room d0
#define RS2regs room

*       Remembered set statistics write record
rem_stat2
        movem.l RS2regs,-(sp)
        move.l  stat_file,d0
        bmi.s   1f
        if      UniFlex
        sys     write,rec_3,rec_3_len
        endif
```

```
        if      UTek
        syscall write,#rec_3,#rec_3_len
        endif
1       movem.l (sp)+,RS2regs
        rts

#undef room
#undef RS2regs


*       initial counts record shows how many objects
*       are in each grade.

#define objCount d0
#define objSize d1
#define grade d2
#define not3 d3
#define currentObject a1
#define spaceEnd a2
#define R4regs d0/d1/d2/d3/a1/a2

init_counts
        movem.l R4regs,-(sp)            ; save regs.
        move.l  #!3,not3                ; rounding constant.
        move.l  not3,initFlag           ; show we did it once.
        move.l  #MAXGRADE,grade         ; first grade to scan.
1       clr.l   objCount                ; start at zero
        move.l  school_grdend:w(,grade.w*4),spaceEnd ; get end of space.
        move.l  school_active:w(,grade.w*4),currentObject ; get it's start.
        add.w   #4,currentObject        ; bump over size word.
        cmp.l   currentObject,spaceEnd  ; any to scan ?
        beq.s   3f                      ; no quit.
2       add.l   #1,objCount             ; bump the count of objects.
        move.w  objectSizeOffset(currentObject),objSize ; get objects size.
        add.w   #3,objSize              ; round up the
        and.w   not3,objSize            ; size of the object.
        add.w   objSize,currentObject   ; bump pointer to object to scan.
        cmp.l   spaceEnd,currentObject  ; done ?
        bmi.s   2b                      ; no, get this one too.
        move.l  objCount,rec_icnt(,grade.w*4)
3       dbra    grade,1b                ; on to next grade.
        move.l  stat_file,d0
        bmi.s   4f
        if      UniFlex
        sys     write,rec_4,rec_4_len
        endif
        if      UTek
        syscall write,#rec_4,#rec_4_len
        endif
4       movem.l (sp)+,R4regs
        rts

*       Start profiling info.
        global  _prof1,_prof2
        extern  addref,GCexception
r5scale equ     16
```

```
r5scalex equ    $01000
r5size  equ     $0900/r5scale

_prof1
        link    a6,#-4
        if      UniFlex
        sys     profil,addref,r5buff,r5size*2,r5scale
        endif
        if      UTek
        syscall profil,r5buff,r5size*2,addref,r5scalex
        endif
        unlk    a6
        rts

*       Write profiling info to the stats file.
_prof2
        link    a6,#-4
        move.l  stat_file,d0
        bpl.s   1f
        if      UniFlex
        sys     create,fileName,$01B
        bcs.s   2f
        endif
        if      UTek
        syscall open,#fileName,#$0201,#$01b6
        bvs.s   2f
        endif
        if      UniFlex
1       sys     write,rec_5,rec_5_len
        endif
        if      UTek
1       syscall write,#rec_5,#rec_5_len
        endif
2       unlk    a6
        rts

#undef objCount
#undef objSize
#undef grade
#undef not3
#undef currentObject
#undef spaceEnd
#undef R4regs


*       Big objects statistics initialization.

#define room d6
#define RS1regs room

bot_stat1
        movem.l RS1regs,-(sp)
        clr.l   rec_botgrd
        move.w  room,rec_botgrd+2
        clr.l   rec_botcnt
        clr.l   rec_botmem
```

```
        clr.l   rec_botfree
        clr.l   rec_botFM
        movem.l (sp)+,RS1regs
        rts

#undef room
#undef RS1regs


#define room d0
#define RS2regs room

*       Big objects statistics write record
bot_stat2
        movem.l RS2regs,-(sp)
        move.l  stat_file,d0
        bmi.s   1f
        if      UniFlex
        sys     write,rec_6,rec_6_len
        endif
        if      UTek
        syscall write,#rec_6,#rec_6_len
        endif
1       movem.l (sp)+,RS2regs
        rts

#undef room
#undef RS2regs

        global  rec_copied,rec_grad,rec_held,rec_merged,rec_msize,rec_cntxt
        global  rec_rssize,rec_rsptrs,rec_rscopy,rec_rsctx
        global  rec_botcnt,rec_botmem,rec_botfree,rec_botFM
        global  rec_1,rec_2,rec_3,rec_4,rec_5

rec_1           dc.w    1
                dc.w    rec_1_len
rec_request     dc.l    0
rec_1_len       equ     *-rec_1


rec_2           dc.w    2
                dc.w    rec_2_len
rec_grade       dc.l    0
rec_sgsize      dc.l    0
rec_snsize      dc.l    0
rec_egsize      dc.l    0
rec_ensize      dc.l    0
rec_copied      dc.l    0
rec_grad        dc.l    0
rec_held        dc.l    0
rec_merged      dc.l    0
rec_msize       dc.l    0
rec_cntxt       dc.l    0
rec_2_len       equ     *-rec_2

rec_3           dc.w    3
```

```
                    dc.w    rec_3_len
rec_rset            dc.l    0
rec_rssize          dc.l    0
rec_rsptrs          dc.l    0
rec_rscopy          dc.l    0
rec_rsctx           dc.l    0
rec_3_len           equ     *-rec_3

rec_4               dc.w    4
                    dc.w    rec_4_len
rec_icnt            dc.l    0
                    dc.l    0
                    dc.l    0
                    dc.l    0
                    dc.l    0
                    dc.l    0
                    dc.l    0
                    dc.l    0
rec_4_len           equ     *-rec_4

rec_5               dc.w    5
                    dc.w    rec_5_len
r5buff              ds.w    r5size
rec_5_len           equ     *-rec_5

rec_6               dc.w    6
                    dc.w    rec_6_len
rec_botgrd          dc.l    0
rec_botcnt          dc.l    0
rec_botmem          dc.l    0
rec_botfree         dc.l    0
rec_botFM           dc.l    0
rec_6_len           equ     *-rec_6

initFlag            dc.l    0
stat_file           dc.l    -1
fileName            fcc     "GCstats.tmp"
                    fcb     0

        endif

        global  izod2
izod2   ds.l    2
```

```
*       Variables used by the storage management routines.
*               The code was prototyped in C. The c code
*               follows as comments.
*
* bote *botfree      ;                  /* free list head for the BOT   */
*
* grade    *toRoom, *gradRoom ;         /* grade rooms to copy to.      */
*
* freemem  *bigfree, *smallfree ;
* char     *smallarea, *smallAreaEnd ;
*
* object *ActivatedContexts[MAXACT] ;   /* context addresses for used   */
* #define LASTACT &(ActivatedContexts[MAXACT])
* object **nextActContext = { ActivatedContexts } ;
* int   mustCompact  ;                  /* need to scavenge this grade  */
*-------------------------------------------------

#include "../include/parameters.i"

                lib     objdefs/MemoryStructs.def

                global  botfree,toRoom,gradRoom
                global  bigfree,smallfree,smallArea,smallAreaEnd
*               global  ActivatedContexts,nextActContext,LastAct
                global  mustCompact,maxCompacted,mustGraduate,compactFlag
                global  bbp,sbp
                global  bcp,scp
                global  bfc,sfcx
                global  bflg,sflg
                global  memend,memlimit
                global  robot,nullbot
                global  CSP
                global  remHash

gradRoom                ds.w    1       ; grade to graduate to
toRoom                  ds.w    1       ; grade to copy to
smallfree               ds.l    1       ; free memory list for small indir. obj
bigfree                 ds.l    1       ; free memory list for big obj.
sbp                     ds.l    1       ; \
bbp                     ds.l    1       ; \
scp                     ds.l    1       ; \
bcp                     ds.l    1       ;  - working vars for free search.
sfcx                    ds.w    1       ;  - working vars for free search.
bfc                     ds.w    1       ; /
sflg                    ds.w    1       ; /
bflg                    ds.w    1       ; /
smallArea               ds.l    1       ; bottom of area for small inirect obj.
smallAreaEnd            ds.l    1       ; end of above area
memend                  ds.l    1       ; end of allocated memory
memlimit                ds.l    1       ; end of all memory
*  folloing supplied by Allen.
* nextActContext        dc.l    *+4     ; where to save adr of next context
* ActivatedContexts     ds.l    MAXACT  ; table of context that have been used
* LastAct                       equ     *       ; end of table
*                       ds.l    4       ; overflow for allen
botfree                 ds.l    1       ; head of big obj free list
```

```
robot                   ds.l    1       ; pointer to reserve tank of bots.
nullbot                 dc.l    0,0,0,0 ; BOT entry for zero remote parts.
CSP                     ds.l    1       ; stash for CSP on entry.
mustCompact             ds.w    1       ; next grade to garbage collect.
compactFlag             dc.w    0       ; compactRem shouldnt do ovfl stuff.
maxCompacted            dc.w    0       ; oldest graded collected this pass.
mustGraduate            dc.w    0       ; flag that must grad obj in this grade
*****************************************************************
*       The school which holds the info for each grade in the           *
*       Storage manager. It is held as parallel arrays rather           *
*       than an array of structures to make indexing instructions       *
*       work better.                                                    *
*****************************************************************
*
* typedef struct
*     {
*     word_t      *grdend   ;     /* The last used word in the active.    */
*     word_t      *grdmax   ;     /* The overflow point in the active.    */
*     remembered *grdremem ;      /* pointer to remembered set            */
*     bote        *bigobjs ;      /* pointer to big obj list              */
*     space       *active ;       /* pointer to space holding objects     */
*     space       *inactive  ;    /* pointer to space available           */
*     int          oldAge ;       /* cycles until objects are promoted    */
*     } grade ;
*
* grade school[MAXGRADE] ;
*
                global  school_grdend,school_grdmax,school_grdremem
                global  school_bigobjs,school_active,school_inactive
                global  school_oldAge
school_grdend           ds.l    MAXGRADE+1
school_grdmax           ds.l    MAXGRADE+1
school_grdremem         ds.l    MAXGRADE+1
school_bigobjs          ds.l    MAXGRADE+1
school_active           ds.l    MAXGRADE+1
school_inactive         ds.l    MAXGRADE+1
school_oldAge           ds.b    MAXGRADE+1    ; includes grad above age.
*       Names for Grade zero allocation routines to use.
        global  allocNext,allocLimit
allocNext       equ     school_grdend
allocLimit      equ     school_grdmax

* some help for C
                global  _school_grdend,_school_grdmax,_school_grdremem
                global  _school_bigobjs,_school_active,_school_inactive
                global  _school_oldAge

_school_grdend  equ     school_grdend
_school_grdmax  equ     school_grdmax
_school_grdremem equ    school_grdremem
_school_bigobjs equ     school_bigobjs
_school_active  equ     school_active
_school_inactive equ    school_inactive
_school_oldAge  equ     school_oldAge

*****************************************************************
```

```
*       A hash table of bits for scratch pad use by                     *
*                compactRememberedSet                                   *
*************************************************************************
remHash         ds.l    REMSIZE


        if      GCdebug

        global  GCDB_collectionsDone,GCDB_collectedGrade,GCDB_whenGrade
        global  GCDB_chkA,GCDB_chkB,GCDB_chkAft,GCDB_chkQ,GCDB_stuff
        global  _GCDB_chkA,_GCDB_chkB,_GCDB_chkAft,_GCDB_stuff
GCDB_stuff              equ     *
_GCDB_stuff             equ     *
GCDB_collectedGrade     dc.l    0,0,0,0,0,0,0,0
GCDB_whenGrade          dc.l    -1,-1,-1,-1,-1,-1,-1,-1
GCDB_collectionsDone    dc.l    0
_GCDB_chkA              equ     *
GCDB_chkA               dc.b    0
_GCDB_chkB              equ     *
GCDB_chkB               dc.b    0
_GCDB_chkAft            equ     *
GCDB_chkAft             dc.b    MAXGRADE+1
GCDB_chkQ               dc.b    0

        endif
```

```
        name    gradParm

        global  gradParm,_gradParm

        data
        dc.q    0
_gradParm
gradParm
        dc.l    128,2
        dc.l    64,2
        dc.l    96,2
        dc.l    128,2
        dc.l    256,4
        dc.l    256,8
        dc.l    512,16
        dc.l    1500,16
```