THE TEXT OF OSPub

by

Christopher Strachey

and

Joseph Stoy

Oxford University

Technical Monograph PRG-9(t)

July 1972

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
Oxford.

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
Oxford. OX2 6PE.

ABSTRACT

The two volumes of this monograph comprise the complete text of an experimental operating system, and a commentary on it. It is published to illustrate the authors' papers describing the system (Stoy, J.E., and Strachey, C.: OS6 — an experimental operating system for a small computer; Comp. J. 15, Nos. 2 and 3, 1972), to give an example of an operating system written in a high level language, and to provide material for discussion about matters of style in programming.

FOREWORD

These books are a supplement to the authors' two papers on OS6 (a full reference is given in the abstract). Though the general design of the system is the work of the authors, other people have, of course, assisted with its implementation. The authors wish to make grateful acknowledgement to

Julia Bayman, Bijit Biswas, Malcolm Harper, Clifford Hones, Peter McGregor and Peter Mosses,

who have all (to a greater or lesser extent) helped with the writing of this system. Julia Bayman and Malcolm Harper have given particularly valuable assistance in preparing the system and these documents for publication. Nevertheless, the design errors which remain are the sole responsibility of the authors.

CONTENTS

(Note: items enclosed in square brackets appear only in the commentary; round brackets denote either a further level of subheading, or that the item also appears again, and is described, elsewhere.)

II: THE TEXT OF THE SYSTEM

II:1. The load-go loop, Run and Load

II:1,1 LGLOOP

```
    ||  This section defines the global routines
    ||  LoadGoLoop, LGLoop and DefaultProg.


 5
    let LoadGoLoop[] be
    §   Run[LGLoop]  repeat    §


10  let LGLoop[] be
    §L
        Reset[In]
        Prog := DefaultProg

15      OutS[Console, '*nOSPub']
        OutDateandTime[Console]
        Load[1]
        Run[Prog]

20      Reset[Output]
        Reset[ReportStream]
    §L


25
    let DefaultProg[] be
        OSReport[111, 'global 1 not set']
```

****

II:1,2 RUN

```
    ||  This section declares the global routines
    ||  Run, TerminateRun and Finish.

    static  $  Terminated = false  $
5


    let Run[Program] be
    $R
10      PrepareforRun[]
        Program[]
        TerminateRun[]  repeatuntil Terminated
    $R


15

    and PrepareforRun[] be
    $PR
        let R = NewVec[RSIZE]           ||  New RunBlock :
20      R!RPRE := FetchCode[RBLOCK]     ||    Predecessor
        R!PPTR := PPtr                  ||    Procedure Pointer
        R!IBLK := IBlock                ||    Information Block
        R!CPTR := CPtr                  ||    Code Pointer
        R!FSV  := FS                    ||    FreeStore Vector
25      R!INPT := In                    ||    Input Stream
        R!OUTP := Output                ||    Output Stream
        R!CON  := Console               ||    Console Stream
        R!REP  := ReportStream          ||    Report Stream
        R!GU   := GiveUp                ||    GiveUp Routine
30      R!GUS  := GiveUpStack           ||    Stack for GiveUp
        R!GUSS := GiveUpStackSize       ||    Size of stack needed
        R!CUC  := ClearUpChain          ||    Used by ClearUp

        NewFreeStore[]
35      CoretoDisc[FSF, FSF!PAGE]  ||  Update last page of FSFile

        unless ClearUpChain = ENDCUCHN do
          $ ClearUpChain!CPRE := lv R!CUC
            ClearUpChain := ENDCUCHN
40        $

        StoreCode[RBLOCK, R]
    $PR
```

```
45

    and TerminateRun[] be
    §TR Terminated := false
        ClearUp[]
50
      § let R = FetchCode[RBLOCK]
        GiveUpStackSize := R!GUSS
        GiveUpStack     := R!GUS
        GiveUp          := R!GU
55      ReportStream    := R!REP
        Console         := R!CON
        Output          := R!OUTP
        In              := R!INPT

60      Unload[R!CPTR]
        RestoreFreeStore[R!FSV]
        StoreCode[RBLOCK, R!RPRE]
        Terminated := true

65
        ClearUpChain := R!CUC
        unless ClearUpChain = ENDCUCHN do
            ClearUpChain!CPRE := lv ClearUpChain

70      unless (IBlock = R!IBLK)^(CPtr = R!CPTR) do
          § OSReport[121, 'TerminateRun wrong:*n
                          IBlock, CPtr = *N, *N', IBlock, CPtr]
            GiveUp[R]     §

75      unless R!RPRE = R do ReturnVec[R, RSIZE]

        if User = NULL do Run[LogIn]
    §TR

80

    and ClearUp[] be
    §CU until ClearUpChain = ENDCUCHN do
        §u let E1 = ClearUpChain
85         ClearUpChain := ClearUpChain!CSUC
           (E1!ROUTINE)[E1]
    §CU
```

```
90
    and LogIn[] be
    §LI OutS[Console, 'Please log in *n']
        LoadSystemFile['LogIn']
        Prog[]
95  §LI



100 let Finish[] be
    §F  PPtr := (FetchCode[RBLOCK])↓PPTR
        return
    §F

105
****
```

II:1.3 LOAD

```
    ||  This section defines the global routines
    ||  Load and Unload.


 5  manifest  ||  Loader Warning Characters
    §   CODE        = 1
        INTERIUDE   = 2
        BINARY      = 3
        DATA        = 4
10      TITLE       = 5
        NEWSECTION  = 6
        ENDLOAD     = 7
    §

15  manifest  ||  Interpreted code instruction
    §   GLOBJUMP    = #140127 §  ||  OG21 (Set Globals) ; GOTOC



20  static
    §   SectId = 100
        GlobalsUnset = true
    §

25
    let Load[NumberofEndloads] be
    §L
      §r
        let WCh = Next[In]    ||  Warning character
30      let Length = Next[In]
        switchon WCh into
         §s case TITLE:       ||  Skip Title
                 for i=1 to Length do WCh := Next[In]
                 endcase

35
             case NEWSECTION:
                 unless Length = 0 do
                   § OSReportN[131];  GiveUp[Length] §
                 LoadSection[SectId]
40               SectId := SectId+1
                 endcase
```

```
            case ENDLOAD:
                 unless Length = 0 do
45                 § OSReportN[132];  GiveUp[Length] §
                 if NumberofEndloads <= 1 return
                 NumberofEndloads := NumberofEndloads-1
                 endcase

50          default :
                 OSReport[133, 'Wrong WCh for Load']
                 GiveUp[WCh]
      §r repeat
    §L
55



60  and LoadSection[Ident] be
    §LS let WCh, Length = 0, 0

        let I = NewVec[ISIZE]     ||  New Information Block :
        I↓CP    := CPtr           ||    Code Pointer
65      I↓CL    := 0              ||    Code Length
        I↓DP    := NOTSET         ||    Data Pointer
        I↓DL    := 0              ||    Data Length
        I↓IPRE  := IBlock         ||    Predecessor
        I↓ISUC  := NONE           ||    Successor
70      I↓ID    := Ident          ||    Section Identifier

        StoreCode[CPtr, I]
        CPtr := CPtr+1
        CFirst := CPtr
75      IBlock↓ISUC := I
        IBlock := I
        GlobalsUnset := true

        §r
80          WCh := Next[In]
            switchon WCh into
             §s case CODE:
                case INTERLUDE:
                     Length := Next[In]
85                   if (CPtr+Length) > FetchCode[MAXC] do
                       § OSReport[134, 'Too much code']
                         GiveUp[1]     §
```

```
                    GlobalsUnset := true
90                  TransferInC[In, CPtr, Length]

                    test WCh = CODE
                      then CPtr := CPtr+Length
                      or § I↓CL := CPtr - (I↓CP + 1)
95                         CPtr[]  ||  enter interlude
                           GlobalsUnset := false
                        §
                    endcase

100

              case DATA:
                    unless I↓DP = NOTSET do
                      § OSReportN[135];  GiveUp[I] §
                        ||  more than one DBlock
105
                    Length := Next[In]
                  § let D = NewVec[Length]  ||  Data Block
                    D↓0 := I
                    I↓DP := D
110                 I↓DL := Length
                    TransferIn[In, D+1, Length]
                    endcase


115           default :
                    I↓CL := CPtr - (I↓CP + 1)
                    PutBack[In, WCh]
                    return  || to Load

120     §r  repeat
    §LS


125


130
```

```
    let Unload[c] be
135 §U
        if c < (FetchCode[RBLOCK])↓CPTR do
          § OSReport[136, 'Unload[*N]', c]
            GiveUp[0]     §

140     while c ≤ IBlock↓CP do
          §w
            if (FetchCode[(IBlock↓CP)+1]=GLOBJUMP)^ ~GlobalsUnset
                do SetGlobals[]    ||  to denest

145         unless IBlock↓DP = NOTSET do
                ReturnVec[IBlock↓DP, IBlock↓DL] ||  return DBlock

            IBlock := IBlock↓IPRE
            ReturnVec[IBlock↓ISUC, ISIZE]       ||  return IBlock
150       §w

        CPtr := IBlock↓CP + IBlock↓CL + 1
        SectId := IBlock↓ID + 1
        IBlock↓ISUC := NONE
155 §U


****
```

II:1.4 SETLAB

```
    ||  This section defines the routines SetLabels and
    ||  SetGlobals, which are normally called only by
    ||  the interlude of a program.

 5  manifest  ||  Relocating information
    $  CSEG         = 0
       SEGBIT       = 8100000
       SEGMASK      = 8 77777
       ENDLABCHN    = 8 77776
10     ||  BINARY is declared in 'LOAD'
    $


    let SetLabels[] be
15  $SL
        let C = IBlock!CP + 1  || Start of program's Code segment
        let D = IBlock!DP + 1  || Start of program's Data segment

        let WCh = Next[In]
20      unless WCh = BINARY do
          $ OSReportN[141]; GiveUp[WCh] $

        for i=1 to Next[In]/2 do
          $f let A = Next[In]
25          let R = Next[In]
            let ASeg = A^SEGBIT
            let RSeg = R^SEGBIT
            let Val = (A^SEGMASK) + (ASeg=CSEG -> C, D)
            let Ref = (R^SEGMASK)
30          test RSeg = CSEG
              then
                $C
                  until Ref = ENDLABCHN do
                    $u let Ptr = FetchCode[Ref+C]
35                     StoreCode[Ref+C, Val]
                       Ref := Ptr
                $C
              or
                $D
40                if IBlock!DP = NOTSET unless Ref = ENDLABCHN do
                    $ OSReportN[142]; GiveUp[IBlock] $
                      ||  DLabels but no DBlock
```

```
                until Ref = ENDLABCHN do
                  $u let Ptr = rv(Ref+D)
                     rv(Ref+D) := Val
                     Ref := Ptr
                $D
   $SL

   || SetGlobals nests the declared globals
   || It is also called by Unload to de-nest them.

   let SetGlobals[] be
   $SG
       let p = IBlock↓CP + IBlock↓CL
           || last word of program code
       let n = FetchCode[p]
           || number of globals to be set
       for i=1 to n do
         $f let Address = FetchCode[p-1]
           let Contents = FetchCode[p-2]
           StoreCode[p-2, rv Address]
           rv Address := Contents
           p := p-2
   $SG
```

II:2. Post-mortem arrangements and interrupts

II:2.1 GIVEUP

```
    ||  This section defines the global routines
    ||  ForcedGiveUp, StandardGiveUp, Dump and EndGiveUp.

    manifest  ||  Dumping information
 5  §    PRNVEC     =   33
         VEC        =   10
         C          =    3
         D          =    2
         DUMPSTART  = 3776
10  §

    let ForcedGiveUp[] be
    §FGU
        test InStack[PrivateStack↓0, GiveUpStack]
15        then PrivateStack↓0 := GiveUpStack↓0
          or   GiveUpStack↓0 := PrivateStack↓0

      §1 let R = FetchCode[RBLOCK]
         while InStack[R↓PPTR, GiveUpStack]     || Unravel
20         do §  TerminateRun[]                  ||  Runs in
                 R := FetchCode[RBLOCK]  §       ||  GiveUpStack.
      §1

         PPtr := GiveUpStack
25       Report§['Forced GiveUp']

         if GiveUp ‡ StandardGiveUp do
           unless GiveUpStackSize < GiveUpStack↓LENGTH do
           §SS let g = GiveUp
30             GiveUp := StandardGiveUp
                    ||  in case FreeStore runs out
               § let V = NewVec[GiveUpStackSize+1]
                 V↓0 := GiveUpStack↓0
                 V↓LENGTH := GiveUpStackSize+1
35               GiveUp := g
                 GiveUpStack := V
                 PPtr := V
           §SS

40       GiveUp[0]
    §FGU
```

```
45 and InStack[P, Stack] = (Stack ≤ P ≤ Stack + Stack↓LENGTH)


   let StandardGiveUp[n] be
50 §SG let Ch = 0
      OutS[Console, 'GiveUp *N *O,  Dump *q ', n, n]
      Ch := Next[Console]
          repeatuntil (Ch='Y')∨(Ch='y')∨(Ch='N')∨(Ch='n')
      test (Ch='Y')∨(Ch='y')
55      then Dump[]
        or StandardPM[]
      EndGiveUp['GiveUp', n]
   §SG

60

   and Dump[] be
   §D  let PRNVec = FetchExecWord[PRNVEC]
       let PRN8 = FetchExecWord[PRNVec + 8]
65     let CBounds = FetchExecWord[PRN8 + C]
       let DBounds = FetchExecWord[PRN8 + D]

       let UC, LC = CBounds^8377, CBounds rshift 8
       let UD, LD = DBounds^8377, DBounds rshift 8
70
       let CSize = (UC-LC)XPAGESIZE
       let DSize = (UD-LD)XPAGESIZE

       SetUpDummyExecStructure[CBounds, DBounds]
75     DumpSegment[CODESEG, DUMPSTART + LC, CSize]
       DumpSegment[DATASEG, DUMPSTART + LD, DSize]
       ReportS['Dumped']
   §D

80

   and FetchExecWord[Addr] = valof
   §FE let Ans = 0
       Exec[READABS, L, lv Ans, Addr]
85  L: resultis Ans
   §FE
```

```
    and SetUpDummyExecStructure[CBounds, DBounds] be
    §DE DiscPage↓PRNVEC := 0  ||  dummy PRN vector at 0
90      DiscPage↓8 := VEC     ||  PRN8
        DiscPage↓9 := VEC     ||  PRN9
        DiscPage↓(VEC+C) := CBounds
        DiscPage↓(VEC+D) := DBounds

95    § let x = FetchCode[DISCWRITEPERMITTED]
        StoreCode[DISCWRITEPERMITTED, true]
        CoretoDisc[DiscPage, DUMPSTART]
        StoreCode[DISCWRITEPERMITTED, x]
    §DE
100
    and DumpSegment[Seg, Page, Size] be
    §DS
        unless Page > DUMPSTART do
          § OSReportN[211]; Wait[] §
105
        §R
            let E = vec ESIZE    ||  Exec Block
            E↓DEVICE     := DISCWRITE
            E↓BUFFER     := 0
110         E↓BUFFSIZE   := Size
            E↓PAGENUMB   := Page
            E↓SEG        := Seg
            E↓ENDMODE    := QUIETEND
            E↓COMPLETED  := false
115         E↓SELFPTR    := E
            ||  The other two elements are not used.

            Exec[TRANSFER, Rj, lv E↓PAGENUMB, E↓PAGENUMB, E]
            §  §  repeatuntil E↓COMPLETED
120
            if E↓PAGENUMB = Page break
                ||  successful transfer

         Rj:||  If the call is rejected or terminates
125         ||  unsuccessfully, Exec overwrites E↓PAGENUMB
            ||  with a failure number.
            OutS[Console, 'Dump failure *O ', E↓PAGENUMB]
            Wait[]
        §R  repeat
130
    §DS
```

```
135
    let EndGiveUp[String, n] be
    §EGU
        OutS[Console, '*S *N *O ', String, n, n]
        Wait[]
140     ForcedFinish[]
    §EGU

145 and ForcedFinish[] be
    §FF
        Reset[DiadRead]
        Reset[ExecConsole]
        Finish[]
150 §FF

****
```

II:2,2 PM

```
    ||  This section defines the global routines StandardPM,
    ||  ReportCallTrace, and ReportFreeStoreState.


 5
    let StandardPM[] be
    §S  ReportCallTrace[2, 10]
        ReportFreeStoreState[FS]
    §S
10



    let ReportCallTrace[n, Max] be
15  §RC let P = PPtr
        let RP = (FetchCode[RBLOCK])↓PPTR
        ReportS['  P    LINK']
        for i = 1 to n do P := rv P
            ||  don't trace most recent n calls
20      until P ≥ RP do P := rv P
            ||  don't trace in GiveUpStack or PrivateStack
        for i = 1 to Max do
        §f  if P < RP break
            ReportS['*N *N', P, rv(P+1)]
25          P := rv P
        §f
        ReportS['P at last Run:  *N', RP]
    §RC


30

    and ReportFreeStoreState[f] be
    §RFS
        ReportS['FreeStore area *N to *N', f↓LB, f↓UB]
35      ReportBlocks['Free', f↓FBC]
        ReportWords ['Free', f↓FWC]
        ReportBlocks['Pending', f↓PBC]
        ReportWords ['Pending', f↓PWC]
    §RFS
40
```

```
    and ReportBlocks[String, b] be
45  $RB let n, w, m = 0, 0, 0
        if b = END return

        until b = END do
        $u  n := n+1
50          if b↓SIZE > m then m := b↓SIZE
            w := w + (b↓SIZE + 1)
            b := b↓NXTB
        $u
        ReportS['*N *S *S*n  *N words*n  Largest block size *N',
55                  n, String, (n=1 → 'Block', 'Blocks'), w, m]
    $RB


60  and ReportWords[String, w] be
    $RW let n = 0
        if w = END return

        until w = END do
65      $   n := n+1
            w := rv w    $
        ReportS['*N *S *S', n, String, (n=1 → 'Word', 'Words')]
    $RW

70
****
```

II:2.3 MPM

```
    ||  This section defines ManualPM (used by Interrupt)

    static § Con = 0 §

5
    let ManualPM[] be
    §MPM
        let Ch, A, N = 0, 0, 0
        §r1
10          let Exch, CSeg, Both, Ret
                = false, false, false, false

            OutS[Con, '*nMPM ']
                ||  Con is set in Interrupt
15
            §r2
                Ch := Next[Con]
                switchon Ch into
                  § case 'N':
20                  case 'n':  ForcedFinish[]

                    case 'E':
                    case 'e':  Exch := true
                               endcase
25
                    case 'Y':
                    case 'y':
                    case 'C':
                    case 'c':  CSeg := true
30                             ||  run on into next case
                    case 'B':
                    case 'b':  Both := true
                               endcase

35                  case 'R':
                    case 'r':  Ret  := true
                               break          || out of §r2
            §r2 repeatuntil ('0'<Ch<'9')v(Ch='-')

40          test Ret
             then Return[]
             or
```

```
            §NR
              A := MPMNextN[Con, Ch]
              ResetState[Con]
              §r3
                  N := CSeg → FetchCode[A], rv A

                  if N=0 unless Exch do
                  §Z  Out[Con, '*n']
                      §r4 A := A + 1
                          N := CSeg → FetchCode[A], rv A
                      §r4 repeatuntil N≠ 0
                  §Z

                  OutS[Con, (Both→ '*N *A *N', '*N *N'),A,N,N]

                  test Exch
                  then
                   §E OutS[Con, ' ']
                      Ch := Next[Con]  repeatwhile Ch = '*s'
                      if (Ch='N')∨(Ch='n') break  || out of §r3
                      Exchange[A, Ch, CSeg]
                   §E
                  or
                   §L Out[Con, '*n']
                      if AnythingTyped[Con] break || out of §r3
                   §L

                  A := A + 1
              §r3 repeat
            §NR
    §r1 repeat
§MPM

and Exchange[Addr, Ch, CSeg] be
§EX let N = 0

    §r  if (Ch='*n') return
        if ('0'≤Ch≤'9')∨(Ch='-') do
          § N := MPMNextN[Con, Ch]
            break  || out of §r
          §
        if (Ch='B')∨(Ch='b') do
          § N := MPMNextO[Con, Ch]
            break  || out of §r
          §
```

```
            Ch := Next[Con]
        $r  repeat
90
        test CSeg then StoreCode[Addr, N]
                  or  rv Addr := N
    $EX

95  and Return[] be
    $RET
        OutS[Con, 'Return *q ']
      § let Ch = Next[Con]
        unless (Ch='Y')v(Ch='y') return || to ManualPM
100
        OutS[Con, 'P, Link, Result (in Decimal) := ']
      § let P = MPMNextN[Con, NULL]
        let Link = MPMNextN[Con, NULL]
        let Result = MPMNextN[Con, NULL]
105
        let PP = PPtr
        let RunLevel = (FetchCode[RBLOCK])↓PPTR
        until  (PP=P)v(PP=RunLevel) do PP := rv PP
        unless PP=P do
110      $U OutS[Con, 'Wrong P*n']
            return  || to ManualPM
         $U

      § let Dummy = JumpTo[P, Link, Result]
115 $RET

    and JumpTo[P, Link, Result] = valof
    $J
        rv PPtr := P          ||  fiddle stack
120     rv (PPtr+1) := Link
        resultis Result       || Does the actual return
    $J

    and AnythingTyped[S] = valof
125 $AT
        let x = State[S]
        if x = NOTHINGTYPED resultis false
        if (x=RETURNt)^(S≠ExecConsole) do
          § ResetState[S]
130         resultis false §
        resultis true
    $AT
```

```
    and MPMNextN[S, FirstCh] = valof
    $NN
135     let Ch, n, Neg = FirstCh, 0, false

        $r1 if '0'<=Ch<='9' break  || out of $r1
            unless (Ch='*s')v(Ch='*4')
              do Neg := (Ch='-')
140         Ch := Next[S]
        $r1 repeat

        $r2 n := 10Xn + (Ch-'0')
            Ch := Next[S]
145     $r2 repeatwhile '0'<=Ch<='9'

        resultis Neg -> -n, n
        ||  Loses the terminating character
    $NN
150
    and MPMNext0[S, FirstCh] = valof
    $N0
        let Ch, m, n = FirstCh, 0, 0
        until '0'<=Ch<='7' do Ch := Next[S]
155
        $r1 n := 8Xn + (Ch-'0')
            Ch := Next[S]
        $r1 repeatwhile '0'<=Ch<='7'

160     until Ch = '*n' do  || look for second byte
        $U  if '0'<=Ch<='7' do
            $B  $r2 m := 8Xm + (Ch-'0')
                    Ch := Next[S]
                $r2 repeatwhile '0'<=Ch<='7'
165
                n := (n lshift 8) + m
                break  || out of $U
            $B
            Ch := Next[S]
170     $U
        resultis n
        ||  Loses the terminating character
    $N0
****
```

II:2.4 INTERRUPT

```
    ||  This section defines the global routine Interrupt


 5  manifest § UNUSED = 0      §

    static   § TempP = UNUSED §

10


    let Interrupt[] be
    §I
15      if TempP = UNUSED do TempP := PPtr
        PPtr := PrivateStack
        unless InStack[TempP, PrivateStack]
            do PrivateStack!0 := TempP
        TempP := UNUSED
20
      § let RFI = FetchCode[REASONFORINTERRUPT]
        StoreCode[REASONFORINTERRUPT, NOREASON]
        Con := Console  ||  Con is a static declared in MPM
        Reset[Con]
25      switchon RFI into
         §s case POWERON:                   ||  auto start
                Out[ExecConsole, '*n']
                ForcedFinish[]

30          case NOREASON:                  ||  probably violation
                OutS[Con, '*nVIOLATED;']
                endcase

            case EXECCONXON:                ||  X-ON from ExecConsole
35              Con := ExecConsole
                endcase

            default :                       ||  Remote console X-ON
                OutS[Con, '*n*nRUN 8;']
40      §s
```

```
        §r  let Ch = 0
45          Out[Con, BELL]
            ResetState[Con]
            Ch := State[Con] repeatwhile Ch = NOTHINGTYPED
            Ch := Ch ^ PARITYMASK
            switchon Ch into
50            § case 'L':     ||  LogIn
              case 'l':
                    User, CurrentIndex := NULL, NULL
                    ||  run on into next case

55            case 'G':     ||  Go
              case 'g':
                    OutS[Con, '*C*n', Ch]
                    ForcedFinish[]

60            case 'F':     ||  ForcedGiveUp
              case 'f':
              case 'M':     ||  ManualPM
              case 'm':
                  § let c = Console
65                Console := Con
                  Out[Con, Ch]
                  OutDateandTime[Con]
                  Console := c      §

70                test (Ch='M')v(Ch='m')
                    then ManualPM[]
                    or   ForcedGiveUp[]
        §r  repeat
    §I
75
****
```

II:3. Storage allocation and PutBack

II:3.1 FREESTORE

```
||  This section defines the global routines and functions
||  NewVec, ReturnVec, NewWord, ReturnWord, MaxVecSize,
||  NewFreeStore and RestoreFreeStore.

let NewVec[n] = valof
§NV
  · if n < 0 do
      § OSReport[311, 'NewVec[*N]', n]
        GiveUp[0]
      §
    if (n=0)^(FS↓FWC ‡ END) do
      § let w = FS↓FWC      ||  first word in FreeWordChain
        FS↓FWC := rv w
        resultis w
      §

  § let BP = lv FS↓FBC      ||  Block Pointer
    let B = rv BP           ||  Block
    §R
      if B = END do
        § OSReport[312, 'NewVec[*N] too big', n]
          GiveUp[FS]
        §
      if B↓SIZE ≥ n break
      BP := lv B↓NXTB
      B  := rv BP
    §R repeat

    test B↓SIZE ≤ n+1
     then
      § rv BP := B↓NXTB
        if B↓SIZE > n do ReturnWord[lv(B↓(n+1))]
      §
     or
      § let SB = lv(B↓(n+1))     ||  SurplusBlock
        SB↓NXTB := B↓NXTB
        SB↓SIZE := (B↓SIZE)-(n+1)
         || The order of the last two assignment statements
         || is important if n=0.    Here NXTB > SIZE.
```

```
            rv BP := SB
          §

      resultis B
45 §NV



   let ReturnVec[V, n] be
50 §RV
      unless FS↓LB ≤ V ≤ V+n ≤ FS↓UB do
      §Pend
          if n < 0 do RVGiveUp[313, V, n, 0]
          unless (V>FS↓UB)∨(V+n<FS↓LB)
55          do RVGiveUp[314, V, n, FS]

        § let f = FS
          while f > f↓FPRE do f := f↓FPRE  || seek first FS
          unless f = f↓FPRE do RVGiveUp[315, V, n, FS]
60        unless (f↓LB ≤ V)^(V+n ≤ f↓UB)
            do RVGiveUp[316, V, n, f]

          test n = 0
           then  § rv V := FS↓PWC
65                 FS↓PWC := V    ||  pending word
                §
           or    § V↓SIZE, V↓NXTB := n, FS↓PBC
                 FS↓PBC := V    ||  pending block
                §
70        return
      §Pend

     § let PW = lv FS↓FWC    ||  Previous Word
       let W = rv PW         ||  Word
75     until (W=END)∨(W ≥ V-1) do
         § PW := W
           W := rv PW
         §
       unless W = END do
80     §1  if W = V-1 do
             § W := rv W
               rv PW := W
               V := V-1
               n := n+1
85           §
```

```
            if V ≤ W ≤ V+n do RVGiveUp[317, V, n, FS]

            if W-1 = V+n do
               § rv PW := rv W
                 n := n+1
               §
         §1

      § let BP = lv FS↓FBC     || Block Pointer
        let B = rv BP          || Block

        until (B=END)∨(B+B↓SIZE ≥ V-1) do
           § BP := lv B↓NXTB
             B  := rv BP
           §
        unless B = END do
        §2  if B+B↓SIZE = V-1 do
            §3  B↓SIZE := B↓SIZE + (n+1)
                if (B↓NXTB)-1 = V+n do
                   § B↓SIZE := B↓SIZE + ((B↓NXTB)↓SIZE +1)
                     B↓NXTB := (B↓NXTB)↓NXTB
                   §
                return
            §3
            if B-1 = V+n do
            §4  V↓SIZE := n + (B↓SIZE +1)
                V↓NXTB := B↓NXTB
                || The order of the last two assignment
                || statements is important if n=0.
                rv BP := V
                return
            §4
            unless B > V+n do RVGiveUp[318, V, n, FS]
        §2

        test n = 0
         then  § rv V := W
                 rv PW := V
               §
         or    § V↓NXTB := B
                 V↓SIZE := n
                 rv BP := V
    §RV
```

```
    and RVGiveUp[r, V, n, g] be
      § OSReport[r, 'ReturnVec[*N, *N]', V, n]
135     GiveUp[g]
      §



140 let NewWord[] = NewVec[0]



    let ReturnWord[w] be ReturnVec[w, 0]
145



    let MaxVecSize[] = valof
    $MVS
150     let M = (FS↓FWC = END)→ NOSTORE, 0
        let B = FS↓FBC

        until B = END do
          § if B↓SIZE > M do M := B↓SIZE
155         B := B↓NXTB
          §
        resultis M
    $MVS

160



    let NewFreeStore[] be
    $NFS
        let f = NewVec[FSVECSIZE]     || new FSBlock
165     let MVS = MaxVecSize[]
        let A = NewVec[MVS]           || new area

        f↓FBC, f↓FWC := A, END
        f↓PBC, f↓PWC := END, END
170     f↓LB, f↓UB := A, (A+MVS)
        f↓FPRE := FS

        A↓NXTB := END
        FS := f
175 $NFS
```

```
    let RestoreFreeStore[FStore] be
    §RFS
        §1  let f = FS
            until (f≤FStore)∨(f≤f↓FPRE) do f := f↓FPRE
180         unless f=FStore do
            § OSReportN[319] ; GiveUp[FStore] §
        §1
        until FS=FStore do
        §U  test FS↓LB≤PBChain<FS↓UB
185           then || PutBack is in current area
                §2  let S = PBChain↓STREAM
                    let V = (S<0)→ (~S), S
                    test FS↓LB≤V<FS↓UB
                      then || Stream also in current area
190                     § PBChain := PBChain↓PBPRE
                          RestoreFreeStore[FS↓FPRE]
                        §
                      or
                        § let Ob = Next[S]  || to remove PutBack
195                       RestoreFreeStore[FS↓FPRE]
                          PutBack[S, Ob]
                  §2
              or
                §3  let f = FS
200               FS := FS↓FPRE
                  ReturnVec[f↓LB,(f↓UB - f↓LB)]

                § let x = f↓PBC          ||  Pending Blocks
                  until x = END do
205                 § let b = x
                      x := b↓NXTB
                      ReturnVec[b, b↓SIZE]
                    §
                  x := f↓PWC             ||  Pending Words
210               until x = END do
                    § let w = x
                      x := rv w
                      ReturnWord[w]
                    §
215
                  ReturnVec[f, FSVECSIZE]
                §3
        §U
    §RFS
****
```

II:3.2 PUTBACK

```
||  This section defines the global routine PutBack.


let PutBack[Str, x] be
$(PB
    let P = NewVec[PBSIZE]
    let S = (Str>0) -> Str, (~Str)!SLOWBLOCK

    P!OB := x
    P!STREAM := Str

    P!PBNEXT, P!PBENDOF := S!NEXT, S!ENDOF
    P!PBCLOSE, P!PBRESET := S!CLOSE, S!RESET
    P!PBSTR := S!STR

    if (Str<0) do
      $( P!PTR := (~Str)!INBFFPTR
         (~Str)!INBFFPTR := ((~Str)!INBFFEND)+1
      $)

    P!PBPRE := PBChain
    PBChain := P

    S!STR, S!ENDOF := P, EndofPB
    test (Str>0)
     then S!NEXT,S!CLOSE,S!RESET :=  NextPB, ClosePB, ResetPB
     or   S!NEXT,S!CLOSE,S!RESET := FNextPB,FClosePB,FResetPB
$)PB

and NextPB[S] = valof
$(NPB
    let P = S!STR
    let x = P!OB

    S!NEXT, S!ENDOF  := P!PBNEXT, P!PBENDOF
    S!CLOSE, S!RESET := P!PBCLOSE, P!PBRESET
    S!STR := P!PBSTR

    if P!STREAM < 0 do
        (~(P!STREAM))!INBFFPTR := P!PTR
```

```
      § let BP = lv PBChain   ||  Block Pointer.
 45     let B = rv BP           ||  Block.
        until (B=END)v(B=P) do
          § BP := lv B↓PBPRE
            B := rv BP
          §
 50     if B = END do
          § OSReportN[321];  GiveUp[FS] §

        rv BP := B↓PBPRE

 55     ReturnVec[P, PBSIZE]
        resultis x
    §NPB

    and FNextPB[FB] = NextPB[FB↓SLOWBLOCK]
 60

    and ClosePB[S] be
    §C  let x = NextPB[S]
        Close[S]
 65 §C

    and FClosePB[FB] be
    §FC let x = FNextPB[FB]
        Close[~FB]
 70 §FC


    and ResetPB[S] be
    §R  let x = NextPB[S]
 75     Reset[S]
    §R

    and FResetPB[FB] be
    §FR let x = FNextPB[FB]
 80     Reset[~FB]
    §FR


    and EndofPB[S] = false
 85
****
```

II:4, Stream primitives and I/O routines

II:4.1 STRPRIMITIVES

```
    ||  This section defines the global routines and functions
    ||  Close, Reset, State, ResetState and StreamError.


  5 let Close[S] be
    §C  test S > 0
          then (S↓CLOSE)[S]
          or § S := ~S
               ((S↓SLOWBLOCK)↓CLOSE)[S]
 10 §C

    let Reset[S] be
    §R  test S > 0
          then (S↓RESET)[S]
 15       or § S := ~S
               ((S↓SLOWBLOCK)↓RESET)[S]
    §R


 20 let Source[S] = valof
    §   if S < 0 do S := (~S)↓SLOWBLOCK
        resultis S↓SOURCE
    §

 25 let State[S] = valof
    §   let Sce = Source[S]
        resultis (Sce↓STATE)[Sce]
    §

 30 let ResetState[S] be
    §   let Sce = Source[S]
        (Sce↓RESETSTATE)[Sce]
    §

 35
    let StreamError[S] be
    §   OSReport[411, 'StreamError']
        GiveUp[S]
    §
****
```

II:4.2 OUTS

```
||  This section defines the global output routines
||  OutS, WriteS and ReportS, and also the
||  system routines OSReport and OSReportN.

let OutS[S, String, a,b,c,d,e,f,g,h,i,j,k,l,m,
                    n,o,p,q,r,s,t,u,v,w,x,y,z] be
    OutString[S, String, lv a]

and WriteS[String, a,b,c,d,e,f,g,h,i,j,k,l,m,
                   n,o,p,q,r,s,t,u,v,w,x,y,z] be
    OutString[Output, String, lv a]

and ReportS[String, a,b,c,d,e,f,g,h,i,j,k,l,m,
                    n,o,p,q,r,s,t,u,v,w,x,y,z] be
§   OutString[ReportStream, String, lv a]
    Out[ReportStream, '*n']
§


and OutString[S, String, ParamList] be
§OS let Ptr, Ch = 0, 0
    let i = 0
    let Length = (String↓0) rshift 8

    until Ptr = Length do
    §u Ptr := Ptr+1
       Ch := GetC[String, Ptr]

       test Ch ‡ '*'
        then Out[S, Ch]
        or
         §1 Ptr := Ptr+1
            if Ptr > Length do
             § OSReportN[421]; return §
            Ch := GetC[String, Ptr]

            test Ch = '*'
             then Out[S, '*']
             or
```

```
                §2 if i > 25 do
45                 § OSReportN[422];  return §

                   test Ch ≠ 'I'
                    then
                     §3 let Function = valof
50                      §V switchon Ch into
                         § case 'A':  resultis OutAddr
                           case 'B':  resultis OutByte
                           case 'C':  resultis Out
                           case 'N':  resultis OutN
55                         case 'O':
                           case 'o':  resultis OutO
                           case 'S':  resultis OutString
                           default :
                            OSReport[423,'OutString: ** *B',Ch]
60                          return
                        §V
                        Function[S, ParamList↓i]
                     §3
                    or
65                   §4 Ptr := Ptr+1
                        if Ptr > Length do
                         § OSReportN[424]; return §
                        Ch := GetC[String, Ptr]

70                      unless '0' ≤ Ch ≤ '9' do
                        § OSReport[425,'OutString: **I *B', Ch]
                          return §
                        OutJustify[S, ParamList↓i, (Ch-'0')]
                     §4
75                 i := i+1
    §u
§OS


80

and GetC[String, n] = (n rem 2 = 0) →
        (String↓(n/2))rshift 8, (String↓(n/2))^LMASK

85
```

```
90
    and OutJustify[S, n, FieldSize] be
    $OJ let Sp = FieldSize - Size[n]
        for i = 1 to Sp do Out[S, '*s']
        OutN[S, n]
95  $OJ


    and Size[n] =   (n > 9)       → 1 + Size[n/10],
                    (n ≥ 0)       → 1,
100                 (n ≠ -32768) → 1 + Size[-n],
                                    6
    ||  -32768 must be a special case
    ||  because of the machine representation.

105

    and OSReport[n, String, a,b,c,d] be
    $   OutS[ReportStream, 'OSReport *N:  ',n]
        ReportS[String, a,b,c,d]
110 $


    and OSReportN[n] be
        ReportS['OS Report *N', n]
115

****
```

II:4.3 OPRTS

```
    || This section defines the global output routines
    || OutAddr, OutByte, OutN, OutO,
    || Write, WriteAddr, WriteByte, WriteN and WriteO.

 5

    let OutP[S, n] be     || outputs positive number
    §P  if n > 9 do OutP[S, n/10]
        Out[S, '0' + n rem 10]
10  §P



    let OutN[S, n] be
15  §N  if n < 0 do
        §n  Out[S, '-']
            if n = -32768 do
              § OutString[S, '32768'] ; return §
              || special case because of machine representation
20          n := -n
        §n
        OutP[S, n]
    §N

25

    let Out9[S, x] be     || outputs 9 bits as octal number
    §9  Out[S, '0' + ((x rshift 6)^ 7)]
        Out[S, '0' + ((x rshift 3)^ 7)]
30      Out[S, '0' + (x ^ 7)]
    §9


    let OutO[S, n] be
35  §O  Out9[S, n rshift 9]
        Out9[S, n]
    §O


40  let OutByte[S, b] be
        Out9[S, b ^ 8377]
```

```
    let OutAddr[S, a] be
45  $A  OutByte[S, a rshift 8]
        Out[S, ':']
        OutByte[S, a]
    $A

50
    let Write[x] be Out[Output, x]

    let WriteN[n] be OutN[Output, n]

55  let WriteO[n] be OutO[Output, n]

    let WriteByte[b] be OutByte[Output, b]

    let WriteAddr[a] be OutAddr[Output, a]
60
****
```

II:4.4 NEXTN

```
    ||  This section defines the global functions
    ||  NextN and NextO.


  5 let NextN[S] = valof
    §N
        let Ch, n, Neg = 0, 0, false

        §r1 Ch := NextCh[S]
 10         if ('0'<Ch<'9') break
            unless (Ch='*s')v(Ch='*4')
              do Neg := (Ch='-')
        §r1 repeat

 15     §r2 n := 10Xn + (Ch-'0')
            Ch := NextCh[S]
        §r2 repeatwhile ('0'<Ch<'9')

        PutBack[S, Ch]
 20     resultis Neg → -n, n
    §N

    and NextO[S] = valof
    §O
 25     let Ch, n = 0, 0
        until ('0'<Ch<'7') do Ch := NextCh[S]

        §r  n := 8Xn + (Ch-'0')
            Ch := NextCh[S]
 30     §r  repeatwhile ('0'<Ch<'7')

        PutBack[S, Ch]
        resultis n
    §O
 35
    and NextCh[S] = valof
    §Ch
        if Endof[S] do
          § OSReportN[441];  GiveUp[S] §
 40     resultis Next[S]
    §Ch
****
```

II:5. Permanent input/output streams

II:5.1 TELETYPE

```
    ||  This section defines the routines and functions
    ||  which are used in 'SETUPTT'.


 5
    let NextTT[S] = valof
    §N  let p = S!LINEBFFPTR
        S!LINEBFFPTR := p+1
        if p < S!LINEBFFLAST resultis rv p
10
        ||  Otherwise refill the buffer.
      § let BuffStart = lv S!LINEBUFFER
        let BuffEnd   = lv S!TELESIZE
        let x = 0
15      p := BuffStart
        Out[S, BELLt]
        ResetState[S]

        §r  x := State[S] repeatwhile x = NOTHINGTYPED
20          ResetState[S]
            switchon x into
             §s default :
                    Out[S, x]  || echo
                    rv p := x
25                  p := p+1
                    if p > BuffEnd do
                      § OutNewline[S]
                        break
                      §
30                  endcase

                case RETURNt:
                case LINEFEEDt:
                    OutNewline[S]
35                  rv p := '*n'
                    p := p+1
                    break

                case ESCAPEt:   ||  terminate input
40                  break       ||  without 'newline'.
```

```
                case RUBOUTt:
                     unless p = BuffStart do
                       $ p := p-1    ||  trailing erase.
45                     Out[S, QUERYt]
                       $
                     endcase

                case XOFFt:
50                   p := BuffStart  ||  kill line.
                     Out[S, SHARPt]
                     ||  run on into next case

                case VTABt:          ||  does 'newline' on paper,
55                   OutNewline[S] ||  but doesn't input it.
                     endcase

        $r  repeat

60      S!LINEBFFLAST := p
            ||  element of buffer after last character
        S!LINEBFFPTR := BuffStart
        resultis Next[S]
    $N
65


    and OutNewline[S] be
    $   Out[S, RETURNt]
70      Out[S, LINEFEEDt]
    $


    let OutTT[S, x] be
75  $O  let i=0
        S!OUTBUFF := x
     Rj:Exec[TRANSFER, Rj, lv i, 0, lv S!WRITEBLOCK]
    $O

80

    let ResetTT[S] be
        S!LINEBFFPTR := S!LINEBFFLAST +1

85
    let EndofTT[S] = false
```

```
    let StateTT[S] = S↓INBUFF
90

    let ResetStateTT[S] be
    $R  let i=0
        S↓INBUFF := NOTHINGTYPED
95      Exec[TRANSFER, L, lv i, 0, lv S↓READBLOCK]
     L: return
    $R
```

****

II:5,2 INTTT

```
    ||  This section defines the global device stream function
    ||  IntcodetoandfromTeletype.

    manifest
 5  $   TTSIZE = 8
        TTSTR  = 7
        FLAG   = 8
    $

10  let IntcodetoandfromTeletype[Str] = valof
    $IT
        let S = NewVec[TTSIZE]
        S!NEXT   := NextTele
        S!OUT    := OutTele
15      S!CLOSE  := CloseTele
        S!RESET  := ResetTele
        S!ENDOF  := EndofTele
        S!TTSTR  := Str
        S!SOURCE := Source[Str]
20      resultis S
    $IT

    and NextTele[S] = valof
    $NT let x = 0
25      $r1 $r2 x := Next[S!TTSTR]
                if S = ExecConsole break || out of $r2
            $r2 repeatuntil EvenParity[S,x]
            x := x ^ PARITYMASK

30          if x = NEWLINEt resultis '*n'
            if x = PRIMEt resultis PRIME
            if x = UPARROWt resultis '↑'
            if ('*s'<x<LEFTARROW)\/('a'<x<'z') resultis x
            if x = BELL resultis BELL
35      $r1 repeat
    $NT

    and EvenParity[S,x] = valof
    $CP if Parity!(x rshift 4)= Parity!(x ^ 817) resultis true
40      OSReport[521, 'Wrong TT parity']
        TryAgain[S!TTSTR]
        resultis false
    $CP
```

```
45
   and OutTele[S, Ch] be
   §OT if S = ExecConsole do
        §E Reset[S]
           if Ch = '*n' do
50           § OutCRLF[S↓TTSTR]
               return
        §E
        if Ch = '*n' do
          §  S↓OUT, S↓FLAG := OutNewLines, false
55         return
          §
        if Ch = '*4' do
          §  for i=1 to 4 do Out[S, '*s']
             return
60        §
        Ch := Ch ^ UlMASK  ||  ignore underlining

        unless ('*s' ≤ Ch ≤ ']')∨('a' ≤ Ch ≤ 'z') do
          Ch := valof
65         §v switchon Ch into
              §    case RETURN :
                   case LEFTARROW :
                   case DELETE :     resultis Ch
                   case PRIME :
70                 case GRAVE :      resultis PRIMEt
                   case '↑' :        resultis UPARROWt
                   case 'X' :        resultis STARt
                   case '|' :
                   case DIV :        resultis SLASHt
75                 case '^' :        resultis AMPERSANDt
                   case RUNOUT :     resultis RUNOUTt
                   default :         resultis BELLt
           §v
        if Parity↓(Ch rshift 4) ≠ Parity↓(Ch ^ 817) do
80        Ch := Ch ∨ PARITYBIT
        Out[S↓TTSTR, Ch]
   §OT


85 and OutCRLF[S] be
   §    Out[S, RETURNt]
        Out[S, LINEFEEDt]
   §
```

```
90 and OutNewLines[S, Ch] be
   §NL
      if Ch = '*n' do
         § S↓FLAG := true
           OutCRLF[S↓TTSTR]
95         return
         §
      if S↓FLAG for i=1 to 40 do Out[S↓TTSTR, RUNOUTt]
      OutCRLF[S↓TTSTR]
      S↓OUT := OutTele
100   OutTele[S, Ch]
   §NL


   and CloseTele[S] be
105 §CT if S↓OUT = OutNewLines do OutCRLF[S↓TTSTR]
      Close[S↓TTSTR]
      ReturnVec[S, TTSIZE]
   §CT

110
   and ResetTele[S] be
   §RT if S↓OUT = OutNewLines do
         § S↓OUT := OutTele
           OutCRLF[S↓TTSTR]
115        §
      Reset[S↓TTSTR]
   §RT


120 and EndofTele[S] = Endof[S↓TTSTR]


   and Source[Str] = valof
   §S  if Str < 0 do Str := (~ Str)↓SLOWBLOCK
125    resultis Str↓SOURCE
   §S
****
```

II:5.3 XFER

```
    ||  This section defines the routine InitiateTransfer.

    manifest
 5  $(   PAUSE0 = 200
      PAUSE1 = 200
    $)


10  let InitiateTransfer[S, n, Message] be
    $(IT let r = 0
         let i = -PAUSE0
         let Last = PAUSE1 X S↓PINGS
         let HeldUp = false
15       let E = lv S↓EXECBLOCK

         E↓BUFFSIZE := n

         $(R  Exec[TRANSFER, Rj, lv r, 0, E]
20            break  ||  if transfer accepted.

          Rj:test 0 <= i < Last
               then if (i rem PAUSE1)=0 do Out[ExecConsole, BELL]
               or $(H if i = Last do
25                      $( HeldUp := true
                           ResetState[ExecConsole]
                         $)
                       if HeldUp ^ (State[ExecConsole]#NOTHINGTYPED)
                         do
30                       $( OutS[ExecConsole, '*S*n', Message]
                           ResetState[ExecConsole]
                    $)H
             i := i+1

35       $)R  repeat

         E↓BUFFER := (E↓BUFFER)↓(-1)
    $)IT

40
****
```

II:5.4 READER

```
    ||  This section defines the routines and functions
    ||  used in 'SETUPRDR' and the global routine TryAgain.


  5 manifest
    $   OPERABLE =    0
        OFFLINE  =   -1
        ONLINE   =    0
        PAUSE2   =  100
 10 $


    let ReadBuff[S] be
        InitiateTransfer[S, READBFFSIZE+1, 'Reader']
 15

    and FirstNextBFPT[FB] = valof
    $F  let S = FB!SLOWBLOCK
        let E = lv S!EXECBLOCK
 20     E!DEVICE := ReaderDev
        E!STOPCH := NOTBYTE
        ReadBuff[S]
        S!NEXT := NextFillBFPT
        FB!INBFFPTR, FB!INBFFEND := EMPTY, EMPTY
 25   $ let Rubbish = Next[~FB]
        resultis Next[~FB]
    $F


 30 and NextFillBFPT[FB] = valof
    $N  let S = FB!SLOWBLOCK
        ReadBuff[S]
        FB!INBFFPTR := S!(EXECBLOCK+BUFFER)
        FB!INBFFEND := FB!INBFFPTR + READBFFSIZE
 35     resultis Next[~FB]
    $N


    and ResetBFPT[FB] be
 40 $R  let S=FB!SLOWBLOCK
        FB!INBFFPTR, FB!INBFFEND := EMPTY, EMPTY
        S!NEXT := ReaderOffLine[] -> FirstNextBFPT, NextWaitBFPT
    $R
```

```
45 and NextWaitBFPT[FB] = valof
   §W  let S = FB↓SLOWBLOCK
       let i, Ch = 0, 0
       let NeverOff = true
       ResetState[ExecConsole]
50
       §R  test ReaderOffLine[]
             then NeverOff := false
             or unless NeverOff break
           Ch := State[ExecConsole]
55         if ((Ch=RETURNt)∨(Ch=LINEFEEDt)) ^ NeverOff break

           test 0 ≤ i < PAUSE2 X S↓PINGS
             then if (i rem PAUSE2)=0 do Out[ExecConsole, BELL]
             or if Ch ‡ NOTHINGTYPED
60               do OutS[ExecConsole, 'Reader*n']
           if Ch ‡ NOTHINGTYPED do ResetState[ExecConsole]
           i := (i<0)→ 0, i+1
       §R  repeat

65     resultis FirstNextBFPT[FB]
   §W


   and StateBFPT[S] = ReaderOffLine[] → OFFLINE, ONLINE
70

   and ReaderOffLine[] = valof
   §0  let Status=0
       Exec[LOOKATRDR, L, lv Status]
75     resultis Status ‡ OPERABLE
    L: resultis true
   §0


80 and EndofBFPT[FB] = false



   let TryAgain[F] be
85 §TA unless F = BytesfromPT do
          § OSReport[541, 'TryAgain on wrong stream']
            GiveUp[F]
          §
```

```
    § let FB = ~F
90    let S = FB↓SLOWBLOCK
      let E = lv S↓EXECBLOCK
      unless S↓CLOSE = S↓RESET do
          § OSReportN[542]; GiveUp[F] §  || PutBack on BFPT

95    OutS[ExecConsole, 'Try again*n']

    § let Ptr, End = FB↓INBFFPTR, FB↓INBFFEND
      S↓NEXT := FirstNextBFPT
      FB↓INBFFPTR, FB↓INBFFEND := EMPTY, EMPTY
100   E↓DEVICE := RDRDEVSUM - ReaderDev  ||  change direction

    § let k = End - Ptr
      let First = (k ≤ 1)→ End-5,
                  (Ptr-3 < E↓BUFFER)→ E↓BUFFER, Ptr-3
105
      for i = First to First+4 do
      §f  OutByte[ExecConsole, rv i]
          if i = Ptr-1 do OutS[ExecConsole, ' wrong']
          Out[ExecConsole, '*n']
110   §f

    § let x = rv (Ptr-1)  ||  offending character
      let Message = 'Backread'
      InitiateTransfer[S, 2, Message]
115   InitiateTransfer[S, READBFFSIZE+1, Message]

    § let y = 0
      let Ly = E↓BUFFER + k
      rv Ly := NOTBYTE
120   InitiateTransfer[S, k+2, Message]
      y := rv Ly  repeatwhile y = NOTBYTE

      test x = y
        then OutS[ExecConsole, 'Char still same']
125     or   OutS[ExecConsole, 'Char now *B', y]
      Wait[]
  §TA


130

****
```

II:5.5 DIADS

```
    ||  This section defines the global stream function
    ||  WordsfromDiads, and the global routine
    ||  TryDiadAgain.

 5

    manifest  ||  Diad Stream
    $(    DSIZE        = 5
          DIADBFFSIZE  = 30
10        DIADBFF      = 6

              ||  Diad Characters
          STOPCODE     = 4
          ESC          = '*33
          ERASE        = '*377
15        WCH          = '*125
    $)


20

    let WordsfromDiads[Str] = valof
    $(WD  let FB = NewVec[FBSIZE]
          let S = NewVec[DSIZE + DIADBFFSIZE]
25

          S!NEXT   := NextBlock
          S!ENDOF  := EndofW
          S!CLOSE  := CloseW
          S!RESET  := ResetW
30        S!STR    := Str
          S!OUT    := StreamError

          FB!SLOWBLOCK := S
          FB!INBFFPTR,  FB!INBFFEND  := EMPTY, EMPTY
35        FB!OUTBFFPTR, FB!OUTBFFEND := EMPTY, EMPTY
          FB!INESC, FB!OUTESC := NOTUSED, NOTUSED

          resultis ~FB
    $)WD
40
```

```
45
   and NextBlock[FB] = valof
   $NB let S = FB!SLOWBLOCK
       let B = lv S!DIADBFF
       let I = S!STR
50
       let Checksum, Blocksize = 0, 0
       let Ch, LByte, RByte = 0, 0, 0

       $L1 $L2
55             Ch := NextCh[I] repeatwhile Ch=NULL
               if Ch = WCH break
               test Ch = STOPCODE
                then Reset[I]
                 or  $ OSReport[551, 'Wrong diad WCH']
60                   TryAgain[I]
           $L2 repeat

           Blocksize := NextByte[I]
           if Blocksize > DIADBFFSIZE do
65           $ OSReportN[552]; GiveUp[Blocksize] $
           Checksum := Blocksize

           for i = 0 to Blocksize-1 do
           $f  LByte := NextByte[I]
70             RByte := NextByte[I]
               B!i := (LByte lshift 8)v RByte
               Checksum := Checksum + LByte + RByte
           $f

75         Ch := NextByte[I]
           Ch := (Ch lshift 8)v NextByte[I]
           if Ch = Checksum break
           OSReport[553, 'Diad sumcheck failed']
           TryDiadAgain[I, Ch, Checksum]
80     $L1 repeat

       FB!INBFFPTR := B
       FB!INBFFEND := B + Blocksize
       resultis Next[~FB]
85 $NB
```

```
90

95

    and NextByte[S] = valof
    $   let x = NextCh[S]
        resultis (x = ESC) → (NextCh[S]+1), x
100 $

    and NextCh[S] = valof
    $   let x = 0
        x := Next[S]  repeatwhile x = ERASE
105     resultis x
    $


    and EndofW[FB] = Endof[(FB↓SLOWBLOCK)↓STR]
110
    and ResetW[FB] be
    $R  let S = FB↓SLOWBLOCK
        FB↓INBFFPTR, FB↓INBFFEND := EMPTY, EMPTY
        Reset[S↓STR]
115 $R

    and CloseW[FB] be
    $C  let S = FB↓SLOWBLOCK
        Close[S↓STR]
120     ReturnVec[S, DSIZE + DIADBFFSIZE]
        ReturnVec[FB, FBSIZE]
    $C


125

130
```

```
    let TryDiadAgain[F, Read, Comp] be
    $TDA
135     unless F = BytesfromPT do
          $ OSReportN[554];  GiveUp[F] $

      $ let FB = ~F
        let S = FB↓SLOWBLOCK
140     let E = lv S↓EXECBLOCK

        OutS[ExecConsole, 'Try diad again*n']

      $ let k = FB↓INBFFEND - FB↓INBFFPTR
145     S↓NEXT := FirstNextBFPT
        FB↓INBFFPTR, FB↓INBFFEND := EMPTY, EMPTY

        ReadBackwards[S, 3]
        ReadBackwards[S, READBFFSIZE]
150     ReadBackwards[S, k+1]

        E↓STOPCH := WCH
        $R  S↓BFF1END := false
            S↓BFF2END := false
155         ReadBackwards[S, READBFFSIZE]
        $R  repeatwhile S↓BFF1END v S↓BFF2END
        E↓STOPCH := NOTBYTE

        OutS[ExecConsole, 'Sum read *O, computed *O', Read, Comp]
160     Wait[]
    $TDA


    and ReadBackwards[S, n] be
165 $RB
        let Message = 'Backread'
        let E = lv S↓EXECBLOCK

        E↓DEVICE := RDRDEVSUM - ReaderDev
170     E↓COMPLETED := false

        InitiateTransfer[S, n+1, Message]
        $  $  repeatuntil E↓COMPLETED
    $RB
175
****
```

II:6. Miscellaneous facilities

II:6.1 CLOCK

```
    ||  This section defines the global routine OutDateandTime,
    ||  and the global function TimeofDay.


 5
    static  §S
    ClockRestarted = false
            §S

10
    let OutDateandTime[S] be
    §DT OutS[S, ' ************** ']
        OutDate[S, FetchCode[DATE]]
        OutTime[S]
15      OutS[S, '*n']
    §DT


    and OutDate[S, d] be
20      OutS[S, '*N/*N/*N',
                d rshift 11, (d rshift 7)^817, d^8177]


    and OutTime[S] be
25  §OT let t = TimeofDay[]
        if ClockRestarted ^ (S=Console) return
      § let h, m = t/60, t rem 60
        OutS[S, ' *N.*N*N', h, m/10, m rem 10]
    §OT
30


    let TimeofDay[] = valof
    §T  let Clock = FetchCode[TIMEOFDAYCLOCK]
35      ClockRestarted := false
        RestartClock[Clock]
        resultis MINSPERDAY - (Clock↓BUFFER)↓TIME
    §T
```

```
40

45
   and RestartClock[E] be
   $R let i=0
      Exec[TRANSFER, Rj, lv i, 0, E]
      ClockRestarted := true
50    (E↓BUFFER)↓TIME := MINSPERDAY - AskTime[Console]
    Rj:|| assumed already going.
   $R

55 and AskTime[Con] = valof
   $A OutS[Con, '*nTime *q ']
     $ let Hrs = NextN[Con]
      let Mins = NextN[Con]
      resultis Hrs X 60  + Mins
60 $A
```

****

II:6.2 MISC

```
    ||  This section defines the global routines
    ||  AddressZero, Copy, EqS, NullProgram, and Wait.


  5 let AddressZero[] be
      § OSReport[621, 'Jump to address Zero']
        GiveUp[0]
      §

 10
    let Copy[V1, V2, n] be
      § let FB = vec FBSIZE
        FB↓INBFFPTR, FB↓INBFFEND := V1, V1+n
        FB↓INESC := NOTUSED
 15     TransferIn[~FB, V2, n]
      §


    let EqS[p, q] = valof
 20 §E  unless p↓0 = q↓0 resultis false
        for i = 1 to (p↓0 rshift 8)/2 do
            unless p↓i = q↓i resultis false
        resultis true
    §E
 25

    let NullProgram[] be § §


 30 let Wait[] be
      § let Ch = 0
        Reset[Console]
        Ch := Next[Console] repeatuntil Ch = '*n'
      §
 35
****
```

II:7. Disc routines (1)

II:7.1 DISCXFER

```
    || This section defines the global routines
    || CoretoDisc and DisctoCore.

 5  manifest § OFFSET = -10 §

    let CoretoDisc[v, p] be
    §CD unless v = DiscPage do Copy[v, DiscPage, PAGESIZE]
10      DiscPage↓SUMCH := Sumcheck[]
        unless FetchCode[DISCWRITEPERMITTED] return
        DiscTransfer[DISCWRITE, p]
    §CD

15
    and DisctoCore[v, p] be
    §DC
        §r  DiscTransfer[DISCREAD, p]
          § let x, y = Sumcheck[], DiscPage↓SUMCH
20          if (x=y)∨FetchCode[SUMCHINHIBITED] break
            OutS[Console, 'Sumcheck page *N:  computed *O,*s
                          recorded *O ', p, x, y]
            Reset[Console]
        §r  repeatuntil Next[Console] = 'I'
25
        unless v = DiscPage do Copy[DiscPage, v, PAGESIZE]
    §DC


30
    and DiscTransfer[Dev, p] be
    §T  let E = DiscPage + OFFSET  || ExecBlock for Disc
        E↓DEVICE := Dev

35      §r  for i = 1 to 2 do
            §f  E↓PAGENUMB := p
                E↓COMPLETED := false
                Exec[TRANSFER, Rj, lv E↓PAGENUMB, E↓PAGENUMB, E]
                § § repeatuntil E↓COMPLETED
```

```
40
                if E↓PAGENUMB = p return  || successful transfer

            || If the transfer ends unsuccessfully, Exec
            || overwrites E↓PAGENUMB with a failure number.
45
                if i = 2 break || out of §f
                Message[ExecConsole, E↓PAGENUMB, p]
                OutDateandTime[ExecConsole]
            §f
50      Rj: Message[Console, E↓PAGENUMB, p]
            Reset[Console]
        §r  repeatuntil Next[Console] = 'I'
    §T

55
    and Message[S, f, p] be
        OutS[S, 'Disc failure *D page *N ', f, p]
```

****

II:7.2 DISCFS

```
    || This section defines the routines
    || NewDiscBlock and ReturnDiscBlock.


5

    let NewDiscBlock[] = valof
    $N
        $r
10          let Scan = FSF↓SCAN
            test Scan = PAGEEND
              then
                $1 test FSF↓BACKLINK = ENDBODY
                     then $ OSReport[721, 'Disc full']
15                          GiveUp[0]
                          $
                     or   $ let Addr = FSF↓PAGE

                            let Link = FSF↓BACKLINK
20                          DisctoCore[FSF, Link]
                            FSF↓NXTPAGE := ENDBODY
                            CoretoDisc[FSF, Link]
                            FSF↓PAGE := Link
                            FSF↓SCAN := FIRSTENTRY
25                          $ let d = FetchCode[DATE]
                            UpdateHead[FSF↓THISFILE, SAME,
                                  FSF↓SERIAL, Link, SAME, d, d]
                            resultis Addr
                $1
30              or
                $2 let Addr = FSF↓Scan
                   FSF↓SCAN := Scan + 1
                   unless Addr = CLAIMED do
                    $3  FSF↓Scan := CLAIMED
35                      DisctoCore[DiscPage, Addr]
                        if DiscPage↓0 = FREE resultis Addr
        $r  repeat
    $N

40
```

```
45 let ReturnDiscBlock[Addr] be
   §R
      let Scan = FSF↓SCAN
      test Scan = FIRSTENTRY
        then
50       §1 FSF↓NXTPAGE := Addr
            CoretoDisc[FSF, FSF↓PAGE]
            FSF↓SERIAL := FSF↓SERIAL + 1
            FSF↓NXTPAGE := ENDBODY
            FSF↓BACKLINK := FSF↓PAGE
55          FSF↓PAGE := Addr
            FSF↓SCAN := PAGEEND
            for i = FIRSTENTRY to LASTENTRY do FSF↓i := CLAIMED
            CoretoDisc[FSF, Addr]
          § let d = FetchCode[DATE]
60          UpdateHead[FSF↓THISFILE, SAME, FSF↓SERIAL,
                                Addr, SAME, d, d]
         §1
        or
         §2 FSF↓(Scan - 1) := Addr
65          FSF↓SCAN := Scan - 1
            DiscPage↓0 := FREE
            CoretoDisc[DiscPage, Addr]
   §R

****
```

II:7.3 DISCIN

```
    ||  This section defines the global Stream functions
    ||  InfromFile and EntriesfromFile.


  5 manifest § EFSIZE = 5 §


    ||  The routine CheckType is used by both
    ||  InfromFile and OuttoFile :
 10
    let CheckType[f,h] be
    §C  if h = NULL do
          § OSReport[731, 'File deleted']
            GiveUp[f] §
 15       unless (h↓TYPE ^ STREAMABLE)= STREAMABLE do
            § OSReportN[732]
              GiveUp[f] §
    §C


 20


    let InfromFile[f] = valof
    §IF
        let h = FindHeading[f]
 25     CheckType[f, h]
      § let FirstPage = h↓FIRSTPAGE
        ReturnVec[h, h↓0 ^ LMASK]
        Update[f, DLR, FetchCode[DATE]]

 30   § let FB = NewVec[FBSIZE]
        let  S = NewVec[IFSIZE]

        FB↓SLOWBLOCK := S
        FB↓INBFFPTR, FB↓INBFFEND := EMPTY, EMPTY
 35     FB↓OUTBFFPTR, FB↓OUTBFFEND := EMPTY, EMPTY
        FB↓INESC, FB↓OUTESC := NOTUSED, NOTUSED

        S↓NEXT  := NextBuffIF
        S↓ENDOF := EndofIF
 40     S↓CLOSE := CloseIF
        S↓RESET := ResetIF
        S↓OUT   := StreamError
```

```
        S↓FILENUMB := f
        (S+IFBUFFER)↓NXTPAGE := FirstPage
45
        resultis ~FB
    $IF


50  and NextBuffIF[FB] = valof
    $N  if EndofIF[FB] resultis ENDOFSTREAMCH
            ||  EndofIF refills buffer
        resultis Next[~FB]
    $N
55

    and EndofIF[FB] = valof
    $E  let S = FB↓SLOWBLOCK
        let B = lv S↓IFBUFFER
60      if (B↓NXTPAGE = NULLBODY)∨(B↓NXTPAGE = ENDBODY) then
            resultis true
        DisctoCore[B, B↓NXTPAGE]
        FB↓INBFFPTR := lv B↓FIRSTDATA
        FB↓INBFFEND := (B↓NXTPAGE = ENDBODY) →
65                     lv B↓(B↓ENDOFDATAPTR), (lv B↓LASTDATA)+1

        resultis (FB↓INBFFPTR = FB↓INBFFEND)
    $E


70
    and CloseIF[FB] be
    $C  ReturnVec[FB↓SLOWBLOCK, IFSIZE]
        ReturnVec[FB, FBSIZE]
    $C
75

    and ResetIF[FB] be
    $R  let S = FB↓SLOWBLOCK
        let h = FindHeading[S↓FILENUMB]
80      (S+IFBUFFER)↓NXTPAGE := h↓FIRSTPAGE
        ReturnVec[h, h↓0 ^ LMASK]
        FB↓INBFFPTR, FB↓INBFFEND := EMPTY, EMPTY
    $R

85
```

```
    let EntriesfromFile[f] = valof
    $EF
 90     let S = NewVec[EFSIZE]
        let h = FindHeading[f]
        unless h!TYPE = INDEX do
          $ OSReport[733, 'Not index file']
            GiveUp[f]  $
 95     ReturnVec[h, h!0 ^ LMASK]

        S!NEXT  := NextEF
        S!ENDOF := EndofEF
        S!CLOSE := CloseEF
100     S!RESET := ResetEF
        S!OUT   := StreamError
        S!STR   := InfromFile[f]

        resultis S
105 $EF


    and NextEF[S] = valof
    $N  let F = S!STR  || = InfromFile[f]
110     let n = Next[F]
        if n = ENDOFSTREAMCH resultis n
      $ let Entry = NewVec[n]
        Entry!0 := n
        TransferIn[F, lv(Entry!1), n]
115     resultis Entry
    $N


    and EndofEF[S] = Endof[S!STR]
120

    and CloseEF[S] be
    $C  Close[S!STR]
        ReturnVec[S, EFSIZE]
125 $C


    and ResetEF[S] be Reset[S!STR]

****
```

II:7.4 DISCOUT

```
    ||  This section defines the global Stream function
    ||  OuttoFile, and also the global routine DeleteBody.

    ||  They are defined in the same section because they
    ||  both use the non-global routines ReturnChain
    ||  and CheckPerm.


let ReturnChain[f, Page] be
$RC
    $r
        DisctoCore[DiscPage, Page]
        unless DiscPage↓THISFILE = f do
          $ OSReport[741, 'Page chain wrong - file *N',f]
            GiveUp[Page] $
      $ let p = DisoPage↓NXTPAGE
        ReturnDiscBlock[Page]
        Page := p
    $r  repeatuntil Page = ENDBODY
$RC


let CheckPerm[f, h] be
$C  let p = h↓PERM
    unless (p=UNRESTRICTED)∨(p=OWNERONLY ^ User=h↓OWNER) do
      $ OSReport[742, 'Protected file']
        GiveUp[f]
$C


let DeleteBody[f] be
$D  let h = FindHeading[f]
    if h = NULL do
      $ OSReportN[743];  GiveUp[f] $
    CheckPerm[f, h]
  $ let p = h↓FIRSTPAGE
    unless p = NULLBODY do
    $1  UpdateHead[f, NULLBODY, 0, NULLBODY,
                       SAME, FetchCode[DATE], SAME]
        ReturnChain[f, p]
```

```
        Charge[h↓OWNER, - h↓NUMBPAGES]
    §1
45  ReturnVec[h, h↓0 ^ LMASK]
§D



50 let OuttoFile[f] = valof
§OF
    let FB = NewVec[FBSIZE]
    let S = NewVec[OFSIZE]
    let B = lv S↓OFBUFFER
55
    let h = FindHeading[f]
    CheckType[f, h]
    CheckPerm[f, h]

60  FB↓SLOWBLOCK := S
    FB↓OUTBFFPTR := lv B↓FIRSTDATA
    FB↓OUTBFFEND := (lv B↓LASTDATA)+1
    FB↓INBFFPTR, FB↓INBFFEND := EMPTY, EMPTY
    FB↓INESC, FB↓OUTESC := NOTUSED, NOTUSED
65
    S↓NEXT  := StreamError
    S↓OUT   := OutBuffOF
    S↓CLOSE := CloseOF
    S↓ENDOF := StreamError
70  S↓RESET := ResetOF

    S↓FILENUMB  := f
    S↓FILEOWNER := h↓OWNER
    S↓OLDBODY   := h↓FIRSTPAGE
75  S↓OLDSIZE   := h↓NUMBPAGES
    S↓OFPAGE    := NewDiscBlock[]
    S↓NEWBODY   := S↓OFPAGE

    ReturnVec[h, h↓0 ^ LMASK]
80
    B↓SERIAL    := 1
    B↓THISFILE  := f
    B↓NXTPAGE   := ENDBODY
    CoretoDisc[B, S↓OFPAGE]  || for safety
85
  § let C = lv S↓CUCHAIN
    C↓ROUTINE := FailClose
```

```
        C↓CSUC := ClearUpChain
        C↓CPRE := lv ClearUpChain
 90     ClearUpChain := C
        unless C↓CSUC = ENDCUCHN do (C↓CSUC)↓CPRE := C

        resultis ~FB
    $OF
 95

    and OutBuffOF[FB] be
    $OB
        let S = FB↓SLOWBLOCK
100     let B = lv S↓OFBUFFER
        TurnPage[S, NewDiscBlock[], B]
        FB↓OUTBFFPTR :=  lv B↓FIRSTDATA
    $OB

105
    and TurnPage[S, NextPage, B] be
    $T
        B↓NXTPAGE := NextPage
        CoretoDisc[B, S↓OFPAGE]
110     S↓OFPAGE := NextPage
        B↓SERIAL := B↓SERIAL + 1
    $T


115 and CloseOF[FB] be
    $C
        let S = FB↓SLOWBLOCK

        let C = lv S↓CUCHAIN
120     (C↓CPRE)↓CSUC := C↓CSUC
        unless C↓CSUC = ENDCUCHN do (C↓CSUC)↓CPRE := C↓CPRE

        ResetOF[FB]
        ReturnDiscBlock[S↓OFPAGE]
125     ReturnVec[S, OFSIZE]
        ReturnVec[FB, FBSIZE]
    $C


130 and ResetOF[FB] be
    $R
        let S = FB↓SLOWBLOCK
```

```
        let B = lv S↓OFBUFFER

135     if (S↓OFFPAGE = S↓NEWBODY)^(FB↓OUTBFFPTR = lv B↓FIRSTDATA)
          || i.e. if nothing output
          do §1
                S↓OLDBODY := NULLBODY
                S↓OLDSIZE := 0
140             DeleteBody[S↓FILENUMB]
                return
            §1

      § let NumbPages = B↓SERIAL
145     Charge[S↓FILEOWNER, NumbPages - S↓OLDSIZE]

        B↓ENDOFDATAPTR := FB↓OUTBFFPTR - B
        B↓NXTPAGE := ENDBODY
        CoretoDisc[B, S↓OFFPAGE]  ||  write last page
150
        UpdateHead[S↓FILENUMB, S↓NEWBODY, NumbPages, S↓OFFPAGE,
                                SAME, FetchCode[DATE], SAME]

        unless S↓OLDBODY = NULLBODY do
155         ReturnChain[S↓FILENUMB, S↓OLDBODY]

        S↓OLDBODY := S↓NEWBODY
        S↓OLDSIZE := NumbPages
        S↓OFFPAGE := NewDiscBlock[]
160     S↓NEWBODY := S↓OFFPAGE
        B↓SERIAL := 1
        FB↓OUTBFFPTR :=  lv B↓FIRSTDATA
    §R

165
    and FailClose[C] be
    §F
        let S = C - CUCHAIN
        let B = lv S↓OFBUFFER
170     B↓NXTPAGE := ENDBODY
        CoretoDisc[B, S↓OFFPAGE]              ||  Complete the chain,
        ReturnChain[S↓FILENUMB, S↓NEWBODY]||  and return it.
    §F

****
```

II:8, Disc routines (2)

II:8.1 CHARGE

```
let Charge[User, NumberofPages] be § return §
****
```

II:8,2 UPDATE

```
    let Update[f, Word, Value] be
    §U
        let Incrementing = (Word < 0)
        let Element = Incrementing → -Word, Word
 5
        let h = FindHeading[f]
        if h = NULL do
          § OSReport[821, 'Heading deleted']
            GiveUp[f]  §
10    § let HdLength = h!0 ^ LMASK
        unless Element ≤ HdLength do
          § OSReportN[822]
            GiveUp[Word]  §
        ReturnVec[h, HdLength]
15
      § let HdAddr = LookUpinMFL[f]
        let DiscWord = HdAddr!HWORD + Element
        DisctoCore[DiscPage, HdAddr!HPAGE]
        if DiscWord > LASTDATA do
20      §1  let NextPage = DiscPage!NXTPAGE
            if NextPage = ENDBODY do
              § OSReportN[823]
                GiveUp[f]  §
            DisctoCore[DiscPage, NextPage]
25          HdAddr!HPAGE := NextPage
            DiscWord := DiscWord - DATASIZE
        §1

        DiscPage!DiscWord :=
30          Incrementing → (DiscPage!DiscWord)+Value, Value
        CoretoDisc[DiscPage, HdAddr!HPAGE]
        ReturnVec[HdAddr, ENTRYSIZE]
    §U

****
```

II:8.3 UPDATEHEAD

```
    ||  This section defines the global routine UpdateHead.


  5 let UpdateHead[f, FirstPage, NumbPages, LastPage,
                   Type, DateLastChanged, DateLastRead] be
    $UH
        let HdAddr = LookUpinMFL[f]
        let ParamList = lv f
 10     DisctoCore[DiscPage, HdAddr↓HPAGE]

        for i=1 to 6 unless ParamList↓i = SAME do
        $F
            let Element = valof
 15          $v switchon i into
                $ case 1 : resultis FIRSTPAGE
                  case 2 : resultis NUMBPAGES
                  case 3 : resultis LASTPAGE
                  case 4 : resultis TYPE
 20               case 5 : resultis DLW
                  case 6 : resultis DLR
              $v
          $ let DiscWord = HdAddr↓HWORD + Element
            if DiscWord > LASTDATA do
 25         $1  let NextPage = DiscPage↓NXTPAGE
                if NextPage = ENDBODY do
                  $ OSReportN[831]
                    GiveUp[f]  $
                CoretoDisc[DiscPage, HdAddr↓HPAGE]
 30             DisctoCore[DiscPage, NextPage]
                HdAddr↓HPAGE := NextPage
                HdAddr↓HWORD := HdAddr↓HWORD - DATASIZE
                DiscWord := DiscWord - DATASIZE
            $1
 35         DiscPage↓DiscWord := ParamList↓i
        $F

        CoretoDisc[DiscPage, HdAddr↓HPAGE]
        ReturnVec[HdAddr, ENTRYSIZE]
 40 $UH

****
```

II:8.4 FINDHEADING

```
||  This section defines the global functions
||  FindHeading and LookUpinMFL.


5

let LookUpinMFL[f] = valof
§L
    if f ≤ 0 resultis NULL
10  DisctoCore[DiscPage, FetchCode[MFLFIRSTPAGE]]

  § let i = 0
    until f ≤ i + ENTRIESPERPAGE do
    §u  if DiscPage↓NXTPAGE = ENDBODY resultis NULL
15      DisctoCore[DiscPage, DiscPage↓NXTPAGE]
        i := i + ENTRIESPERPAGE
    §u

  § let w = (ENTRYSIZE+1)X((f-1)-i) + FIRSTDATA
20  if DiscPage↓w = NOTENTRY resultis NULL

  § let Entry = NewVec[ENTRYSIZE]
    Copy[lv (DiscPage↓w), Entry, ENTRYSIZE+1]
    resultis Entry
25 §L


30

let FindHeading[f] = valof
§F
    let HdAddr = LookUpinMFL[f]
    if HdAddr = NULL do
35    § OSReport[841, 'No entry in MFL']
      GiveUp[f]  §

    DisctoCore[DiscPage, HdAddr↓HPAGE]
  § let Word = HdAddr↓HWORD
40  ReturnVec[HdAddr, ENTRYSIZE]
```

```
    § let Length = DiscPage↓Word ^ LMASK
      let h = NewVec[Length]
      let n = LASTDATA - Word
45
      test Length ≤ n
        then  Copy[lv (DiscPage↓Word), h, Length+1]
        or §1
              let NextPage = DiscPage↓NXTPAGE
              if NextPage = ENDBODY do
50               § OSReportN[842]
                    GiveUp[f]  §
              Copy[lv (DiscPage↓Word), h, n+1]
              DisctoCore[DiscPage, NextPage]
55            Copy[lv (DiscPage↓FIRSTDATA), lv (h↓(n+1)),
                                                  Length-n]
          §1

      if h↓TYPE = DELETED do
60    §2  ReturnVec[h, Length]
          resultis NULL
      §2

      resultis h
65  §F
```

****

II:8.5 LOOKUP

```
    || This section defines the global function LookUp

    manifest § ENTRYLENGTH = 20 §

  5 let LookUp[Name1, Name2, i] = valof
    §L
      if i = NULL resultis NULL
    § let h = FindHeading[i]
      if h = NULL do
 10     § OSReport[851, 'Index deleted']
        GiveUp[i]  §
      unless h↓TYPE = INDEX do
        § OSReport[852, 'File not index']
        GiveUp[i]  §
 15   ReturnVec[h, h↓0 ^ LMASK]
    § let S = InfromFile[i]
      let Length = ENTRYLENGTH
      let v = NewVec[Length]
      until Endof[S] do
 20   §u
        let n = Next[S]
        if n > Length do
        §1  ReturnVec[v, Length]
          Length := n
 25       v := NewVec[Length]
        §1
        TransferIn[S, v+1, n]
        unless v↓STATUS = DELETED do
          if EqS[Name1, v+v↓N1] ^ EqS[Name2, v+v↓N2] do
 30         §2
            Close[S]
          § let NotLinked = (v↓LINKING ≥ 0)
            let f = NotLinked → v↓FILE,
                        LookUp[v-v↓N3, v+v↓N4,
 35                       LookUp[v+v↓N5, v+v↓N6, SystemIndex]]
            ReturnVec[v, Length]
            resultis f
      §u
      Close[S]
 40   ReturnVec[v, Length]
      resultis NULL
    §L
****
```

II:8.6 LOADFILE

```
    ||  This section defines the global routines
    ||  LoadFile and LoadSystemFile.


 5


    let LoadFile[f] be
    $LF
10      let i = In
        let n = MaxVecSize[] - INFILESIZE
        let v = NewVec[n]

        In := InfromFile[f]
15      ReturnVec[v, n]

        Load[i]   ||  from In
        Close[In]
        In := i
20  $LF




25
    let LoadSystemFile[String] be
    $LSF
        let f = LookUp[String, 'IC', SystemIndex]
        if f = NULL do
30       $ OSReport[861, '*'*S*' *'IC*' not system file', String]
          GiveUp[0]  $

        LoadFile[f]
    $LSF
35
****
```

OS/SF

```
        DEC 6;  DEC 0;                  || NEWSECTION

        DEC 1;  DEC 17;                 || CODE, 17 words :
   *L2; IP2; LDPRG; OSY; ST; EXIT;      || FetchCode     (global  2)
   *L3; OP3; LDC; IP2; STPRG; EXIT;     || StoreCode     (global  3)
   *L4; EXEC; EXIT;                     || Exec          (global  4)

   *L15; OG335; SUMCK;                  || SumCheck      (global 15)
                                        ||  (global 335 = DiscPage)

   *L17; NEXT;                          || Next          (global 17)
   *L16; OUTZ;                          || Out           (global 16)
   *L20; ENDOF;                         || Endof         (global 20)
   *L13; GET; *PUTZ;                    || TransferIn    (global 13)
   *L14; GET; *PUTY;                    || TransferInC   (global 14)
   *L7;  TROT;                          || TransferOut   (global  7)

   *L100; DEC 2;  DEC 21;               || INTERLUDE, 21 words :
   *L101; OG8;  LDC;   OG2;  ST;        || OG8 = CFirst (global  8)
          ON3;  ADDT;  OG3;  ST;
          ON3;  ADDT;  OG4;  ST;
          ON1;  ADDT;  OG15; ST;
          ON2;  ADDT;  OG17; ST;
          ON1;  ADDT;  OG16; ST;
          ON1;  ADDT;  OG20; ST;
          ON1;  ADDT;  OG13; ST;
          ON2;  ADDT;  OG14; ST;
          ON2;  ADDT;  OG7;  ST;
          EXIT;

  *L999; END;


   ||
   || Note that if the method of setting globals is changed,
   || then the System dumper must also be changed.
   ||
```

```
40
   let SumCheck[] = valof
   §S let n = 0
      for i=0 to LASTDATA do n := AddwithEAC[n, DiscPage↓i]
      resultis n
45 §S

   and AddwithEAC[a,b] = a+b+ (((a±(a+b))<0<(a±b)) → 1, 0)


50


   let Next[S] = valof
   §N if S>0 resultis (S↓NEXT)[S]
      S := ~S
55    if S↓INBFFPTR>S↓INBFFEND resultis ((S↓SLOWBLOCK)↓NEXT)[S]
      unless S↓INESC = NOTUSED do
          if rv(S↓INBFFPTR)= S↓INESC then
                              resultis ((S↓SLOWBLOCK)↓TRAP)[S]
    § let Ch = rv(S↓INBFFPTR)
60    S↓INBFFPTR := S↓INBFFPTR + 1
      resultis Ch
   §N


65


   let Out[S, x] be
   §O test S>0
        then (S↓OUT)[S, x]
70      or
        §1  S := ~S
            test S↓OUTBFFPTR ≥ S↓OUTBFFEND
              then  ((S↓SLOWBLOCK)↓OUT)[S, x]
              or
75              §2  rv (S↓OUTBFFPTR) := x
                    S↓OUTBFFPTR := S↓OUTBFFPTR + 1
                    test S↓OUTBFFPTR ≥ S↓OUTBFFEND
                      then  ((S↓SLOWBLOCK)↓OUT)[S, x]
                      or unless S↓OUTESC = NOTUSED  do
80                         if S↓OUTESC=x do
                                ((S↓SLOWBLOCK)↓OUT)[S, x]
   §O
```

```
85

90
    let Endof[S] = valof
    $(  if S>0 resultis (S!ENDOF)[S]
        S := ~S
        if S!INBFFPTR >= S!INBFFEND then
95                              resultis ((S!SLOWBLOCK)!ENDOF)[S]
        unless S!INESC = NOTUSED do
            if rv(S!INBFFPTR) = S!INESC then
                                resultis ((S!SLOWBLOCK)!ENDOF)[S]
        resultis false
100 $)


    let TransferIn[S,v,n] be
        for i=0 to n-1 do v!i := Next[S]
105

    let TransferInC[S,v,n] be
        for i=0 to n-1 do StoreCode[v+i, Next[S]]

110
    let TransferOut[S,v,n] be
        for i=0 to n-1 do Out[S, v!i]
****
```

III: SET-UP PROGRAMS

III:1, System set-up

III:1.1 SYSSETUP

```
    || This section defines the routine PInterrupt, which
    || is entered when the System is first loaded from Disc.


 5  manifest
    §   INVALIDPAGE = 9999
        SYSTEMINDEX =    4
        SYSTEM      =    1   || User code
        STARTOFMFL  =    2
10      GUSSIZE     =  110
        PSSIZE      =  110

        OUTOFRANGE  =    5   || Exec reject qualifier
        FIRSTNSG    =  401
15      LASTNSG     =  500
        OFFSET      = -(ESIZE+1)
    §

    static  §S
20  DiscUsable = false
    DeadArea = 0;  DALength = 0
            §S


25  let PInterrupt[] be
    §PI
        StoreCode[DISCWRITEPERMITTED, false]

        || The order of this sequence is important
30      SetUpFS[]
        SetUpDiscPage[]
        SetUpPMStacks[]
        SetUpStreams[]
        SetUpSundryItems[]
35      SetUpTimeofDayClock[]
        SetDateandTime[]
        SetUpRunBlock[]
        SetStackBase[]
```

```
40      DiscUsable := CheckDiscOn[]
        test DiscUsable
          then DiscUsable := QuickValFSF[]
          or   OutS[Console, 'DISC OFF:  ']

45      StoreCode[REASONFORINTERRUPT, NOREASON]
        StoreCode[INTERRUPTADDRESS, lv Interrupt]

        ReleaseNonSystemGlobals[]

50      test DiscUsable
          then $  FSFtoCore[]
                  Run[LogIn]
                  StoreCode[DISCWRITEPERMITTED, true]
               $
55        or    $  FSF↓PAGE := INVALIDPAGE
                  User, CurrentIndex := SYSTEM, SystemIndex
                  OutS[Console, 'Abnormal set-up*n']
               $

60      return     ||  enters LoadGoLoop (see SetStackBase)
    $PI



65  and SetUpFS[] be
    $FS let f = FS + (FSVECSIZE + 1)

        FS↓FBC, FS↓FWC :=   f, END
        FS↓PBC, FS↓PWC := END, END
70      FS↓LB , FS↓UB  :=   f, FSLim
        FS↓FPRE := FS

        f↓SIZE := FSLim - f
        f↓NXTB := END
75  $FS


    and SetUpDiscPage[] be
    $DP let Blocksize = ESIZE + 1
80      let v = NewVec[Blocksize + 2XPAGESIZE]
        let Endv = lv v↓(Blocksize + 2XPAGESIZE)

        let D = ((v-1)/PAGESIZE + 1)XPAGESIZE
```

```
        ||  D now points to the first Hardware page
 85     ||  in the range (v, Endv).

        if D-Blocksize < v do D := D + PAGESIZE
        ||  allow room for ExecBlock.

 90     DiscPage := D

        DeadArea := v
        DALength := ((D - Blocksize)- v) - 1
        ||  The DeadArea is returned
 95     ||  after the PM stacks are set-up.
        ReturnVec[(D+PAGESIZE), (Endv - (D+PAGESIZE))]

      § let E = D - Blocksize  ||  so OFFSET = - Blocksize
        E↓BUFFER    := DiscPage
100     E↓BUFFSIZE  := PAGESIZE
        E↓SEG       := DATASEG
        E↓ENDMODE   := QUIETEND
        E↓SELFPTR   := E
        ||  E↓DEVICE, E↓PAGENUMB and E↓COMPLETED are set before
105     ||  use (e.g. in DiscTransfer).  The other two elements
        ||  of the ExecBlock are not used in this command.
    §DP


110 and SetUpPMStacks[] be
    §S  GiveUpStack := NewVec[GUSSIZE]
        GiveUpStack↓LENGTH := GUSSIZE
        GiveUpStackSize := GUSSIZE-1

115     PrivateStack := NewVec[PSSIZE]
        PrivateStack↓LENGTH := PSSIZE

        ReturnVec[DeadArea, DALength] || claimed in SetUpDiscPage
    §S
120

    and SetUpSundryItems[] be
    §SI
        GiveUp := StandardGiveUp
125     PBChain := END
        ClearUpChain := ENDCUCHN

        FSF := NewVec[FSFSIZE]
```

```
        StoreCode[MFLFIRSTPAGE, STARTOFMFL]
130     SystemIndex := SYSTEMINDEX

        StoreCode[SUMCHINHIBITED, false]

        IBlock↓ISUC := NONE
135
      § let MaxC = (CPages-1)XPAGESIZE - OFFSETC
                 || allows for multiplexor page
        let MaxD = DPages X PAGESIZE
        StoreCode[MAXC, MaxC]
140     StoreCode[MAXD, MaxD]
    §SI



145 and SetUpTimeofDayClock[] be
    §T  let E = NewVec[ESIZE]
        let B = NewVec[CLOCKBUFFSIZE - 1]

        E↓DEVICE     := CLOCK
150     E↓BUFFER     := B
        E↓BUFFSIZE   := CLOCKBUFFSIZE
        E↓SEG        := DATASEG
        E↓ENDMODE    := QUIETEND
        E↓COMPLETED  := false
155     E↓SELFPTR    := E
        || The other three elements of the ExecBlock
        || are not used in this device and command.

        B↓TIME := NULL  || set in AskTimeofDay
160     B↓PERIOD := ONEMINUTE

        StoreCode[TIMEOFDAYCLOCK, E]
    §T

165

    and SetUpRunBlock[] be
    §R  let R = NewVec[RSIZE]

170     || Reset FS so that RunBlock is outside FreeStore:

        NewFreeStore[]
        FS↓FPRE := FS
```

```
175     R!RPRE := R
        R!PPTR := FSLim + 1  ||  start of stack
        R!IBLK := IBlock
        R!CPTR := CPtr
        R!FSV  := FS
180     R!INPT := In
        R!OUTP := Output
        R!CON  := Console
        R!REP  := ReportStream
        R!GU   := GiveUp
185     R!GUS  := GiveUpStack
        R!GUSS := GiveUpStackSize
        R!CUC  := ClearUpChain

        StoreCode[RBLOCK, R]
190 $R


    and SetStackBase[] be
    $S  let StackBase = FSLim + 1
195     let Link = StackBase + 1
        rv StackBase := StackBase
        rv Link := LoadGoLoop
    $S


200
    and CheckDiscOn[] = valof
    $DO let E = DiscPage + OFFSET
        E!DEVICE := DISCREAD
        E!PAGENUMB := INVALIDPAGE
205     Exec[TRANSFER, Rj, lv E!PAGENUMB, E!PAGENUMB, E]
        Rj: ||  Exec will reject the command, either
            ||  because the Disc-Controller is off-line,
            ||  or because the page is out of range.
            ||  The reason is placed in E!PAGENUMB.
210     resultis (E!PAGENUMB = OUTOFRANGE)
    $DO


    and ReleaseNonSystemGlobals[] be
215 $RG global $ GlobalZero : 0 $
        let GlobVec = lv GlobalZero
        for g = FIRSTNSG to LASTNSG do GlobVec!g := NULL
    $RG
```

```
220
    and FSFtoCore[] be
    $F  let H = FindHeading[LookUp['FSF', 'SYS', SystemIndex]]
        DisctoCore[FSF, H↓LASTPAGE]
        FSF↓SCAN := FIRSTENTRY
225     FSF↓PAGE := H↓LASTPAGE
        ReturnVec[H, (H↓0 ^ LMASK)]
    $F


230 and LogIn[] be
    $LI LoadSystemFile['LogIn']
        Prog[]
    $LI
```

****

III:1.2 SETDANDT

```
    || This section defines the routine SetDateandTime,
    || which is called by 'SYSSETUP'.


 5  manifest  $M
    JAN = 1;  FEB = 2;  MAR = 3;  APR = 4;  MAY = 5;  JUN = 6
    JUL = 7;  AUG = 8;  SEP = 9;  OCT = 10;  NOV = 11;  DEC = 12
    IMPOSS = -1
             $M
10

    let SetDateandTime[] be
    $SDT
        let t = TimeofDay[]
15      let d = Date[]
        StoreCode[DATE, d]
        Reset[Console]
    $SDT

20
    and Date[] = valof
    $D
        OutS[Console, 'Date *q ']
      $ let Day   = NextN[Console]
25      let Month = NextN[Console]
        let Year  = NextN[Console] rem 1900

        unless 1 <= Year <= 99
            do $ Wrong['Year']; resultis Date[] $
30      unless JAN <= Month <= DEC
            do $ Wrong['Month']; resultis Date[] $

      $ let DaysinMonth, Fudge = 0, 0
        switchon Month into
35      $S  case JAN: DaysinMonth := 31
                      Fudge := (Leap[Year]-> 6, 0) ; endcase
            case FEB: DaysinMonth := (Leap[Year]-> 29, 28)
                      Fudge := (Leap[Year]-> 2, 3) ; endcase
            case MAR: DaysinMonth, Fudge := 31, 3 ; endcase
40          case APR: DaysinMonth, Fudge := 30, 6 ; endcase
            case MAY: DaysinMonth, Fudge := 31, 1 ; endcase
            case JUN: DaysirMonth, Fudge := 30, 4 ; endcase
            case JUL: DaysinMonth, Fudge := 31, 6 ; endcase
```

```
            case AUG: DaysinMonth, Fudge := 31, 2 ;  endcase
            case SEP: DaysinMonth, Fudge := 30, 5 ;  endcase
            case OCT: DaysinMonth, Fudge := 31, 0 ;  endcase
            case NOV: DaysinMonth, Fudge := 30, 3 ;  endcase
            case DEC: DaysinMonth, Fudge := 31, 5 ;  endcase
        $S

        unless 1 <= Day <= DaysinMonth
            do $ Wrong['Day']; resultis Date[] $

      $ let DayofWeek = (Year + Year/4 + Day + Fudge)rem 7
        || Fudge is a correction because of the Month

        OutS[Console, 'Day of week *q ']
      $ let Ch = NextLetter[Console]
        if Ch='S' v Ch='T' do
          $ let x = NextLetter[Console]
            unless x='U' do Ch := x
          $

        unless DayofWeek = (Ch='S' -> 0, Ch='M' -> 1, Ch='T' -> 2,
                            Ch='W' -> 3, Ch='H' -> 4, Ch='F' -> 5,
                            Ch='A' -> 6, IMPOSS)
            do $ Wrong['Day of week']; resultis Date[] $

        resultis (Day lshift 11)+(Month lshift 7)+ Year
$D


    and Wrong[String] be
        OutS[Console, '*s wrong *n', String]

    and Leap[Year] = (Year rem 4 = 0)


    and NextLetter[S] = valof
    $NL let Ch = 0
        Ch := Next[S] repeatuntil ('A'<=Ch<='Z')v('a'<=Ch<='z')
        resultis Ch ^ CASEMASK  || translate into capitals
    $NL


****
```

III:1.3 QUICKVAL

```
manifest § MAXPAGE = 3000 §

static §S
Result = true
         §S

let QuickValFSF[] = valof
§2
    let f = LookUp['FSF', 'SYS', SystemIndex]
    if f = NULL do
      § ReportS['FreeStoreFile not found']
      resultis false §

  § let h = FindHeading[f]
    let Serial, Page, PreviousPage = 1, h↓FIRSTPAGE, ENDBODY

    §R unless 1 ≤ Page ≤ MAXPAGE do
         § ReportS['FSF: forward link *N; previous page *N',
                                Page, PreviousPage]
           resultis false §

      DisctoCore[DiscPage, Page]
      unless DiscPage↓SERIAL = Serial do
        ReportMessage['FSF: page *N wrong serial', Page]
      unless DiscPage↓THISFILE = f do
        ReportMessage['FSF: page *N wrong file no.', Page]
      unless DiscPage↓BACKLINK = PreviousPage do
        ReportMessage['FSF: page *N backlink wrong', Page]

    § let n = 0
      for i=FIRSTENTRY to LASTENTRY do
        unless 1 ≤ DiscPage↓i ≤ MAXPAGE do
          if DiscPage↓i ≠ CLAIMED do n := n + 1

      if n > 0 do ReportMessage['FSF: page *N, *N *S*s
                        outside page limits', Page, n,
                        (n > 1 → 'entries', 'entry')]

      if DiscPage↓NXTPAGE = ENDBODY break
      Serial, PreviousPage:= Serial + 1, Page
      Page := DiscPage↓NXTPAGE
    §R repeat
```

```
45
        unless Serial = h!NUMBPAGES do
          ReportMessage['FSF: no. of pages wrong']
        unless Page = h!LASTPAGE do
          ReportMessage['FSF: last page number wrong']
50
        ReturnVec[h, h!0 ^ LMASK]
        resultis Result
    $R

55


    and ReportMessage[String, x, y, z] be
    $R  ReportS[String, x, y, z]
60      Result := false  $R
```

****

III:2. Set up streams

III:2.1 SETUPSTR

```
    manifest
    $   VECSIZE = 15        || 16 possible values
        PARBITS = 864626    || for 4 bits.
    $
 5

    let SetUpStreams[] be
    $SUS
        SetUpParityTable[]
10
        SetUpExecConsole[]
        Console := ExecConsole
        Output := ExecConsole
        ReportStream := ExecConsole
15
        SetUpReader[]
        In := DiadRead
    $SUS

20
    and SetUpParityTable[] be
    $P  let P = NewVec[VECSIZE]
        let PBits = PARBITS
        for i = 0 to VECSIZE do
25        $ P!i := PBits ^ 1
          PBits := PBits rshift 1
          $
        Parity := P
    $P
30

****
```

III:2.2 SETUPTT

```
let SetUpExecConsole[] be
§  Teletype := TT[]
   ExecConsole := IntcodetoandfromTeletype[Teletype]
   ExecConsole↓CLOSE := ExecConsole↓RESET
§


and TT[] = valof
§TT
   let S = NewVec[TELESIZE]
   S↓NEXT          := NextTT
   S↓OUT           := OutTT
   S↓CLOSE         := ResetTT
   S↓STR           := NULL
   S↓ENDOF         := EndofTT
   S↓RESET         := ResetTT
   S↓SOURCE        := S
   S↓STATE         := StateTT
   S↓RESETSTATE    := ResetStateTT

 § let R = lv S↓READBLOCK
   R↓DEVICE        := TTREAD  || without echo
   R↓BUFFER        := lv S↓INBUFF
   R↓BUFFSIZE      := 2       || one word is required by Exec
   R↓STOPCH        := NOTBYTE
   R↓SEG           := DATASEG
   R↓ENDMODE       := QUIETEND
   R↓COMPLETED     := false
   R↓SELFPTR       := R
   ||  The other two elements of the ExecBlock
   ||  are not used in this command.
```

```
40    § let W = lv S↓WRITEBLOCK
        W↓DEVICE       := TTWRITE
        W↓BUFFER       := lv S↓OUTBUFF
        W↓BUFFSIZE     := 1
        W↓STOPCH       := NOTBYTE
45      W↓SEG          := DATASEG
        W↓ENDMODE      := QUIETEND
        W↓COMPLETED    := false
        W↓SELFPTR      := W
        || The other two elements of the ExecBlock
50      || are not used in this command.

        S↓LINEBFFPTR, S↓LINEBFFLAST := EMPTY, EMPTY

        resultis S
55  $TT
```

****

III:2.3 SETUPRDR

```
manifest $ RPINGS = 3 $


5
let SetUpReader[] be
$R  ReaderDev := READERLEFTTORIGHT
    BytesfromPT := BFPT[]
    DiadRead := WordsfromDiads[BytesfromPT]
10  $ let Diad = (~DiadRead)!SLOWBLOCK
    Diad!CLOSE := Diad!RESET
$R


15

and BFPT[] = valof
$BPT
    let FB = NewVec[FBSIZE]
20  let S = NewVec[BFPTSIZE]
    let E = lv S!EXECBLOCK

    FB!SLOWBLOCK := S
    FB!INBFFPTR, FB!INBFFEND  := EMPTY, EMPTY
25  FB!OUTBFFPTR, FB!OUTBFFEND := EMPTY, EMPTY
    FB!INESC, FB!OUTESC := NOTUSED, NOTUSED

    ||  S!NEXT is set by the call of ResetBFPT below
    S!OUT         := StreamError
30  S!CLOSE       := ResetBFPT
    S!ENDOF       := EndofBFPT
    S!RESET       := ResetBFPT
    S!SOURCE      := S
    S!STATE       := StateBFPT
35  S!RESETSTATE  := NullProgram
    S!PINGS       := RPINGS

  $ let Buff1, Buff2 = lv S!BUFF1, lv S!BUFF2
    Buff1!(-1) := Buff2
40  Buff2!(-1) := Buff1
```

```
        E↓DEVICE        := ReaderDev
45      E↓BUFFER        := Buff1
        E↓BUFFSIZE      := 0  || set in InitiateTransfer
        E↓STOPCH        := NOTBYTE
        E↓SEG           := DATASEG
        E↓ENDMODE       := QUIETEND
50      E↓COMPLETED     := false
        E↓SELFPTR       := E
        ||  The other two elements of the ExecBlock
        ||  are not used in this command.

55      ResetBFPT[FB]   ||  to set S↓NEXT

        resultis ~FB
    $EPT

60
****
```

IV: DECLARATIONS

IV:1 DECLARATIONS

```
    get 'GLOBALS'
    get 'PRIVATE GLOBALS'
    get 'CONSTANTS'

  5
****
```

IV:1.1 GLOBALS

```
global §G

BytesfromPT: 387

Close: 18; Console: 27; Copy: 36; CPtr: 9
CurrentIndex: 300

DeleteBody: 326; DefaultProg: 398; DiadRead: 338

EndGiveUp: 373; Endof: 20; EntriesfromFile: 320
EqS: 352; ExecConsole: 383

FetchCode: 2; FindHeading: 318; Finish: 395

GiveUp: 29; GiveUpStack: 363; GiveUpStackSize: 353

In: 25; InfromFile: 321

Load: 397; LoadFile: 378; LoadSystemFile: 365
LookUp: 317

MaxVecSize: 37

NewVec: 60; NewWord: 62; Next: 17; NextN: 358
NextO: 357; NullProgram: 359

Out: 16; OutAddr: 347; OutByte: 346
OutDateandTime: 348; OutN: 379; OutO: 380
Output: 26; OutS: 381; OuttoFile: 322

Prog: 1; PutBack: 391

ReportCallTrace: 375; ReportFreeStoreState: 374
ReportS: 30; ReportStream: 28; Reset: 68; ResetState: 349
ReturnVec: 61; ReturnWord: 63; Run: 399

StandardGiveUp: 394; StandardPM: 393; Start: 1
State: 350; StoreCode: 3; StreamError: 69
SystemIndex: 303

TimeofDay: 364; TransferIn: 13; TransferInC: 14
TransferOut: 7; TryAgain: 355
```

```
   Unload: 396;  User: 310;
45
   Wait: 372;  WordsfromDiads: 340;  Write: 343
   WriteAddr: 345;  WriteByte: 344;  WriteN: 45;  WriteO: 38
   WriteS: 44

50          $G
****
```

IV:1.2 PRIVATE GLOBALS

```
    global  $G

    AddressZero: 371
    CFirst: 8;  Charge: 306;  ClearUpChain: 332;  CoretoDisc: 334
  5 DiscPage: 335;  DisctoCore: 333;  Dump: 66
    Exec: 4
    ForcedGiveUp: 32;  FS: 24;  FSF: 329
    IBlock: 23;  InitiateTransfer: 10
      IntcodetoandfromTeletype: 79;  Interrupt: 351
 10 LGLoop: 64;  LoadGoLoop: 370;  LookUpinMFL: 319
    NewDiscBlock: 330;  NewFreeStore: 390
    OSReport: 33;  OSReportN: 34
    Parity: 67;  PBChain: 337;  PPtr: -3;  PrivateStack: 356
    ReaderDev: 382;  RestoreFreeStore: 389
 15   ReturnDiscBlock: 331
    SetGlobals: 21;  SetLabels: 11;  Sumcheck: 15
    Teletype: 376;  TerminateRun: 392;  TryDiadAgain: 336
    Update: 312;  UpdateHead: 311

 20
    || temporary globals:

    CPages: 416
    DPages: 417
 25 EndofBFPT: 401;  EndofTT: 402
    FSLim: 412
    NextTT: 403
    OutTT: 404
    PInterrupt: 411
 30 QuickValFSF: 414
    ResetBFPT: 405;  ResetTT: 406;  ResetStateTT: 407
    SetDateandTime: 413;  SetUpExecConsole: 409
      SetUpReader: 410;  SetUpStreams: 415
      StateBFPT: 418;  StateTT: 408
 35

            $G
****
```

IV:1.3 CONSTANTS

```
    manifest  ||  General
    §   NULL       = 0
        UNDEFINED  = 0
    §
 5
    manifest  ||  Machine constants
    §   PAGESIZE   =  256
        OFFSETC    =   30
        LMASK      = #377
10  §

    manifest  ||  Exec commands
    §   CANCEL     =  108
        LOOKATRDR  =  248
15      READABS    =  240
        TRANSFER   =   28
    §

    manifest  ||  Exec block for TRANSFER command
20  §   ESIZE = 9

          DEVICE     = 0
          BUFFER     = 1
          BUFFSIZE   = 2
25        PAGENUMB   = 3  || for disc transfers
          STOPCH     = 3  || for other transfers
          SEG        = 4
                           || element 5 is never used
          ENDMODE    = 6
30        COMPLETED  = 7  || used if endmode is QUIETEND
          SELFPTR    = 8
          INTREASON  = 9  || used if endmode is INTERRUPT
    §

35  manifest  ||  Standard contents of Exec block elements
    §   NOTBYTE    = -1  || stopch
        CODESEG    =  1  || seg
        DATASEG    =  0  || seg
        INTERRUPT  =  3  || endmode
40      QUIETEND   =  2  || endmode
    §
```

```
   manifest  ||  Device numbers
45 §    TTWRITE              = 1
        TTREAD               = 2  || without echo
        READERLEFTTORIGHT    = 4
          RDRDEVSUM          = 7
        DISCREAD             = 26
50      DISCWRITE            = 27
        CLOCK                = 29
   §

   manifest  ||  Code segment addresses
55 §    INTERRUPTINHIBITED = -29
        REASONFORINTERRUPT = -28

        MAXD                 = -8
        MAXC                 = -7
60      SUMCHINHIBITED       = -6
        DISCWRITEPERMITTED   = -5
        MFLFIRSTPAGE         = -4
        DATE                 = -3
        RBLOCK               = -2
65      TIMEOFDAYCLOCK       = -1

        INTERRUPTADDRESS     =  3
   §

70
   manifest  ||  Reasons for interrupt
   §    NOREASON     =  0
        EXECCONXON   =  1
        POWERON      =  2
75 §


   manifest  ||  Clock
   §    CLOCKBUFFSIZE =  2
80
          TIME          =  0
          PERIOD        =  1

        ONEMINUTE       =  120
85      MINSPERDAY      =  60X24
   §
```

```
    manifest  ||  Run block
90  §   RSIZE = 12

        RPRE =  0  ||  Predecessor of block
        PPTR =  1  ||  Procedure pointer
        IBLK =  2  ||  Information block
95      CPTR =  3  ||  Code pointer
        FSV  =  4  ||  Free store vector
        INPT =  5  ||  Input Stream
        OUTP =  6  ||  Output Stream
        CON  =  7  ||  Console Stream
100     REP  =  8  ||  Report Stream
        GU   =  9  ||  GiveUp routine
        GUS  = 10  ||  Stack for GiveUp
        GUSS = 11  ||  Size of stack needed for GiveUp
        CUC  = 12  ||  used by ClearUp
105 §


    manifest  ||  Information block
    §   ISIZE = 6
110
        CP   =  0  ||  Code pointer
        CL   =  1  ||  Code length
        DP   =  2  ||  Data pointer
        DL   =  3  ||  Data length
115     IPRE =  4  ||  Predecessor of block
        ISUC =  5  ||  Successor of block
        ID   =  6  ||  Section identifier

      NOTSET = -1
120   NONE   = -1
    §


    manifest  ||  Free store
125 §   FSVECSIZE = 6

        FBC  =  0  ||  Free block chain
        FWC  =  1  ||  Free word chain
        PBC  =  2  ||  Pending block chain
130     PWC  =  3  ||  Pending word chain
        LB   =  4  ||  Lower bound of current area
        UB   =  5  ||  Upper bound of current area
        FPRE =  6  ||  Predecessor of vector
```

```
135     SIZE   =  0
        NXTB   =  1
          ||  SIZE and NXTB cannot be interchanged without
          ||  alteration to the program.   (See 'FREESTORE').

140     END    =  0
        NOSTORE= -1
    §


145 manifest  ||  ClearUp Chain
    §
        CSUC     =  0  ||  Successor of block
        CPRE     =  1  ||  Predecessor of block
        ROUTINE  =  2  ||  ClearingUp routine
150
        ENDCUCHN  =  877774
    §


155 manifest  ||  Stream vector
    §
        NEXT         =  0
        OUT          =  1
        CLOSE        =  2
160     STR          =  3  || except source or bilateral streams
        ENDOF        =  4
        RESET        =  5
        SOURCE       =  6
        STATE        =  7  || only source streams
165     RESETSTATE   =  8  || only source streams

        NOTHINGTYPED = -1
    §

170
    manifest  ||  Stream elements used by InitiateTransfer
    §
        PINGS      =  3
        EXECBLOCK  =  9
175 §
```

```
    manifest  ||  Fast stream block
180 $   FBSIZE = 6

        SLOWBLOCK = 0  || pointer to stream vector
        INBFFPTR  = 1
        INBFFEND  = 2
185     INESC     = 3
        OUTBFFPTR = 4
        OUTBFFEND = 5
        OUTESC    = 6

190
      EMPTY   = 0
      NOTUSED = 0
    $

195
    manifest  ||  Teletype stream
    $   TELESIZE = 105

        READBLOCK   =  9
200     WRITEBLOCK  = 19
        INBUFF      = 29
        OUTBUFF     = 31
        LINEBFFPTR  = 32
        LINEBFFLAST = 33
205     LINEBUFFER  = 34
    $


    manifest  ||  Size of reader buffer
210 $   READBFFSIZE = 30   $


    manifest  ||  Reader stream
    $   BFPTSIZE = 22 + 2XREADBFFSIZE
215
        BUFF1   = 20
        BFF1END = 50
        BUFF2   = 52
        BFF2END = 82
220 $
```

```
225 manifest  ||  PutBack vector
    §   PBSIZE = 8

          PBNEXT   = 0
          PBCLOSE  = 1
230       PBSTR    = 2
          PBENDOF  = 3
          PBRESET  = 4
          OB       = 5  ||  Object put back
          STREAM   = 6
235       PTR      = 7
          PBPRE    = 8  ||  Predecessor
    §

    manifest  ||  Internal Character Code
240 §     ULMASK       = #177
          CASEMASK     = #137

          BELL         =   7
          DELETE       = #177
245       DIV          =  #30
          GRAVE        = #140
          LEFTARROW    = #137
          PRIME        =  #24
          RETURN       =  #15
250       RUNOUT       =   3
    §

    manifest  ||  Teletype Character Code
    §     PARITYMASK   = #177
255       PARITYBIT    = #200

          AMPERSANDt   =  #46
          BELLt        = #207
          ESCAPEt      =  #33
260       LINEFEEDt    =  #12
          NEWLINEt     =  #12
          PRIMEt       =  #47
          QUERYt       =  #77
          RETURNt      = #215
265       RUBOUTt      = #377
          RUNOUTt      =   0
          SHARPt       = #243
          SLASHt       =  #57
```

```
        STARt       = 852
270     UPARROWt    = 8136
        VTABt       = 8213
        XOFFt       = 8223
    §

275
    manifest || stack element
    § LENGTH = 1 §


280 manifest || Filing System
    §
        SERIAL      =   0
        THISFILE    =   1
        NXTPAGE     =   2
285     FIRSTDATA   =   3
        LASTDATA    = 254
        ENDOFDATAPTR = 254 || on last page of streamable files
        SUMCH       = 255

290     DATASIZE =   252
        FREE     =     0
        ENDBODY  = 877776
    §

295
    manifest || Headings
    §
        FIRSTPAGE =   1
        NUMBPAGES =   2
300     LASTPAGE  =   3
        TYPE      =   4
        OWNER     =   5
        CREATED   =   6   || Data created
        DLW       =   7   || Date Last Written
305     DLR       =   8   || Date Last Read
        PERM      =   9   || Permission
        MFLNUMB   =  10   || File number

        SAME      =  -9
310     NEVER     =   0
        NULLBODY  =  -1
    §
```

```
315
    manifest ||  Index Entries
    §
         ENTRYEND  =   0
320      STATUS    =   1
         FILE      =   3
         LINKING   =   3
         N1        =   1
         N2        =   2
325      N3        =   3
         N4        =   4
         N5        =   5
         N6        =   6
    $
330

    manifest ||  Some file types
    §    DELETED     =   0
335      INDEX       =  8b13
         STREAMABLE  =  8b10
    $

    manifest ||  File permissions
340 §    UNRESTRICTED =   0
         OWNERONLY    =   1
         RESTRICTED   =   2
    $

345

    manifest ||  MasterFile List
    §    ENTRYSIZE = 1

350        HPAGE = 0
           HWORD = 1

         NOTENTRY = 0
         ENTRIESPERPAGE = 126
355 $
```

```
360

    manifest  ||  Free Store File
    $  FSFSIZE = 257

365       BACKLINK    =   3
          FIRSTENTRY  =   4
          LASTENTRY   = 254
          PAGEEND     = 255
          SCAN        = 256
370       PAGE        = 257

      CLAIMED = 5000
    $

375 manifest  ||  InfromFile
    $  IFSIZE = 261

         FILENUMB    =   3  || also used by OuttoFile
         IFBUFFER    =   6
380
      ENDOFSTREAMCH = -1
      INFILESIZE    = 269  || (IFSIZE+1)+(FBSIZE+1)
    $

385 manifest  ||  OuttoFile
    $  OFSIZE = 269

       || FILENUMB  =  3  (defined above)

390       CUCHAIN   =  6

          FILEOWNER =  9
          OLDBODY   = 10
          OLDSIZE   = 11
395       OFPAGE    = 12
          NEWBODY   = 13
          OFBUFFER  = 14
    $

400
****
```

V: SEGMENTATION OF THE SYSTEM FOR COMPILATION

OS/1

```
§Pub1

get  'DECLARATIONS'
get  'LGLOOP'
get  'RUN'
get  'LOAD'
get  'SETLAB'

§Pub1
```

OS/2

```
§Pub2

get  'DECLARATIONS'
get  'GIVEUP'
get  'PM'
get  'MPM'
get  'INTERRUPT'

§Pub2
```

OS/3

```
§Pub3

get  'DECLARATIONS'
get  'FREESTORE'
get  'PUTBACK'

§Pub3
```

OS/4

```
$Pub4

get  'DECLARATIONS'
get  'STRPRIMITIVES'
get  'OUTS'
get  'OPRTS'
get  'NEXTN'

$Pub4
```

OS/5

```
$Pub5

get  'DECLARATIONS'
get  'TELETYPE'
get  'INTTT'
get  'XFER'
get  'READER'
get  'DIADS'

$Pub5
```

OS/6

```
$Pub6

get  'DECLARATIONS'
get  'CLOCK'
get  'MISC'

$Pub6
```

OS/7

```
$Pub7

get  'DECLARATIONS'
get  'DISCXFER'
get  'DISCFS'
get  'DISCIN'
get  'DISCOUT'

$Pub7
```

OS/8

```
$Pub8

get  'DECLARATIONS'
get  'CHARGE'
get  'UPDATE'
get  'UPDATEHEAD'
get  'FINDHEADING'
get  'LOOKUP'
get  'LOADFILE'

$Pub8
```

OS/SU

```
$PubSU

get   'DECLARATIONS'
get   'SYSSETUP'
get   'SETDANDT'
get   'QUICKVAL'

$PubSU
```

OS/SUS

```
$PubSUS

get   'DECLARATIONS'
get   'SETUPSTR'
get   'SETUPTT'
get   'SETUPRDR'

$PubSUS
```

VIII: SOME LIBRARY FILES

VIII:1 LogIn

```
get 'DECLARATIONS'

manifest || elements of Usercode entry :
§  ENTSIZE  = 0
   USERNAME = 1
   NAME1    = 2
   NAME2    = 3
   USERNUMB = 4
§

manifest
§  MAXSTRINGSIZE = 127  §


let Prog[] be
§P User, CurrentIndex := NULL, NULL
   OutS[Console, 'Name *q ']
  § let Name = String[Console]
   let v = LookUpUser[Name]
   ReturnVec[Name, MAXSTRINGSIZE]
   if v = NULL do
     § ReportS['No such user']
       Prog[]
       return
     §
   User := v!USERNUMB
   CurrentIndex := LookUp[v+v!NAME1, v+v!NAME2, SystemIndex]
   ReturnVec[v, v!ENTSIZE]
§P


and LookUpUser[Name] = valof
§L let UserCodes = LookUp['UserCodes', 'SYS', SystemIndex]
   let S = EntriesfromFile[UserCodes]
   until Endof[S] do
   §u let v = Next[S]
      if EqS[Name, v+v!USERNAME] do
        § Close[S]
          resultis v §
      ReturnVec[v, v!ENTSIZE]
   §u
```

```
        Close[S]
        resultis NULL
    §L
45
    and String[S] = valof
    §S  let v = NewVec[MAXSTRINGSIZE]
        let n, i = 0, 0
        let Ch = Next[S]
50      v!0 := 0
        until Ch='*n' do
        §u  n := n+1
            test n rem 2 = 1
              then v!i := (v!i) v Ch
55            or § i := i+1
                    v!i := Ch lshift 8 §
            Ch := Next[S]
        §u
        v!0 := (n lshift 8) v v!0
60      resultis v
    §S

****
```

VIII:2 MakeNewFile

```
get 'DECLARATIONS'

global || defined in this segment
§  DeleteFile       : 327
   MakeNewFile      : 328
   UpdatePermission : 313
§

global || defined in 'DiscRts'
§  AddVectoFile : 307
   NewLocation  : 308
§

manifest
§  ASIZE     = 1
     APAGE   = 0
     AWORD   = 1
   UNLOADED  = 0
   TITLE     = 11
§

manifest
§  MAXHDSIZE= 127 + TITLE   §

static
§  HeadingFile  = UNDEFINED
   MFLFile      = UNDEFINED
   StartMFLScan = UNDEFINED
§



let MakeNewFile[Title, Type] = valof
§M  if StartMFLScan = UNDEFINED do
     §1 HeadingFile := LookUp['Headings', 'SYS', SystemIndex]
        MFLFile     := LookUp['MFL', 'SYS', SystemIndex]
        StartMFLScan := FetchCode[MFLFIRSTPAGE]
      §1
```

```
        LoadDiscRtsifNec[]
      $ let HdAddr = NewLocation[HeadingFile]
        let File = NewMFLEntry[HdAddr↓APAGE, HdAddr↓AWORD]
        CreateNewHead[Title, Type, File, User, HdAddr]
45      ReturnVec[HdAddr, ASIZE]
        resultis File
    $M

    and LoadDiscRtsifNec[] be
50      if NewLocation=UNLOADED do LoadSystemFile['DiscRts']

    and NewMFLEntry[Page, Word] = valof
    $N  let p = vec (PAGESIZE-1)
        $r  DisctoCore[p, StartMFLScan]
55        $ let i = FIRSTDATA
            let f = 1 + ENTRIESPERPAGEX((p↓SERIAL)-1)
            until i > LASTDATA do
            $u  if p↓i = NOTENTRY do
                $1  p↓(i+HPAGE) := Page
60                  p↓(i+HWORD) := Word
                    CoretoDisc[p, StartMFLScan]
                    resultis f
                $1
                i := i + (ENTRYSIZE+1)
65              f := f+1
            $u
            if p↓NXTPAGE = ENDBODY do
            $2  p↓NXTPAGE := NewMFLPage[(p↓SERIAL)+1]
                CoretoDisc[p, StartMFLScan]
70          $2
            StartMFLScan := p↓NXTPAGE
        $r  repeat
    $N

75  and NewMFLPage[Serial] = valof
    $P  let Page = NewDiscBlock[]
        DiscPage↓SERIAL   := Serial
        DiscPage↓THISFILE := MFLFile
        DiscPage↓NXTPAGE  := ENDBODY
80      for i=FIRSTDATA to LASTDATA do DiscPage↓i := NOTENTRY
        CoretoDisc[DiscPage, Page]
        UpdateHead[MFLFile, SAME, Serial, Page,
                                  SAME, FetchCode[DATE], SAME]
        resultis Page
85  $P
```

```
    and CreateNewHead[Title, Type, File, Owner, HdAddr] be
    §C  let v = vec MAXHDSIZE
        let TitleLength = (Title%0 rshift 8)/2
90
        v%0 := (TITLE lshift 8)|v(TITLE + TitleLength)
        v%FIRSTPAGE   := NULLBODY
        v%NUMBPAGES   := 0
        v%LASTPAGE    := NULLBODY
95      v%TYPE        := Type
        v%OWNER       := Owner
        v%CREATED     := FetchCode[DATE]
        v%DLW         := NEVER
        v%DLR         := NEVER
100     v%PERM        := OWNERONLY
        v%MFLNUMB     := File
        Copy[Title, lv v%TITLE, TitleLength+1]

        AddVectoFile[HeadingFile, HdAddr, v, (TITLE+TitleLength)]
105 §C


    let UpdatePermission[f, Perm] be
    §u  CheckLegality[f, 'update permission']
110     Update[f, PERM, Perm]
    §u

    and DeleteFile[f] be
    §D  CheckLegality[f, 'delete file']
115     DeleteBody[f]
        Update[f, TYPE, DELETED]
    §D

    and CheckLegality[f, Description] be
120 §C  let h = FindHeading[f]
        if (h=NULL)|(User#h%OWNER) do
          § ReportS['Illegal attempt to *S', Description]
            GiveUp[f] §
        ReturnVec[h, h%0 ^ LMASK]
125 §C

****
```

VIII:3 Index Ops

```
get 'DECLARATIONS'

global  ||  defined in this segment
§   DeleteEntry : 314
    Enter       : 316
    Link        : 315
§

global  ||  defined in 'DiscRts'
§   AddVectoFile : 307
    NewLocation  : 308
§

manifest
§   ASIZE = 1
    UNLOADED = 0
§

static  ||  used by Link
§   p=0; q=0; i=0   §

let Enter[f, Name1, Name2, Ind] be
§E  if f<0 do
      § ReportS['Enter[*N, *S, *S, *N]', f, Name1, Name2, Ind]
        GiveUp[f] §
    CheckPermission[Ind]

  § let g = LookUp[Name1, Name2, Ind]
    if g#NULL do DeleteEntry[Name1, Name2, Ind]

    AddEntry[f, Name1, Name2, Ind]
§E
```

```
45 and AddEntry[f, Name1, Name2, Ind] be
   §AE LoadDiscRtsifNec[]
     § let L1, L2 = Size[Name1], Size[Name2]
       let n = 3+L1+L2
       let v = NewVec[n]
50
       v↓ENTRYEND := n
       v↓N1 := 4
       v↓N2 := 4+L1
       v↓FILE := f
55     Copy[Name1, v+v↓N1, L1]
       Copy[Name2, v+v↓N2, L2]

     § let Addr = NewLocation[Ind]
       AddVectoFile[Ind, Addr, v, n]
60     ReturnVec[Addr, ASIZE]

       ReturnVec[v, n]
   §AE

65
   and Link[Name1, Name2, Na, Nb, Nc, Nd, Ind] be
   §L  CheckPermission[Ind]
       CheckLinkDoesntLoop[Name1, Name2, Na, Nb, Nc, Nd, Ind]

70   § let g = LookUp[Name1, Name2, Ind]
       if g≠NULL do DeleteEntry[Name1, Name2, Ind]

       AddLinkedEntry[Name1, Name2, Na, Nb, Nc, Nd, Ind]
   §L
75

   and CheckLinkDoesntLoop[Name1, Name2, Na, Nb, Nc, Nd, Ind] be
   §CL p, q, i := Name1, Name2, Ind
     § let Dummy = Check[Na, Nb, Check[Nc, Nd, SystemIndex]]
80 §CL

   and Check[Name1, Name2, Ind] = valof  ||  version of LookUp
   §C
       if EqS[Name1, p] ^ EqS[Name2, q] ^ (Ind=i) do
85       § ReportS['Attempted loop in Link']
           GiveUp[Ind] §
```

```
    if Ind = NULL resultis NULL
  § let S = EntriesfromFile[Ind]
90  until Endof[S] do
    §u  let v = Next[S]
        if v↓STATUS ‡ DELETED do
          if EqS[Name1, v+v↓N1] ^ EqS[Name2, v+v↓N2] do
            §1  Close[S]
95              § let NotLinked = (v↓LINKING ≥ 0)
                  let f = NotLinked → v↓FILE,
                            Check[v-v↓N3, v+v↓N4,
                             Check[v+v↓N5, v+v↓N6, SystemIndex]]
                  ReturnVec[v, v↓ENTRYEND]
100               resultis f
                §1
        ReturnVec[v, v↓ENTRYEND]
    §u
    Close[S]
105 resultis NULL
  §c


  and AddLinkedEntry[Name1, Name2, Na, Nb, Nc, Nd, Ind] be
110 §A  LoadDiscRtsifNec[]
    § let L1, L2 = Size[Name1], Size[Name2]
      let La, Lb = Size[Na], Size[Nb]
      and Lc, Ld = Size[Nc], Size[Nd]
      let n = 6+L1+L2+La+Lb+Lc+Ld
115   let v = NewVec[n]

      v↓ENTRYEND := n
      v↓N1 := 7
      v↓N2 := 7+L1
120   v↓N3 := v↓N2 + L2
      v↓N4 := v↓N3 + La
      v↓N5 := v↓N4 + Lb
      v↓N6 := v↓N5 + Lc
      Copy[Name1, v+v↓N1, L1]
125   Copy[Name2, v+v↓N2, L2]
      Copy[Na, v+v↓N3, La]
      Copy[Nb, v+v↓N4, Lb]
      Copy[Nc, v+v↓N5, Lc]
      Copy[Nd, v+v↓N6, Ld]
130   v↓LINKING := -(v↓LINKING)
          ||  negative sign indicates linked entry
```

```
      § let Addr = NewLocation[Ind]
        AddVectoFile[Ind, Addr, v, n]
135     ReturnVec[Addr, ASIZE]
        ReturnVec[v, n]
    §A


140
    and LoadDiscRtsifNec[] be
        if NewLocation = UNLOADED do LoadSystemFile['DiscRts']

    and CheckPermission[f] be
145 §P  let h = FindHeading[f]
        let Perm = h↓PERM and Owner = h↓OWNER
        ReturnVec[h, h↓0^LMASK]
        if (Perm=RESTRICTED)v((Perm=OWNERONLY)^(User≠Owner)) do
          § Reports['Illegal attempt to write to file *N', f]
150       GiveUp[Owner] §
    §P

    and Size[String] = 1 + ((String↓0) rshift 8)/2

155


    let DeleteEntry[Name1, Name2, Ind] be
    §D
160     let Input = EntriesfromFile[Ind]
        let Output = OuttoFile[Ind]

        until Endof[Input] do
        §u  let v = Next[Input]
165         let Size = v↓ENTRYEND
            if v↓STATUS ≠ DELETED do
              unless EqS[Name1, v+v↓N1] ^ EqS[Name2, v+v↓N2] do
                     TransferOut[Output, v, Size+1]
            ReturnVec[v, Size]
170     §u
        Close[Output]
        Close[Input]
    §D

****
```

VIII:4 File Vectors

```
get 'DECLARATIONS'

global || defined in this segment
§   AddMoreVectoFile : 325
    VectorfromFile   : 323
    VectortoFile     : 324
§

global || defined in 'DiscRts'
§   AddVectoFile : 307
    NewLocation  : 308
§

manifest
§   ASIZE = 1
    UNLOADED = 0
§

let VectortoFile[v, f] be
§V  let S = OuttoFile[f]
    TransferOut[S, lv(v!1), v!0]
    Close[S]
§V

let VectorfromFile[f] = valof
§V  let h = FindHeading[f]
    if h = NULL do
      § ReportS['Vectorfrom (deleted) file']
        GiveUp[f]
      §
    DisctoCore[DiscPage, h!LASTPAGE]
  § let Unused = LASTDATA - (DiscPage!ENDOFDATAPTR - 1)
    let n = (DATASIZE X h!NUMBPAGES) - Unused
    ReturnVec[h, h!0^LMASK]

  § let v = NewVec[n]
    v!0 := n
  § let S = InfromFile[f]
    TransferIn[S, lv(v!1), v!0]
    Close[S]
    resultis v
§V
```

```
45
   let AddMoreVectoFile[v, f] be
   §A let h = FindHeading[f]
      CheckPerm[f, h]
      ReturnVec[h, h↓0^LMASK]
50    LoadDiscRtsifNec[]
    § let Addr = NewLocation[f]
      AddVectoFile[f, Addr, lv(v↓1), (v↓0 - 1)]
      ReturnVec[Addr, ASIZE]
   §A
55


   and LoadDiscRtsifNec[] be
60    if NewLocation = UNLOADED do LoadSystemFile['DiscRts']


65
   and CheckPerm[f, h] be
   §C let Perm = h↓PERM and Owner = h↓OWNER
      if (Perm=RESTRICTED)v((Perm=OWNERONLY)^(User≠Owner)) do
        § ReportS['Illegal attempt to AddMoreVectoFile *N', f]
70        GiveUp[Owner]
   §C
```

****

VIII:5 DiscRts

```
get 'DECLARATIONS'

global  ||  defined in this segment
$   AddVectoFile : 307
    NewLocation  : 308
$

manifest  ||  the size and elements of the location vector
$   ASIZE = 1
        APAGE = 0
        AWORD = 1
$

static  $ Vec = 0 $

let NewLocation[f] = valof
$NL let Addr = NewVec[ASIZE]
    let h = FindHeading[f]
    let Owner = h!OWNER
    let Page = h!LASTPAGE and Word = 0
    ReturnVec[h, h!0^1MASK]

    test Page = NULLBODY
      then $ Page := MakeOnePageBody[f, Owner]
             Word := FIRSTDATA
           $
      or   $ DisctoCore[DiscPage, Page]
             Word := DiscPage!ENDOFDATAPTR
           $
    Addr!APAGE, Addr!AWORD := Page, Word
    resultis Addr
$NL

and MakeOnePageBody[f, Owner] = valof
$AP let Page = NewDiscBlock[]
    DiscPage!SERIAL   := 1
    DiscPage!THISFILE := f
    DiscPage!NXTPAGE  := ENDBODY
    DiscPage!ENDOFDATAPTR := FIRSTDATA
    CoretoDisc[DiscPage, Page]
```

```
        UpdateHead[f, Page, 1, Page, SAME, FetchCode[DATE], SAME]
        Charge[Owner, 1]
45      resultis Page
    $AP


    let AddVectoFile[f, Addr, v, n] be
50  $AV let Page, Word = Addr↓APAGE, Addr↓AWORD
        Vec := NewVec[PAGESIZE-1]
        DisctoCore[Vec, Page]
        for i=0 to n do
        $f  if Word > LASTDATA do
55              $ Page := TurnPage[Page]
                  Word := FIRSTDATA
                $
            Vec↓Word := v↓i
            Word := Word+1
60      $f

        if Word > LASTDATA do
           $ Page := TurnPage[Page]
             Word := FIRSTDATA
65         $
        Vec↓ENDOFDATAPTR := Word
        CoretoDisc[Vec, Page]

      $ let h = FindHeading[f]
70      UpdateHead[f, SAME, Vec↓SERIAL, Page,
                           SAME, FetchCode[DATE], SAME]
        Charge[h↓OWNER, (Vec↓SERIAL - h↓NUMBPAGES)]
        ReturnVec[h, h↓0^LMASK]
        ReturnVec[Vec, (PAGESIZE-1)]
75  $AV

    and TurnPage[p] = valof
    $TP let Page = NewDiscBlock[]
        Vec↓NXTPAGE := Page
80      CoretoDisc[Vec, p]
        Vec↓SERIAL := Vec↓SERIAL + 1
        Vec↓NXTPAGE := ENDBODY
        resultis Page
    $TP
85

****
```

VIII:6 LinePrinter

```
    get 'DECLARATIONS'

    global || defined in this segment
 5  $  GeneralLinePrinter : 80
       LinePrinter        : 98
    $

10
    manifest || BytestoLP stream
    $  BLPSIZE = 23
          CLEARUP = 19
          BUFF    = 23
15     LPPINGS =  5
       PRINTER = 28
       PAUSE   = 1000
    $

20  manifest || GeneralIntcodetoLP
    $  LPSIZE = 10
          ULBUFF      = 7
          ULPTR       = 8
          ERROR       = 9
25        PAGETHROWS =10

       LPLINELENGTH = 80
          FLAG = 0
    $
30

    manifest || Line Printer Character Code
    $  BARp       = 8 41  ;  PAGETHROWp = 8 14
       DOWNARROWp = 8 46  ;  PRIMEp     = 8 47
35     GRAVEp     = 8 43  ;  RETURNp    = 8 15
       LAMBDAp    = 8134  ;  ROUNDOp    = 8 60
       LOGORp     = 8 42  ;  RTARROWp   = 8100
       MULTp      = 8140  ;  SHARPp     = 8174
       NARROWOp   = 8117  ;  SPACEp     = 8 40
40     NEQp       = 8 44  ;  ULp        = 8137
       NEWLINEp   = 8 12  ;  UPARROWp   = 8 45
    $
```

```
45 manifest  ||  Extra Internal Character Code
   $    ULBIT       = 8200  ;  EXCLAM     = 8 41
        ACUTE       = 8 47  ;  HOOK       = 8 22
        AT          = 8100  ;  LAMBDA     = 8 34
        BACKSTROKE  = 8134  ;  LETTERO    = 8117
50      DIGITO      = 8 60  ;  SHARP      = 8 43
        DOLLAR      = 8 44  ;  STOPCODE   =   4
   $

   manifest  ||  everything else
55 $    DEFAULT  = 0
        MASKUL   = 8177577
        COMPLETE = 0
        UNPRINTABLE = 1
        OVERFLOW    = 2
60      PAPERWASTE  = 3
        IRRELEVANT = 0
   $


65
   let LinePrinter[] = GeneralLinePrinter[DEFAULT]


70 and GeneralLinePrinter[ErrorFunction]
       = GeneralIntcodetoLP[BytestoLP[], ErrorFunction]


75 and BytestoLP[] = valof
   $B  let v = NewVec[BLPSIZE]

       v!NEXT        := StreamError
       v!OUT         := OutBLP
80     v!CLOSE       := CloseBLP
       v!PINGS       := LPPINGS
       v!ENDOF       := StreamError
       v!RESET       := NullProgram
       v!SOURCE      := v
85     v!STATE       := StreamError
       v!RESETSTATE  := StreamError
```

```
    § let E = lv v↓EXECBLOCK
      E↓DEVICE    := PRINTER
90    E↓BUFFER    := lv v↓BUFF
      E↓BUFFSIZE  := 1
      E↓STOPCH    := NOTBYTE
      E↓SEG       := DATASEG
      E↓ENDMODE   := QUIETEND
95    E↓SELFPTR   := E
      || The other three elements of the Execblock are not used

    § let c = lv v↓CLEARUP
      c↓CSUC     := ClearUpChain
100   c↓CPRE     := lv ClearUpChain
      c↓ROUTINE  := ClearUpLP
      ClearUpChain := c
      unless c↓CSUC=ENDCUCHN do (c↓CSUC)↓CPRE := c

105   v↓(BUFF-1) := lv v↓BUFF  || used by InitiateTransfer

      resultis v
    §B

110

    and OutBLP[v, x] be
    §OB v↓BUFF := x
      InitiateTransfer[v, 1, 'Printer']
115 §OB


    and CloseBLP[v] be
    §CB let c = lv v↓CLEARUP
120   (c↓CPRE)↓CSUC := c↓CSUC
      unless c↓CSUC=ENDCUCHN do (c↓CSUC)↓CPRE := c↓CPRE

      for i=1 to PAUSE do § §
      Cancel[]
125
      ReturnVec[v, BLPSIZE]
    §CB


130 and ClearUpLP[c] be Cancel[]
```

```
    and Cancel[] be
    $C  let i = 0
135     Exec[CANCEL, L, lv i, 0, PRINTER]
    L:  $C


    and GeneralIntcodetoLP[Str, ErrorFn] = valof
140 $GI let S = NewVec[LPSIZE]
        S!NEXT    := StreamError
        S!OUT     := TrapPageThrow  || top of page assumed
        S!CLOSE   := CloseLP
        S!STR     := Str
145     S!ENDOF   := StreamError
        S!RESET   := ResetLP
        S!SOURCE  := Source[Str]

        S!PAGETHROWS := 1
150     S!ERROR   := (ErrorFn=DEFAULT) -> StandardErrorFn, ErrorFn

      $ let B = NewVec[LPLINELENGTH]  || buffer for underlining
        S!ULBUFF  := B
        S!ULPTR   := 0
155     B!FLAG    := false

        resultis S
    $GI

160


    and OutLP[S, x] be
    $O  let B = S!ULBUFF
165     let CharUnderlined = ((x^ULBIT)#0)
        let y = x^MASKUL
        let Ch = valof
        $v  switchon y into
            $s  default :
170                 test '(' <= y <= DELETE
                      then resultis y
                      or endcase  || unprintable character

                case '*4' :
175                 for i=1 to 4 do
                        OutLP[S, CharUnderlined -> '_', '*s']
                    resultis COMPLETE
```

```
                case '*n' :
180                 OutputUnderlines[S]
                    Out[S↓STR, NEWLINEp]
                    S↓ULPTR := 0
                    resultis COMPLETE

185             case RETURN :
                    OutputUnderlines[S]
                    Out[S↓STR, RETURNp]
                    S↓ULPTR := 0
                    resultis COMPLETE
190
                case '*p' :
                    OutputUnderlines[S]
                    Out[S↓STR, PAGETHROWp]
                    S↓ULPTR := 0
195                 S↓OUT := TrapPageThrow
                    S↓PAGETHROWS := 1
                    resultis COMPLETE

                case '*b' :
200                 if S↓ULPTR = 0 resultis COMPLETE
                    OutputUnderlines[S]
                    Out[S↓STR, RETURNp]
                    S↓ULPTR := S↓ULPTR-1
                    for i=1 to S↓ULPTR do
205                   § Out[S↓STR, SPACEp]
                        B↓i := SPACEp     §
                    resultis COMPLETE


210             case '*s' :         resultis SPACEp
                case PRIME :        resultis PRIMEp
                case '→' :          resultis RTARROWp
                case '∨' :          resultis LOGORp
                case '×' :          resultis MULTp
215             case LAMBDA :       resultis LAMBDAp
                case '≠' :          resultis NEQp
                case '↑' :          resultis UPARROWp
                case '↓' :          resultis DOWNARROWp
                case EXCLAM :       OutLP[S, '.']
220                                 OutLP[S, '*b']
                                    resultis PRIMEp
                case SHARP :        resultis SHARPp
```

```
                case DOLLAR :       OutLP[S, '$']
                                    OutLP[S, '*b']
                                    resultis BARp
                case DIGIT0 :       resultis NARROWOp
                case LETTER0 :      resultis ROUNDOp
                case GRAVE :        resultis GRAVEp
                case '|' :          resultis BARp

                case AT :
                case BACKSTROKE :
                case LEFTARROW :    endcase  || all unprintable
            §s

            x := (S!ERROR)[S, UNPRINTABLE, x]
            unless x = COMPLETE do OutLP[S, x]
            resultis COMPLETE
        §v

        if S!ULPTR ≥ LPLINELENGTH do
            Ch := (S!ERROR)[S, OVERFLOW, Ch]

        if Ch = COMPLETE return

        Out[S!STR, Ch]
        S!ULPTR := S!ULPTR + 1
        B!(S!ULPTR) := CharUnderlined → ULp, SPACEp
        B!FLAG := CharUnderlined ∨ B!FLAG
    §0




and OutputUnderlines[S] be
    §UL let B = S!ULBUFF
        unless B!FLAG return
        B!FLAG := false
        Out[S!STR, RETURNp]
        TransferOut[S!STR, lv (B!1), S!ULPTR]
    §UL



and TrapPageThrow[S, x] be
    §TP test x = '*p'
          then S!PAGETHROWS := S!PAGETHROWS + 1
```

```
        or § let p = S!PAGETHROWS
             S!PAGETHROWS := 0
             S!OUT := OutLP
             if p>1 do p := (S!ERROR)[S, PAPERWASTE, p]
             OutLP[S, x]
    §TP

and CloseLP[S] be
    §C  let B = S!ULBUFF
        let p = S!PAGETHROWS
        OutputUnderlines[S]
        ReturnVec[B, LPLINELENGTH]
        if p>1 do p := (S!ERROR)[S, PAPERWASTE, p]
        if S!PAGETHROWS = 0 do Out[S!STR, PAGETHROWp]
        Close[S!STR]
        ReturnVec[S, LPSIZE]
    §C

and ResetLP[S] be
    §R  unless S!ULPTR=0 do Out[S, '*n']
      § let p = S!PAGETHROWS
        if p>1 do
          § p := (S!ERROR)[S, PAPERWASTE, p]
            S!PAGETHROWS := 1
          §
        Reset[S!STR]
    §R

and StandardErrorFn[S, Reason, x] = valof
    §E  switchon Reason into
        §s  case UNPRINTABLE:
                §1  let y = x^MASKUL
                    if y=ACUTE resultis PRIME\/(x^ULBIT)
                    if (x=RUNOUT)\/(x=STOPCODE) resultis COMPLETE

                    test HOOK <= y <= DELETE
                      then § PrinterReport['*O unprintable', x]
                              resultis '*s' \/(x^ULBIT)
                           §
                      or   § PrinterReport['*O invalid', x]
                              resultis COMPLETE
                §1
```

```
            case OVERFLOW:
315             Out[S, '*n']
                PrinterReport['Line too long']
                resultis x

            case PAPERWASTE:
320             PrinterReport['*N consecutive pagethrows', x]
                resultis IRRELEVANT

            default :
                PrinterReport['Invalid call of StandardErrorFn']
325             resultis NULL
    $E

    and PrinterReport[String, a, b, c, d] be
    $PR OutS[ReportStream, '*4Printer: ']
330     ReportS[String, a, b, c, d]
    $PR



335 and Source[str] = valof
    $S  if Str<0 do Str := (~Str)↓SLOWBLOCK
        resultis Str↓SOURCE
    $S

340
****
```

INDEX