# THE TEXT OF $OSP_{ub}$

by

Christopher Strachey

and

Joseph Stoy

Oxford University

(COMMENTARY)

Technical Monograph PRG-9(c)

July 1972

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

© 1972 Christopher Strachey and Joseph Stoy.

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 [1, 2], 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 [1, 2] on OS6. 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.)


| | Text | Comm |
|---|---|---|
| Foreword | | |
| [I. Introduction | | |
| Aims of publication, Differences from 'OS6' papers, Style, Guide to these books.] | | 1 |
| II. The text of the system | | |
| 1. The load-go loop, Run and Load | 1 | 8 |
| 1.1 LGLOOP | 1 | 8 |
| LoadGoLoop, LGLoop, [Prog,] DefaultProg. | 1 | 8 |
| 1.2 RUN | | |
| Run, PrepareforRun, TerminateRun, ClearUp, [ClearUpChain,] LogIn, Finish. | 2 | 9 |
| 1.3 LOAD | | |
| [Note on binary format, Format of compiled BCPL segment,] Load, [SectId, IBlock, Format of loaded section, CPtr, CFirst,] LoadSection, Unload. | 5 | 11 |
| 1.4 SETLAB | | |
| SetLabels, SetGlobals. | 9 | 15 |
| 2. Post-mortem arrangements and interrupts | 11 | 18 |
| 2.1 GIVEUP | 11 | 18 |
| [GiveUp, GiveUpStackSize, GiveUpStack,] ForcedGiveUp, InStack, StandardGiveUp, Dump, FetchExecWord, SetUpDummyExecStructure, DumpSegment, EndGiveUp, ForcedFinish. | | |
| 2.2 PM | | |
| StandardPM, ReportCallTrace, ReportFreeStoreState, ReportBlocks, ReportWords. | 15 | 22 |
| 2.3 MPM | | |
| ManualPM, Exchange, Return, JumpTo, AnythingTyped, MPMNextN, MPMNextO. | 17 | 23 |
| 2.4 INTERRUPT | | |
| [PPtr, PrivateStack,] Interrupt. | 21 | 26 |

3. Storage allocation and PutBack 23 29
3.1 FREESTORE 23 29
[FS,] NewVec, ReturnVec, RVGiveUp, NewWord, ReturnWord, MaxVecSize, NewFreeStore, RestoreFreeStore.
3.2 PUTBACK 28 35
[PBChain,] PutBack, NextPB, FNextPB, ClosePB, FClosePB, ResetPB, FResetPB, EndofPB.

4. Stream primitives and I/O routines 30 38
4.1 STRPRIMITIVES 30 38
Close, Reset, Source, State, ResetState, StreamError.
4.2 OUTS 31 40
OutS, WriteS, ReportS, OutString, GetC, OutJustify, Size, OSReport, OSReportN.
4.3 OPRTS 34 42
OutP, OutN, Out9, OutO, OutByte, OutAddr, Write, WriteN, WriteO, WriteByte, WriteAddr.
4.4 NEXTN 36 44
NextN, NextO, NextCh.

5. Permanent input/output streams 37 45
5.1 TELETYPE 37 45
[Teletype,] NextTT, OutNewLine, OutTT, ResetTT, EndofTT, StateTT, ResetStateTT.
5.2 INTTT 40 48
[ExecConsole,] IntcodetoandfromTeletype, NextTele, EvenParity, OutTele, OutCRLF, OutNewLines, CloseTele, ResetTele, EndofTele, (Source).
5.3 XFER 43 51
InitiateTransfer.
5.4 READER 44 53
[BytesfromPT, ReaderDev,] ReadBuff, FirstNextBFPT, NextFillBFPT, ResetBFPT, NextWaitBFPT, StateBFPT, ReaderOffLine, EndofBFPT, TryAgain.
5.5 DIADS 47 59
[Note on diad format, DiadRead,] WordsfromDiads, NextBlock, NextByte, NextCh, EndofW, ResetW, CloseW, TryDiadAgain, ReadBackwards.

6. Miscellaneous facilities 51 64
6.1 CLOCK 51 64
OutDateandTime, OutDate, OutTime, TimeofDay, RestartClock, AskTime.
6.2 MISC 53 65
AddressZero, Copy, EqS, NullProgram, Wait.

7. Disc routines (1) 54 67
7.1 DISCXFER 54 67
[DiscPage,] CoretoDisc, DisctoCore, DiscTransfer, Message.
7.2 DISCFS 56 70
[FSF,] NewDiscBlock, ReturnDiscBlock.
7.3 DISCIN 58 74
CheckType, InfromFile, NextBuffIF, EndofIF, CloseIF, ResetIF, EntriesfromFile, NextEF, EndofEF, CloseEF, ResetEF.
7.4 DISCOUT 61 77
ReturnChain, CheckPerm, DeleteBody, OuttoFile, OutBuffOF, TurnPage, CloseOF, ResetOF, FailClose.

8. Disc routines (2) 65 82
8.1 CHARGE 65 82
Charge.
8.2 UPDATE 66 82
Update.
8.3 UPDATEHEAD 67 83
UpdateHead.
8.4 FINDHEADING 68 84
LookUpinMFL, FindHeading.
8.5 LOOKUP 70 85
LookUp.
8.6 LOADFILE 71 88
LoadFile, LoadSystemFile.

9. Special functions in WIC 72 89
FetchCode, StoreCode, Exec [(TRANSFER (the executive teletype, the paper tape reader, the paper tape punch, the line printer, the disc, the clock, remote consoles), CANCEL, LOOKATRDR, READABS)], Sumcheck, Next, Out, Endof, TransferIn, TransferInC, TransferOut.

| | | |
|---|---|---|
| III. Set-up programs | 75 | 97 |
| 1. System set-up | 75 | 97 |
| 1.1 SYSSETUP | 75 | 97 |
| [FSLim, CPages, DPages,] PInterrupt, SetUpFS, SetUpDiscPage, SetUpPMStacks SetUpSundryItems, SetUpTimeofDayClock, SetUpRunBlock, SetStackBase, CheckDiscOn, ReleaseNonSystemGlobals, FSFtoCore, LogIn. | | |
| 1.2 SETDANDT | 81 | 103 |
| SetDateandTime, Date, Wrong, Leap, NextLetter. | | |
| 1.3 QUICKVAL | 83 | 105 |
| QuickValFSF, ReportMessage. | | |
| 2. Set up streams | 85 | 106 |
| 2.1 SETUPSTR | 85 | 106 |
| SetUpStreams, [Parity,] SetUpParityTable, [Output, ReportStream, Console, In]. | | |
| 2.2 SETUPTT | 86 | 107 |
| SetUpExecConsole, TT. | | |
| 2.3 SETUPRDR | 88 | 108 |
| SetUpReader, BFPT. | | |
| IV. Declarations | 90 | 109 |
| 1. DECLARATIONS | 90 | 109 |
| 1.1 GLOBALS | 91 | 109 |
| 1.2 PRIVATE GLOBALS | 93 | 109 |
| 1.3 CONSTANTS | 94 | 110 |
| General, Machine constants, Exec commands, Exec block for TRANSFER command, Standard contents of TRANSFER block elements, Device numbers, Code segment addresses (INTERRUPTINHIBITED, REASONFORINTERRUPT, MAXD, MAXC, SUMCHINHIBITED, DISCWRITEPERMITTED, MFLFIRSTPAGE, DATE, RBLOCK, TIMEOFDAYCLOCK), Reasons for interrupt, Clock, Run block, Information block, Free store, ClearUpChain, Stream vector, Stream elements used by InitiateTransfer, Fast stream block, Teletype stream, Reader stream, PutBack vector, Internal character code, Teletype character code, Stack element, Filing system, Headings, Index entries, Some file types, File permissions, Master file list, Free store file, InfromFile, OuttoFile. | | |

V. Segmentation of the system for compilation 103 115
OS/1 - OS/8, OS/SU, OS/SUS.

[VI. List of failure reports] 116

[VII.The system index 119
SystemIndex, Files with second name 'IC',
Files with second name 'Index', Other files.]

VIII.Some library files 107 125
1. LogIn 107 125
[User, CurrentIndex,] Prog, LookUpUser,
String.
2. MakeNewFile 109 127
MakeNewFile, LoadDiscRtsifNec, NewMFLEntry,
NewMFLPage, CreateNewHead, UpdatePermission,
DeleteFile, CheckLegality.
3. Index Ops 112 129
Enter, AddEntry, Link, CheckLinkDoesntLoop,
Check, AddLinkedEntry, (LoadDiscRtsifNec,)
CheckPermission, Size, DeleteEntry.
4. File Vectors 116 133
VectortoFile, VectorfromFile,
AddMoreVectoFile, (LoadDiscRtsifNec,)
CheckPerm.
5. DiscRts 118 134
NewLocation, MakeOnePageBody, AddVectoFile,
TurnPage.
6. LinePrinter 120 135
Line Printer, GeneralLinePrinter, BytestoLP,
OutBLP, CloseBLP, ClearUpLP, Cancel,
GeneralIntcodetoLP, OutLP, OutputUnderlines,
TrapPageThrow, CloseLP, ResetLP,
StandardErrorFn, PrinterReport, (Source).

[References] 144
[Appendix: BCPL] 145
Index 128 152

## I: INTRODUCTION

The text of this operating system is published as a supplement to the authors' two papers [1,2] on OS6 - an experimental operating system for a small computer. We had originally hoped to publish the text of OS6 as it stood; however, when the staff working on the system began preparing it for publication they decided that it could not possibly be published until some of the more horrible parts had been revised and the whole system generally tidied up. The result of this tidying (which involved the production of OS7 as an intermediate stage) is the present system, called OSPub.

However, although OSPub was produced expressly for publication, it is not an untested system. It has, in fact, for some months been the standard system in general use on our Modular One computer. The same files on the disc have been used both as a source for the printed text of the system and as input for the compiler to produce the system in the machine.

### Aims of Publication

We have three reasons for publishing the complete text of this system. In the first place, of course, we wish to illustrate in detail the principles outlined in the description of the system in [1] and [2].

In the second place, we wish actually to publish an example of a complete operating system written in a high level language. The only way a mathematical theorem becomes accepted is by publishing the proof, so that other workers in the field either pull it to pieces and expose the flaws in its argument, or convince themselves of its validity. We believe that in the long run the same sort of criteria must be applied to programs, and we hope that this publication is at least a step towards this goal. It has been said before (by Dijkstra [3], for example) that the fact that a program has been successfully tested proves only that it processes its test data correctly; the validity of a program in general can only be really accepted by someone who has convinced himself by examining the program's structure in detail. If this is to be possible with sufficient mathematical precision, the program text must be made available in a high level

programming language; neither an informal description in English on the one hand nor a machine code listing on the other is satisfactory.

We are not pretending that BCPL, or indeed any present day programming language, is a perfect vehicle of expression. A language ought to have the property that every important relationship within a program is either stated explicitly in the notation or taken for granted by the 'educated reader'. Current languages, with hidden side effects and similar features, come nowhere near fulfilling this condition. But we should not worry too much about this inadequacy. Our theoretical understanding of system programming techniques, and our experience of expressing them in anything more disciplined than machine code, are both so limited that an attempt to refine our languages now might well make matters worse. We would probably cut out the wrong things, and introduce fresh complications as we tried to break out of the straitjacket into which we would have tied ourselves. What we should do now, instead, is learn to express ourselves as clearly and naturally as possible with the tools available, eschewing 'clever tricks' and taking care, as a matter of programming style, that the important points affecting the validity of the program are made obvious in the program text. Then, when sufficient experience and understanding have been gained, we shall be able to add mathematical rigour to our intuitive arguments, by adapting the semantics of our languages to capture exactly what we will have discovered we want to say, and to leave the rest not merely unsaid, but absent even by implication.

So a third aim of this publication is to present an extended and practical piece of programming as an example for discussion of programming style. Again, we by no means pretend that it is flawless: indeed, for any work of this size, composed by several people, perfection or even uniformity of style would be impossible (cf., for example, [4]). We hope that the reader will identify for himself some of the worst passages (and perhaps think of improvements), and also, more importantly, that he will consider whether any more general defects, in the language itself or in our use of it, are impairing the clarity of expression.

## Differences from 'OS6' papers

Regrettably, there still remain several discrepancies between the system published here and the description in [1] and [2]. We

list them all together here, with explanations. As might be expected, most of them concern I/O.

| | |
|---|---|
| [1] §0.0 | Our configuration now includes 56K of core. Most of this has a cycle time of 1.5 microseconds. However, since the interpreter is still stored in fast core, the slower core has very little effect on the speed of the system. |
| [1] §0.2 | The size of the interpreter is now about 400 instructions. The increase is due mainly to the extra instructions described in [2] §2.6. |
| [1] §1.2.1 | The standard version of GiveUp has changed (as forecast in [1] §3.1). It now offers the choice either of the standard diagnostic information as before or of dumping a core image onto the disc for subsequent analysis. |
| [1] §3.1 | Sleuth has not kept up with the changes to the virtual machine code and is now obsolete. |
| [2] §2.1 | Note that in fact BytestoPT is a stream function, not a stream. |
| [2] §2.4.4 | State and ResetState are now defined on BytesfromPT as well as on the bilateral streams. |
| [2] §2.5.3 | PutBack needed enhancement to deal with fast streams (see [2] §2.6). It now requires an extra element of the PutBack vector to store the current value of a fast stream's input buffer pointer. |
| [2] §3.1.2 | The objects called disc vectors have not earned themselves a place in the system, though they are privately available. |
| [2] §3.2 | EntriesFrom is now called EntriesfromFile. |
| [2] §3.4 | The contents of the first two words of the housekeeping information at the start of a page of file body have been interchanged. This is because the first word is overwritten when the page is returned to free store. If this happens by accident it is more helpful when salvaging the situation to know the file to which the page used to belong than what serial number it had; so the serial now comes first.<br>The mistake concerning the address of the first page of the MFL body has now been rectified; this value is now kept in a reserved word of the code segment. |
| [2] §3.4.1 | The page of the free store file in core is not kept in the program segment: this was pure wishful thinking on the part of the authors. We have not yet put our wishes into effect as we are considering further |

redesign of the free storage system (see DiscFS, II:7.2).

The page in core is written back to the disc at the start of each Run, not the end. It is more likely to be correct when the system is preparing to start a user program than when it is clearing up the remains.

[2] §3.4.2 A user may legitimately replace the main index in which he is working by another. UserIndex is therefore more properly named CurrentIndex.

The accounting system is not implemented - see Charge, II:8.1.

## Style

Having now discussed the content of the system, we return briefly to the question of style. We hope that BCPL is such that the program text may be understood even by someone having no formal acquaintance with the language; nevertheless, some of its details which might otherwise cause confusion are listed in the appendix, and the reader is referred to the reference manual [5] for further information. Here we draw attention to one or two matters concerning our overall treatment of the system.

It is perhaps worth noting that the word '<u>goto</u>', though available in the language, is not used at all, and the only labels in the system are those required by the specification of the pseudo-hardware Exec routine (II:9). So we avoid altogether what is becoming recognised (see, for example, Wulf, Russell and Habermann [6]) as one of the features most likely to introduce error and confusion; we make do instead with conditional commands and expressions, subroutines, switches, and the wide range of loop commands, all of which are semantically much more regular constructions. We do not feel that this exclusion of '<u>goto</u>' has resulted in any unnaturalness, nor that its inclusion could improve the clarity of any particular case.

We have not hesitated, when it appeared convenient, to use recursion. Indeed, as we remark in II:4.3, our recursive routine OutP, for the output of positive integers, occupies fewer words of core, and takes fewer cycles of machine time (on our machine), than any other routine of equivalent specification.

It should be unnecessary to remark that attention to layout can greatly improve the readability of a program. But a

generation of starting in column 8 has left its effect, so we draw attention to our use of indentation, blank lines, and the grouping of material generally, to bring out the structure of the routines.

A related question is the amount of comment and ancillary documentation that is desirable. On the whole, a program in a high-level language should be its own documentation. Even so, some additional remarks (!) are sometimes helpful, just as a mathematical proof often gains in clarity when the formal steps are tempered with informal connecting explanation. In mathematics, the formal and the informal statements are often interspersed, and it is indeed possible to treat a program in the same way (for an example of this approach, see [2] §2.5.2). But the structure of a program is usually more complex than that of a proof (which is usually practically linear, with the occasional splitting into cases, but with heavy use of 'subroutine calls' to previously proved results), and the program's structure may be obscured by a continuing flow of comment. We have therefore included rather few comments with the program, and have separated out most of the description into the commentary. As a rough guide (though we are not at all consistent), the comments in the text are about things which might otherwise escape the notice or the memory of someone who knows the system fairly well, while the commentary is meant to guide the reader coming to the system for the first time. Some parts of the system need more description than others, in particular those which control the peripherals and humour the operator. In general, however, we have tried to keep the amount of extra description to the minimum.

Another matter to which we have given considerable attention is the choice of names. We have tried to reject outright the convention that all names should contain some characters to indicate in what section of the operating system they are defined. This convention may be very desirable in assembly code, but it is unnecessary in a more sophisticated language. Names with purely local significance should be prevented from misuse elsewhere by the scope rules of the language; names with wider application should be chosen so that they clearly and concisely evoke the object's significance (as, for example, do GiveUp, MakeNewFile, or OutN). On the whole we try to keep names, particularly local names, as short as possible - as indeed they are in mathematics. It is easier to appreciate the structure of a construction if the recognition and correlation of its components is a straightforward and simple process.

It is our convention that names in block capitals are reserved for manifest constants.

One class of objects for which we have been unable to avoid names with extra characters to indicate their field of use is elements of data structures. For example, many of the structures used in the system contain pointers to their predecessors (e.g. FS blocks, Run-blocks, elements of ClearUpChain, etc.). The position in the vector occupied by this pointer is not the same for all the structures. The structures are used throughout the system, and the names of their elements must therefore be global in scope. So we are forced to use different names for the various kinds of predecessor (FPRE, RPRE, CPRE etc.). In a language with types we could avoid this difficulty by careful design of the data structure facilities (see, for example, PASCAL [7]), but we cannot do any better with BCPL.

## Guide to these books

Part II of this book contains the text of the operating system itself. It is divided into nine chapters, each of which covers some broad aspect of the system's activity. Each chapter is subdivided into sections (which are stored as separate files on the disc), and at the finest level of subdivision come the individual definitions themselves.

Part III contains the text of the programs used to initialise the system when it is read into the core. Like the previous part, it is subdivided into chapters and sections.

Part IV contains the declarations of global variables and manifest constants, which provide the environment for the programs in Parts II and III. Part V shows how the sections of text are grouped together for compilations: in fact, each chapter of Parts II and III, with the declarations, forms a separate segment. Part VI is a list of the failure reports which can be generated by the system.

The remaining two parts are concerned with the system library. Part VII lists and briefly describes the entries in the system index, and in Part VIII a few selected library files are printed in full and described.

We recommend that the reader should first orient himself by studying papers [1] and [2]. Then he will be ready to delve into Part II, aided perhaps by the Appendix on BCPL. Parts IV, V and VI are mainly for reference.

Cross references in the commentary have been made not by page number but by section number, as these are the same in both text and commentary. Phrases like 'see above' refer within the same section.

## II: THE TEXT OF THE SYSTEM

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

#### II:1.1 LGLOOP

LoadGoLoop (global 370)

This is the heart of the operating system, and is what the system actually does. It is described in [1] §1.2.3.

LGLoop (global 64)

This routine, which is Run by LoadGoLoop above, is given in a simplified form in [1] §1.2.3.

Lines 12, 20, 21: Since each activation of LGLoop is in principle independent, it is appropriate to reset the standard input stream before starting, and the standard output streams at the end.
13: To avoid catastrophe in case the loaded program neglects to set the global variable Prog, it is preset to a default value.

Prog (global 1)

This global variable is reserved to hold the steering program of a user's job (and also sometimes of a system file: see LogIn, VIII:1). A job is executed by calling this routine. It is sometimes also called Start.

DefaultProg (global 398)

This failure routine is the default value of Prog. It is global, so that it may be used in other load-go sequences.

## II:1.2 RUN

**Run** (global 399)

See [1] §1.2. Run applies its parameter as a parameterless routine, after taking precautions to enable the *status quo* to be restored.

Line 12: The possible repetition of TerminateRun is in case the system is interrupted while this routine is in progress (perhaps because one of the routines in the ClearUpChain is faulty).

**PrepareforRun**

See Run.

Line 35: A correct copy of the current state of disc space allocation (which is normally kept in core) is written back to the disc.
37-40: Since the old ClearUpChain is chained in both directions (Fig. 1), it must be altered, unless it is null, in order to move its handle from the *global* ClearUpChain to the new Run-block.
42: The new Run-block is made the current one.

**TerminateRun** (global 392)

See Run.

Line 70: If the unloading operation (line 60) did not restore IBlock and CPtr to exactly their previous values, then the error is reported.
77: When the system is interrupted in order that another user may log in, the routine Interrupt (II:2.4) sets the *global* variable User to NULL, so that TerminateRun will subsequently Run the logging-in program. This seems a rather stupid complication: it is a relic from the time when we were very short of core, and it was usually necessary to unload some code (i.e. end a Run) in order for the LogIn program to fit in.

ClearUp
ClearUpChain (global 332)

See [1] §1.2.2. The structure of the ClearUpChain is shown in Fig 1. The ROUTINE word in each entry is a routine which is intended to be applied to the entry itself. Each entry is removed from the chain before its routine is applied, so that if the routine fails the entry will not be dealt with again.

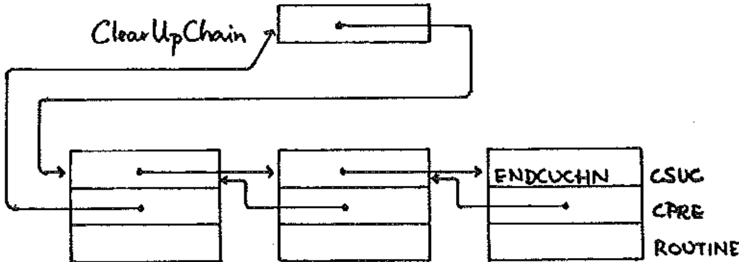



Fig. 1 Structure of ClearUpChain

LogIn

See TerminateRun. The system library program 'LogIn' is described below (VIII:1).

Finish (global 395)

This routine, described in [1] §1.2.1, forces exit from a program being Run. Fig.2 shows the structure of the BCPL stack. Using information in the Run block, Finish alters the procedure pointer to point to the activation level of Program[] (called in Run, line 11). Then, when return is obeyed (written explicitly, for stylistic reasons, line 102) the effect is as if the activation of Program[] had terminated normally.

Fig. 2 Structure of BCPL stack

## II:1.3 LOAD

### Note on binary format

A binary word stream, whether from tape, file or anywhere else, comes in blocks, each containing at least two words. The first word is a warning character, and specifies the type of information in the block. The second word is a count of the number of words in the block over and above the first two. The remaining words are information words.

The warning characters recognised by the loader routines are

TITLE NEWSECTION CODE
INTERLUDE DATA BINARY
ENDLOAD.

Of these, NEWSECTION and ENDLOAD introduce marker blocks, which are precisely two words long, so that the value of the second word is always 0. The significance of information in a BINARY block depends on context.

Other warning characters occur in binary information for other purposes (e.g. archiving files).

### Format of compiled BCPL segment

A compiled BCPL segment consists of blocks in the following order:

| | |
|---|---|
| TITLE | the information words contain a packed BCPL string; |
| NEWSECTION | |
| CODE | these may vary in length and number, depending on the compiler; |

| | |
|---|---|
| DATA | one of these at most; |
| INTERLUDE | contains relocating code, described below; |
| BINARY | contains label-setting directives, described below |
| (ENDLOAD | after the final section). |

The INTERLUDE block is, of course, generated by the compiler, and its effect is the same as the following:

```
§ SetLabels[]
  CFirst[]
  return §
```

SetLabels is described in II:1.4; CFirst is the first word of the program's code area, which usually contains a jump to SetGlobals, also described in II:1.4.

The format of the BINARY block is described in II:1.4, under SetLabels.

Load (global 397)

This routine is for loading binary programs. In is assumed to be a binary word input stream, in the format described above, and the routine loads program until it has encountered the number of ENDLOAD blocks specified as the parameter.

SectId

The routine LoadSection expects as parameter a word to identify the section loaded, while it is in the machine. Load and Unload together maintain the variable SectId, which Load uses (and increments) whenever it requires a parameter for LoadSection.

IBlock (global 23)
Format of Loaded Section

With each section of loaded program is associated an information block. This specifies the areas in the code and data segments of store occupied by the program. The first word of each area points back to the information block. The information blocks are chained, and the end of the chain is kept in the global IBlock. Since the information blocks are used for diagnostic purposes, it is convenient to chain them in both directions.

Although there may be several CODE blocks in a binary program, they are concatenated into one area when loaded. This is not easily possible for DATA blocks, as the free store is not managed as a stack; so only one DATA block is allowed per section. The format of a loaded section is therefore as shown in Fig.3.

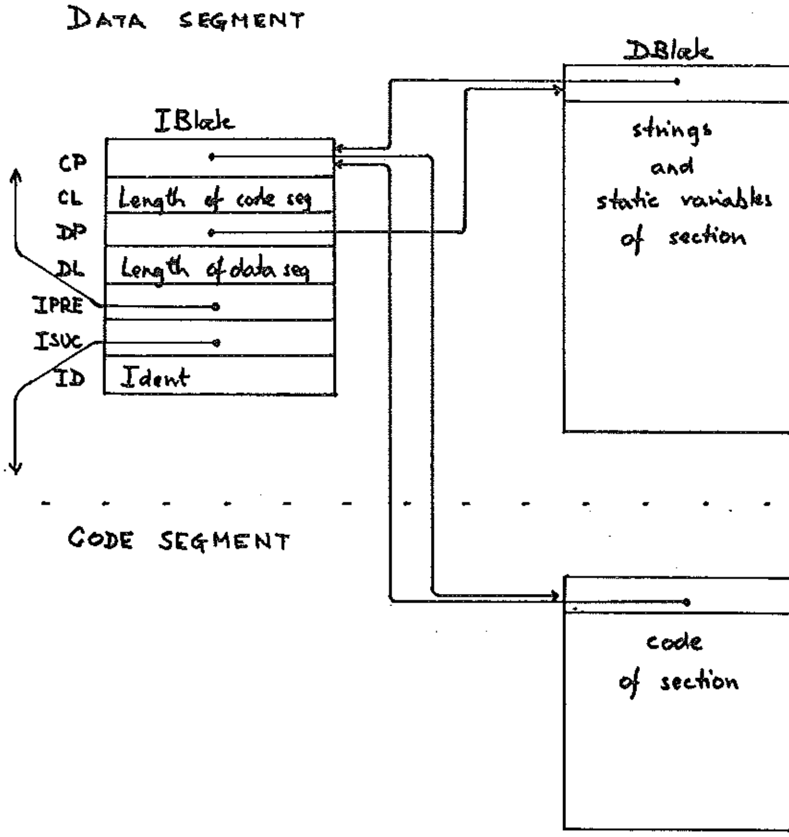



Fig. 3 Structure of a loaded section

CPtr (global 9)

This variable holds a pointer to the first free word in the code segment which, as stated in [1] §1.1, is managed as a last-in-first-out stack. CPtr is global so that programs may note its current state in order to give it later as a parameter for Unload (see below).

CFirst (global 8)

This points to the first word of code of the latest segment to be loaded. It is set by LoadSection and used by interludes.

LoadSection

This routine is called by Load when a NEWSECTION block is detected.

Line 77: The Unload routine tests this flag to ensure that it does not attempt to de-nest globals (see [1] §1.1) which may not have been set, as this could cause system corruption.
90: The loop to read code into the code area is given extra hardware assistance.
94-97: After the IBlock information is brought up to date, interludes are entered immediately. Since interludes lead to the setting up of the globals, it is afterwards felt safe to make GlobalsUnset false. However, this is perhaps not the most logical place to do so.
111: Hardware assistance is also used to load the data block.
115: The occurrence of any unrecognised block indicates the end of the section.

Unload (global 396)

This routine unloads sections of program in the reverse order to that in which they were loaded, until the code pointer has been moved back at least to the value specified by the parameter c. Only the code loaded during the current Run may be unloaded.

Line 140-150: Each cycle round the loop §w unloads one section of code.
142: SetGlobals (II:1.4) is not called to de-nest the globals if the code was generated before 'nested globals' were introduced.

## II:1.4 SETLAB

SetLabels (global 11)

This parameterless routine is called by the INTERLUDE block of a compiled program, and is used to relocate addresses in both the code and the data areas. It obtains information from the BINARY block, which follows the INTERLUDE block in the binary stream.

The BINARY block, of length n (excluding the first two words), contains n/2 entries, of the format shown in Fig.4.

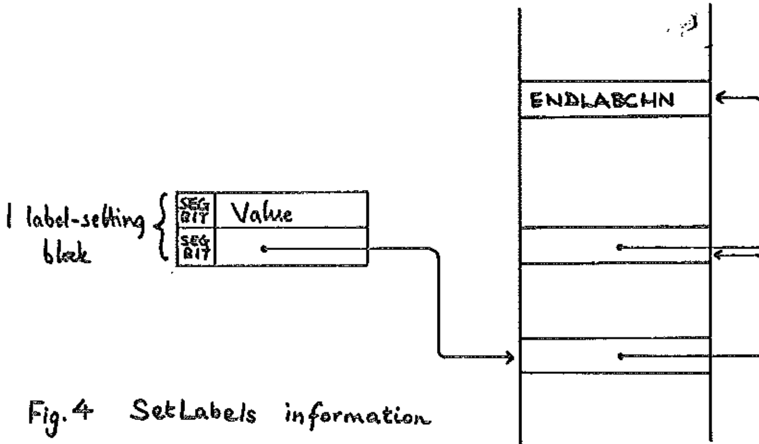



Fig.4 SetLabels information

Lines 24, 25: A specifies the required contents of some word containing an address; R specifies the address of this word.

26, 27: ASeg and RSeg, the most significant bits of the two words in the entry, specify whether the addresses (contained in the remaining bits) refer to the code area or the data area; in both cases the address is relative to the start of the area occupied by the section of program being relocated.

28: The actual value to be substituted (Val) is computed from A by adding the appropriate starting address, after masking out the segment bit.

29: Ref points to a chain of words (in the segment specified by RSeg), each of which is required to contain Val.

30-48: The routine scans the chain, updating each element.

40: In the case of a chain in the data area, it is necessary to check that such an area (which is optional) is in fact present.

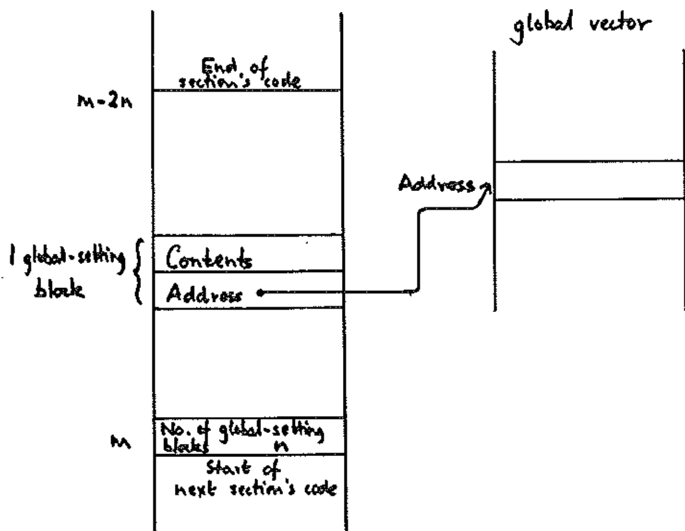



Fig. 5 SetGlobals information

SetGlobals (global 21)

This parameterless routine is used to nest the globals declared in a section of program, as described in [1] §1.1. A jump to this routine is compiled into the first words of the code area of a program, so that it may be entered by the routine call

CFirst[]

contained in the Interlude (described above, II:1.3), which is activated during the loading of a program. SetGlobals is also called by Unload (II:1.3) in order to de-nest the globals.

The information required by this routine is stored at the end of the code area of the loaded program, and is in the form shown in Fig.5. Each entry consists of a pair of words, containing the address of a global variable and its new contents. The final word contains the number of entries.

Line 58: SetGlobals obtains the address (p) of this final word from the information block.

62-67: It then scans the entries (backwards), exchanging the present contents of the global with its new value. Thus when Unload calls it for the second time it will restore the previous contents.

Note that SetGlobals must be called after SetLabels, as the contents of words in the entries are likely to require relocation.

## II:2. Post-mortem arrangements and interrupts

### II:2.1 GIVEUP

GiveUp (global 29)

GiveUp is a variable routine with one argument (see [1] §1.2.1). It is intended for use in programs at points where an error has been detected and there is no obvious way of continuing; a call of GiveUp may also be forced by the operator. GiveUp is initially set to StandardGiveUp (see below). It may be changed by the program when a private GiveUp routine would provide more appropriate post-mortem information or better facilities for continuation.

A program may use the argument of GiveUp as a means of identification of which error has occurred. More usually, however, a message indicating the nature of the fault is output before GiveUp is called; in this case, the parameter of GiveUp may be used to provide more diagnostic information. GiveUp normally ends with a call of EndGiveUp (see below).

GiveUpStackSize (global 353)

A programmer who replaces GiveUp with a private routine should set in this global variable an estimate of the amount of stack required. This enables the system to set up some private stack in the free store for a call of ForcedGiveUp to use, in order that it will work whatever the state of the machine. GiveUpStackSize, like GiveUp itself, is preserved in the Run-blocks and restored on exit from a Run.

GiveUpStack (global 363)

This is a global vector from the free store area, used as private stack when ForcedGiveUp calls GiveUp. It is set up when the system is loaded; but if a program indicates, by altering GiveUpStackSize, that it has changed GiveUp to a version needing more stack, then ForcedGiveUp will claim a larger vector for GiveUpStack.

The format of GiveUpStack, and PrivateStack (II:2.4), is shown in Fig.6. Word 0 is a P-pointer back to the previous stack and word 1 contains the length of the vector.

Note that GiveUpStack is used only by ForcedGiveUps: calls of GiveUp from within a program use the normal stack.

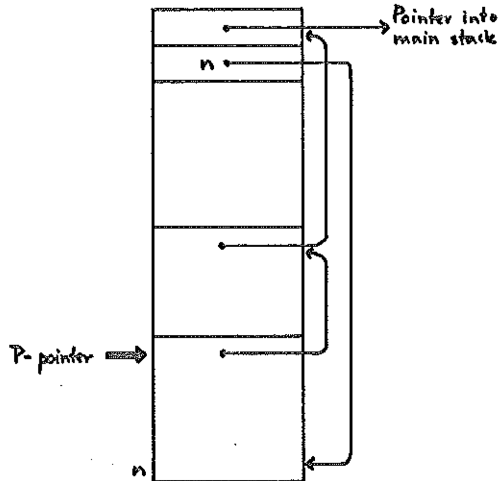



Fig. 6 Structure of a private stack in use

ForcedGiveUp (global 32)

This routine is called when the operator indicates that he wishes to force a call of GiveUp (see Interrupt, II:2.4). It sets up GiveUpStack and calls the routine GiveUp.

Line 14-16: If PrivateStack points back into GiveUpStack, a previous ForcedGiveUp has been interrupted (this is quite a common occurrence). These lines ensure that whether this has happened or not, both PrivateStack and GiveUpStack point back to the normal stack.

18-22: It is possible, though unusual, that GiveUp may itself have Run further programs.

30: GiveUpStack is already certainly big enough to accommodate StandardGiveUp.

32, 34: The extra words leave room for the housekeeping information in GiveUpStack.

InStack

This routine is used internally by ForcedGiveUp and Interrupt.

StandardGiveUp (global 394)

StandardGiveUp is the initial value of the variable routine GiveUp. In this version, the user is given a choice either of dumping a core image onto the disc or of outputting some standard post-mortem information.

Dump (global 66)

This routine dumps an image of both the code and the data segments onto a reserved area of the disc, so that they can be subsequently examined by a suite of post-mortem programs. This suite must also deal with dumps produced by a second method which is sometimes used when the system has collapsed completely. This second method, by means of a tiny program in Modular One machine code read in like the initial bootstrap, dumps an image of the entire Modular One core onto the disc by a single transfer. In this second case, the post-mortem suite has to use information in the dump of the Computer Technology Executive program to determine where in the core image the code and data segments are, so the Dump routine also dumps this information, in the same format. Because of hardware restrictions, this routine cannot dump the whole core, but must dump the code segments and the data segments one at a time in the appropriate places.

Lines 63-66: The routine interrogates the data structures in the Executive program to determine the absolute address in core of the code and data segments.
68, 69: These variables are the upper and lower bounds of the two segments, expressed as page addresses in core.
74: A simplified form of Executive's data structures is constructed and dumped, so that the post-mortem programs will be able to work out the disc addresses of the two segments.

Note that this routine, which is very implementation-dependent, does not use the disc filing system, but a specially reserved area of the disc. (There was not enough disc space for our original intention, which was to allow users to keep core dumps in the filing system.)

FetchExecWord

This function obtains the contents of any word in core, given its absolute Modular One address (see Exec, II:9).

SetUpDummyExecStructure

This routine constructs in DiscPage enough of a data structure for the post-mortem programs which analyse a core image dumped on the disc to determine the addresses of the code and data segments.

Lines 97-100: Although the dump is outside the filing system, it is convenient to use the filing system primitive CoretoDisc to write the information onto the disc; one must, however, ensure that the transfer is effective even if writing to the filing system is inhibited.

DumpSegment

This routine uses the Exec TRANSFER command (II:9) to dump a complete segment onto the disc.

Line 103: A safety check is made that the transfer is to be in the core image area, which is at the high-address end of the disc.

EndGiveUp (global 373)

A call of this routine may conveniently be made at the end of a GiveUp routine. It outputs a message on Console (the diagnostic information from GiveUp will, naturally, have been output to ReportStream), and waits to allow the operator to take any further action, such as manual post-mortem.

ForcedFinish

This routine is used when the operator has forced a Finish by interrupting the system, or at the end of a GiveUp. It calls the ordinary routine Finish ([1] §1.2.1, and II:1.2), but it has also proved convenient in practice to reset the constant streams.

II:2.2 PM

StandardPM (global 393)

This routine may be used to output some standard diagnostic information. It is one of the alternative actions of the standard GiveUp (II:2.1).

ReportCallTrace (global 375)

This routine outputs the stack housekeeping information, which helps in sorting out the current state of the hierarchy of routine activations. The stack pointers and links are output as far back as the last call of Run, or until Max have been output.

Line 18: The first n links are ignored, to avoid the output of information about the diagnostic procedures themselves.
20: If ReportCallTrace is called while the system is in a private part of stack (e.g. during a ForcedGiveUp), then it is assumed that information about the main stack is required; the routine will not output any links in the private stack, unless indeed the whole of the current Run has used it.

ReportFre eStoreState (global 374)

This routine outputs post-mortem information about the state of the free store.

ReportBlocks

This routine is used by ReportFreeStoreState to output information about the free block chain and the pending block chain. It outputs the number of blocks, the total number of words in the whole chain, and the size of the largest block.

Line 46: If the chain is NULL, nothing is output.

ReportWords

ReportFreeStoreState uses this routine to output information about the free word chain and the pending word chain. Unless the chain is null, its length is output.

## II:2.3 MPM

ManualPM

An entry to this routine is one of the possible courses of action when the system has been interrupted (see Interrupt, II:2.4). In response to commands input on the console stream (Con, set up by Interrupt), this routine allows any word in the code or data segment to be examined or changed.

The simplest command is merely a decimal number, which causes the output of the addresses and contents of successive words in the data segment, beginning at the specified address. This continues until something is typed on the console. Each word is output on a new line, but a single blank line is output in place of any block consisting of one or more adjacent words whose contents are zero.

The command may contain one or more letters before the number. The effect of these letters is as follows:

- B ('Both' formats) The content of each word is output in address format (that is, as two 8-bit bytes in octal) as well as in decimal.
- C Words are output starting at the specified address in the Code segment. In this case output is automatically in both formats.
- E ('Exchange') This mode is used to alter the contents of words in either segment. After the output of each word, the system pauses for the operator to type one of the following:
  - (i) A newline: the word is left unchanged and the routine continues with the next word.
  - (ii) A decimal number followed by newline: this specifies the new contents of the word, and the system continues with the next word.

(iii) B, followed by one or two octal numbers, followed by a newline (B stands, perhaps, for 'both', or 'binary'): this also specifies the new contents. If there is one number it is simply interpreted as an octal number; if there are two they are interpreted as two 8-bit bytes.

(iv) N: the exchanging loop is broken and the routine waits for another command.

There are two other single-letter commands:

N ('No more') The routine calls ForcedFinish.

R This provides facilities for Returning to a specified point in the program. After checking that the 'R' was not typed by mistake, the system asks what values it should set for
the P-pointer,
the link (that is, the instruction address),
the result (that is, the contents of the result register, as this facility is often used to simulate the return from a function call).
The system checks that the value proposed for the P-pointer in fact points to the stack for some routine activation in the current Run.

Lines 26, 27: For historical reasons concerning the Modular One hardware, the code segment was originally called the Y-segment.

44 (and also lines 81, 85, 102-104): since this is a post-mortem routine it must work even if the free store has been exhausted. The normal functions NextN and NextO are therefore unavailable, as they use PutBack, which needs free store. Special versions are therefore provided.

Exchange

This routine deals with the response to each word when ManualPM is operating in exchange mode, except for case 'N', which is dealt with by ManualPM itself.

Return

This routine asks the necessary questions for the return mode of ManualPM.

JumpTo

When, in the return mode of ManualPM, Return has obtained satisfactory new values for the P-pointer, the return link and the result, this function actually makes the jump.

Lines 119, 120: It alters its own return link on the stack, so that when it returns (line 121) it ends up at the specified place.

AnythingTyped

This boolean function examines the state of its parameter stream, to determine whether anything has been typed on the device.

Line 128: The newline key on the Olivetti remote terminals generates three characters (CR, LF, CR), and the final carriage return can be a nuisance unless, as here, it is ignored.

MPMNextN

This routine is the ManualPM version of NextN, which must not use PutBack (see comment on line 44, above). Since the number may have been reached before the routine is called, the first character to be scanned, though not necessarily significant, is handed over as the extra parameter, FirstCh.

Line 138: Minus signs are significant only if nothing, except possibly spaces, separates them from the number.
148: Since PutBack is impossible the terminating character is lost.

MPMNextO

This is the ManualPM version of NextO, which is necessary for the same reasons as MPMNextN. Unlike the standard function NextO, it also accepts two octal numbers on the same line as specifying one word in two 8-bit bytes.

## II:2.4 INTERRUPT

When the operating system is interrupted for any reason, the hardware forces a jump to word 2 of the code section, words 0 and 1 being reserved to trap erroneous jumps to 0 (see [1] §1.3). Words 2 and 3 of the code segment, initialised during system setup, contain a jump to the routine Interrupt. The hardware also places a value, specifying the reason for the interruption, in REASONFORINTERRUPT, a reserved word in the code segment.

PPtr (global -3)

As it happens, in our implementation this register, which points to the base of the portion of stack belonging to the current routine activation, is accessible as a global. In real hardware this would probably be a processor register, in which case more special machine-code instructions (see II:9) would be required to manipulate it.

PrivateStack (global 356)

Since it is necessary that Interrupt should work whatever the state of the machine, and in particular if the stack has become corrupted or exhausted, Interrupt uses this vector as its stack. PrivateStack is set up during system setup. Its format, shown in Fig.6 (II:2.1), is the same as that of GiveUpStack. PrivateStack is also used by ManualPM.

Interrupt (global 351)

This routine is automatically invoked whenever the operating system is interrupted; it is normally never called explicitly by programs.

The action taken depends on the reason for the interruption. If the machine has just been switched on (case POWERON) a call of Finish is forced, so that the system automatically leaves the interrupted Run and continues with the outer program which called it. In all other cases (storage bound violations, or an Interrupt forced by the operator's pressing the X-ON key on a

console) the routine outputs an appropriate message on the appropriate stream and waits for the operator to tell it what action to take by typing one of four letters:

- G (Go) The routine forces a call of Finish in order to abandon the interrupted Run, and otherwise continues.
- F (ForcedGiveUp) A call is forced of the programmer's diagnostic routine GiveUp (see II:2.1).
- M (ManualPM) The routine ManualPM is entered (II:2.3).
- L (LogIn) This indicates that a new user wishes to begin to use the system. Apart from giving him an opportunity to log in, the action is the same as G.

Lines 15-19: The routine changes the P-pointer to use PrivateStack; and word 0 of PrivateStack (which should contain a pointer back to the previous stack) is correctly set, unless the system was in PrivateStack already. In such a case, the system would have been interrupted while dealing with a previous interruption, and word 0 of PrivateStack would already contain a pointer back to the normal program stack. This method, using the static variable TempP, ensures that the process works even if a further interruption occurs while it is taking place.

23: Con is the stream used for all the dialogue of Interrupt and ManualPM.

30: case NOREASON: this implies that the hardware has detected that the system has stopped, and has restarted it. The most probable cause is violation of storage bounds.

35: This allows the system to be rescued from ExecConsole even if, for example, Console is some remote console stream which has become corrupted.

52: This prompts TerminateRun to call the LogIn program (see TerminateRun, II:2.1).

64, 65, 68: This is necessary because the clock routines may ask for the initial time on Console.

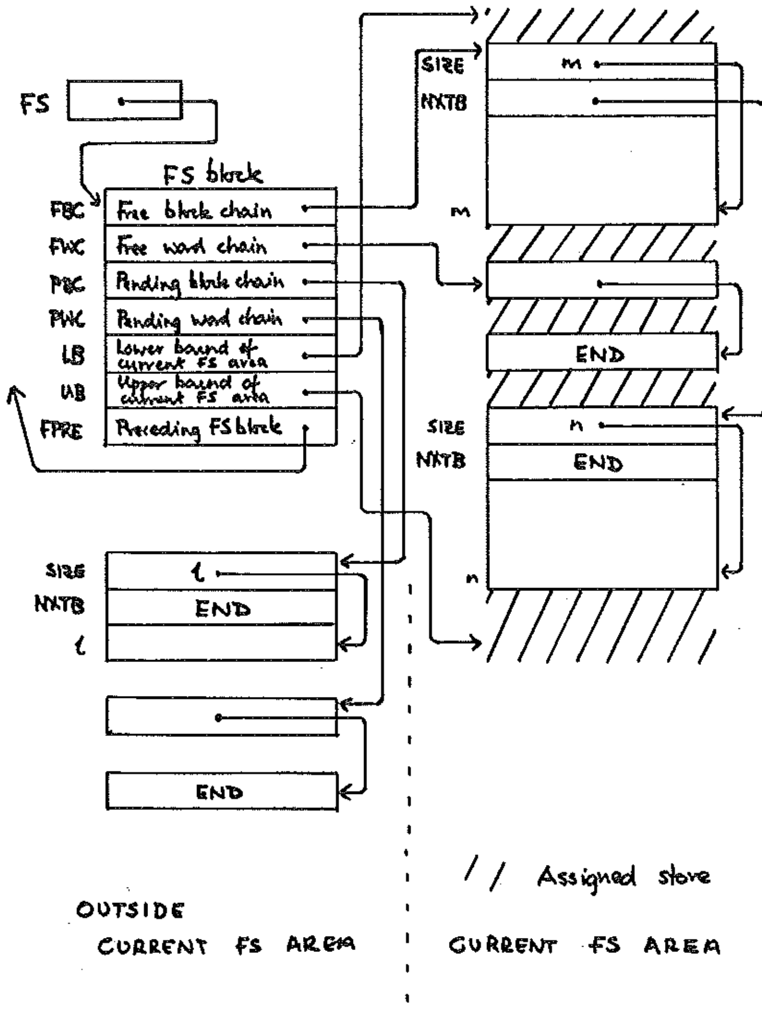



Fig. 7 Structure of Free Store

# II:3. Storage allocation and PutBack

## II:3.1 FREESTORE

This section of the operating system deals with the provision of off-stack core storage (see [1] §2.2.1).

The free store area employed for a Run is the largest free vector in the storage which belongs to the program invoking the Run. (The first Run is allocated a fixed amount of free store when the system is set up.) At the end of the Run all its free store area is forcibly reclaimed.

The free vectors in a free store area are chained together in order of location. The free single words are kept in a separate chain, also in order of location.

FS (global 24)

The state of a free store area is kept in a 7-word vector. The global FS points to the current one. The format of the whole structure is shown in Fig.7.

NewVec (global 60)

This function is used to claim a block of specified size from the current free store area. The parameter n gives the size of the block (i.e. the block v is to have elements v↓0 to v↓n, and will therefore be of length n+1).

Line 8: A validity check on the parameter.
12-16: If a single word is required and the free word chain is not empty, one of its elements is used.
18-28: The free block chain is scanned until a big enough block is found. B is the block being considered, BP is the lv of the pointer to it, usually from the previous block. See Fig.8.
30: Unless the block is exactly the right size, NewVec needs to return the surplus. This could be done by calling ReturnVec, but (unless the surplus is a single word) all the required information is known already. A lot of unnecessary work can

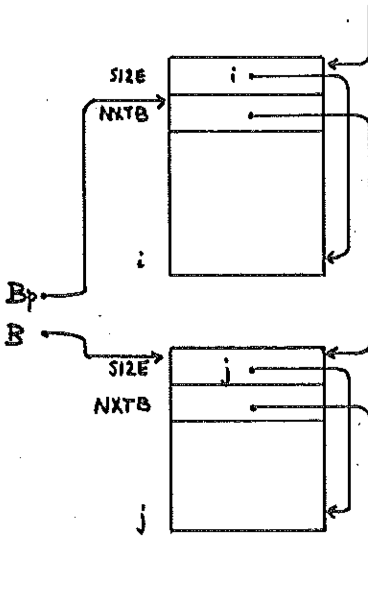



Fig. 8 NewVec

therefore be avoided by merely resetting the pointers instead.

32: If the block is the right size or if there is only one surplus word the block is removed from the block chain.

33: A single surplus word is returned.

36-42: The chain is adjusted to contain the surplus block (Note that B↓NXTB and SB↓SIZE are the same location if n = 0, hence the comment).

ReturnVec (global 61)

ReturnVec is a routine to return a specified block of store to the free store. It takes two parameters, V (a vector) and n (its size), and it returns the area V↓0 to V↓n inclusive.

The block is merged with any block (or word) already free with which it is contiguous. If the block is outside the current free store area it is placed on a 'pending chain', to be returned when the system reverts to the earlier (larger) area from which the block was obtained.

Lines 52-71: These are obeyed if the block is outside the current free store area, or if the parameter n is invalid.

53: A validity check for n.

54: A check that the vector does not straddle a boundary of the current area.

57-61: A check that the vector is inside the primaeval free store area.

63-70: The vector is a valid pending word or block; it is added to the appropriate chain.

73: The normal case begins here; the vector is inside the current free store area.

73-78: The free word chain is searched to see if any free word occurs immediately before or after V. Two pointers are used,

PW (the 'previous' word) and W (rv PW, the word being considered). The search continues until W reaches or passes the word before V, or reaches the end of the chain.

81-85: Here there is a free word immediately before V (Fig.9(a)). The word is removed from the free word chain and added onto V (Fig.9(b)).

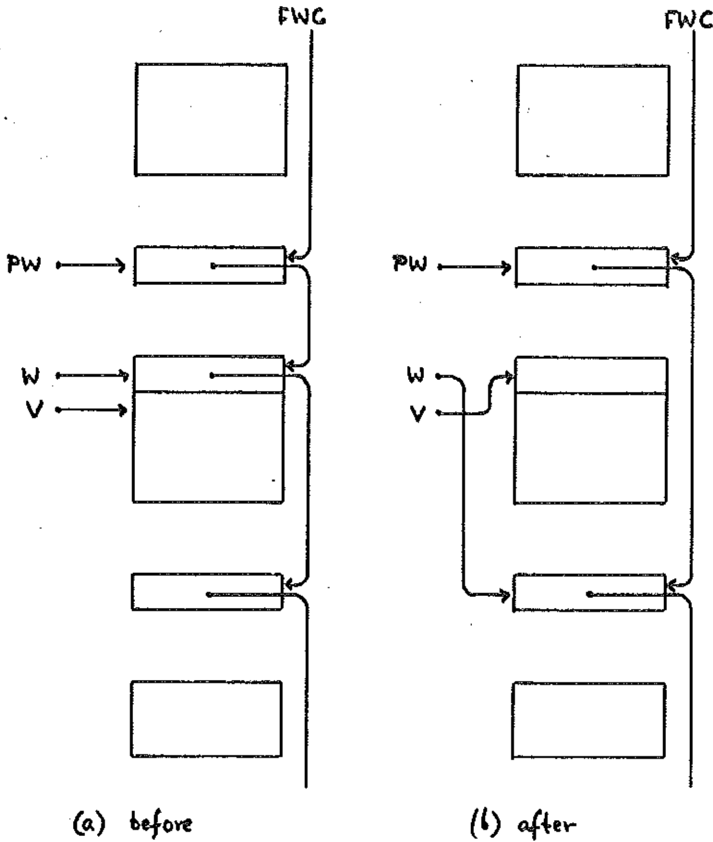



Fig. 9 ReturnVec

86: A safety check that the vector V does not contain any word on the free word chain.
88-91: If a free word immediately follows V, it is removed from the free word chain and added onto V.
94: The routine now attempts to concatenate the (possibly enlarged) vector V with blocks on the free block chain. The method is similar to that for the free words, except that it is the free block which is augmented rather than the vector (this avoids removing blocks from the free block chain, only to return them to the same place later).
94-100: The search uses the pointers B and BP described in NewVec (see Fig.8).
102, 103: If the block B immediately precedes V, it is enlarged to include V.
104-107: If the enlarged block B is now contiguous with the next block on the free block chain they are amalgamated.
110-115: If B immediately follows V then it is enlarged to include V (if n = 0 then V↓NXTB and B↓SIZE are the same location).
118: A validity check that V does not overlap a free block (we already know that the upper bound of B is greater than or equal to the lower bound of V).
121-127: V is not contiguous with anything, so it is inserted as a new link in the free word chain or the free block chain as appropriate. Note that we already have pointers to the correct positions.

RVGiveUp

This is a private error routine used by ReturnVec.

NewWord (global 62)

This is a function to obtain a single word from the free store area.

ReturnWord (global 63)

This is a routine to return a single word to the free store.

<u>MaxVecSize</u> (<u>global</u> 37)

This is a parameterless function to find the size of the biggest unallocated block in the current free store. If both the free block chain and the free word chain are empty, the result is the constant NOSTORE. Note that in order to interpret the result it is necessary that NOSTORE < 0, and the function itself uses this assumption.

<u>Line 150:</u> If the free word chain is not empty there is at least a vector of size 0.
<u>151-156:</u> A search down the free block chain, comparing the size of each block with the maximum already found.

<u>NewFreeStore</u> (<u>global</u> 390)

This parameterless routine is normally called only by PrepareforRun (II:1.2). It constructs a new FS block to describe a new free store area formed from the largest available block in the existing free store. The <u>global</u> FS is altered to point to the new FS block.

<u>Lines 168, 169:</u> The entire area is the only free block; the free word chain and the pending block and word chains are all empty.
<u>171:</u> The new FS block is chained to the previous one.

<u>RestoreFreeStore</u> (<u>global</u> 389)

This routine is normally used only by TerminateRun (II:1.2). Its parameter is an FS block, and the routine restores the free store to the state described by this block. It has to deal with the pending chains and also with the 'PutBack' problem (see [2] §2.5.3).

<u>Lines 178-182:</u> A check that the parameter block is a valid FS block (i.e. in the chain of FS blocks).
<u>183-218:</u> For each cycle round this loop (§U), FS takes one step back along the FS block chain.
<u>184:</u> If any elements of PBChain are in the current free store area, we know that they will be the latest elements to have been added to the chain. If such an element exists, this activation of the routine does nothing more, in this cycle of

loop §U, than deal with one element of PBChain; any remaining elements and the actual step back along the FS block chain are dealt with by a recursive call of RestoreFreeStore.

187: S may be either a fast stream or a slow stream.

188: In either case, V is the core vector to which S refers.

190: If the stream is in the current free store area (which we know by line 183 is not the area we are trying to reach) then the stream is about to be abolished anyway; so the PEChain element can safely be ignored.

191: We recurse to complete the action necessary in this cycle of loop §U. Note that since this is a call of RestoreFreeStore itself there will be some extra unnecessary validity checking of the parameter. This could be avoided at the cost of a little extra complexity, but this routine is used comparatively rarely and the time wasted is unimportant. Actually, for this call, but not for the other recursive call (line 195), it would be possible, and a little more efficient, to give FStore as the parameter. But it is more important to preserve symmetry, in order to keep the structure of this routine as simple as possible, as it is already rather complicated.

194-196: If the stream will survive this cycle round the loop then the object in the PutBack block must also be preserved. We therefore retrieve it (line 194), recurse in order to complete the reversion to the previous element of the FS block chain (line 195), and put the object back again, the necessary vector being claimed now from the earlier free store (line 196). Note that the recursive call cannot revert all the way to FStore, as S might be abolished at some intermediate stage.

199: There are now no PutBack blocks to be considered.

200: FS is stepped back to the previous element in the chain.

201: The complete area of the free store just abolished is returned to the previous free store.

203-214: All the elements on the pending chains are returned. If they do not belong to the free store which is now current, they will merely go onto its pending chains.

216: The FS block of the free store just abolished is returned.

## II:3.2 PUTBACK

**PBChain** (global 337)

This holds the chain of PutBack vectors. See below.

**PutBack** (global 391)

PutBack is described in [2] §2.4.5, and an outline of its implementation is given in [2] §2.5.3. The version given here, however, differs slightly from that described in the paper, in order that it may deal with fast streams. An extra element is required in the PutBack vector in addition to those shown in [2] Fig.2, and the structure of a fast stream with one object put back is now as shown in Fig.10. Note that the pointer from the slow block to the PutBack block is in the position STR, normally reserved for the parameter stream. This is satisfactory for pure input streams, but it causes difficulty with bilateral streams, where the output part must work normally even when an object is put back to the input part. The parameter stream for bilateral streams must therefore be placed elsewhere in the vector. See, for example, IntcodetoandfromTeletype (II:5.2).

Line 8: If Str > 0 then Str is not a fast stream.
17-20: If Str is a fast stream (see [2] §2.6) then its input buffer pointer is stored in the extra element of the PutBack vector, and the buffer pointer is then overwritten to point beyond the end of the buffer. This ensures that when Next is next applied the function FNextPB, stored in the slow block, will be called.
22, 23: The vector is added to PBChain.
26-28: The functions which replace the NEXT, CLOSE and RESET elements of the stream vector differ according to whether the stream is a fast stream.

NextPB

The action of this function is described in [2] §2.5.3.

Lines 44-53: The PutBack vector is removed from PBChain; a failure occurs if it was not there.

FNextPB

The fast-stream version of NextPB.

ClosePB
FClosePB
ResetPB
FResetPB
EndofPB

These routines are described in [2] §2.5.3, though except for the last they must be given in two versions, for slow and fast streams.

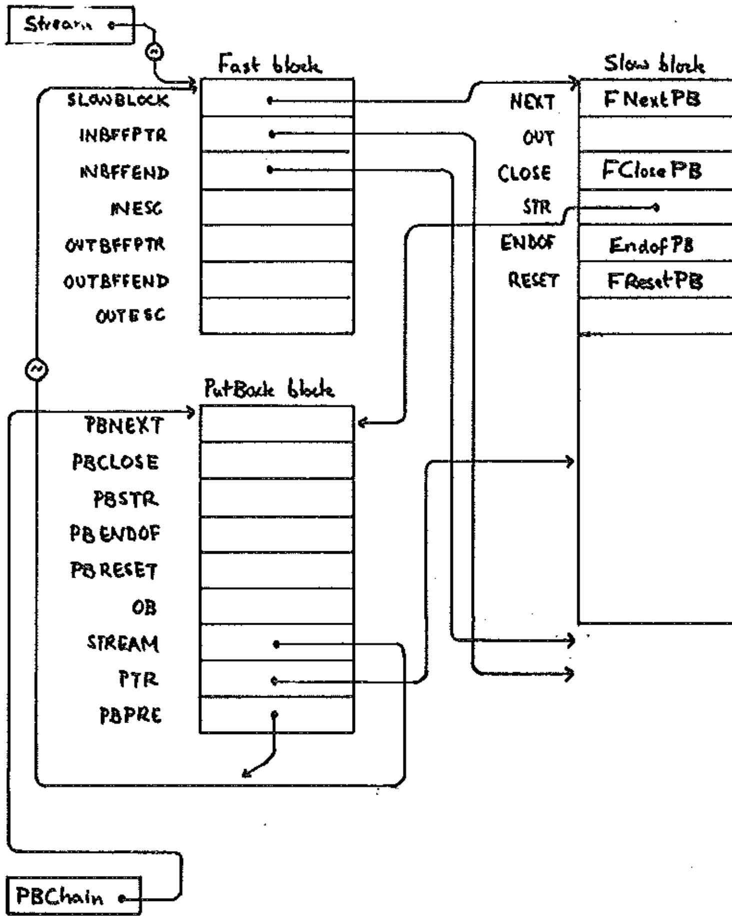



Fig. 10 Structure of a fast input stream with one object put back

## II:4. Stream primitives and I/O routines

### II:4.1 STRPRIMITIVES

This section contains the definitions of some of the primitive routines and functions operating on streams. They are introduced in [2] §2.1 and §2.4; their implementation is outlined in [2] §2.5. Next, Out and Endof are given hardware assistance (for reasons given in [2] §2.6), and so their definitions appear elsewhere (II:9).

These primitives have to deal with both fast streams and slow streams. If the numerical value of the stream is positive it is a slow stream and its value is the address of the stream vector; if the value is negative the stream is a fast stream and the value is the logical complement (~v) of the fast stream vector (v). The formats of both kinds of stream are shown in Figs. 11 and 12. The functions and routines of a slow stream take the stream vector itself as their first parameter; those of a fast stream take the address of the fast stream vector (that is, the logical complement of the stream itself).

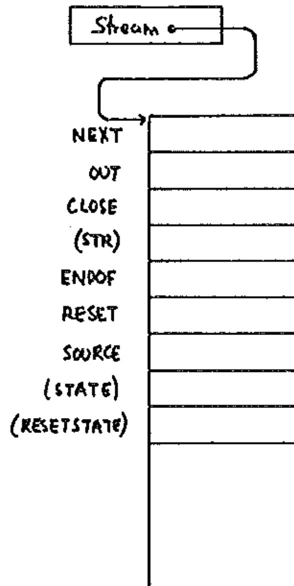



Fig. 11 Structure of a slow stream

**Close** (global 18)
**Reset** (global 68)

See [2] §2.4.3 and §2.4.2.

**Source**

If a stream function is applied to a stream, the actions of Next or Out on the argument stream and on the result stream will usually be different; however, the actions of State and ResetState will probably be the same.

So in the implementation of State and ResetState we refer directly to the 'source stream', which is usually the stream handling the actual device. Every stream vector contains a pointer to its source stream (in the case of a source stream itself it will point to its own vector). The function Source obtains this element for either a slow or a fast stream.

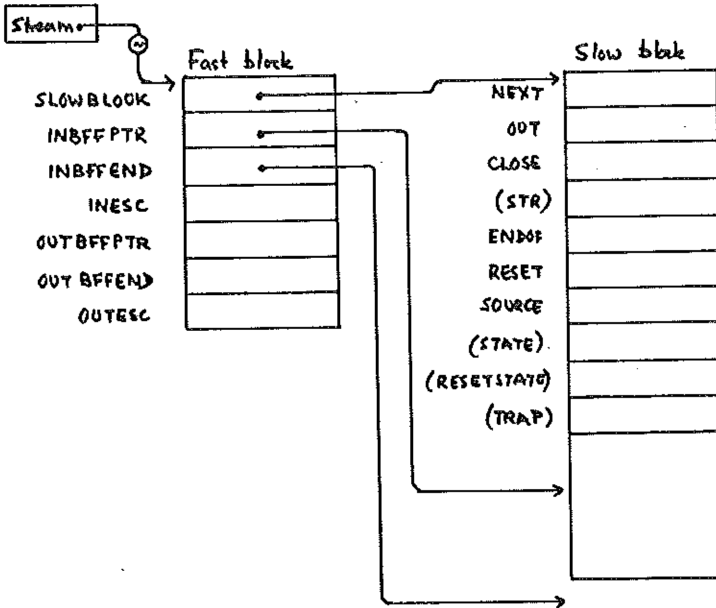



Fig. 12 Structure of a fast input stream

State (global 350)
ResetState (global 349)

See [2] §2.4.4.

Source streams have two extra reserved elements (STATE and RESETSTATE) for the implementation of these primitives.

StreamError (global 69)

This system error routine is available for general use. It is intended for positions in a stream vector corresponding to primitives which would otherwise be undefined (such as Out for an input stream).

## II:4.2 OUTS

OutS (global 381)

In principle, this routine outputs the BCPL string String to the character output stream S. The character '*' is treated specially, however, and it and the character after it are interpreted, whenever it occurs, as specifying some special action. This action will usually involve the next in the list of extra parameters a to z. The repertoire of special actions is as follows:

- *A The next parameter is output in the format of OutAddr.
- *B The next parameter is output as an 8-bit byte in octal.
- *C The next parameter is output as a single character.
- *I The character following the 'I' must be a single digit, n. The next parameter is output as a decimal integer, right-justified in a field of length n characters, or however many more are required to accommodate it.
- *N The next parameter is output as a decimal integer with no spaces.
- *O The next parameter is output as an unsigned octal integer.
- *S The next parameter is output as a string (the action is undefined if this second string contains the '*' character).
- ** A single * is output.

Lines 6, 7: BCPL does not require a routine definition and a call of the same routine to have the same number of parameters; so we allow for an excess of them here, in order that the action of the routine itself might not overwrite any.

8: The extra parameters a to z are handed over as a vector, as the third parameter of OutString.

WriteS (global 44)
ReportS (global 30)

These routines have the same action as OutS, except that they output to streams Output and ReportStream respectively. It has also been found convenient that ReportS should finally output an extra newline. Note that these routines do not call OutS explicitly, as the long parameter list of OutS would use an excessive amount of stack, which would be embarrassing particularly if a private stack were in use.

OutString

This is a private routine used by OutS, WriteS and ReportS, and also by OutN (II:4.3). It outputs the string String to the output stream S, using extra parameters from the vector ParamList when it encounters the escape character '*'. These are treated as specified in the description of OutS above.

Line 25: Ptr is a character pointer into the string.
26: i is used to index the ParamList vector.
27: The length of the string in characters.
34: The normal case.
39: Ch is the character after a '*'.
49-61: The appropriate routine is chosen,
62: and applied.
65-74: This deals with the case of *I.
68: Ch is the character after the I, which ought to be a digit.

GetC

A private routine used by OutString to fetch the nth character of a string.

OutJustify

A private routine used by OutString when dealing with *I. It outputs the integer n to the stream S, right-justified in a field of which the third parameter specifies the length.

Size

A private function used by OutJustify to determine the length in digits of the decimal representation of an integer.

OSReport (global 33)

This routine is used by the system to output messages about errors detected by system routines. The first parameter is a failure number; the second is an explanatory string to be output by ReportS, and it may be followed by up to four extra parameters to be used if the string contains any '*' characters.

OSReportN (global 34)

This routine is used for the less common system failures and merely outputs a failure number. It will be necessary for the user to refer to the list of failures (Part VI) to find out what has happened. Whether this routine or the previous one is used in any particular case can, of course, be altered in the light of experience.

## II:4.3 OPRTS

OutP

This is a private routine used by OutN. It outputs a positive integer n along the character output stream S, as a decimal integer with insignificant zeros suppressed. Note that on our machine this recursive routine is both faster and shorter than any other.

Line 9: This assumes that the decimal digits have consecutive sequential values in internal code. Then, if $0 \leq x \leq 9$, the digit which represents x is given, in internal code, by '0' + x.

OutN (global 379)

This routine outputs the integer n, as a decimal number, to the character stream S.

Line 17: In a 16-bit 2's complement machine the range of a single word is -32768 to +32767. So the sign of -32768 cannot be reversed, and this number has to be dealt with separately.

OutQ

This private routine is used by OutO and OutByte. It outputs the least significant nine bits of x in octal to the stream S. (See the note referring to line 9 above.)

OutO (global 380)

This routine outputs an integer n in octal to the character stream S.

OutByte (global 346)

This routine outputs the eight least significant bits of b in octal (three digits) to the character stream S.

OutAddr (global 347)

This routine outputs the integer a in octal to the character output stream S, as two 8-bit bytes separated by a colon. It is so called because this format corresponds slightly to the address format of Modular One (page:word).

Write (global 343)
WriteN (global 45)
WriteO (global 38)
WriteByte (global 344)
WriteAddr (global 345)

These routines are analogous to Out, OutN, OutO, OutByte and OutAddr respectively, but they all use the global output stream Output. See [2] §2.7 and §2.8.

## II:4.4 NEXTN

NextN (global 358)

NextN is a function to read the next signed decimal integer from a character input stream S. Characters in the stream are ignored until an integer is reached. The character after the end of the integer is left PutBack on the stream.

Lines 11, 12: A minus sign is significant only if nothing (except possibly spaces) separates it from the number.

NextO (global 357)

This function reads the next octal integer from the character input stream S. Characters are ignored until such an integer is reached. The character after the integer is left PutBack on the stream.

NextCh

This private function used by NextN and NextO obtains the next character from the input stream S, but fails (that is, calls GiveUp) if it detects the end of the stream.

## II:5. Permanent input/output streams

### II:5.1 TELETYPE

**Teletype** (global 376)

This is a constant stream, of bytes to and from the executive teletype. It is very rarely used explicitly by programs, however, as they usually access the teletype *via* the internal code stream ExecConsole (II:5.2). Teletype is set up when the system is initialised (III:2.2). This section contains the definitions of the internal functions and routines used by the stream. They are given temporary *global* numbers where necessary, as after the system is initialised they are all in the stream vector, and their *global* place is no longer required. The format of the vector is shown in Fig.13. Note that the input part of this stream could be a fast stream, but since waiting for the operator is an inefficient operation anyway we did not bother to optimise it.

**NextTT** (temporarily global 403)

This is the 'Next' function for the teletype stream.

Lines 7-9: If there is a character in the line buffer it is used.
12-62: Otherwise a complete line is read from the teletype to refill the buffer. During the reading of this line some characters have special effects, as described below.
16: A ping to energise the operator.
19: Wait for a character to come in.
20: Set the device to read the next character.
22: The normal case. The character is echoed and placed in the buffer.
26-29: If the buffer is full (which is unlikely) a newline is output, and the buffer (without a newline) is input. Note that in all this loop (§r) the command *break* terminates the filling of the buffer.
32, 33: Both these characters have the same effect. A newline is input, and the filling of the buffer terminates.
39: This escape character allows the input of a buffer to be terminated without a newline. The character itself is neither input nor echoed.

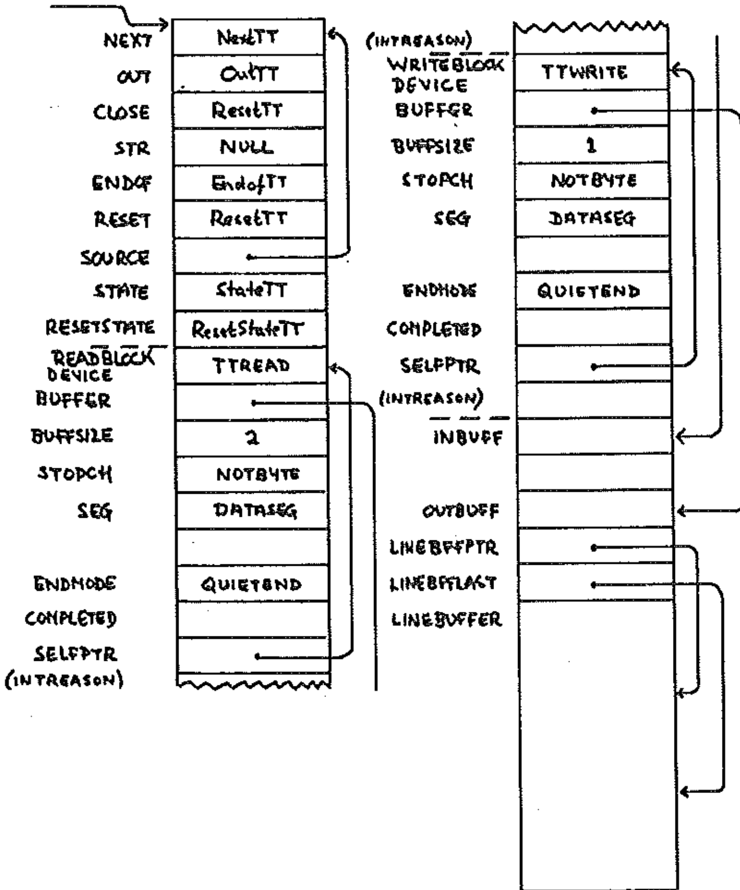



Fig. 13 Teletype

42: This is the trailing erase character.
43: It is ignored if the buffer is empty already.
44: The last character input is removed.
45: The erase character is echoed as a query.
49: This character (CTRL S) cancels the line so far. It is echoed as a sharp followed by a newline, and the buffer pointer is reset to the beginning.
54: This character (CTRL K) is for use when an excessive number of trailing erases have made the line longer on paper than it is in fact. A newline is output but there is no effect on the buffer.
60, 62: The pointers are set to refer to the new bufferful.
63: A recursion to obtain the first character.

OutNewLine

A private routine to output a newline on the teletype. Note that this routine is only used by the input part of the teletype stream, If it were to be used at computer-limited speeds it would be necessary to include an extra carriage-return character, to give the teletype carriage time to complete its movement.

OutTT (temporarily global 404)

The 'Out' routine for the teletype stream.

Line 77: The routine loops at this command until it is accepted.

ResetTT (temporarily global 405)

The 'Reset' routine for the teletype stream. There is nothing to do for the output half, but the input pointers are altered to ensure that the next call of NextTT reads a fresh line from the device.

EndofTT (temporarily global 402)

One can always ask the operator for more.

StateTT (temporarily global 406)

The teletype stream is a source stream, in the sense of the description of State and ResetState (II:4.1). The state of the stream is defined to be the contents of the input buffer.

ResetStateTT (temporarily global 407)

Line 94: The input buffer is initialised.
95: An input transfer is started to the input buffer.
96: If the transfer command is rejected it is assumed that one is in progress already, so there is nothing more to do.

## II:5.2 INTTT

ExecConsole (global 383)

This is a constant stream, of internal code characters to and from the executive teletype. It is the normal means for programs to communicate with the machine room operator. It is created during system set-up (III:2.2), by the application of the function IntcodetoandfromTeletype (see below) to the stream Teletype (II:5.1).

IntcodetoandfromTeletype (global 79)

This is an example of a stream function for code conversion, and it converts a stream of bytes in teletype code to an internal code character stream. It is bilateral: that is, it is defined on both input and output streams. It is used during system setup for the definition of ExecConsole (III:2.2), but it is also available for general use (e.g. with the reader or punch, to deal with paper tape in teletype

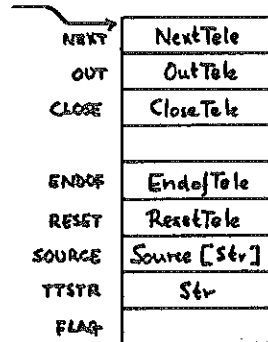



Fig. 14 Intcodetoand-fromTeletype[Str]

code). In the latter case it is recommended that the more appropriate of the alternative names for this function be used - IntcodefromTeletype or IntcodetoTeletype.

The format of the stream vector is shown in Fig.14. Note that the parameter stream is not stored in the usual position, so that the output part might still work when something is put back to the input part. See PutBack (II:3.2).

NextTele

This is the 'Next' function.

Line 26: Parity errors are tolerated only if the stream is the on-line executive console stream ExecConsole.
27: Otherwise the function insists on a character of correct parity. Note that the parity testing function EvenParity can call TryAgain as a side-effect.
28: The parity bit is removed.
33: This is the normal case.
35: Invalid and unprintable characters are ignored.

EvenParity

This private function, used by NextTele, tests whether x contains an even number of bits. It uses the vector Parity (III:2.1).

Line 41: If the test is unsuccessful the function as a side-effect applies TryAgain to the stream function's parameter stream.

OutTele

This is the 'Out' routine for the stream function.

Line 48: Whenever output occurs to the executive console the stream is reset, thus losing any unread input. This is to ensure that when the system asks a question it does not accept earlier input as the answer.
49-51: Normally, as defined below (lines 53-56, etc.), if several newlines are output the final one is preceded by four inches

of runout. These lines avoid this effect occurring on the on-line teletype.

54: The output routine itself is replaced by a special version to deal with newlines. S↓FLAG is set false to indicate that only one newline has so far been output to the stream.

64-78: See lines 30-34 for another way of doing the same sort of thing.

79, 80: The parity bit is inserted if necessary.

OutCRLF

A private routine for outputting newlines.

OutNewLines

This is the special version of the output routine for the stream function, used for the output of newlines. It is placed in the vector when one newline has already been output to the stream but has not yet been passed on to the parameter stream.

Line 93: S↓FLAG is set true to indicate that more than one newline has been output.

94: A newline is output. Note that we still have one newline in hand.

97: We reach here when a character other than newline is detected. If more than one newline has been output this line outputs four inches of runout.

98: The final newline is output (leaving one newline until after the runout ensures that if the paper tape is spliced at this gap the print-out will still be satisfactory).

99: The normal output routine is restored.

100: The character after the final newline is output.

CloseTele

The 'Close' routine of the stream function.

Line 105: If there is a newline in hand it is output.

ResetTele

This is the 'Reset' routine of the stream function.

<u>Lines 112-115:</u> If the special output routine is in use it is set back to normal and the newline in hand is output. In this case there is no runout as it is quite likely that the program will be generating some itself.

<u>EndofTele</u>

This is the 'Endof' function for the stream function. Its result depends on the parameter stream.

<u>Source</u>

A copy of the function in II:4.1.

## <u>II:5.3 XFER</u>

<u>InitiateTransfer</u> (<u>global</u> 10)

This routine is used internally by the paper tape reader stream BytesfromPT, and the other double-buffered library stream functions BytestoPT (for the paper tape punch), and LinePrinter. It has three parameters. The first parameter is a stream vector S; the routine makes some assumptions about the contents of S, which should be as shown in Fig.15.

As the matters requiring description in this routine are rather complicated and also rather mixed-up, we give a 'guided tour' here, in place of the usual line-by-line commentary.

E (line 15) is a parameter block suitable for a TRANSFER call of Exec (II:9). The routine amends this block (line 17) to specify a transfer of n characters (n is the second parameter) and attempts to start the transfer (line 19). If the attempt is unsuccessful, the routine loops (§R, lines 19-35), continually incrementing a count i (line 33). Each time i becomes zero (about every three minutes), a series of pings is output on ExecConsole (lines 22, 23). The interval between pings is PAUSE1 increments of i (line 23). The number of pings in the series (3 for the reader, 5 for the printer and 7 for the punch) is specified in S↓PINGS, which is used (line 13) to compute the

interval of i ($0 \leq i < Last$, line 22) during which pings are emitted. Since (in order to avoid bullying the operator) several minutes elapse between each series of pings, the operator may find out what is holding up the machine by typing a character on ExecConsole: if the routine detects such a character (line 28), it outputs the string Message (the third parameter) on ExecConsole (line 30).

The cycle begins with i preset (line 12) to a small negative value (-PAUSEO). This causes the routine to wait for a short while, so that any transfer already in progress on the device has time to finish, before it begins to complain by pinging. At the end of the first series of pings, the boolean HeldUp is set (line 25). HeldUp is used (line 28) to ensure that the state of ExecConsole is neither tested nor reset unless a hold-up actually occurs: this is to avoid, normally, affecting similar use of ExecConsole for some other purpose (e.g. to break out of a tape-copying loop, as suggested in [2] §2.4.4).

The loop is broken when Exec accepts the TRANSFER request (line 20), and the routine finally amends the parameter block, exchanging the buffers participating in the double-buffering arrangements (line 37). (It is assumed that the word before each buffer is a pointer to the other buffer.)

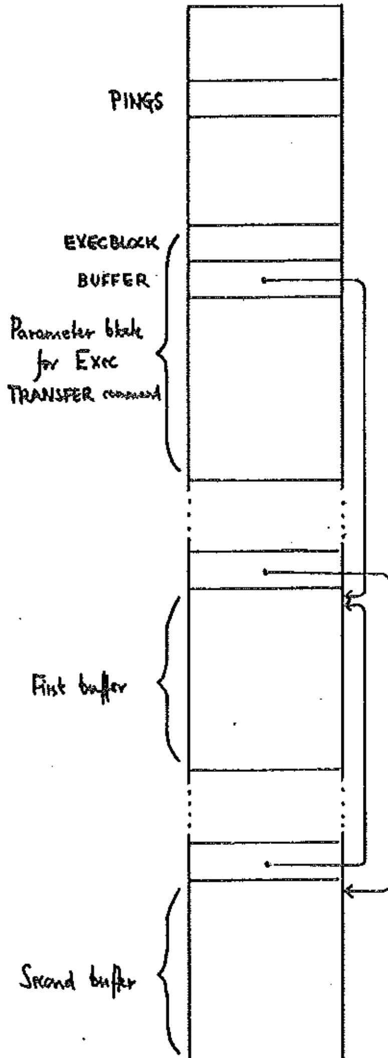



Fig. 15 Structure of stream assumed by InitiateTransfer

## II:5.4 READER

**BytesfromPT** (global 387)

BytesfromPT is a constant input stream, of 8-bit bytes from the paper tape reader. All input from the tape reader uses this stream. It is set up when the system is initialised (III:2.3). It is because the setting-up code is elsewhere that some of the component routines of BytesfromPT are temporarily global.

BytesfromPT is a fast stream, in the sense of [2] §2.6. It is double-buffered, and its structure is shown in Fig 16. Note that since this stream is permanent, the Close routine is set equal to the Reset routine.

**ReaderDev** (global 382)

The paper tape reader can read tape in either direction. EXEC allows this to be specified by using two different device numbers for the reader (see II:9). Since it is occasionally useful to be able to change the normal direction of reading, the device number is held in a global, rather than being specified as a manifest constant.

**ReadBuff**

Using InitiateTransfer (II:5.3), this routine initiates the input of a buffer. In order to read n characters, Exec requires a buffer of length n+1, because it uses the last word to indicate whether the transfer ended because the buffer was filled or because a specified termination character was read (see II:9, and the description of TryDiadAgain (II:5.5) which uses this facility). In this section, READBFFSIZE specifies the number of characters transferred, so 1 must be added to it for the second parameter of InitiateTransfer.

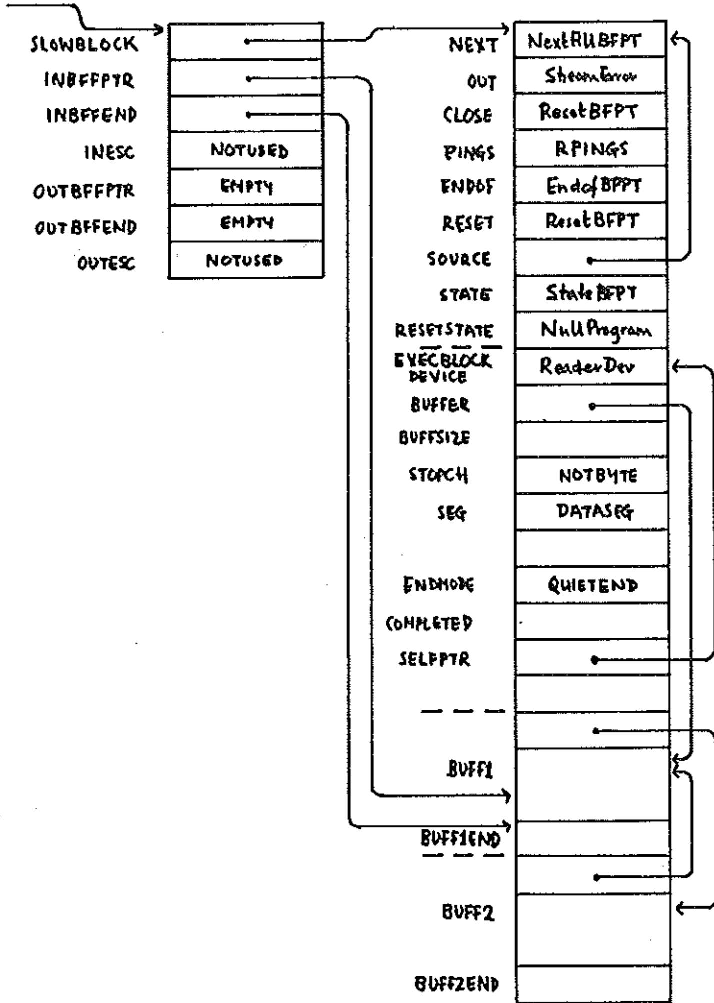



Fig. 16 BytesfromPT

FirstNextBFPT

This function is called the first time Next is applied to BytesfromPT after setting up, or after the stream has been reset. A call is forced by presetting suitable values in the fast block (see Fig.16).

Line 17: The parameter FB is the fast block.
18: S is the other, 'slow' vector.
20: The routine places the device number of the reader (obtained from ReaderDev) in the appropriate word of the Exec parameter block.
21: This is needed if ever TryDiadAgain (II:5.5) is interrupted.
22: It then initiates the reading of the first bufferful.
23: It overwrites the entry of itself (in the NEXT element of S) with NextFillBFPT, which will normally be used to refill the buffers.
24: It sets the elements of FB so that an entry to NextFillBFPT will be forced the next time Next is applied to the stream. This gets the double-buffering under way, by ensuring that the transfer of the second buffer will have been started, and therefore that reading the first buffer will have been completed, before any characters are removed from the first buffer.
25: There then follows an extra call of Next. In the version of Exec which we use, the reader is normally stopped by preventing the interrupts which input each character. This does not have any effect until the next character comes along, and the method avoids applying the brakes if another transfer is initiated soon enough. If not, the reader stops with this extra character in the reader's electronics trying to get into the processor (the reader repeats the attempt to interrupt until the ban is lifted, when the character is input, the brake relaxed and reading resumed). Though this is normally satisfactory, it means that when a new tape is loaded the first character input is usually the character after the last one read from the previous tape, which might well be invalid. Since a new tape is usually read by a reset stream, we can overcome this problem satisfactorily by ignoring one character in FirstNextBFPT. This is done by the extra Next call.
26: The function finally recurses, to return a character.

NextFillBFPT

This function is normally called when a buffer has been emptied.

Line 32: It calls ReadBuff to refill the emptied buffer and also (see InitiateTransfer, II:5.3) to swap the buffers.

33-35: It sets the fast block elements to point to the other buffer, and returns a character as a result. Notice that by starting the refilling of one buffer before beginning the processing of the other we ensure that a buffer is not processed until it is completely read.

ResetBFPT (temporarily global 405)

This routine is called when BytesfromPT is reset, or when an attempt is made to close it.

When the stream is reset, the reader may require operator attention, usually to load the next tape. So we arrange not to continue reading until the reader has been switched off-line (and, of course, on-line again).

Line 41: ResetBFPT sets the elements of the fast block FB to force an entry to a buffer-filling routine.

42: It then tests whether the reader is currently off-line. If so, there is no problem, and the NEXT element of the slow vector S is merely overwritten with FirstNextBFPT (see above). Otherwise, it is overwritten with NextWaitBFPT, which will wait for the reader to go off-line and on-line again.

NextWaitBFPT

This is the routine which waits until the reader is switched off-line and on-line again. Like InitiateTransfer, it also provides series of pings to energise the operator, and a means for the operator to enquire what is holding up the system. In addition, for cases where the operator knows the tape to be correctly positioned, there is provision for continuing without touching the reader (by typing a newline on the ExecConsole).

II:5.4

The structure of NextWaitBFPT is very similar to that of InitiateTransfer. There is a loop (§R, lines 51-63) during which i is incremented, and the reader tested. The loop is broken when the reader becomes on-line after having been off-line (line 53), or when a RETURN or LINEFEED character is detected on the ExecConsole (line 55). The pinging is managed as in InitiateTransfer and the operator enquiry very similarly, though there is no need for the message to the operator to be a parameter.

Line 62: Since this loop takes a longer time than InitiateTransfer's, it is allowed to cycle only halfway round the full range.

65: When the looping finally ends, FirstNextBFPT is called to begin reading.

(temporarily global 418)

StateBFPT

BytesfromPT is a source stream, in the sense of the description of State and ResetState (II:4.1). Here, the state of the reader is defined to be whether it is on-line or off-line.

ReaderOffLine

This is a boolean function to test whether the paper tape reader is off-line. It uses a special Exec command (see II:9), which sends a status request to the reader. The reader is considered to be off-line if either the request is rejected or the status turns out to be anything other than operable.

(temporarily global 401)

EndofBFPT

The 'Endof' function for the stream BytesfromPT. Its result is always false: see [2] §2.4.1.

(global 355)

TryAgain

TryAgain is a routine to be used when

Next[BytesfromPT]

has produced an invalid character. It moves the tape back to the offending character, so that a further call of Next will result in

another attempt to read it.

Lines 85-88: The routine first checks that its parameter is BytesfromPT, and otherwise gives up.

92: It also checks that no characters are PutBack (see II:3.2) to the stream, since then the offending character would have come from program, rather than from tape.

95: It then announces itself on ExecConsole. This gives time for the reading of the next buffer, which will already have started, to finish, and for the tape to stop.

97-99: After the current state of the buffer pointers are noted (line 97), the stream is effectively reset (but without causing the reader to wait), so that if TryAgain is interrupted BytesfromPT will recover satisfactorily.

100: The Exec parameter block (see II:9) is then amended to reverse the direction of the tape movement.

102-110: The routine now prints out (in octal) five characters from the buffer, including the offending one, which is marked 'wrong'. (The routine attempts to have the offending character as the middle of the group, but if this is impossible because of the buffer boundary it does the best it can. This is the complication in the definition of First (lines 103, 104).)

112-121: It is now necessary to read (backwards):

(1) The 'extra' character in the reader electronics (see ResetBFPT above) (line 114: remember that the buffer size must be augmented by one word for InitiateTransfer - see ReadBuff, above).

(2) The buffer read after the offending one (line 115).

(3) The offending buffer up to and including the incorrect character (line 120: the second parameter is k+2 because k=0 would imply that the last character in the original buffer was wrong, and hence that one character must now be read). Before doing this, the address Ly where the offending character is to go in the buffer is computed, and the routine then waits (line 121) until the character arrives there.

123-126: After printing its value (which may or may not be the same as before) the routine calls Wait (II:6.2) to allow the operator, if he wishes, to examine the physical tape.

Note that neither in this routine nor in the similar one TryDiadAgain (II:5.5) do we re-read automatically without operator intervention. If we did, there would be no incentive to summon the maintenance engineer until the reader became quite unusable.

## II:5.5 DIADS

### Note on diad format

Diad format is a way in which 16-bit words may be punched on 8-bit paper tape. Sumcheck information is included so that mispunches or misreads may be detected.

Information on a diad tape is arranged in blocks. Blank tape is ignored between blocks. ERASE characters (8377) are always ignored, between blocks and within them, since the punch hardware can generate them automatically for overpunching an error.

The STOPCODE character (8004) is also allowed between blocks. It causes BytesfromPT to be reset and the reader to wait until the operator intervenes: it may therefore be used to prevent the reader from running off the end of a tape. It is otherwise ignored.

Each block begins with the warning character 8125. The next byte is interpreted as an integer n ($n \leq 30$), which indicates the number of 16-bit words to be constructed from the block. To this byte and to all succeeding bytes in the block an escape rule applies: if the value of any byte is 8033, it is itself ignored, but it indicates that 1 is to be added to the following byte. The escape rule must be used to specify 8377, as ERASEs are ignored everywhere, and also to specify the escape character (8033) itself.

The next n pairs of bytes (after application of the escape rule, if appropriate) form words (more significant half, followed by less significant). Each byte, including the count n, is added to a checksum (after the escape rule is applied): the sum is output as the final two bytes in the block (again applying the escape rule if necessary).

### DiadRead (global 338)

Since reading paper tape punched in diad format was a very common operation, especially before we had a disc, this constant input stream was provided for the purpose. It is created during system set-up (III:2.3), by applying WordsfromDiads (see below) to BytesfromPT (II:5.4).

WordsfromDiads (global 340)

This is a stream function which takes a byte input stream (usually BytesfromPT) as an argument; the result is a fast word input stream. The layout of the stream vectors is as shown in Fig.17.

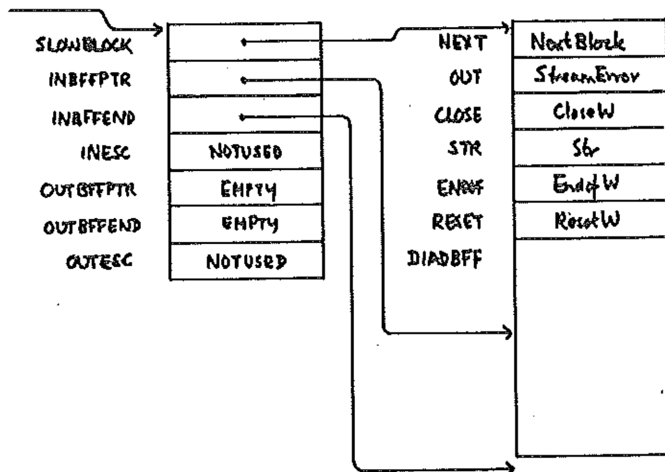



Fig. 17 WordsfromDiads[Str]

NextBlock

This is called to read a diad block into the buffer. S is the slow vector and I the input stream (that is, the parameter stream of the original stream function call).

Lines 54-61: Loop §L2 searches for a warning character WCH, using TryAgain (II:5.4) if it encounters anything invalid.
63-77: The block is then read according to the rules.
78-80: If the sumcheck fails TryDiadAgain (see below) is called.

82-84: Finally the fast block pointers are set, and the first word returned as the result.

NextByte

This is used by NextBlock above to obtain bytes. It applies the escape rule.

NextCh

This is used by NextByte to obtain paper tape characters. It applies the 'ignore ERASE' rule.

Note that by using these functions we have involved ourselves in two extra function calls for each tape row read: this is, perhaps, a little profligate in its use of machine time, and may be worth optimising.

EndofW

The primitive Endof function (II:9) returns false if the fast buffers are not empty. If they are empty, this function tests the parameter stream.

ResetW

The fast block pointers are set to force a call of NextBlock when Next is applied, and the parameter stream is reset.

CloseW

The parameter stream is closed and the working space returned.

TryDiadAgain (global 336)

This routine is used when a WordsfromDiads stream detects a sumcheck failure in a diad block. Its purpose is to move the tape to the beginning of the block which failed, so that the

stream can have another attempt to read it.

Lines 135-150: Its first part is very like TryAgain (II:5.4) (except that here there is no printing of characters in octal); it brings the tape back to just before the last character processed (which would be the final character of the sumcheck word).

148: In fact the first backwards read 'unreads' 3 characters, rather than TryAgain's 1. This causes the tape to move a little further, so that even if the error occurred because the reader omitted a character the tape will still be moved back to a position within the offending block.

152-156: The next section moves the tape back to the first warning character WCH detected. As we mentioned in the description of ReadBuff, the Exec TRANSFER command uses the final word of an input buffer to indicate how the transfer terminates. If the transfer ends because the buffer is full, the last word is set to true. Otherwise (if it terminates because a specified terminating character is read) the final word is untouched. So we alter the Exec parameter block to specify WCH as the terminating character, and then enter Loop §R. In this loop we set the final words of both buffers false, and start a transfer. When the transfer has finished we test to see whether either of the buffers was filled, and repeat until this is not so.

157: We suspend the terminating character facility.

159: We now output details of the sumcheck failure (we defer it until this stage so that it provides time for the tape to come to a firm stop).

160: Finally Wait is called, to allow the operator the possibility of examining the tape.

ReadBackwards

This is a private routine, applied to BytesfromPT by TryDiadAgain, to perform a backwards transfer of length n characters (or until a terminating character is reached).

Line 169: The reader's direction is set to read backwards.

170: The flag which indicates the end of a transfer is reset.

173: This null loop continues until the transfer initiated in the previous line is completed.

Note that this routine works only if the reader is not busy when the routine is entered. If it is, then E↓COMPLETED will be set when the previous transfer ends, and the transfer initiated by this routine will only just have started. TryDiadAgain tries to ensure that this condition is satisfied, by typing a message on ExecConsole at a suitable moment (line 142), in order to allow the reader time to come to a halt. Matters would, of course, be far cleaner if we were able to find out whether a transfer was in progress by asking the reader directly: but, because of a hardware design peccadillo, this information is not part of the reader's 'status' (see ReaderOffLine, II:5.4).

## II:6. Miscellaneous facilities

### II:6.1 CLOCK

**OutDateandTime** (global 348)

This routine outputs the date and the time to the character output stream S, preceded by seven asterisks and followed by a newline.

**OutDate**

This routine outputs the date d to the output stream S in the format day/month/year.

**OutTime**

This routine outputs the current time to the character output stream S.

Line 26: If ClockRestarted is true it indicates that the call of TimeofDay in the previous line will have started the clock and asked for the current time on the Console stream. So there is no point in immediately re-outputting the time on the Console.

**TimeofDay** (global 364)

This function gives the current time in minutes from the preceding midnight.

Line 34: Clock is a vector; its format is given in the description of the Exec commands (II:9).
36: If the clock is already going this call will have no effect.
37: The word (Clock↓BUFFER)↓TIME contains the number of minutes left in the day.

RestartClock

This routine uses Exec (II:9) to attempt to restart the time-of-day clock. Its parameter is the Exec parameter block for the clock device, the format of which is described in II:9.

Lines 49, 50: If the attempt is successful, the flag ClockRestarted is set to true and the buffer word (E↓BUFFER)↓TIME (which the clock will periodically decrement) is initialised.
51: If the Exec command is rejected it is assumed that the clock is already going, so no further action need be taken.

AskTime

A function which obtains the time of day by asking the operator.

II:6.2 MISC

AddressZero (global 371)

When the system is constructed words 0 and 1 of the code segment are set to a jump to this failure routine.

Copy (global 36)

This routine copies n words (v↓0 to v↓(n-1)) from vector V1 to vector V2. It uses the special hardware assistance provided for fast streams and for the function TransferIn (see II:9).

Lines 12-14: The vector FB is set up to look like a fast input stream referring to V1 as its buffer.
15: TransferIn (II:9) places n words from this stream into V2.

EqS (global 352)

This function tests two strings for equality.

Line 20: A given pair of strings will probably have either different lengths or different initial characters. This line therefore immediately disposes of most cases without the paraphernalia of setting up a loop.

21: (p↓0 rshift 9) would be slightly more efficient, but less perspicacious, than (p↓0 rshift 8)/2. The shift extracts the first byte (the length of the string in characters), and the division converts it to the length in words.

22: The loop is broken as soon as two words fail to match.

NullProgram (global 359)

A routine which does nothing is sometimes useful.

Wait (global 372)

A routine to wait until the operator types a newline on the console.

## II:7. Disc routines (1)

### II:7.1 DISCXFER

DiscPage (global 335)

The length of a transfer to or from the disc is an integral number of pages, a page being 256 words. For reasons of hardware economy Modular One insists that the absolute core address of the start of a transfer area should be a multiple of 256. We consider such a restriction intolerable, so we arrange to set up one page, DiscPage, to fulfil the condition (see III:1.1); all the system's disc transfers are then to or from this page, the information being copied to or from the required place in core with no restriction on its address. To save space, some system routines use DiscPage explicitly themselves for accessing the disc, but since all disc transfers are *via* DiscPage it is necessary to make quite certain that information is used before it is overwritten.

CoretoDisc (global 334)

This routine transfers the contents of the vector v (length 256 words) to page p on the disc.

Line 9: If v is DiscPage itself it is unnecessary to copy the information. We assume that the time wasted by the unnecessary test when v is not DiscPage is less than the time that would be wasted by unnecessarily copying the information if it is. Whether this is true depends on the relative frequency of the two situations.

10: A sumcheck is written to the last word of DiscPage. The calculation of this sumcheck is performed by a special hardware function (see II:9).

12: DISCWRITEPERMITTED, in the code segment, is a switch which may be used to prevent writing to the disc in dangerous situations.

MFL file
f-1
f
f+1

Heading file

Body of f
1
f
Sumcheck

3
f
ENDBODY
n
unfilled space
n
Sumcheck

2 SERIAL
f THISFILE
NXTPAGE
Sumcheck

An index file
f
Name1
Name2

Fig. 18 Structure of the disc filing system

DisctoCore (global 333)

This routine transfers the contents of page p on the disc to the vector v in core.

Line 20: The routine breaks out of loop §r if either the sumcheck test is satisfied or the code segment word SUMCHINHIBITED is set. SUMCHINHIBITED may therefore be used as a switch to ignore the sumcheck test if the information on the disc is non-standard.
24: By typing 'I', the sumcheck test may be manually overridden from the console.
26: See the remark on line 9 above.

DiscTransfer

This routine transfers information between the core vector DiscPage and page p on the disc. The parameter Dev is the device number for the Exec TRANSFER command (II:9), and it indicates whether the information is to be transferred to or from the disc.

Line 38: The routine attempts to start the transfer by an Exec TRANSFER command. This command may be rejected; if so, Exec places a value giving the reason for the failure in the word whose address is given by the third parameter, and as usual jumps to the label Rj. It is also possible that the transfer, though successfully initiated, may terminate unsuccessfully, for example because of a parity failure in the disc hardware. If this happens, Exec automatically overwrites the word of the parameter block containing the disc page address (E↓PAGENUMB). Since in our version of Exec the two kinds of failure indicator can be distinguished (the second kind are negative), we find it convenient to direct the first kind of indicator to E↓PAGENUMB as well; so its address is given as the third parameter of the Exec call.
39: If the transfer is successfully initiated, the routine waits until it is completed. Since our disc is very fast and our interpreter comparatively slow, this wastes a negligible amount of time.
41: If E↓PAGENUMB has not been overwritten, the transfer has terminated successfully.
46: The hardware may be expected occasionally to perform unsuccessful transfers (the manufacturers suggest that one a day is tolerable), so when this occurs we automatically repeat

the attempted transfer once (§f), as next time it will probably be successful.

47, 48: Before repeating, the failure is reported on the executive console, to keep the system management informed about the current reliability of the disc.

50: If the transfer cannot be initiated, or if it repeatedly fails to terminate successfully, the operator is informed via Console.

52: The operator has three options: by typing a new line he can cause the routine to have another attempt at the transfer (loop §r), possibly having removed the cause of the failure (e.g. he may have noticed that the disc was not switched on); he can interrupt the system; or by typing 'I' he can cause the system to continue as if the transfer had been successful. Notice that since the 'I' option can be dangerous if not used circumspectly an unusual letter is employed, so that it should not happen by mistake.

Message

A private routine for outputting disc failure messages.

## II:7.2 DISCFS

The routines in this and the next few sections are concerned with the disc filing system, which is described in [2] §3. The general structure of the system is shown in Fig.18, which is very similar to [2] Fig.4 (see the introduction, Part I, for a list of the differences between papers [1] and [2] and this published text). A file heading is shown in Fig.24 (II:8.3).

This section deals with disc storage allocation. The addresses of all the free pages on the disc are kept in a file, the free store file. The body pages of this file have an extra word of housekeeping information: when storage is allocated alterations are made to the last page of the file, and in case the page runs out it is convenient to have a pointer from each page of the file back to its predecessor. The structure of the file is therefore as shown in Fig.19.

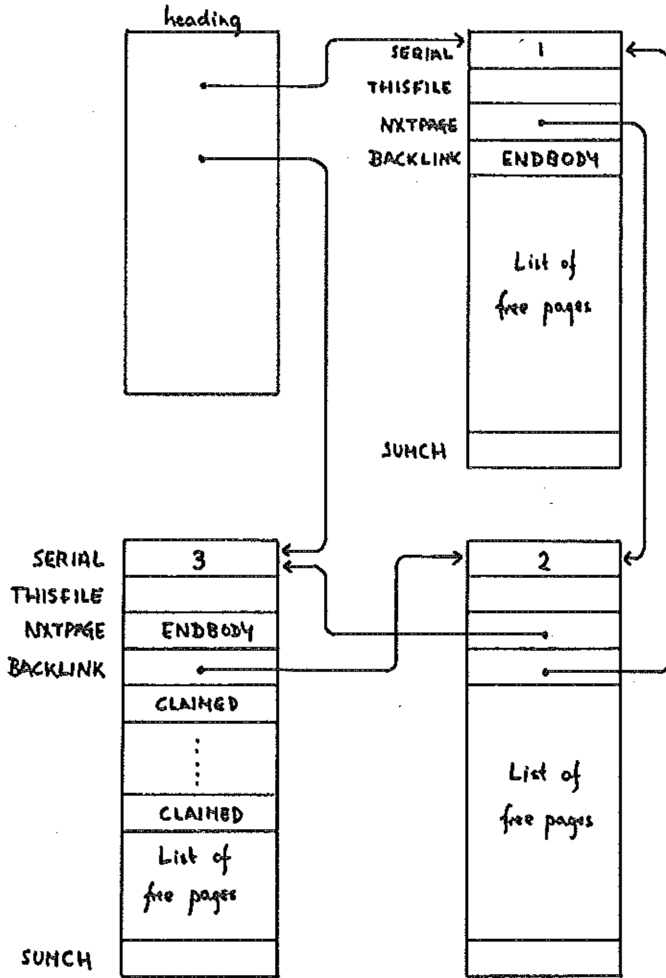



Fig. 19 Structure of FreeStoreFile

Note, that this is not the best structure of the free store file. A better one would use a bit map, which for our disc could be accommodated on one page. This method would avoid the possibility, which exists at present, of multiple entries of the same free page.

The first word of every free page is set to zero. NewDiscBlock can therefore check that a page it is about to allocate is indeed free, and can hence guard against the effects of multiple entries. But, as is explained below, this check is not infallible.

FSF (global 329)

For efficiency's sake the last page of the free store file is kept and manipulated in core, in the vector FSF. The first 256 words of FSF contain the free store file page, and a further two words are used to keep housekeeping information, as shown in Fig.20. These words include a pointer to the first entry in the page. Note that this pointer is not kept on the disc; instead, empty words in the page contain the constant CLAIMED. The pointer in core can therefore be initially set to the start of the page.

The page in core is written back to the disc at frequent intervals (at the beginning of each Run (by PrepareforRun, II:1.2), and whenever the page kept in core changes to another page).

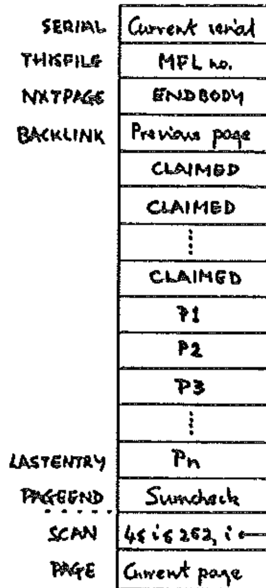



Fig. 20 Structure of FSF vector in core

NewDiscBlock (global 330)

This parameterless function is used by filing system routines to claim a page from the disc free storage.

Line 10: This is the pointer into the core copy of the last page of the disc free store file.
11-29: If the free store file page in core is empty, it is necessary to change to its predecessor.
13-16: If there is no predecessor, the filing system is full.
17-28: Otherwise, the predecessor is established in core as the new last page of the file, and the old last page is handed out as the result.
17: This is to be the result.
21: The housekeeping in the new core page is altered to make it the last page of the file.
22: It is updated on the disc.
25-27: The heading of the file is also updated to keep step with the change.
30-36: This is the more usual case, when the page in core is not empty.
31: This if valid will be the result.
33: By this test the function ignores empty words in the page.
35, 36: The page about to be allocated is examined to check that it is in fact free. This is, however, not an infallible check against page-sharing: if the page has already been allocated (because of a multiple entry), it need not necessarily yet have been overwritten.
37: The whole process is repeated until a result is obtained.

ReturnDiscBlock (global 331)

This routine is used to return pages to the disc free store.
Lines 48-62: If the free store file page in core is already full, the page being returned becomes the new (empty) last page for the file.
51: The page in core, now no longer the last page, is written back to the disc.
57: All the words in the new last page are set empty.
58: The new last page is written to the disc,
59-61: and the heading is updated.
64, 65: If the page in core is not full, then the page being returned is entered in it.
66, 67: The page being returned is itself marked free.

## II:7.3 DISCIN

### CheckType

This is a private routine used by InfromFile and OuttoFile. It takes two parameters, a file and its heading, and checks that the file is a valid parameter for the stream function.

Line 15: One bit in the type of a file (STREAMABLE) is used to indicate whether the file is such that its information may be meaningfully accessed by a stream.

### InfromFile (global 321)

This function takes a file as parameter; the result is a word input stream from the file. This is a fast stream and its structure is shown in Fig.21.

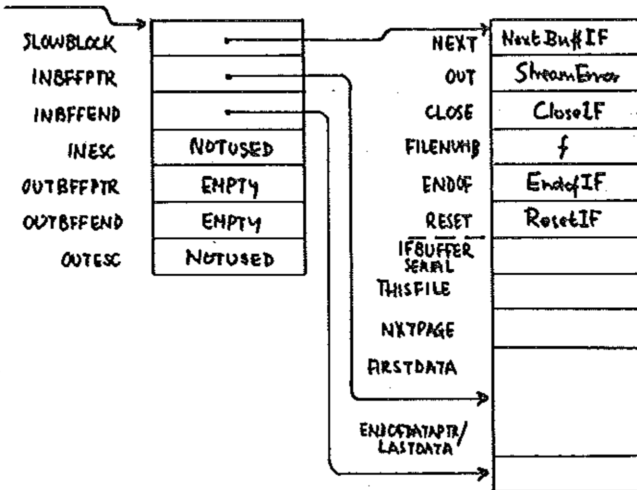



Fig. 21 InfromFile [f]

Line 25: A validity check on the file.
27: The vector containing the heading is discarded.
28: The file's 'date last read' is updated.
33-36: The fast block is initialised, both buffers being set empty.
38-44: The slow block is initialised.
44: The stream is set up so that it appears to be at the end of a page before the first page in the file's body: that is to say, the 'next page' element points to the first page of the file. This is to force the first call of Next (that is, of NextBuffIF and EndofIF) to read the first page into the buffer.
46: The value of a fast stream is the logical complement of the fast block vector.

NextBuffIF

The fast stream hardware calls this function to refill the stream's buffer, if possible. The result is the first character of the new buffer or, if the stream has now finished, the constant ENDOFSTREAMCH.

Line 51: Most of the work of this routine is done as a side-effect by EndofIF.
53: The fast stream hardware is activated (recursively) to produce the first character.

EndofIF

The fast stream hardware automatically produces the result false for the Endof function if the buffer is not empty (see II:9). If it is empty, this function is called (it is also called explicitly by NextBuffIF above). As Fig.18 (II:7.2) shows, the final page of a normal file has a pointer indicating how much of the final page contains information. It is possible that the final page might be completely empty (this is a consequence of the fact that a word is reserved for the pointer only on the final page). It is therefore necessary for EndofIF (which only deals with the case of an empty buffer) to look at the next page of the file, in case it turns out to be an empty last page.

Line 57: FB is the fast block vector.

58: S is the slow block vector.
59: B is the buffer itself.
60: If B↓NXTPAGE = NULLBODY the file has no body, and this value will have been set (line 44 above) when the stream was created. If B↓NXTPAGE = ENDBODY then the buffer just emptied contained the last page of the file.
62: The next page is read.
63-65: The fast block pointers are set.
64, 65: If the page is the last page, the end of the information in the buffer is determined by the pointer word B↓ENDOFDATAPTR; otherwise it is the end of the buffer.
67: This result is true only for an empty last page.

CloseIF

The 'Close' routine for the stream: storage is returned.

ResetIF

This is the 'Reset' routine for the stream. Various quantities are set back to their initial condition, using information from the file heading, so that subsequent calls of Next will recommence the reading of the file from the beginning.

Line 80: See comment on line 44.

EntriesfromFile (global 320)

This stream function may be applied only to an index file; as described in [2] §3.2, each time Next is applied to the result stream it produces a vector containing an entry in the index.

Lines 91-95: A check that the parameter file f is an index.
102: S↓STR is set to an ordinary word input stream from the file.

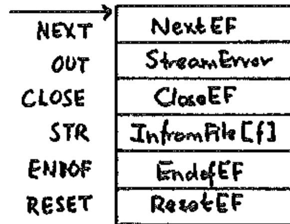



Fig. 22 EntriesfromFile[f]

NextEF

The 'Next' function for the stream.

Line 110: Unless the stream has ended, n will be the length of the next entry (see Fig.25, II:8.5).
111: If, however, n = ENDOFSTREAMCH, it is handed straight out.
114: Hardware assistance to fill the vector.

EndofEF
CloseEF
ResetEF

The 'Endof', 'Close' and 'Reset' routines for the stream.

## II:7.4 DISCOUT

ReturnChain

This private routine is used to return a chain of file body pages to free storage. The parameter Page is the first page of the chain; f, the other parameter, is the file to which the body chain belongs, and is used in a validity check.

Line 14: This is the check that each page returned belongs to file f: if it were not done, an invalid link could cause decimation of the filing system.
20: The loop continues until the end of the chain is reached.

CheckPerm

This private routine is used by DeleteBody and OuttoFile. Its parameters are a file and its heading, and it checks that the current user of the system is allowed to overwrite the file.

DeleteBody (global 326)

A routine to remove the body of a file.

Fig. 23 OuttoFile[f]

Line 35: A check that the file exists.
37: Deleting the body is a special case of overwriting the file.
39-44: The main part of the routine (lines 40-44 are omitted if the file already has no body).
40, 41: The body is detached from the heading, and the 'date last written' is updated.
42: The body chain is returned. Note that this is done after the body is detached from the heading, to avoid catastrophe in case of interruption.
43: The owner of the file is credited with the number of pages returned. Remember that h contains the heading as it used to be, before the call of UpdateHead (line 40).

OuttoFile (global 322)

This stream function produces a word output stream to the file f. The stream is a fast stream, and its structure is shown in Fig.23.

Line 52: FB is a vector for the fast block.
53: S is for the slow block.
54: B is the part of S reserved for the buffer.
57: A check that a stream may validly be created to the file.
58: A check that the user is allowed to overwrite the file.
60-64: The fast block is initialised.
66-77: The slow block is initialised.
76: This word is to contain the address of the page for which the current buffer is destined.
77: S↓OFPAGE at present contains the first page of the new body. Note that the old body of the file is not abandoned until the stream is explicitly closed. As remarked in [2] §3.1.1, this is designed to minimise accidents in the event of error.
81-83: The buffer is initialised with the housekeeping information for the first page.
84: For safety's sake this page is written back to the disc, to ensure that it is no longer marked free, which would keep it liable to be re-allocated somewhere else. If we did not do this, the page would be at risk until the buffer had been filled, when it would of course be overwritten. Only the first page of the file has this extra overwriting: efficiency forbids it for the other pages, which are therefore left at risk for a while. But it is the start of a program which tends to be the most precarious part. In any case, the disc free store system should not make this kind of mistake at all

(see II:7.2).

86: As was mentioned in [1] §1.2.2, these streams are one of the places where the ClearUpChain must be used, so that final housekeeping action may be taken if the stream is prematurely abandoned. C is the part of the slow block reserved for the entry in ClearUpChain.

87-91: C is initialised, and entered in the chain.

93: The value of a fast stream is the logical complement of the fast block vector.

OutBuffOF

The fast stream hardware calls this routine to output a full buffer, for which it uses the private routine TurnPage.

Line 101: The second parameter of TurnPage will be the next page of the file.

102: The fast buffer pointer is reset.

TurnPage

This is a private routine used by OutBuffOF.

Line 108: The housekeeping information in the buffer is set to point to the next page.

109: The current buffer is written to the disc.

110: The next page becomes the current page.

111: The buffer's housekeeping is updated.

CloseOF

This routine is called when the stream is closed.

Lines 119-121: Since the stream is being closed properly, no clearing up action will be required later; so the entry is removed from ClearUpChain.

123: This call will attach to the heading the body created by the stream as the new body of the file, and will return the old body chain. Since it is the 'Reset' routine, it will also set the stream up for further output, by providing a new first page for a new body.

124: This new first page is returned.

125, 126: The free storage used by the stream is returned.

ResetOF

This 'Reset' routine attaches to the file heading the body chain so far created by the stream, thus making it the body of the file, and it returns the old body to free storage. It then prepares the stream for further output.

Lines 135-142: If nothing has been output since the stream was created, the new file is to be null. The simplest procedure is merely to delete the file's old body and to alter the places in the stream's structure where the old body is described.
145: The owner is charged for the amount by which the size of the new body exceeds the old size.
147: The pointer in the last page of the file is set up.
148, 149: The new file body is completed.
151, 152: The new body is attached to the heading, and the date last written is updated.
154, 155: Unless the old body was null it is returned.
157-162: Various quantities in the stream's structure are re-initialised.

FailClose

This routine is for the stream's entry in ClearUpChain (see [1] §1.2.2 and II:1.2). It is called at the premature end of the Run in which the stream is set up; it returns to free storage the incomplete new body under construction, leaving the file itself unchanged.

Line 168: The value of the slow block is calculated from the parameter C, which is the ClearUpChain entry itself.
170, 171: The last page of the chain so far exists only in the buffer, and it is necessary to write it to the disc before the chain is returned.

# II:8. Disc routines (2)

## II:8.1 CHARGE

This section manages the accounting system for space on the disc.

**Charge** (global 306)

This routine should charge the user specified as the first parameter for the number of pages specified by the second. A more elaborate algorithm than the one incorporated here was in fact devised, but was rejected because we were short of core space. In any case, we find that a bit of personal moral pressure applied by the system management is sufficiently effective at keeping users' disc space within reasonable limits, and a good deal more civilised.

## II:8.2 UPDATE

**Update** (global 312)

This routine is used by the system for altering a single field in a file heading. The file is specified by the first parameter. The word specified by the second parameter may either be altered to the value specified by the third, or be incremented by that value. The format of a heading is shown in Fig.24.

Line 3: The sign of the second parameter specifies which of the two actions is required.
4: Element is the word in the heading which is to be changed.
6-14: Validity checks on the parameters.
6-9: A check that the file exists.
10-13: A check that the required Element exists; that is, that it is within the bounds of the heading.
16: HdAddr is a vector containing the disc address of the heading.
17: DiscWord is the address in its page of the element under consideration.
18: The page containing the heading is transferred to DiscPage

(see II:7.1).

19-27: If the word required is past the end of this page, so that it is on the next page of the file, then the next page is brought into DiscPage and the addresses in HdAddr and DiscWord relocated.

21-23: A system failure if there is no next page of the file.

29, 30: The required word is altered.

31: The page is written back to the disc.

## II:8.3 UPDATEHEAD

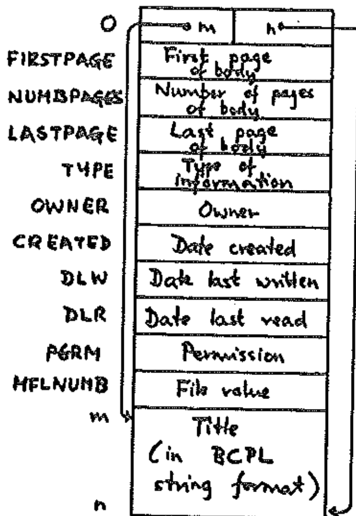



Fig. 24 Structure of a file heading

UpdateHead (global 311)

This routine is used by the system for changing several fields of a file heading at once. The new values are specified by the last six parameters, but for any field which is to remain unchanged the corresponding parameter may take the value SAME.

Line 9: The parameter list is treated as a vector. This is specifically allowed in the BCPL language (see the Appendix).

10: The page containing the heading is brought into core.

12: Loop §F is not obeyed for fields which are to remain unchanged.

14-22: The appropriate word in the heading is selected.

23: Its address in DiscPage is calculated.

24-34: If the address is beyond the end of the page, the next page is brought into core.

26-28: A failure occurs if the next page does not exist.
29: The page already in core, which may have been altered, is written back to the disc.
31, 32: The heading address is relocated so as to refer to the new page in core.
33: The address of the word under consideration is relocated.
35: The content of the required word is changed.
38: The heading page is written back to the disc.

## II:8.4 FINDHEADING

LookUpinMFL (global 319)

This function scans the Master File List (MFL) to find the entry corresponding to the parameter file. Each entry consists of two words giving the disc address (page and word) of the start of the file's heading. The MFL is scanned page by page (by the until loop, lines 13-17). This admittedly involves more disc transfers than are absolutely necessary; however, the MFL is only used when an access route to the file (e.g. a stream) is set up, and not during the normal course of its use. This inefficiency therefore happens relatively infrequently, and there are other more important things to optimise.

If no entry is found for the file the result is NULL. It might be thought that such an outcome would be grounds for failure, and in normal use this is so. But it is convenient for some administrative programs (e.g. the disc housekeeper) to use this function when scanning through all the files on the disc, so the decision on whether to fail or not is left to an outer level.

Line 9: If the file value is negative the file does not exist.
14: If the end of the MFL is reached before the file is found, the file does not exist.
19: w is the address in DiscPage of the entry required.
20: If the first word of the space reserved for the entry contains the constant NOTENTRY, the file does not exist.
22, 23: The entry is copied into a new vector. Note that it is assumed that an entry can never overflow onto the next page of the MFL.
24: The new vector is the result.

FindHeading (global 318)

As described in [2] §3.1, this function produces a vector in core containing the heading of a given file.

Lines 34-36: A failure if the file does not exist.
38: The page containing the heading is brought into core.
43: The vector h is to contain the heading.
46: This tests whether all the heading is in the page in core.
47: If so, it is copied into h.
49-52: Otherwise, a check is made on the existence of the next page of the heading file, which contains the remainder of the heading.
53: The part of the heading in the current page is copied into h.
54: The next page of the heading is brought into core.
55, 56: The remainder of the heading is copied into h.
59-62: When a file is deleted, the TYPE field in the heading is overwritten with the value DELETED; the heading and the MFL entry are finally removed when the heading file is compacted by the disc housekeeper. So, if the file is found to have been deleted, a null result is returned for the heading.

## II:8.5 LOOKUP

LookUp (global 317)

As described in [2] §3.2, the result of this function is the file associated with the names Name1 and Name2 in the index file i. The index is scanned until an entry with the specified names is found (loop §u, lines 19-38).

The first version of this function used the stream EntriesfromFile[i] (see II:7.3). It was found, however, that this was rather slow; in particular, the provision of little vectors for each item of the stream of entries made heavy use of the core storage allocation system (II:3.1). As stated in [1] §2.2.1, our storage allocation system is designed to be efficient in its use of available storage, but it is fairly expensive in execution time. We therefore decided that there was a case for streamlining this function, and the version given here uses an ordinary word stream rather than a stream of entries.

There are two forms of index entries, which are described in [2] §3.2.1. There is also a form for an entry which has been deleted but not yet removed from the index (this third form is not generated by the current system, but earlier versions which do so are still extant). All three formats are shown in Fig.25. Note that for no logical reason, but merely to economise on space, several items of information may share the same field in an entry. Thus the status of an entry (i.e. whether it is deleted) shares with the first name; the file number of a normal entry shares with the third name of a linked entry, and the sign of this word is used as a flag, accessed using the name LINKING, to indicate which of the two forms of entry is being used.

Line 7: An attempt to look up an entry in a null index might be expected to lead to failure (as in the two following tests, lines 9 and 12). However, the treatment given here implies that a search for a file will produce an overall result instead of failing catastrophically, even if the search is in several stages and proves unsuccessful in an early stage when searching for a particular index.

18: This is a vector probably large enough to accommodate any entry which may occur.

19-38: Each cycle round this loop examines one entry.

21: The first word of an entry is its length.

22-26: If v is too small for the entry, it is replaced by another vector large enough.

27: Hardware assistance to read the entry into v.

28: The entry is not examined if it has been deleted.

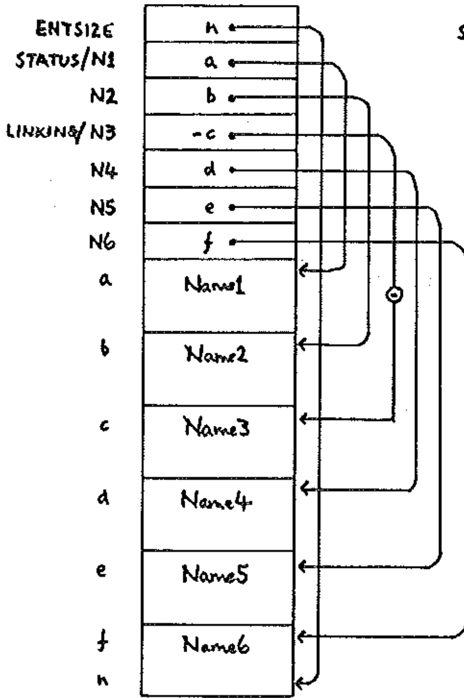
29: Note that BCPL is such that if one name does not match the other name will not be tested (let those who invoke important side-effects in innocent-looking boolean functions beware).

32: NotLinked indicates whether the normal form of entry is in use.

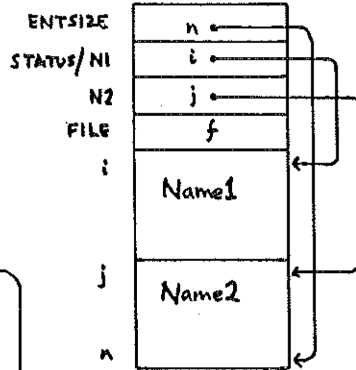
33: In the normal case the required file is specified explicitly in the entry.

34, 35: In the case of a linking entry it is necessary to look up the further names. The minus sign in line 45 is needed because reversal of the sign of v↓LINKING, which is also v↓N3, is used to indicate a linked entry.

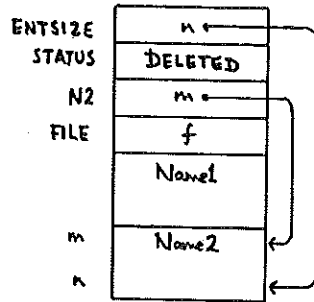
41: If no entry is found with the required names the result is NULL. See [2] §3.2.




(b) Linking entry




(a) Normal entry




(c) Deleted entry

Fig. 25 Index entries

## II:8.6 LOADFILE

LoadFile (global 378)

This routine is for loading a program contained in an IC file f.

Lines 11, 12, 15: These three lines are a trick to prevent fragmentation of the core free store. If they were omitted, the storage required for the input stream (set up in line 14) would probably come from the start of the available space, and the storage required for the loading of the program (the I-blocks and possibly D-blocks) would occupy the adjacent area. Then, when the input stream was closed (line 18), the space returned would be isolated from the rest of the available space. By claiming a large vector v before setting up the stream, the stream vector is forced to the other end of the available space. By returning v before loading the program, the program is able to occupy the start of the original space, so that when the stream is closed the largest possible contiguous space remains.

LoadSystemFile (global 365)

This routine is used to load library programs entered in the SystemIndex.

Lines 29-31: A check that the specified file does in fact exist.

## II:9. Special functions in WIC

### OS/SF

This section defines the seventeen words of machine code in the operating system. Though the 'assembly code' text is given, we do not propose to describe the assembly code in any detail. It has none of the sophisticated facilities of modern assembly codes: there is in fact no point in making it easy to read or write, as it is used very rarely.

Line 1: This specifies a NEWSECTION block (see the note on binary format, II:1.3).

3: This introduces a block of code seventeen words long.

4-16: This defines the seventeen words of code.

4-6: These are the three 'essential routines' described in [1] §2.1. They therefore cannot be expressed in BCPL, but their action is described below. (The mnemonics LDPRG and STPRG arise because FetchCode used to be called LoadProg, and StoreCode StoreProg. But with a name like 'LoadProg' one was never quite sure whether the prog was going to the store or from it.)

8-16: These lines describe seven routines which use hardware assistance merely to make them run faster. Equivalent texts of these routines are given in BCPL after the assembly code part, and they are described below.

18-29: This is an interlude (see II:1.3) intended to initialise the global variables allocated to the routines defined above, when this section is loaded. In fact this interlude is never obeyed: instead, it is analysed by the program (the system dumper) which constructs the operating system as a core image on the disc (see III:1.1), and it is the dumper program which sets up the globals appropriately. This is the reason for the comment (lines 35, 36).

We now describe the individual routines.

### FetchCode (global 2)

y := FetchCode[x]

The result of this function call is the content of the word at

address x in the code segment.

<u>StoreCode</u> (<u>global</u> 3)

StoreCode[x, y]

This routine updates the content of the word at address x in the code segment; the new value is y.

<u>Exec</u> (<u>global</u> 4)

Exec[Command, RejLab, <u>lv</u> Result, a, b]

This routine invokes a command to the Computer Technology Limited Executive program E2. The significance of the five parameters is as follows:

Command — This specifies which particular Executive command is being given (each of the possible commands is described individually below).

RejLab — If a command is accepted by Executive, the routine Exec ends in the normal way. If the command is rejected, the routine ends with a jump to RejLab (Exec takes care of the problem of the value of the P-pointer, which is what normally prevents labels from being passed as parameters). However, the Modular One machine code conventions are such that if an Exec command must be always accepted, so that no special exit is required, the exit path is the one which in the more general case is reserved for the special exit. This means that if a command can <u>never</u> be rejected, it <u>always</u> ends with a jump to RejLab. This is the sort of thing that happens if you try to be logical about machine code.

<u>lv</u> Result — The result of a successful Exec command, or the reason for rejection if unsuccessful, is placed in the address given by this parameter.

a, b — These are used to contain parameters of the Exec command itself. The values required for each particular command are given below.

We now give the descriptions of the individual Exec commands used by the system. Readers who are conversant with the Modular One software will notice that the system uses only two of the thirty-two commands provided by the E2 Executive: TRANSFER (which Computer Technology calls READI or WRITEI) and CANCEL. In the case of TRANSFER the specification is affected somewhat by the

interpreter. The interpreter also makes internal use of one or two further standard commands. The other two commands (LOOKATRDR and READABS) provide 'hardware' facilities required by the operating system. They were not available in the standard E2, so we amended it to provide them.

TRANSFER

This account of the TRANSFER command is intended merely to give enough information for the reader to understand the text of the system. Many details are therefore omitted. The command is used to initiate transfers to and from every peripheral device in the system. b is a vector containing the parameters of the transfer, and is described in the next paragraph. If the command is rejected, the reason for the rejection is placed in Result: the possible reasons are tabulated below. If the command is accepted, Result contains the value of a, which is not otherwise used.

The format of the parameter vector (conventionally referred to in the text as E) is shown in Fig.26.

E↓DEVICE specifies which device is to be accessed (a list of devices together with special comments appertaining to each is given below).

E↓BUFFER and E↓BUFFSIZE specify the buffer which is to be used for the transfer.

For transfers involving the disc, E↓PAGENUMB gives the number of the page on the disc at which the transfer is to start. For other devices, the element is known as E↓STOPCH: the transfer will be terminated if a character equal to this element is transferred. In order that a program controlling an input transfer may know why the transfer terminated, the final word of the buffer is reserved for this purpose: if the transfer ends because the buffer is full, the last word is set to true; otherwise (if the character specified by E↓STOPCH is detected), the final word is untouched (TryDiadAgain, II:5.5, uses this facility).

E↓SEG specifies whether the buffer is in the code or the data segment and also, if the transfer is of characters, whether they are to be packed two to a word. (We do not use the character-packing facility, and the only time we transfer from the code segment is when the entire segment is dumped for post-mortem purposes - see DumpSegment, II:2.1.) Note that when specifying a transfer involving the code segment, E↓BUFFER must contain the absolute hardware address of the

buffer within the segment: our origin of co-ordinates for addressing this segment is offset by a small value (OFFSETC) from the beginning of the segment.

E↓ENDMODE specifies the action that is to be taken when the transfer finishes. If it has the value QUIETEND, all that happens is that the value of E↓COMPLETED is replaced by its logical complement. If the value of E↓ENDMODE is INTERRUPT, however, when the transfer ends the system is interrupted (see Interrupt, II:2.4).

E↓COMPLETED is used only to indicate the completion of the transfer when E↓ENDMODE equals QUIETEND.

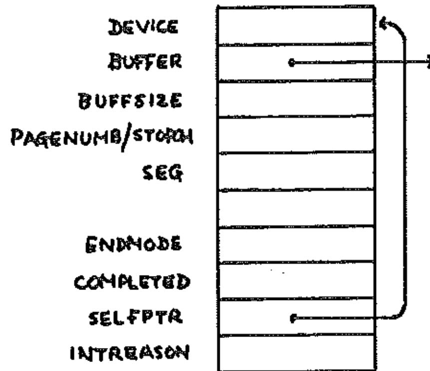



Fig.26 Parameter block for Exec TRANSFER command

E↓SELFPTR always contains E itself: this pointer is required by the interpreter.

E↓INTREASON contains the value which will be given as the reason for interrupt if E↓ENDMODE specifies that the system should be interrupted at the end of the transfer.

The following devices may be accessed by the TRANSFER command:

<u>The executive teletype</u>

This is regarded as two separate devices, TTWRITE and TTREAD. The Executive also provides for a third device, for reading with automatic echoing, but we do not use this facility, as our echoing is organised by NextTT (II:5.1).

<u>The paper tape reader</u>

This is also regarded as two devices. The choice between them depends on the direction in which the tape is required to move. We normally use READERLEFTTORIGHT.

<u>The paper tape punch</u>

<u>The lineprinter</u>

The disc

This again is regarded as two devices, DISCREAD and DISCWRITE. The least significant eight bits of E↓BUFFER and E↓BUFFSIZE are ignored, so that the buffer begins at a page boundary and occupies an integral number of pages. With this device it is possible that though a transfer may be initiated successfully, a hardware failure (such as a parity failure) may occur while it is taking place. If this occurs, the Executive overwrites E↓PAGENUMB with a value giving the cause of the failure, and proceeds as though the transfer had terminated successfully.

The clock

Four devices are available as independent clocks, though we use only one, called CLOCK. The Executive's internal clock ticks with a fixed period, half a second in our system. The buffer for commands involving the clock is always two words. The contents of both words are preset by the program. The first word is decremented at regular intervals; the period is specified by the second word, in units of the period of the Executive's internal clock. The transfer terminates when the first word becomes zero.

Remote consoles

Four devices are associated with each remote console; three of them are like the three devices belonging to the executive teletype (READ, WRITE and READECHO); the fourth, called the XON device, has a rather different purpose. A transfer from this device terminates when an XON character is typed at the remote console, regardless of whatever other transfers involving that console may be in progress at the time. The buffer is not used (so any valid buffer may be specified). We use these devices, with E↓ENDMODE taking the value INTERRUPT, to enable users to interrupt the system from a remote console.

CANCEL

This command may be used to stop any transfer in progress on any particular device, specified by parameter b. The command is rejected if the device does not exist; otherwise it is accepted, whether or not any transfer was in progress on the device.

LOOKATRDR

This commands Executive to ask the paper tape reader its current status. The command is rejected if the reader itself

rejects the request (e.g. becuse it is switched off or off-line). If the command is accepted, the value of Result gives the reader's status (e.g. whether it is loaded with a paper tape). This command is used by ReaderOffLine (II:5.4).

READABS

This command may be used to obtain the content of any word in the Modular One core, given its absolute address. This is one of the commands, mentioned above, which can never be rejected, so that it always ends with a jump to RejLab. The address is given as the parameter and the command places the contents in Result. This command is used by the function FetchExecWord (II:2.1).

We now come to the seven functions and routines which are implemented in 'hardware' merely for efficiency's sake.

Sumcheck (global 15)

This function computes the sumcheck value for the information in the vector DiscPage (II:7.1). In the equivalent BCPL text (lines 41-45), the function AddwithEAC performs addition with 'end-around-carry'. In fact this operation, which ensures that each bit in a word is checked equally effectively, is implemented using the hardware overflow register; however, a possible BCPL definition (assuming 2's complement arithmetic) is also given (line 47).

Next (global 17)

This is the principal primitive function operating on input streams. It is described in [2] §2.1, §2.5.1 and §2.6. This commentary refers to the equivalent BCPL text.

Line 53: For a slow stream the effect is exactly as described in [2] §2.5.1.

54: S now points to the fast block.

55: If the buffer is empty, the NEXT element of the slow block is invoked to refill it, and incidentally to provide the result of Next.

56, 57: If the next character in the buffer is the escape character, the function in the TRAP element of the slow block

is invoked to deal with the situation. This test may be inhibited by setting the INESC element of the fast block to equal NOTUSED.

59-61: Otherwise, the next character is obtained from the buffer and the pointer is incremented.

At present, NOTUSED has the value 0; so this instruction cannot be used, for example, to detect blank tape as the escape condition. This is a mistake, but we have not yet corrected it, as we have further plans for the escape facility (see the last paragraph of [2] §2.6).

Out (global 16)

This, the principal primitive routine operating on output streams, is also described in [2] §2.1, §2.5.1 and §2.6.

Line 69: The standard definition for slow streams.
71: S is now the fast block.
73: If the buffer is already full it is output, using the appropriate routine in the slow block. This case should never arise.
75, 76: The object being output is placed in the buffer and the pointer incremented.
78: If the buffer is now full it is output.
79-81: If the character, just output is the escape character the buffer is output. As in the case of 'Next', this facility can be inhibited.

Endof (global 20)

This boolean function tests for the end of the information in an input stream. See [2] §2.4.1 and §2.6.

Line 92: The standard definition for a slow stream.
93: S is now the fast block.
94, 95: If the buffer is empty the corresponding function in the slow block is invoked to test for further information.
96-98: The function in the slow block is also invoked if the next character is the escape character, provided that the escape facility is not inhibited.
99: Otherwise, there is more information in the buffer and the result is false.

<u>TransferIn</u> (<u>global</u> 13)

This is a routine for filling a vector from an input stream.

<u>TransferInC</u> (<u>global</u> 14)

This is a similar routine for filling a vector in the code segment from an input stream. It is used principally by the loader.

(Note that both these routines need two IC instructions. This is because the call Next[S] requires the execution of BCPL program, and this cannot be invoked halfway through a machine instruction. When the stack is set up for the call of Next, the link is set so that control each time subsequently re-enters the second instruction of the pair.)

<u>TransferOut</u> (<u>global</u> 7)

A routine to output the contents of a vector to an output stream.

(This routine does not require two IC instructions, as the excursion into BCPL program occurs at the end of each cycle round the loop. The link on the stack is set so that the instruction is re-entered each time.)

# III: SET-UP PROGRAMS

## III:1. System set-up

### III:1.1 SYSSETUP

This part contains the programs which initialise the system when it is first read into the core.

The segments of the operating system, made up as shown in Part V, are compiled in the normal way into IC files. The system is constructed by a special version of the loader, which loads the segments into a core image on the disc. The six setting-up segments in this Part are also loaded, but only temporarily: that is, the various pointers (such as CPtr, II:1.3) are not adjusted to take account of the presence of these temporary segments, and they will be eventually overwritten by programs loaded by the system. Global variables required by these temporary segments are placed in the part of the global vector normally available to user programs, and they are cleared when the system has been initialised.

The core image may or may not be (but usually is) converted into a more compact form on the disc. Bootstrap programs (about a foot of paper tape, read directly into the core by special-purpose hardware) cause these core images to be read into the core. The second pair of words in the code segment (which normally contain a jump to Interrupt, II:2.4) are set up in the core image to contain a jump to the routine PInterrupt, which is therefore obeyed as soon as the machine is switched on after reading the system into core.

FSLim (temporarily global 412)

This variable is initialised in the core image to contain the upper bound of the free store area in the data segment. This of course implies that the size of the free store is specified when the system is constructed in the core image.

CPages (temporarily global 416)
DPages (temporarily global 417)

These two variables are initialised in the core image to contain the size (in pages) of the two segments.

PInterrupt (temporarily global 411)

As stated above, this routine is the first to be obeyed after the system is read into core. It performs all the initialisation required for the system, before finally entering the load-go loop.

Line 27: The software lock preventing writing to the disc is set, so that if this initialisation process is interrupted no overwriting of the filing system is permitted unless the filing system routines in core have been properly initialised.

30-38: The action of these routines will be described under their separate headings, but the order in which they are called is important:

30: many of this sequence of routines claim free storage, so the free store system must be set up first;

31, 32: at one stage in the system's history the disc occasionally used to evince a hardware failure causing it to overwrite the page in core after DiscPage; so it is arranged that DiscPage is followed by the private stacks, the overwriting of which is least likely to prove catastrophic (see the descriptions of these two routines, below, for further trickery connected with this positioning);

35, 36: the clock must be set up before the time is obtained;

37: this must be the last of the routines to claim free store, as it computes the amount of store to be made available for the Running of LoadGoLoop;

38: This routine sets a link to LoadGoLoop, so it is not called until an activation of LoadGoLoop would be meaningful.

40: Though when modifying the system this is easy to forget, it must remain possible to initialise the system even if the filing system on the disc is unusable; otherwise a hardware disc crash would kill the system for ever. The core image of this sytem is archived (on paper tape) and special bootstrap arrangements, not using this system, enable the core image to be established onto an otherwise empty disc and thence to be read into core as usual. It would be possible to archive a special version of the system, designed to deal with the special circumstance of the non-availability of the filing system. However, when the normal system was modified, this

special system would be liable to be forgotten, and so rapidly to become incompatible. We therefore make provision for the special case when setting up the normal system. (Note that since the setting-up code is only temporary, concern for the special case does not waste permanent core space.)

The operator specifies that the filing system is not available simply by switching the disc controller off after the system has been read into core. The system tests for this condition, using the result to set the flag DiscUsable. Note that since special action has to be taken by the operator in the special case, in normal circumstances he does not have even to consider the matter.

42: If the disc is available a quick validity check is performed on the free store file. Since a page of this file is written back from the core fairly frequently, it is possible that it may have become garbled after an erroneous program had corrupted the core. If this is detected the function QuickValFSF says so, and the flag DiscUsable is unset.

43: If the disc is off the system says so.

45: The REASONFORINTERRUPT word is cleared (its previous value will of course have been POWERON).

46: The second pair of words in the code segment is converted to a jump to the normal Interrupt routine (II:2.3).

48: The temporary globals are cleared, which means that the remainder of this routine (lines 50-60) must not use any temporary globals.

51-54: If the filing system is available it is initialised, the user is asked to log in, and the software lock on writing to the disc is removed.

55-58: If the filing system is not available, the address (held in core) of the last page of the free store file is set to an invalid value, so that any attempt to write the free store file page back to the disc will lead to failure (note that this will only arise if the operator has explicitly overwritten the software lock DISCWRITEPERMITTED). Then the user 'System' is forcibly logged in and the operator is warned of the abnormal conditions.

60: Using the link set by SetStackBase this command enters the load-go loop, and the system begins its normal activity.

SetUpFS

When the system is loaded into the core image, the free storage required is allocated consecutively in the data segment

(after the global vector). The global FS is initialised in the core image to point to the first word of the space which is left. This routine sets up this space, of which the beginning is specified by FS and the end by FSLim (see above), as a proper free store as described in II:3.1.

Line 66: f will be the start of the new area: space is reserved for the FS block itself.
68-71: The FS block is initialised.
73, 74: The remaining space is initialised as the only element of a free block chain. NewVec, ReturnVec etc. (II:3.1) are now available for normal use.

SetUpDiscPage

As discussed above (II:7.1), the hardware required for the vector DiscPage begins on a hardware page boundary; that is, the address of its first word is a multiple of 256. This routine obtains a vector fulfilling this condition; it also sets up a parameter block for the Exec commands which perform disc transfers.

Lines 80, 81: v and Endv are the two bounds of a vector which certainly contains a complete hardware page.
83: D points to the start of the first hardware page beginning in the vector v.
87: The parameter block is in a fixed position relative to the start of DiscPage, so that DiscTransfer (II:7.1) can find out where it is. At present it immediately precedes DiscPage, so it is necessary to ensure that there is room for it.
92-96: These lines deal with the remaining parts of the vector v after the part required for DiscPage and the parameter block has been reserved. The part after DiscPage can be returned immediately (line 96) and will be concatenated with the rest of the available space. However, since we wish to ensure that the PM stacks come immediately after DiscPage (see PInterrupt, above), we do not return the free space before DiscPage yet, but leave its parameters in the static variables DeadArea and DALength. SetUpPMStacks will return this area after it has claimed its own space.
98-106: The parameter block is initialised (see the description of the Exec TRANSFER command, II:9).

SetUpPMStacks

This routine sets up PrivateStack and the first GiveUpStack (see II:2.4 and II:2.1). As GiveUpStack is more likely to overflow because of programmer error it is placed first, since overwriting PrivateStack will not cause much trouble.

Line 118: When the private stacks have been claimed, immediately after DiscPage, the free space before DiscPage (see SetUpDiscPage, above) can be returned to the free store system.

SetUpSund ryItems

This routine initialises various things that are not dealt with elsewhere.

Lines 125, 126: The two chains are set empty.
128: This is the vector for the free store file page kept in core.
129, 130: These are the two 'handles' to the disc filing system.
134: This ensures that the temporary setting-up code is cast completely into limbo.
136-140: The two code segment words containing the maximum addresses in the two segments are initialised.
136: OFFSETC is a consequence of the fact that the code segment word with address 0 is not the first word of the segment (see the code segment address constants in IV:1.3). The -1 indicates that there is one page of the segment beyond the maximum address. This page is used as private working space by Computer Technology's communications multiplexer hardware. It is occasionally useful to be able to access this space, which is why it is not outside the segment altogether. But ensuring that it is above the maximum address prevents the loader from overwriting it with code.

SetUpTimeofDayClock

This routine sets up the data structure (parameter block and buffer) required for the clock which keeps the time of day. The format of a clock is given in the description of the Exec routine (II:9).

Line 162: It is for historical reasons that the pointer to the parameter block is kept in the code segment while the block itself is in the data segment.

SetUpRunBlock

This routine initialises the first of the chain of Run-blocks (see Run II:1.2 and [1] §1.2).

Line 172: A new free store is set up.
173: The FS block's predecessor is itself. So this is the 'primaeval' free store (see ReturnVec, II:3.1). All this ensures that there is no possibility of returning this Run-block or the other permanent system vectors to the free store.
175: This is the loop at the end of the Run-block chain (see [1] §1.2.3, footnote).

SetStackBase

This routine sets the link at the start of the stack, which is immediately after the free store.

Line 196: A loop is created at the base of the stack.
197: The link is set to the start of LoadGoLoop. This link is used at the completion of the initiation of the system, and also if ever by chance an interruption occurs during the very brief periods when the system is at the outer level of Run.

CheckDiscOn

This boolean function determines whether or not the disc is on-line, which is the test for whether the filing system is available. The function attempts to start a transfer from a non-existent page on the disc. If the disc is on-line the attempt will be rejected because of the invalid page address; otherwise it will be rejected because the disc is off-line.

Line 210: The result of the test depends on the reason for the rejection.

ReleaseNonSystemGlobals

This routine clears the part of the global vector which is available for use by user programs, but which also contains the temporary globals used in the setting-up of the system. Leaving them in a standard state may simplify the analysis of failing user programs.

FSFtoCore

This routine initialises the core copy of the last page of the free store file.

Line 223: The last page is read into core.
224, 225: The extra words in FSF are initialised.

LogIn

This routine asks the first user to log in to the system. It uses the system file described in VIII:1.

III:1.2 SETDANDT

SetDateandTime (temporarily global 413)

This routine initialises the code segment word containing the date, and the time-of-day clock.

Line 14: This call of TimeofDay will cause the clock to be started and initialised.
17: This gets rid of any extra characters which may have been input in answer to Date's questions.

<u>Date</u>

This function asks the date from the console, to which the answer is given as three numbers (day, month, year). Since operators sometimes make mistakes about this, the function confirms the date by asking for the day of the week and checking that the replies agree.

<u>Line 26:</u> The year may be specified either in full or by the last two digits.
<u>28-31:</u> The year and the month are checked for validity. If they are wrong a recursive call of Date[] is used to try again.
<u>51, 52:</u> The day is checked for validity.
<u>54:</u> The day of the week is calculated.
<u>58:</u> The day of the week is obtained from the console.
<u>59, 60:</u> In the case of Sunday or Saturday and Tuesday or Thursday the day is not uniquely specified by the first letter,
<u>61:</u> but in the case of Tuesday or Sunday it is not uniquely specified by the second letter.
<u>64-67:</u> A check that the calculated day of the week agrees with that given.
<u>69:</u> The date is packed into a single word.

<u>Wrong</u>

A private routine used by Date to output a failure message.

<u>Leap</u>

Leap[Year] tests whether Year is a leap-year.

<u>NextLetter</u>

The result of this function, for which the parameter is a character input stream, is the next letter to come on the stream.

<u>Line 83:</u> If the letter is in lower-case the result is the corresponding capital.

## III:1.3 QUICKVAL

QuickValFSF (temporarily global 414)

This function performs a quick validity check on the free store file. It does not check whether every entry in the file is in fact free, though this is regularly done by the separate Disc Validate program.

Lines 10-12: Failure if there is no free store file.
17-42: Each cycle of this loop (§R) checks one page of the free store file.
17-20: The specified page is not within the filing system.
22: The specified page is brought into core.
23-28: The housekeeping words are checked.
31-33: A check that each entry in the page is either the constant CLAIMED or points to a page inside the filing system.
35-37: A report if any invalid entries were found.
40, 41: The test moves onto the next page.
46-49: A check that the housekeeping information in the heading agrees with that obtained by scanning the file.
52: The result is the value of the static variable Result, which is preset to true (line 4) and set false by any call of ReportMessage.

ReportMessage

A private routine called by QuickValFSF to output failure messages and to set the result of the whole test false.

## III:2. Set up streams

### III:2.1 SETUPSTR

SetUpStreams (temporarily global 415)

This routine sets up the permanent streams of the system and the initial values of the variable streams. The component functions and routines of these streams were defined elsewhere, in the permanent code of the system (II:5), but some of them were assigned temporary global numbers. The values of these latter functions and routines are copied into the stream vectors when the streams are initialised, and they are not afterwards accessed through the global variables.

13-14: The variable output streams are initially set to ExecConsole.
17: The variable input stream is initially set to DiadRead (for reading programs from paper tape).

Parity (global 67)

This is a vector used in the determination of the parity of paper tape codes. If $0 \leq i \leq$ VECSIZE, then Parity↓i is the parity (1 for odd, 0 for even) of the number of bits in the binary representation of i. At present VECSIZE is 15, so that i can be any four-bit integer. Thus the parity of an eight-bit character can be obtained by testing each half separately and combining the results.

SetUpParityTable

This routine sets up the vector Parity.
Line 23: Since the number of elements in the vector is the same as the number of bits in a word, the contents of the vector can be conveniently specified by the constant PARBITS.

Output (global 26)
ReportStream (global 28)
Console (global 27)

These variables are intended to hold streams to be used, respectively, for a program's normal output, for reports of errors and similar messages, and for dialogue with an operator. See [2] §2.7.

In (global 25)

This variable is intended to hold the normal input stream for a program. The loader (II:1.3) takes its input from here. See [2] §2.7.

## III:2.2 SETUPTT

SetUpExecConsole (temporarily global 409)

This routine sets up the streams which use the executive console: the raw byte stream Teletype, and the internal code stream ExecConsole.

Line 4: ExecConsole is to be a permanent stream, so its 'Close' routine is replaced by the 'Reset' routine.

TT

The result of this stream function is the stream of raw bytes to and from the executive teletype.

Lines 16-24: The conventional locations in the stream vector are initialised.
18: This is a permanent stream, so the 'Close' routine is the same as the 'Reset' routine.
22: This is a source stream, so this element points to the stream itself.
23, 24: These routines must be provided for source streams.
26-36: The parameter block is set up for read transfers from the teletype.

40-50: The parameter block for write transfers to the teletype (see the description of Exec routine, II:9).
52: The pointers for the input buffer are initialised.

The working of this stream is described in II:5.1.

III:2.3 SETUPRDR

SetUpReader (temporarily global 410)

This routine sets up the stream of bytes from the paper tape reader, BytesfromPT, and also DiadRead, the permanent stream of words from paper tape punched in diad format.

Line 7: ReaderDev is initialised, to specify the paper tape reader reading in the normal direction.
10, 11: Since DiadRead is to be a permanent stream the 'Close' routine in its slow block is set to the 'Reset' routine.

BFPT

This function sets up the reader stream BytesfromPT.
Lines 23-26: The fast block is initialised.
28: BytesfromPT is an input stream; but the element NEXT will be initialised by the call of ResetBFPT (line 55), as that routine has access to other non-global functions.
29-36: The conventional locations in the slow block are initialised.
30: This is a permanent stream, so the 'Close' routine is set to the 'Reset' routine.
33: This is a source stream, so this pointer points to the stream itself.
34, 35: These two routines are required by source streams, though for ResetState there is nothing to be done.
36: This element is used by InitiateTransfer (II:5.3).
38: The buffer pointers are initialised.
39, 40: As required by InitiateTransfer (II:5.3), the word before each buffer points to the other buffer.
44-53: The parameter block for the transfer is initialised (see the description of Exec, II:9).
55: This call completes the setting up of the initial conditions.
57: The value of a fast stream is the logical complement of the fast block vector.

# IV: DECLARATIONS

## IV:1 DECLARATIONS

This file is inserted (by a get directive - see the Appendix) at the head of every separate segment for compilation (see Part V), in order to insert the declarations of all the global variables and manifest constants which are of more than merely local relevance. It uses get directives itself, in order to split the complete list into three sublists.

### IV:1.1 GLOBALS

The globals declared in this section are those the users of the system are encouraged to use. Each of the globals declared in this section is individually described: the reader is referred to the index.

### IV:1.2 PRIVATE GLOBALS

This section contains the declarations of all the globals used by the system which are not defined in 'GLOBALS' (IV:1.1). The first part contains those permanent globals which are intended for use only by other system routines, though of course there is nothing to prevent anyone from using them if he so wishes. The second part declares temporary globals which are used only when the system is being set up and are then cleared. Some of the values in these globals, however, have permanent significance, as they are copied into stream vectors when the permanent streams are initialised.

Each of the globals declared in this section is individually described: the reader is referred to the index.

## IV:1.3 CONSTANTS

This section contains the definitions of all the manifest constants which are of relevance to more than one local part of the system.

Line 2: This constant is used in many contexts throughout the system.
3: This is conventionally used, e.g.
**let** x = UNDEFINED
when it is necessary to declare a variable before its initial value is known.
6-10: These constants depend on the particular machine.
8: The amount by which the origin of co-ordinates for addressing the code segment is displaced from the beginning of the segment.
9: A mask to obtain the less significant half of a word.
12-17: See Exec (II:9).
19-33: The elements of the parameter block for the Exec TRANSFER command (II:9).
35-41: See the description of the Exec TRANSFER command (II:9).
36: E↓STOPCH takes this value if the terminating character facility is not required.
44-52: These are the devices controlled by routines in the system itself (see Exec, II:9).
48: This is the sum of the values of the two devices referring to the paper tape reader, so that if one is known the other may be obtained by subtraction.
54-65: These are the addresses of the reserved words in the code segment, which occur between the beginning of the segment and the origin for addressing. Of the unlisted addresses, some are reserved for constants and working variables required by the interpreter, and the remainder are spare.
The first two of these reserved words (lines 55, 56) are accessed directly by the hardware; the rest are kept in the code segment for safety's sake. These reserved words are in effect global variables, and so we now proceed to describe them separately.

INTERRUPTINHIBITED

This is a boolean, normally false, which may be altered by software. When it is set true, any attempt to interrupt the

system will be held up until it is reset to false. This allows the system to contain 'non-interruptable' code. Interrupts due to storage bound violations or switching the system off cannot of course be inhibited: when such an interrupt occurs this flag is reset to false. (We have never in fact used this interrupt inhibition facility.)

REASONFORINTERRUPT

Whenever the system is interrupted, the value of this word is replaced by the hardware with one specifying the reason for the interruption (this is a design error: it would be better to 'or' the new value with the previous contents). See Interrupt (II:2.4). If the interruption is for reasons like storage bound violations, this word is left unchanged. Possible values for this word are given below (lines 71-75); other values may possibly occur when the system is interrupted at the end of a peripheral transfer, as specified by E↓INTREASON in the Exec TRANSFER command (II:9).

MAXD
MAXC

These words contain the maximum addresses of words in the data and code segments. See SetUpSundryItems (III:1.1), where they are initialised.

SUMCHINHIBITED

This is a boolean, normally false. If it is true, the sumcheck for information read from the disc is ignored; it is therefore a flag which may be specifically set by software when coping with a corrupt disc.

DISCWRITEPERMITTED

This is a boolean, normally true. When it is false, the system ignores any request to write information to the disc. It is sometimes useful to use this flag when running untested programs, and also when dealing with a corrupt filing system, as otherwise the system's automatic updating of the housekeeping

information might make matters worse.

MFLFIRSTPAGE

This contains the handle to the disc filing system; it is used whenever a file is accessed. See [2] §3.4 - note that by using this reserved word we have corrected the mistake mentioned in that section.

DATE

This word contains the date. It is initialised by SetDateandTime (III:1.2). Its format is given in Date (III:1.2), line 69.

RBLOCK

This points to the Run-block belonging to the current Run. See Run (II:1.2) and [1] §1.2.1.

TIMEOFDAYCLOCK

This points to the parameter block (suitable for Exec TRANSFER commands, II:9) describing the clock which tells the time of day. See TimeofDay (II:6.1).

We now continue with notes on the constant declarations.

Line 67: When the system is interrupted a jump occurs to the second pair of words in the code segment, which initially contains a jump to PInterrupt (III:1.1). The word INTERRUPTADDRESS is updated during system setup (PInterrupt, III:1.1), so that the words become a jump to Interrupt (II:2.4).

73-75: Values placed by the hardware in the REASONFORINTERRUPT word (line 56).

78-86: Constants concerning the clock.

81, 82: The elements of the clock buffer.

84: In units of the period of the Executive's internal clock (half seconds).

89-105: The size of the Run-block and the names of its elements.
109-117: The size of the information block and the names of its elements.
119: The values of the code and data pointers when no code or data block exists.
120: The value of the ISUC element for the last information block.
125-133: The size of the free store block and the names of its elements.
135, 136: The housekeeping words in a free block of free store (see 'FREESTORE', II:3.1).
140: The value of the NXTB word at the end of the free block chain, etc.
141: The result of MaxVecSize when there is no free store.
147-149: The components of an element of the ClearUpChain.
151: The value of CSUC at the end of the chain.
157-165: The elements reserved for standard items in stream vectors.
160: This element is reserved for the parameter stream of a stream function call, except for source streams (which have none) and bilateral streams (in which care must be taken to ensure that output to the stream still works when elements are PutBack to the input part, as the PutBack block is also held in this element).
167: The value returned as the state of a console stream when nothing has been typed since the last ResetState call.
171-175: InitiateTransfer expects these elements to be used.
180-188: The size of a fast stream block and the names of its elements.
191: This value, when assigned to both a buffer pointer and the pointer to the corresponding buffer end, indicates that the buffer is empty.
192: The value for FB↓INESC or FB↓OUTESC which indicates that the test is not to be applied (see Next and Out, II:9).
196-206: The size of the teletype stream vector and the names of some of the elements. See SetUpExecConsole (III:2.2) and TELETYPE (II:5.1).
209-220: Constants for the reader stream BytesfromPT. Note that the size of the stream vector depends on the size of the buffers. See SetUpReader (III:2.3) and READER (II:5.4).
225-237: The size of a PutBack block and the names of its elements. See PutBack (II:3.2).
240: A mask to remove the underline bit from an internal code character.
241: A mask to convert an internal code letter to a non-underlined upper case letter.

243-250: Various internal code characters which cannot be expressed as quoted characters on this output device.
248: This character could be quoted, but it is given explicitly as otherwise it is easy to confuse it with ACUTE.
244, 245: Constants to remove and insert the parity bit for teletype code characters.
257-272: Characters in teletype code.
277: The word which contains the length of a private stack.
282-288: Particular words in a file body page.
287: This only applies to the last page of 'streamable' files.
290: The number of words occupied by data.
291: The value of the first word of a free page.
292: The value of the NXTPAGE element of the last page of a file.
298-307: The fields of a heading.
309: The parameter value for UpdateHead (II:8.3) for those fields which are not to be changed.
310: The default initial value of a date field.
311: The value of the FIRSTPAGE and LASTPAGE fields for a file with no body.
317-329: The fields of index entries.
334: This value in the type field of a file heading indicates that the file has been deleted.
336: This bit in the value of a file type indicates that streams may validly be formed to or from the file.
339-343: Values for the permission field in a file heading.
347-355: These are all concerned with entries in the MFL.
348-351: The size of each entry and the names of its elements.
353: This value in the first word of an entry indicates that the entry is vacant.
354: The number of entries in each page of the MFL.
362-373: These constants are all concerned with the disc storage allocation system.
363: The size of the vector in core.
365: The extra word of housekeeping information in this file (see II:7.2).
369, 370: The extra elements in the FreeStoreFile vector, apart from the copy of the file page.
372: The value placed in empty words of the FreeStoreFile.
376-379: The size of the InfromFile stream vector and the names of some of the elements.
381: The end of stream character for this stream.
382: The total amount of free storage required by this stream (this constant is used by LoadFile, II:8.6).
385-398: The size of the OuttoFile stream vector and the names of some of its elements.

## V: SEGMENTATION OF THE SYSTEM FOR COMPILATION

Each of the little blocks in this part is a complete segment for compilation. The individual files are concatenated using get directives. Each segment contains the files from one chapter of Part II or Part III, together with the declarations in Part IV.

### OS/1 to OS/8

These correspond to II:1 to II:8.

### OS/SU and OS/SUS

These correspond to III:1 and III:2.

## VI: LIST OF FAILURE REPORTS

These failure reports are numbered according to where they occur in the system. The more significant two digits of the report number give the section of Part II in which the calling code occurs, and the least significant digit gives the place in that section. For example, OSReport 312 is called by NewVec (when the free store runs out); NewVec is defined in II:3.1, in 'FREESTORE', and this report is the second occurring in that section.

OSReport

| | |
|---|---|
| 111 | global 1 not set |
| 121 | Terminate Run wrong (probably RunBlock overwritten); GiveUp[address of RunBlock] |
| 131 | In Load, no Zero after NEWSECTION |
| 132 | no Zero after ENDLOAD |
| 133 | Warning Character not TITLE, NEWSECTION or ENDLOAD. GiveUp shows value |
| 134 | In LoadSection, too much code loaded; GiveUp[InformationBlock] |
| 135 | more than one DATA block in section |
| 136 | In Unload, request to unload code loaded outside current area |
| 141 | In SetLabels, BINARY warning character expected |
| 142 | label setting instructions for Data, but no DBlock |
| 311 | In NewVec, parameter is negative |
| 312 | insufficient free store |
| 313 | In ReturnVec, second parameter is negative |
| 314 | vector to be returned overlaps current free store area |
| 315 | chain of FS blocks wrong |
| 316 | vector outside free store |
| 317 | free word inside vector |
| 318 | vector overlaps free block |
| 319 | In RestoreFreeStore, parameter not valid FSBlock |

321 In NextPB, PutBackBlock not found on PBChain

411 StreamError: usually outputting to input stream or
inputting from output stream

421 In OutString, string ends with '*'
422 too many parameters for OutS, etc.
423 invalid character following '*'
424 string ends with '*I'
425 character following '*I' not digit

441 In NextN or NextO, end of stream encountered

521 In NextTele, odd parity found

541 In TryAgain, parameter stream not BytesfromPT
542 parameter stream has something PutBack

551 In NextBlock, Diad WCH expected
552 Diad block too big (>30)
553 Diad sumcheck failed
554 In TryDiadAgain, parameter stream not BytesfromPT

621 AddressZero: probably global routine or function not set

721 In NewDiscBlock, no free pages left on FSF

731 In InfromFile or OuttoFile, file deleted
732 file type wrong for streaming
733 In EntriesfromFile, file not index

741 In ReturnChain, page chain wrong
742 In OuttoFile or DeleteBody, User does not have permission
to write to this file
743 In DeleteBody, heading found deleted

```
821   In Update, heading of file deleted
822              element to be updated not inside heading
823              end of heading file reached

831   In UpdateHead, end of heading file reached

841   In FindHeading, no entry in MFL
842                   end of heading file reached

851   In LookUp, index deleted
852              file not index

861   In LoadSystemFile, file not in SystemIndex
```

# VII: THE SYSTEM INDEX

SystemIndex (global 303)

This global variable holds the system index for the current system. It may be used both by system routines (such as LookUp, II:8.5, and LoadSystemFile, II:8.6) and also by user programs.

In the remainder of this part we list the entries in the system index, with brief descriptions of the purpose of the various files.

## Files with second name 'IC'

Almost all these files contain library programs or routines. It is not the complete available repertoire, as other programs which are mainly the concern of one user, but are nevertheless available for more general use, appear in their authors' personal indexes. An asterisk attached to a name indicates that the file will be described in detail in Part VIII.

'LogIn'*
This program is used to set the system variables User and CurrentIndex to indicate which user is using the system. This program is used by LogIn (II:1.2, and also III:1.1). See VIII:1.

'MakeNewFile'*
'Index Ops'*
'File Vectors'*
These three files contain various filing system routines which are not used often enough to earn them a place in the system itself. See VIII:2 to VIII:4.

'DiscRts'*
A file of private routines used by the previous three files. See VIII:5.

'Unchecked Disc'
This file contains versions of CoretoDisc and DisctoCore which differ from the normal versions (II:7.1) in that no mention is made of sumchecks: all the words in the page are treated as information words. These versions are primarily used for dealing with core images.

'DISC VALIDATE'
This program, loaded and driven by a small steering program on

paper tape, is the program which thoroughly checks all the housekeeping information in the disc filing system.

'DAISY'

This program, also driven by a paper tape steering program, is the disc housekeeper and garbage collection program (it is called DAISY because that seemed an appropriate name for a housekeeper).

'BCPLComp'

This contains the steering routine which drives the BCPL compiler. The other bricks of the compiler are to be found in the compiler's own index.

'Edit Routines'

A simple line-numbered editor.

'EC1'
'EC2'

('Edit and Compile') These files contain various routines of a system which edits text files stored on the disc, and compiles the edited versions, storing the binary on the disc. The system is driven by a steering program on paper tape, and the name of the text and binary files, and the editing commands, are also supplied on paper tape. The program uses 'Edit Routines' and the BCPL compiler steering program 'BCPLComp'.

'BytestoPT'

BytestoPT[] is a stream of 8-bit bytes to the paper tape punch, and is analogous to BytesfromPT (II:5.4). (Note the asymmetry: BytesfromPT is a permanent stream, BytestoPT is a library stream function.)

'LinePrinter'*

LinePrinter[] is an output stream of internal code characters to the line printer.

'TERMINALS'

Terminal[n] is a bilateral internal code console stream to the remote console numbered n in our configuration.

'IntcodefromSeven'
'IntcodefromTerminal'

These define input stream functions to produce internal code streams from streams of bytes from the flexowriter or a remote terminal respectively. Most frequently they are applied to BytesfromPT to read tape prepared off-line on one of these devices, but the stream Terminal mentioned above, for example, also uses IntcodefromTerminal.

'IntcodefromOlivetti'

This is a similar input stream function to convert a byte stream from tape punched on some Olivetti terminals with character sets which differ from our normal set. (The

machines have since been converted, but some old tapes are still in existence.)

'Find Input'

The result of the parameterless function defined in this file is an internal code character input stream from the paper tape reader. The function examines the first byte of information on the paper tape and tries to decide on which device the tape was punched; if it gets stuck it asks the operator on Console. It then loads the appropriate stream function file and creates the stream.

'IntcodetoSeven'

'IntcodetoTerminal'

These define output stream functions which produce internal code output streams from streams of bytes, usually to the paper tape punch. They are analogous to IntcodefromSeven and IntcodefromTerminal respectively.

'MaptoSevenhole'

'MaptoTerminal'

These both define output stream functions; their arguments and results are both internal code output streams. They are used specifically when BCPL text is to be output to a particular device, and their job is to replace the non-printable characters for that device with others which are synonymous in BCPL. For example, MaptoSevenhole would replace a question-mark, which cannot be printed on the flexowriter, by *q. The object is to ensure that programs can be output and re-input satisfactorily, so that they then compile to the same code.

'WordstoDiads'

This defines the output stream function analogous to WordsfromDiads (II:5.5) on input. The argument stream is an output stream of bytes, usually BytestoPT[], and the result is a word output stream. (Diad format is described in II:5.5).

'Legible'

This defines the output stream function IntcodetoLegible which converts a byte stream, destined for the paper tape punch, to an internal code stream. The internal code characters are output in legible form on the paper tape. This stream may be used for labelling paper tape output. (When the stream is set up it reads the byte patterns for the various characters from the file 'LegibleCodes' 'Binary', mentioned below.)

'LineNumbering'

This contains the definitions of several internal code stream functions.

AddLineNumbers is an output stream function which causes the

stream to count the lines being output and prefix each line with a line number.

Paginate, also an output stream function, splits up text being output into pages of convenient length.

RemoveLineNumbers is an input stream function, which causes the stream to ignore the line number at the beginning of each line.

'LinePrinter Line Numbering'

This contains a different version of the AddLineNumbers intended for output to the line printer, where the possibility of re-input need not be considered, so that a more elegant appearance of the line numbering can be contrived. For example, only every fifth line is line numbered: the program text in this publication was numbered using this routine. Also, the pagination of the text is performed automatically by the printer hardware.

'NextS'

This contains the definiton of a function, analogous to NextN and NextO (II:4.4), which reads the next quoted string from its parameter input stream.

'Layout'

This defines a set of routines which provide finer control over the layout of material within a line being output. Particular items may be left- or right-justified or centred in specified fields on the line.

'BCPLS1'

This file contains a few library routines, used by the BPCL compiler, but also available for general use.

'Line Shortener'

This file contains a program now little used, which splits up lines of BCPL text originally punched on a device with a rather wide carriage (e.g. the flexowriter) so that it may be output on smaller devices. The program attempts to choose the best position to break the line.

'BCPLProg'

This is a work file into which the BCPL compiler outputs its compiled programs.

<u>Files with second name 'Index'</u>

'System'
'OSPubSystem'
'OS7System'

More than one version of the operating system are usually in existence at any one time. Since these may require different versions of some of the library files, each version has its own system index. Each system index is entered in itself as 'System' 'Index', but in addition each system index is entered in all of them under a more specific name.

'Compiler'

This index contains the bricks of the BCPL compiler.

'Constructor'

This contains a set of programs for manipulating the text and binary files of the operating system itself, and for constructing operating systems as core images on the disc (see System set-up, III:1).

'POST MORTEM'

This contains a set of programs for analysing a core image dumped on the disc by StandardGiveUp (II:2.1). Various useful items of information may be output to the line printer.

'ESTAB'

This contains the File Establisher, a program which transfers information (supplied on paper tape) to files on the disc. It also has many other facilities.

'Console'

This contains an experimental single-user console system, with a context editor and facilities for compiling and executing programs. The console command language is BCPL.

'Offline'

This contains an experimental version of the operating system, in which the LoadGoLoop takes programs from a queue on the disc. It is intended to function in the complete absence of any operator.

'OS'

This is the root of an over-complicated hierarchy of indexes which contain the text and binary of all the versions of the operating system.

The remainder of the files with second name 'Index' are those belonging to the users of the system (e.g. 'CS' 'Index', 'Julia' 'Index'). There is also one index ('45' 'Index') for use by people who have not yet become official users of the system; the body of this index is liable to be deleted from time to time.

Other files

'Headings' 'SYS'

This is the file containing the headings of all the files in the system.

'MFL' 'SYS'

This file contains the master file list.

'FSF' 'SYS'

This is the free store file and contains a list of the free pages on the disc.

'UserCodes' 'SYS'

This file contains details for each user of the system, such as the names of his index. It is used by the LogIn program (VIII:1).

'BCPL Textfile' 'Text'
'BCPLWork' 'Gen'
'BCPLNames' 'Work'

These are work files used by the BCPL compiler.

'LegibleCodes' 'Binary'

This contains the private information used by the stream function defined in 'Legible' 'IC' above.

'GLOBALS1' 'Text'
'GLOBALS2' 'Text'

These files contain selections of global declarations which programmers may cause to be inserted at the top of their programs to declare the more important system routines.

## VIII: SOME LIBRARY FILES

### VIII:1 LogIn

This program sets the system variables User and CurrentIndex to indicate which user is using the system. It is called by LogIn (II:1.2, and also III:1.1). It asks the user for his name on the console, and finds his entry in the file 'UserCodes' 'SYS'. The format of the entries of this file is shown in Fig.27 and the names of its elements are defined in lines 3-9.

<u>User</u> (<u>global</u> 310)

When someone is logged in to the system, this variable contains the unique value, associated with him in the UserCodes file, by which he is known to the system. It is used, for example, in checking whether the user is allowed to overwrite a particular file (see CheckPerm, II:7.4).

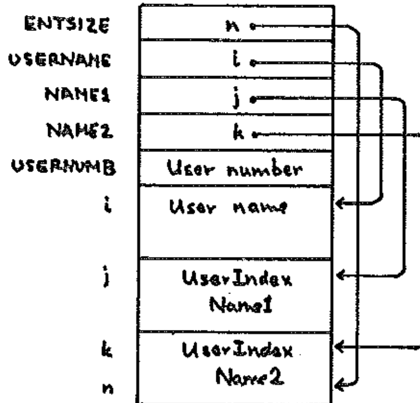


Fig. 27 Entry in UserCodes file

<u>CurrentIndex</u> (<u>global</u> 300)

This variable holds the index file in which the user is currently working. It is set, when a user logs in, to his own index as specified in the UserCodes file, but he may subsequently change it, for example to a sub-index. Certain facilities (for example, the <u>get</u> directive in the BCPL compiler: see the Appendix) involve searches for files in this index, unless they are instructed otherwise.

Prog (global 1)

This is the steering program.

Line 18: Name is a vector (size MAXSTRINGSIZE) containing as a BCPL string the reply typed on the console.
19: v is the entry in the UserCodes file corresponding to Name.
22-25: If no entry was found the program recurses to have another try.
26, 27: The variables are set.
28: The entry vector is returned.

LookUpUser

This function has one parameter, a string, which is the name of a user. Its result is a vector containing the corresponding entry in the file 'UserCodes' 'SYS'.

Line 34: The entries are obtained one by one using the stream function EntriesfromFile. At present this means that the type of the UserCodes file has to be INDEX, which is not ideal.

String

This function reads a string from the parameter input stream S. The string is assumed not to be quoted, and is terminated by a new line (not part of the string): it therefore differs from the library function NextS, which searches for a quoted string. The result is a vector, always of length MAXSTRINGSIZE, containing the string.

Line 50: The vector is initialised to contain a string of zero length.
53-56: This part of the program assumes that the characters are packed two to a word: i.e. it is implementation-dependent.
54: A character is inserted in the less significant half of a word.
56: A character is placed in the more significant half of a word.
59: The final length of the string is placed at the top.

## VIII:2 MakeNewFile

This program contains the filing system routines for creating and deleting files, and for altering the 'permission' of a file which specifies who is allowed to write to it.

Lines 10-13: These two routines, also used by other library files, are defined in the file 'DiscRts'. This file is loaded by MakeNewFile if necessary.
17-19: The size of the vector returned as result by NewLocation and the names of its elements.
20: The value left by the system constructor in unused global variables.
21: The element of a file heading where a title begins. The title is usually accessed indirectly via the first word of the heading (see Fig.24, II:8.2), but its position must of course be specified explicitly when the heading is created.
25: The maximum size of a heading, assuming the title is of maximum length.

### MakeNewFile (global 328)

See [2] §3.1.3. The result of this function is a newly created empty file with title and type as specified by the two parameters.

Lines 36-40: The three static variables are initialised if this has not already been done.
41: The file 'DiscRts' is loaded if not already present.
42: A disc address is obtained for the new heading.
43: The heading address is entered in the MFL. File is the position of this new entry.
44: A new heading, with no body attached, is constructed at the specified disc address. Note that the owner of the file is the current user of the system.
46: The value of a file is its position in the MFL.

### LoadDiscRtsifNec

This private routine loads the system file 'DiscRts' if it is not already present. To determine this, it tests the global variable corresponding to one of the routines defined in the file,

to see whether it is loaded. Note that 'nested globals' ([1] §1.1) means that this global will revert to its previous value UNLOADED when 'DiscRts' is unloaded.

NewMFLEntry

This function searches the MFL for a vacant entry and initialises it to the values given by the parameters. The result is the entry's position in the MFL.

Line 54: StartMFLScan specifies the first page in the MFL which might contain a vacant entry. It is initialised (line 39) to the first page of the MFL.
55-66: Each entry in the page is scanned. i is the position of the entry in the page; f is the entry's overall position in the MFL.
59, 60: If a vacant entry is found it is initialised.
61: The page with the new entry is written back to the disc.
67-71: If no vacant entry is found, the function moves onto the next page.
67-70: If no next page exists, it must be created and linked onto the current page.
72: The whole process is repeated until a vacant entry is found.

NewMFLPage

This function is called when a new page must be attached to the MFL. It claims a page and initialises its contents; it then updates the heading of the MFL to refer to it as the last page. The parameter is the serial number of the new page; the result is the page address. It is left to the calling routine (NewMFLEntry) to link the page to the previous page in the body.

CreateNewHead

This routine is used by MakeNewFile to construct a new heading at the address specified by the last parameter.

Line 88: The heading will first be constructed in this vector before being written to the disc.
89: The length of the title in words.
91-102: The fields of the heading are initialised.

TITLE+TitleLength is the overall length of the heading.

102: The title is copied into the heading vector at the appropriate place using hardware assistance (see Copy, II:6.2).

104: This private routine, defined in 'DiscRts', writes the vector to the disc.

UpdatePermission (global 313)

This routine may be used by the owner of a file to update the permission field of its heading, which specifies who is allowed to write to the file.

Line 109: A check that the user is allowed to alter the field.

DeleteFile (global 327)

This routine may be used by the owner of a file to delete it from the system.

Line 114: A check that the user is entitled to delete the file.

115, 116: The body if any is deleted, and the heading is updated to indicate that the file itself has been deleted. Headings of deleted files are eventually removed by the Disc Housekeeper.

CheckLegality

This routine checks that the file exists and that the current user of the system is its owner. If not it gives up.

## VIII:3 Index Ops

This file contains the routines for making and deleting entries in indexes. The format of entries in an index was shown in Fig.25 (II:8.5).

Lines 10-13: These two routines are defined in the file 'DiscRts', which is loaded if necessary by AddEntry and AddLinkedEntry.

17: The size of the vector returned as result by NewLocation.
18: The value left by the system constructor in unused global variables.

Enter (global 316)

This routine enters the file f in the index Ind under the names Name1, Name2. See [2] §3.2.

Lines 29-31: A validity check on f.
32: A check that the user is allowed to overwrite the index.
34, 35: If the two names are already entered in the index that entry is deleted.
37: The new entry is constructed.

AddEntry

It is this private routine, used by Enter, that actually enters the file in the index, after Enter has prepared the way.

Line 46: The file 'DiscRts' is loaded if not already present.
48: n is the length of the new entry.
49-56: The new entry is constructed in core.
58, 59: The disc address where the new entry will go is calculated and the entry is written there.

Link (global 315)

This routine makes a special entry in the index Ind under the names Name1, Name2 such that it refers to the entry under names Na, Nb in a second index, which is entered in SystemIndex under the names Nc, Nd. See [2] §3.2.1.

Line 67: A check that the user is allowed to overwrite the index.
68: A check that the proposed entry would not result in a loop of circular references.
70, 71: If an entry already exists under names Name1, Name2, then it is deleted.
73: The new entry is made.

CheckLinkDoesntLoop

This routine is used when setting up a linked entry, to ensure that the chain of links does not lead to a loop of references to each other. The parameters of the proposed entry are placed in static variables, and then the entry to which the new entry is to refer is looked up using Check, a special version of LookUp.

Check

This is a version of LookUp (II:8.5) called by CheckLinkDoesntLoop.

Lines 84-86: Before performing the look-up, the parameters are checked against the parameters for the proposed linked entry, previously stored in the static variables p, q and i. If all three match, then the entry would lead to a loop of references.

88-105: Except that the recursive calls (lines 94, 95) are to Check, this might be the definition of LookUp itself. In fact the version of LookUp in the system (II:8.5) is a somewhat optimised version of this, in order to reduce the time spent in the storage allocation system. Check, however, is not used sufficiently frequently to warrant such optimisation here.

AddLinkedEntry

This private routine used by Link actually makes the new entry in the index. It is very similar to AddEntry above.

Line 130: This sign reversal indicates that the entry is linked. Note that the element LINKING shares the same location as the element N3.

LoadDiscRtsifNec

See VIII:2.

CheckPermission

This routine checks whether the user is allowed to write to the parameter file.

Size

A private function to calculate the size in words of a BCPL string.

DeleteEntry (global 314)

This routine removes a specified entry from an index. In the past this was achieved merely by overwriting one of the fields in the entry with the value DELETED, leaving the disc housekeeper eventually to remove the entry completely. However, it has proved preferable in practice to remove the entry at once, by copying the rest of the index back onto itself: this process can now be given hardware assistance by using fast streams and TransferOut ([2] §2.6). Since earlier versions of the system might still be in use, the routines which scan indexes (LookUp (II:8.5) and EntriesfromFile (II:7.3)) must still allow for the possibility of the older form of deleted entry (Fig.25, II:8.5).

Lines 160, 161: An output stream to a file does not overwrite the file until the stream is closed, so both input and output streams may be used together.
163-170: Each entry in the index is scanned.
166: If the entry is the older form of deleted entry it is ignored. This check is necessary because the elements STATUS and N1 share the same field of an entry, so that if this field had been overwritten by DELETED one of the parameters for EqS (line 167) would be undefined.
167, 168: The entry is copied back to the index, unless it is the one which is to be deleted.
171, 172: The output stream is closed first because it is the more important.

## VIII:4 File Vectors

This file defines three routines for transferring information between files and vectors in core. Two of these, VectorfromFile and VectortoFile, are described in [2] §3.1.2; the other, AddMoreVectoFile, appends the contents of a vector to a file. All these routines assume the convention, for a core vector v, that v↓0 contains n, the length of the vector, and that the information itself is in v↓1 to v↓n.

### VectortoFile (global 324)

This routine outputs the contents of the vector v to the file f.

Line 21: The transfer is given hardware assistance.

### VectorfromFile (global 323)

This function reads a file into a vector of the appropriate size obtained from free storage. The vector is returned as the result.

Lines 27-30: A check that the file exists.
31-33: The size of the required vector is calculated.
32: Unused is the size of the unused portion of the last page of the file.
36: The vector is claimed.
37-40: Its contents are initialised.

### AddMoreVectoFile (global 325)

This routine appends to the file f the information in the vector v.

Lines 47-49: A check that the user is allowed to write to the file.
50: 'DiscRts', the file of private routines, is loaded if not already present.
51: Addr is the disc address where the new information is to go.

52: This private routine, defined in 'DiscRts', copies the new information to the disc starting at the specified address.

LoadDiscRtsifNec

See VIII:2.

CheckPerm

This is practically a copy of the private routine in 'DISCOUT' (II:7.4).

## VIII:5 DiscRts

This file contains private routines used by the three files 'MakeNewFile', 'File Vectors' and 'Index Ops'. It is loaded by routines in those files whenever necessary.

Note that these routines are intended only as private routines: they therefore make no checks on the validity of the parameters, nor do they check before writing to a file that such an action is permitted.

NewLocation (global 308)

This function is used when extra information is to be added onto the end of a file body; it calculates the disc address for the start of the new information.

Line 22: The page address will be the last page of the file body.
26-28: If no body exists one is created.
29-31: Otherwise, the address of the first free word is obtained from the pointer which is stored in the last page of the file body (at least for all the files to which this function may be applied).

MakeOnePageBody

This is a private function which constructs a new one page body for a file; its result is the address of the page used for this purpose.

AddVectoFile (global 307)

This routine appends the information in v↓0 to v↓n onto the end of the file f, beginning at the address specified by the vector Addr.

Line 51: Vec is a one page buffer. It is kept in a static variable (line 16) in order that it may also be used by TurnPage. Note that DiscPage cannot be used instead of Vec, in particular because TurnPage calls NewDiscBlock, which could cause DiscPage to be overwritten.
54-57: If the end of the page is reached, a new page must be added to the file.
72: The owner is charged for any new pages. Note that h↓NUMBPAGES refers to the heading before the updating in lines 70, 71.

TurnPage

This function constructs a new last page for a file and links it onto the previous last page. The result is the address of the new page. Note that the heading of the file is not changed: this must be done by the calling routine.

VIII:6 LinePrinter

This file defines the global functions LinePrinter and GeneralLinePrinter. Their definitions use the non-global stream functions BytestoLP and GeneralIntcodetoLP. This is an example of the method of dealing with errors outlined in [2] §2.3.

Lines 11-18: These are constants used in the function BytestoLP.
12-14: The size of the vector and the names of some of the elements.

15: The values of S↓PINGS, used by InitiateTransfer (II:5.3).
16: The device number of the LinePrinter. See the Exec TRANSFER command (II:9).
20-29: Constants used by the function GeneralIntcodetoLP.
21-25: The size of the vector and the names of some of the elements.
27: The length of the vector required to hold a line buffer.
28: The name of a special element of this vector.
32-42: Characters in line printer code.
45-52: Internal code characters which cannot be printed on the devices used for manipulating program texts. (DIGITO and LETTERO can of course be printed but are given explicitly here to avoid possible confusion.)

LinePrinter (global 98)

This is the function normally used to obtain access to the line printer. The result is an internal code output stream. It uses the more general function GeneralLinePrinter; the parameter DEFAULT specifies that the system's default error function is required.

GeneralLinePrinter (global 80)

The result of this function is also an internal code output stream to the line printer, but the action required in the event of error may be specified by the parameter function. See the system's default function StandardErrorFn below for details of the conventions which these error functions must satisfy.

BytestoLP

This is a parameterless stream function. The result is an output stream to the line printer of bytes in printer code. The stream uses InitiateTransfer (II:5.3) in order that appropriate action may be taken if the printer is held up. However, it was written to be as simple as possible and has only a single one-word buffer, unlike the other streams (e.g. BytesfromPT, II:5.4) which use InitiateTransfer. Although the efficiency of this stream would undoubtedly be improved by using double buffers of greater length with the fast stream mechanism to fill them, we have not yet felt the need to optimise here.

Lines 76-86: A vector is claimed and the standard elements initialised.
83: Note that no action is required for Reset.
88-96: Initialisation of the part of the vector reserved for the Exec parameter block (see II:9).
98-101: Initialisation of the part of the vector reserved for the ClearUpChain entry (see II:1.1).
102, 103: The entry is attached to ClearUpChain.
105: When double-buffering, InitiateTransfer requires the word before each buffer to point to the other buffer. When as here both buffers are the same, the word before points to the buffer itself.

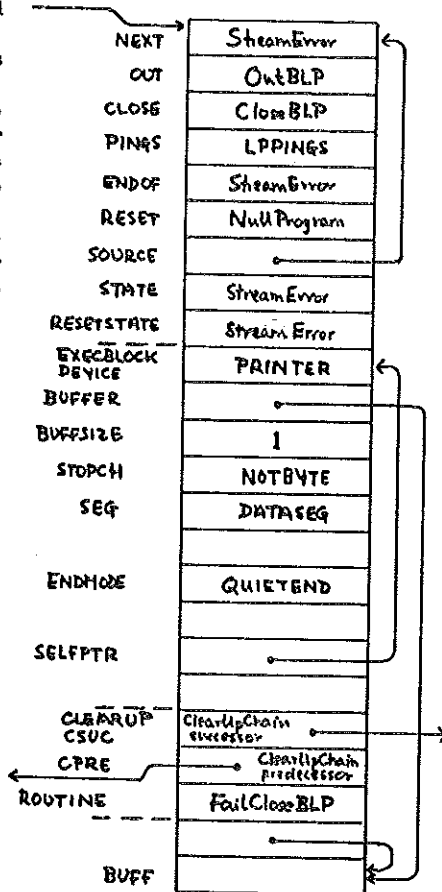



Fig. 28 BytestoLinePrinter[]

OutBLP

The 'Out' routine to output each byte.

CloseBLP

The 'Close' routine.
Lines 119-121: The entry is removed from ClearUpChain.
123: A pause to allow time for completion of any transfer currently in progress.

The problem is as follows: if the printer is still on-line when the stream is closed it is convenient to end with a page throw, so that the output may be easily removed from the device; but frequently by this time the printer will have been set off-line and the paper already removed by the impatient programmer. In this case to force a page throw out would be not only superfluous but also annoying, as the operator would have to switch the printer on again in order to allow it to happen, and to leave the transfer pending might cause catastrophe if the space occupied by the stream is overwritten before the transfer ends (see ClearUpLP, below). However, the interface designed to connect the printer to the main computer does not let the computer know whether the printer is on-line or not. This is the sort of nonsense that software designers are always having to cope with. So the character conversion stream function GeneralIntcodetoLP (see below) outputs a page throw anyway, and the TRANSFER command which this generates (in InitiateTransfer, II:5.3) is accepted by Exec, which doesn't know any better. CloseBLP waits a moment to allow the page throw to happen if it can, and then (line 124) cancels it. Note that we have to ensure that there is no risk of our cancelling the user's own output, which we would have no right to do: but since we output to the printer one character at a time, no special action need be taken - this would need consideration if we were using more sophisticated buffering.

ClearUpLP

This is the value of the ROUTINE field of the ClearUpChain entry, and it is called if the stream is prematurely abandoned. It is necessary to cancel any transfer in progress, to avoid possible catastrophe in case the space occupied by the Exec parameter block (see II:9) is re-allocated and overwritten before the transfer ends, as Exec refers to it in order to decide what action to take on completion (II:9). See also [1] §1.2.2.

Cancel

This routine uses the Exec CANCEL command (II:9) to abandon any transfer currently in progress to the line printer.

GeneralIntcodetoLP

This is a character conversion output stream function. Its first parameter is a byte output stream destined for the line printer. The result is an internal code output stream. The action required on detection of an error condition is specified by the second parameter. This may be a function which will be called in case of error, or it may take the value DEFAULT to indicate that the standard error action is required, defined with the stream function itself.

Underlining is achieved on our line printer by overprinting, so any underlines required are stored in a special buffer and output at the end of the line.

As line printer paper is not cheap, a group of two or more page throws output consecutively is treated as an error. This is achieved by setting the OUT element of the stream to a special routine whenever a page throw is output.

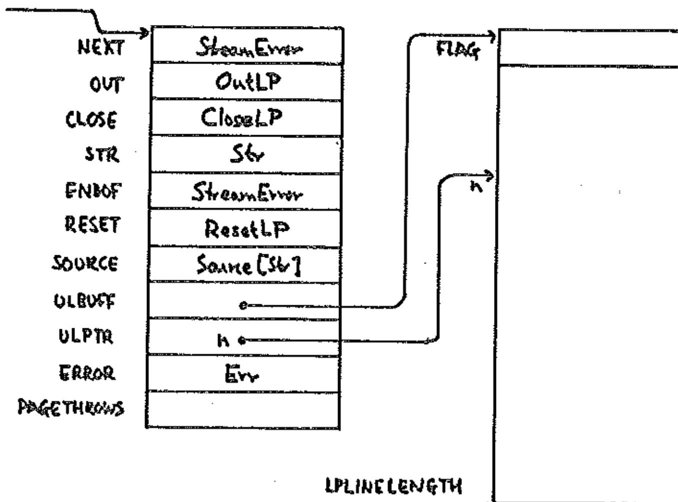



Fig. 29. GeneralIntcodetoLP[Str, Err]

Lines 140-147: A vector is claimed and the standard elements initialised.

142: It is assumed that when the stream is set up the printer is at the top of the page. This depends on the establishment's operating techniques and is usually true.

150: StandardErrorFn is the default error function supplied by the system (see below).

152-155: The buffer for storing underlines is claimed and the pointers initialised. B↓FLAG is used to indicate whether the buffer contains any underlines.

OutLP

This is the normal 'Out' routine for the stream.

Line 164: B is the underline buffer.

165: A boolean stating whether or not the character is underlined.

166: The character without the underline bit.

167: The value of Ch is either the line printer code for the character or the special constant COMPLETE, which indicates that any required action has been performed as a side-effect in the valof block on the right hand side of the definition.

169-172: This is the usual case. The internal code table is split into two at '('. In one part most of the characters have the same value in line printer code, in the other most of them are unprintable.

174-177: Recursion to output the four-space character as four separate spaces.

179-183: The newline character causes the underline buffer to be output, and then the newline character in printer code.

185-189: This deals similarly with the carriage return character, (which, together with the back space character may be used for more elaborate overprinting).

191-197: The page throw character is dealt with in a similar fashion, but in addition the page throw count is started and the 'Out' routine altered, to trap any further page throws.

199-207: If already at the beginning of a line the backspace character has no effect. Otherwise, the effect is the same as a carriage return followed by sufficient spaces to bring the pointer to one place before its original position.

210-229: These are the printable characters which cannot be covered by the default case above.

219: An exclamation mark is constructed by overprinting a prime on a fullstop. The fullstop and the backspace are output by recursive calls.

222-235: Similarly, the dollar is constructed by overprinting an S with a bar.

226, 227: The other digits and letters are dealt with by the default case. However, the hardware character set of the printer has a narrow O for the letter and a fatter one for the digit, whereas our convention is the reverse. So we put it right explicitly here.

236: We reach this point only with unprintable characters, so the error function is called to deal with them. The result of this call may be another internal code character to be output instead.

241, 242: At this stage Ch has been satisfactorily defined. We now check that the line is not already full. If it is, the error function is called to cope with the problem.

244: In this case no further work is required.

246: Otherwise the character is output.

247: The appropriate character, an underline or a space, is placed in the underline buffer.

248: The flag indicates whether anything has yet been underlined.

OutputUnderlines

This routine is called at the end of each line (and when processing certain other characters, such as backspace) to overprint underlines on any characters that require them.

Line 257: If nothing has been underlined there is nothing to be done.

259, 260: Otherwise a carriage return is output, followed by the contents of the underline buffer.

TrapPageThrows

This is the special 'Out' routine which is placed in the stream vector whenever a page throw is output. Its purpose is to trap two or more consecutive page throws and to treat such a condition as an error.

Line 267: If more page throws are output the routine keeps a count.

269, 270: The count is reset and the normal 'Out' routine replaced.

271: If more than one page throw has been output, the error function is called. Note that this must happen after the normal 'Out' routine has been restored in case an error report is required, as ReportStream might be to the printer.

168: The character after the page throw or page throws is output as usual.

CloseLP

The 'Close' routine for the stream.

Lines 279, 280: The underline buffer is output (if necessary) and returned.

281: The error function is called if there have just been several consecutive page throws.

282: A page throw is output unless the printer is at the top of a page already. See the commentary on CloseBLP line 123 above.

ResetLP

The 'Reset' routine for the stream.

Line 289: A new line is output unless the printer is already at the beginning of a line.

291-294: The error function is called if there have just been several consecutive page throws, so that the stream continues with a clean slate.

StandardErrorFn

This is the standard error function called when the IntcodetoLP stream detects an error condition, unless the user has specified an error function of his own. The three parameters are the stream, the reason for the error condition, and any further information (depending on the condition) which might be helpful.

Lines 301-312: This part deals with unprintable characters.
302: y is the non-underlined version of the character.
303, 304: These are treated as valid characters. ACUTE is printed as PRIME, and RUNOUT and STOPCODE characters (meant for paper tape streams) are ignored. However, since these characters are not strictly applicable to the printer, it is up to the error function to decide how to deal with them.
308: A space is left for valid characters which are unprintable merely because they are not in the printer's repertoire. They can therefore be inserted later by hand if required.
311: Invalid characters are reported, but otherwise ignored.
323: This case could only arise after a wild jump.

PrinterReport

A private routine for outputting error reports about the line printer stream.

Source

See II:4.1.

## References

[1] J.E. STOY and C. STRACHEY
OS6 - an experimental operating system for a small computer
Part 1: general principles and structure
Computer Journal 15, pp 117-124 (1972).

[2] ibid.
Part 2: input/output and filing system
Computer Journal 15, No.3 (1972).

[3] E.W. DIJKSTRA
Structured Programming
Software Engineering Techniques
(ed. J.N. Buxton and B. Randell)
pp 84-88: Nato Science Committee (1970).

[4] THE NEW ENGLISH BIBLE
Oxford University Press and Cambridge University Press (1970).

[5] M. RICHARDS
The BCPL Reference Manual
Technical Memorandum 69/1
University of Cambridge Computer Laboratory (1969).

[6] N. WIRTH
The Programming Language Pascal
Acta Informatica 1, pp 35-63 (1971).

[7] W.S. WULF, D.B. RUSSELL and A.N. HABERMANN
BLISS: A Language for Systems Programming
Comm.A.C.M. 14, pp 780-790 (1971).

## Appendix: BCPL

This has no pretence to be a complete description of BCPL, or even one adequate for the intending programmer. For both of these the reader is referred to the reference manual [5]. Our purpose here is merely to mention points which might otherwise prevent understanding of the text of OSPub.

In this description we sometimes use e to stand for any expression, c for a command, b for a block, x for a name, and k for a constant expression (one that the compiler is able to evaluate at compile-time).

### Representation and syntax

get, followed by one or more quoted character strings, indicates that the text in the file specified (in some implementation-dependent way) by the strings is to be inserted at this point.

Comments are introduced by || and extend to the end of the line.

Commands are, in principle, separated by semicolons. However, the compiler will insert one automatically at the end of a line in suitable contexts.

Similarly, a then (or do, which is regarded as synonymous) is properly required in if commands and similar constructions; but the compiler will sometimes insert one automatically if need be.

### Expressions and operators

8 indicates that the number following is in octal.

The value of a quoted string containing just one character ('A') is the internal code for that character. (The internal code is given in Fig.30.)

The value of a longer string is a vector containing the characters packed into words. The number of bytes per word is implementation-dependent (2 for us); the first byte contains the length of the string in characters.

Some non-printable characters may be represented in strings as follows:

*n stands for newline
*s stands for space (or a space itself may be used)
*4 stands for the 4-spaces character

<table>
<tr><th></th><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th></tr>
<tr><td>00</td><td></td><td></td><td></td><td>Runout</td><td>Stop code</td><td></td><td></td><td>Bell</td></tr>
<tr><td>01</td><td>Back space</td><td>Tab</td><td>New line</td><td></td><td>Page throw</td><td>Carriage return</td><td></td><td>4-spaces</td></tr>
<tr><td>02</td><td></td><td></td><td>¬</td><td>{</td><td>'</td><td>}</td><td>→</td><td>$_{10}$</td></tr>
<tr><td>03</td><td>÷</td><td>£</td><td>∨</td><td>×</td><td>λ</td><td>≠</td><td>↑</td><td>↓</td></tr>
<tr><td>04</td><td>Space</td><td>!</td><td>"</td><td>#</td><td>$</td><td>%</td><td>&</td><td>´</td></tr>
<tr><td>05</td><td>(</td><td>)</td><td>*</td><td>+</td><td>,</td><td>-</td><td>.</td><td>/</td></tr>
<tr><td>06</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td></tr>
<tr><td>07</td><td>8</td><td>9</td><td>:</td><td>;</td><td><</td><td>=</td><td>></td><td>?</td></tr>
<tr><td>10</td><td>@</td><td>A</td><td>B</td><td>C</td><td>D</td><td>E</td><td>F</td><td>G</td></tr>
<tr><td>11</td><td>H</td><td>I</td><td>J</td><td>K</td><td>L</td><td>M</td><td>N</td><td>O</td></tr>
<tr><td>12</td><td>P</td><td>Q</td><td>R</td><td>S</td><td>T</td><td>U</td><td>V</td><td>W</td></tr>
<tr><td>13</td><td>X</td><td>Y</td><td>Z</td><td>[</td><td>\</td><td>]</td><td>∧</td><td>←</td></tr>
<tr><td>14</td><td>`</td><td>a</td><td>b</td><td>c</td><td>d</td><td>e</td><td>f</td><td>g</td></tr>
<tr><td>15</td><td>h</td><td>i</td><td>j</td><td>k</td><td>l</td><td>m</td><td>n</td><td>o</td></tr>
<tr><td>16</td><td>p</td><td>q</td><td>r</td><td>s</td><td>t</td><td>u</td><td>v</td><td>w</td></tr>
<tr><td>17</td><td>x</td><td>y</td><td>z</td><td>§</td><td>|</td><td>$</td><td>~</td><td>Delete</td></tr>
</table>

The most significant bit (<u>8</u>200) indicates that the character given by the other seven bits is underlined.

Fig. 30 Internal Character Code

*q stands for a question mark
*b stands for backspace
*p stands for pagethrow
** stands for *
*' stands for '

lv and rv: In general, an expression may be evaluated in one of two modes, called Lmode and Rmode. For example, in the assignment command

x := y

the left-hand side is evaluated in Lmode to give the address of some storage cell, and the right-hand side is evaluated in Rmode, to give the new value for the contents of that cell. The results of these evaluations are called the Lvalue of x and the Rvalue of y. (Of course, some expressions cannot be evaluated in Lmode, as they cannot specify the address of a cell.)

The operator lv allows an address to be treated as a value. An expression

lv e

causes e to be evaluated in Lmode; the Lvalue of e is then the Rvalue of the overall expression. So

x := lv y

will assign the address of y as the new contents of x. (The left-hand sides of assignment statements and the operands of lv are, in fact, the only contexts where expressions are evaluated in Lmode.)

rv provides the converse operation. In an expression

rv e

the Rvalue of e is taken as the Lvalue of the overall expression; if the Rvalue of the complete expression is wanted, the contents are obtained of the cell specified by the Lvalue. So if, as above, x = lv y, then the command

rv x := z

will alter the contents of y. The command

w := rv x

will assign the contents of y to w. So rv(lv x) is identical to x.

The elements of a vector v, written v↓0, v↓1, ..., v↓n, have consecutive storage cells. The Rvalue of v itself is the Lvalue of v↓0. So

v↓i is the same as rv(v+i).

The parameter list of a function or routine may, according to BCPL, be treated as a vector. So, within the definition of f[x,y,z], one may let v = lv x, and then refer to the parameters as v↓0, v↓1 and v↓2.

E1 → E2, E3 means the same as Algol 60's

if E1 then E2 else E3 .

E1 rem E2 means the remainder when E1 is divided by E2. / and rem are mutually consistent; in our implementation the remainder has the same sign as the denominator.

The operators ^ (and), v (or), ~ (not), ≡ (equivalent) and ≢ (not equivalent) perform logical operations on bit patterns. However, the representations of true and false are such that these operators may also correctly be applied to booleans.

Extended relations (such as 1 ≤ x ≤ n) have their normal mathematical meanings.

Evaluation of an expression of the form

valof b

where b is a block, causes execution of b to commence, and to go on until a command of the form resultis e. Execution of b thereupon ceases, and the Rvalue of e is returned as the Rvalue of the valof expression.

The parameter list of a function call is enclosed in square brackets. These brackets must appear, even if the function takes no parameters, in order to distinguish the value obtained by applying the function from the value of the function itself. Parameters of functions (and routines) are always by Rvalue; if the Lvalues are to be handed over, the operators lv and rv must be employed.

## Commands

The symbols § and $ may be used to bracket commands, and are equivalent to Algol 60's begin and end. They may be tagged to indicate matching: when a particular block is closed, it implies the closing of any blocks inside it, even if they are not closed explicitly.

The multiple assignment, for example

x, y := E1, E2

indicates that the separate assignments

x := E1; y := E2

will be performed sequentially (not simultaneously). The order of performing the assignments is undefined.

<u>test</u> e <u>then</u> C1 <u>or</u> C2 is equivalent to Algol 60's

<u>if</u> e <u>then</u> C1 <u>else</u> C2;

In BCPL, <u>if</u> is used only when there is no <u>else</u> part.

The loop command

c <u>repeat</u>

indicates that c is to be repeated indefinitely.

The loop commands

<u>while</u> e <u>do</u> c

and

c <u>repeatwhile</u> e

differ in that the body of a <u>while</u> loop is not obeyed at all if the condition is not initially satisfied; the body of a <u>repeatwhile</u> loop is always obeyed at least once.

In <u>for</u> commands, which are loop commands of the form

<u>for</u> i = E1 <u>to</u> E2 <u>do</u> c

where c is a command, a new variable i is defined, its scope being c. The increments in the value of i are always unity; the limits E1 and E2 are evaluated once only, at the start of the command.

In switch commands, of the form

<u>switchon</u> e <u>into</u> b

the block b contains case-labels, of the form

<u>case</u> k:

Control passes to the case-label for which the value of k equals the value of e. If there is no match, control passes to the default label

<u>default</u>:

if there is one, otherwise b is not entered and control passes to the point after the switch command.

<u>break</u> causes execution of the innermost loop command to cease; control passes to the point after the loop.

<u>return</u> causes execution of the innermost routine activation to cease; control passes to the point after the routine call.

<u>endcase</u> causes execution of the innermost switch command to cease; control passes to the point after the switch command.

<u>resultis</u> e causes evaluation of the innermost <u>valof</u> expression to cease, and indicates that the Rvalue of e is to be the Rvalue of the <u>valof</u> expression.

The parameters of a routine call must be enclosed in square brackets, even if the parameter list is null.

Definitions

In BCPL, variables are of two kinds: static and dynamic. A dynamic variable has storage allocated and its value initialised when its definition is obeyed during the execution of the program. Like Algol 60 variables, they last only until the end of the block in which they are declared, and recursive activations of the same block give rise to separate dynamic variables. Static variables are given storage and initialised when the program is loaded, and last until it is unloaded; so recursion does not give rise to new static variables.

Dynamic variables are created by definitions of the form

let x = e

Vector definitions of the form

let x = vec k

create a vector with elements x↓0 to x↓k. Both vector and elements are dynamic.

Static variables are created by labels, by function definitions of the form

let f[x,y,z] = e ,

by routine definitions of the form

let r[x,y,z] be c ,

by global definitions (see below), or explicitly by definitions of the form

static § x = k §

A manifest definition, of the form

manifest § x = k §

indicates that the Rvalue to be associated with x is to be k. x is a constant, not a variable, and cannot be evaluated in Lmode; but it can be a component of constant expressions. (The compiler will replace occurrences of x by its value.)

A global definition, of the form

global § x: k §

indicates that x is the kth element of the global vector. x is a static variable, though its Rvalue is not initialised (but see 'Important exception', below). The global vector is the mechanism for communication between separately compiled segments.

Definitions introduced by <u>let</u> may be joined by using <u>and</u> in place of the second <u>let</u>. They are then mutually recursive.

Scope rules: all names, however they are defined, are governed by conventional scope rules: that is to say, objects (variables or constants) may be defined within the scope of other objects of the same name.

Important exception: a definition of a static variable (function, routine, etc.) within the scope of a <u>global</u> variable of the same name is treated not as defining a new variable, but merely as specifying the initial Rvalue of the <u>global</u>. (This applies to the 'definitions' of all the system routines defined in OSPub, for example.)

# INDEX


| | Txt | Com |
|---|---|---|
| Accounting | 65 | 82 |
| Activation pointer | | 26 |
| AddEntry | 113 | 130 |
| AddLinkedEntry | 114 | 131 |
| AddMoreVectoFile | 117 | 133 |
| AddressZero | 53 | 65 |
| AddVectoFile | 119 | 135 |
| Aims of publication | | 1 |
| Allocation - core | 23 | 29 |
| Allocation - disc | 56 | 70 |
| AMPERSANDt | 99 | |
| AnythingTyped | 19 | 25 |
| Appendix: BCPL | | 145 |
| AskTime | 52 | 65 |
| Assembly code | 72 | 89 |
| BACKLINK | 102 | |
| BCPL | | 145 |
| BELL | 99 | |
| BELLt | 99 | |
| BFF1END | 98 | |
| BFF2END | 98 | |
| BFPT | 88 | 108 |
| BFPTSIZE | 98 | |
| Binary format | | 11 |
| Bootstraps | 75 | 97 |
| Bricks for compilation | 103 | 115 |
| BUFF1 | 98 | |
| BUFF2 | 98 | |
| BUFFER | 94 | 91 |
| BUFFSIZE | 94 | 91 |
| BytesfromPT | 88 | 53 |
| BytestoLP | 121 | 136 |
| CANCEL | 94 | 93 |
| Cancel | 123 | 138 |
| CASEMASK | 99 | |
| Central loop of system | 1 | 8 |
| CFirst | | 14 |
| Character codes (see also Fig.30) | 99 | |
| Charge | 65 | 82 |
| Check | 113 | 131 |
| CheckDiscOn | 79 | 102 |
| CheckLegality | 111 | 129 |
| CheckLinkDoesntLoop | 113 | 131 |
| CheckPerm | 61 | 77 |
| CheckPermission | 115 | 132 |
| CheckType | 58 | 74 |
| CL | 96 | |
| CLAIMED | 102 | |
| Clearing up | 3 | 10 |
| ClearUp | 3 | 9 |
| ClearUpChain constants | 97 | |
| ClearUpLP | 122 | 138 |
| CLOCK | 95 | 93 |
| Clock | 51 | 64 |
| Clock constants | 95 | |
| CLOCKBUFFSIZE | 95 | |
| CLOSE | 97 | |
| Close | 30 | 38 |
| CloseBLP | 122 | 137 |
| CloseEF | 60 | 77 |
| CloseIF | 59 | 76 |
| CloseLP | 126 | 142 |
| CloseOF | 63 | 80 |
| ClosePB | 29 | 36 |
| CloseTele | 42 | 50 |
| CloseW | 49 | 61 |
| Code segment addresses | 95 | 110 |

| | Txt | Com |
|---|---|---|
| CODESEG | 94 | |
| Compilation of the system | 103 | 115 |
| Compiled segment - format | | 11 |
| COMPLETED | 94 | 92 |
| CON | 96 | |
| Console (see also ExecConsole) | | 107 |
| Constants | 94 | 110 |
| Copy | 53 | 65 |
| Core storage | 23 | 29 |
| CoretoDisc | 54 | 67 |
| CP | 96 | |
| CPages | | 98 |
| CPRE | 97 | 6 |
| CPTR | 96 | |
| CPtr | | 14 |
| CREATED | 100 | |
| CreateNewHead | 111 | 128 |
| Creation of files | 109 | 127 |
| CSUC | 97 | |
| CUC | 96 | |
| CUCHAIN | 102 | |
| CurrentIndex | | 125 |
| DATASEG | 94 | |
| DATASIZE | 100 | |
| DATE | 95 | 112 |
| Date | 81 | 104 |
| DBlock | | 13 |
| Debugging facilities | 11 | 18 |
| Declarations | 90 | 109 |
| DefaultProg | 1 | 8 |
| DELETE | 99 | |
| DeleteBody | 61 | 77 |
| DELETED | 101 | |
| DeleteEntry | 115 | 132 |
| DeleteFile | 111 | 129 |
| DEVICE | 94 | 91 |
| Device numbers | 95 | |
| Diad format | | 59 |
| DiadRead | | 59 |
| Diads | 47 | 59 |
| Diagnostics | | 116 |
| Differences from 'OS6' papers | | 2 |
| Disc (see also II:7-8, VIII:2-4) | | 93 |
| Disc accounting | 65 | 82 |
| Disc input/output | 58 | 74 |
| Disc routines | 54 | 67 |
| Disc storage | 56 | 70 |
| Disc streams | 58 | 74 |
| Disc validation | 83 | 105, 119 |
| Disc vectors | | 3 |
| DiscPage | | 67 |
| DISCREAD | 95 | 93 |
| DiscRts | 118 | 134 |
| DisctoCore | 54 | 69 |
| DiscTransfer | 54 | 69 |
| DiscVectorElement | | 3 |
| DiscVectorfromFile | | 3 |
| DISCWRITE | 95 | 93 |
| DISCWRITEPERMITTED | 95 | 111 |
| DIV | 99 | |
| DL | 96 | |
| DLR | 100 | |
| DLW | 100 | |
| DP | 96 | |

| | Txt | Com |
|---|---|---|
| DPages | | 98 |
| Dump | 12 | 20 |
| DumpSegment | 13 | 21 |
| EMPTY | 98 | |
| END | 97 | |
| ENDBODY | 100 | |
| ENDCUCHN | 97 | |
| EndGiveUp | 14 | 21 |
| ENDMODE | 94 | 92 |
| Endof | 74 | 95 |
| ENDOF | 97 | |
| EndofBFPT | 45 | 57 |
| ENDOFDATAPTR | 100 | |
| EndofEF | 60 | 77 |
| EndofIF | 59 | 75 |
| EndofPB | 29 | 36 |
| ENDOFSTREAMCH | 102 | |
| EndofTele | 42 | 51 |
| EndofTT | 38 | 47 |
| EndofW | 49 | 61 |
| Enter | 112 | 130 |
| EntriesfromFile | 60 | 76 |
| ENTRIESPERPAGE | 101 | |
| Entry-point for program (see Prog) | | 8 |
| Entry-point for system | 75 | 98 |
| ENTRYEND | 101 | |
| ENTRYSIZE | 101 | |
| EqS | 53 | 66 |
| Error reports | | 116 |
| ESCAPEt | 99 | |
| ESIZE | 94 | |
| EvenParity | 40 | 49 |
| Exchange | 18 | 24 |
| Exec | 72 | 90 |
| Exec commands | 94 | 90 |
| EXECBLOCK | 97 | |
| ExecBlock for TRANSFER command | 94 | 92 |
| ExecConsole | | 48 |
| EXECCONDON | 95 | |
| Executive | | 90 |
| Executive teletype | | 48 |
| FailClose | 64 | 81 |
| Failure of programs | 11 | 18 |
| Failure reports | | 116 |
| Fast stream block constants | 98 | |
| FBC | 96 | |
| FBSIZE | 98 | |
| FClosePB | 29 | 36 |
| FetchCode | 72 | 89 |
| FetchExecWord | 12 | 21 |
| Fig.1 | | 10 |
| Fig.2 | | 11 |
| Fig.3 | | 13 |
| Fig.4 | | 15 |
| Fig.5 | | 16 |
| Fig.6 | | 19 |
| Fig.7 | | 28 |
| Fig.8 | | 30 |
| Fig.9 | | 31 |
| Fig.10 | | 37 |
| Fig.11 | | 38 |
| Fig.12 | | 39 |
| Fig.13 | | 46 |
| Fig.14 | | 48 |
| Fig.15 | | 52 |

| | Txt | Con |
|---|---|---|
| Fig.16 | | 54 |
| Fig.17 | | 60 |
| Fig.18 | | 68 |
| Fig.19 | | 71 |
| Fig.20 | | 72 |
| Fig.21 | | 74 |
| Fig.22 | | 76 |
| Fig.23 | | 78 |
| Fig.24 | | 83 |
| Fig.25 | | 87 |
| Fig.26 | | 92 |
| Fig.27 | | 125 |
| Fig.28 | | 137 |
| Fig.29 | | 139 |
| Fig.30 | | 146 |
| FILE | 101 | |
| File creation | 109 | 127 |
| File deletion | 111 | 129 |
| File permissions | 101 | |
| File types | 101 | |
| File Vectors | 116 | 133 |
| FILENUMB | 102 | |
| FILEOWNER | 102 | |
| Files | | |
| input/output streams | 58 | 74 |
| Filing system constants | 100 | |
| FindHeading | 68 | 85 |
| Finish | 4 | 10 |
| FIRSTDATA | 100 | |
| FIRSTENTRY | 102 | |
| FirstNextBFPT | 44 | 55 |
| FIRSTPAGE | 100 | |
| FNextPB | 29 | 36 |
| ForcedFinish | 14 | 21 |
| ForcedGiveUp | 11 | 19 |
| Format | | |
| of compiled segment | | 11 |
| Format of diads | | 59 |
| Format of loaded section | | 12 |
| FPRE | 96 | 6 |
| FREE | 100 | |
| Free storage | 23 | 29 |
| Free store constants | 96 | |
| Free store file | | 72 |
| Free store file constants | 102 | |
| FResetPB | 29 | 36 |
| FS | | 29 |
| FSF | | 72 |
| FSFSIZE | 102 | |
| FSFtoCore | 80 | 103 |
| FSLin | | 97 |
| FSV | 96 | |
| FSVECSIZE | 96 | |
| FWC | 96 | |
| Garbage collection | | 120 |
| General constants | 94 | |
| GeneralIntcodetoLP | 123 | 139 |
| GeneralLinePrinter | 121 | 136 |
| get files | 103 | 115 |
| GetC | 32 | 41 |
| GiveUp | | 18 |
| GiveUpStack | | 18 |
| GiveUpStackSize | | 18 |
| Global-setting directives | 10 | 16 |
| globals | 91 | 109 |
| GRAVE | 99 | |
| GU | 96 | |
| Guide to these books | | 6 |
| GUS | 96 | |
| GUSS | 96 | |
| Hardware instructions | 72 | 89 |

| | Txt | Con |
|---|---|---|
| Headings | | 83 |
| Headings constants | 100 | |
| HPAGE | 101 | |
| HWORD | 101 | |
| IBLK | 96 | |
| IBlock | | 12 |
| ID | 96 | |
| IFBUFFER | 102 | |
| IFSIZE | 102 | |
| In | | 107 |
| INBFFEND | 98 | |
| INBFFPTR | 98 | |
| INBUFF | 98 | |
| INDEX | 101 | |
| Index | 128 | 152 |
| Index entries constants | 101 | |
| Index of system files | | 119 |
| Index operations | 112 | 129 |
| Index structure | | 87 |
| Indexes | | 86 |
| INESC | 98 | |
| INFILESIZE | 102 | |
| Information block | | 12 |
| Information block constants | 96 | |
| InfromFile | 58 | 74 |
| InfromFile constants | 102 | |
| Initialising the system | 75 | 97 |
| InitiateTransfer | 43 | 51 |
| INPT | 96 | |
| Input functions | 36 | 44 |
| Input/output routines | 30 | 38 |
| InStack | 12 | 19 |
| IntcodetoandfromTeletype | 40 | 48 |
| Interlude | 6 | 12 |
| Internal character code | 99 | 146 |
| INTERRUPT | 94 | 92 |
| Interrupt | 21 | 26 |
| INTERRUPTADDRESS | 95 | 112 |
| INTERRUPTINHIBITED | 95 | 110 |
| INTERRUPTINHIBITED | 95 | 110 |
| Interrupts | 11 | |
| INTREASON | 94 | 92 |
| Introduction | | 1 |
| IPRE | 96 | 6 |
| ISIZE | 96 | |
| ISUC | 96 | |
| JumpTo | 19 | 25 |
| Kernel of system | 1 | 8 |
| Labels (setting) | 9 | 15 |
| Language | | 2 |
| LASTDATA | 100 | |
| LASTENTRY | 102 | |
| LASTPAGE | 100 | |
| LB | 96 | |
| Leap | 82 | 104 |
| LEFTARROW | 99 | |
| LENGTH | 100 | |
| LGLoop | 1 | 8 |
| Library files | 107 | 119 |
| Line-printer | 120 | 92 |
| LINEBFFLAST | 98 | |
| LINEBFFPTR | 98 | |
| LINEBUFFER | 98 | |
| LINEFEEDt | 99 | |
| LinePrinter | 121 | 135 |
| Link | 113 | 130 |
| LINKING | 101 | |
| LMASK | 94 | |
| Load | 5 | 12 |
| LoadDiscRtsifNec | 110 | 127 |

| | Txt | Con |
|---|---|---|
| LoadFile | 71 | 88 |
| LoadGoLoop | 1 | 8 |
| Loading of programs | 5 | 11 |
| Loading the system | 75 | 97 |
| LoadSection | 6 | 14 |
| LoadSystemFile | 71 | 88 |
| Log in | 107 | 125 |
| LogIn | 4 | 9 |
| LogIn Prog | 107 | 126 |
| LOOKATRDR | 94 | 93 |
| LookUp | 70 | 85 |
| LookUpinMFL | 68 | 84 |
| LookUpUser | 107 | 126 |
| Machine code instructions | 72 | 89 |
| Machine constants | 94 | |
| Main loop of system | 1 | 8 |
| MakeNewFile | 109 | 127 |
| MakeOnePageBody | 118 | 135 |
| manifests | 94 | 110 |
| ManualPH | 17 | 23 |
| Master file list | | 84 |
| Master file list constants | 101 | |
| MAXC | 95 | 111 |
| MAXD | 95 | 111 |
| MaxVecSize | 26 | 33 |
| Message | 55 | 70 |
| MFL | | 84 |
| MFLFIRSTPAGE | 95 | 112 |
| MFLNUMB | 100 | |
| MINSPERDAY | 95 | |
| Miscellaneous facilities | 51 | 64 |
| MPMNextN | 20 | 25 |
| MPMNextO | 20 | 25 |
| N1 (etc.) | 101 | |
| NEVER | 100 | |
| NEWBODY | 102 | |
| NewDiscBlock | 56 | 73 |
| NewFreeStore | 26 | 33 |
| NEWLINEt | 99 | |
| NewLocation | 118 | 134 |
| NewMFLEntry | 110 | 128 |
| NewMFLPage | 110 | 128 |
| NewVec | 23 | 29 |
| NewWord | 26 | 32 |
| NEXT | 97 | |
| Next | 73 | 94 |
| NextBlock | 48 | 60 |
| NextBuffIF | 59 | 75 |
| NextByte | 49 | 61 |
| NextCh (Diads) | 49 | 61 |
| NextCh (NextN) | 36 | 44 |
| NextEF | 60 | 77 |
| NextFillBFPT | 44 | 56 |
| NextLetter | 82 | 104 |
| NextN | 36 | 44 |
| NextO | 36 | 44 |
| NextPB | 28 | 36 |
| NextTele | 40 | 49 |
| NextTT | 37 | 45 |
| NextWaitBFPT | 45 | 56 |
| NONE | 96 | |
| NOREASON | 95 | |
| NOSTORE | 97 | |
| NOTBYTE | 94 | |
| NOTENTRY | 101 | |
| NOTHINGTYPED | 97 | |
| NOTSET | 96 | |
| NOTUSED | 98 | |
| NULL | 94 | |
| NULLBODY | 100 | |

| | Txt | Con |
|---|---|---|
| NullProgram | 53 | 66 |
| NUMBPAGES | 100 | |
| NXTB | 97 | |
| NXTPAGE | 100 | |
| OE | 99 | |
| OFBUFFER | 102 | |
| OFFSETC | 94 | 92 |
| OFPAGE | 102 | |
| OFSIZE | 102 | |
| OLDBODY | 102 | |
| OLDSIZE | 102 | |
| ONEMINUTE | 95 | |
| Operating system text | 1 | 8 |
| Operator intervention (see Interrupt) | 21 | 26 |
| Operator's console (see ExecConsole) | | 48 |
| Operator's teletype (see Teletype) | | 45 |
| OS6 - discrepancies | | 2 |
| OSReport | 33 | 42 |
| OSReportN | 33 | 42 |
| OSReports (list) | | 116 |
| OUT | 97 | |
| Out | 73 | 95 |
| Out9 | 34 | 43 |
| OutAddr | 35 | 43 |
| OUTBFFEND | 98 | |
| OUTBFFPTR | 98 | |
| OutBLP | 122 | 137 |
| OUTBUFF | 98 | |
| OutBuffOF | 63 | 80 |
| OutByte | 34 | 43 |
| OutCRLF | 41 | 50 |
| OutDate | 51 | 64 |
| OutDateandTime | 51 | 64 |
| OUTESC | 98 | |
| OutJustify | 33 | 41 |
| OutLP | 123 | 140 |
| OutN | 34 | 43 |
| OutNewLine | 38 | 47 |
| OutNewLines | 42 | 50 |
| OutO | 34 | 43 |
| OutP | 34 | 42 |
| OUTP | 96 | |
| Output | | 107 |
| Output routines | 34 | 42 |
| OutputUnderlines | 125 | 141 |
| OutS | 31 | 40 |
| OutString | 31 | 41 |
| OutTele | 41 | 49 |
| OutTime | 51 | 64 |
| OuttoFile | 62 | 79 |
| OuttoFile constants | 102 | |
| OutTT | 38 | 47 |
| OWNER | 100 | |
| OWNERONLY | 101 | |
| PAGE | 102 | |
| PAGEEND | 102 | |
| PAGENUMB | 94 | 91 |
| PAGESIZE | 94 | |
| Paper tape punch | | 92 |
| Paper tape reader (see also II:5.4) | | 92 |
| Parity | | 106 |
| PARITYBIT | 99 | |
| PARITYMASK | 99 | |
| PEC | 96 | |
| PEChain | | 35 |
| PECLOSE | 99 | |
| PEENDOF | 99 | |

| | Txt | Con |
|---|---|---|
| PBNEXT | 99 | |
| PBPRE | 99 | 6 |
| PBRESET | 99 | |
| PBSIZE | 99 | |
| PBSTR | 99 | |
| PERIOD | 95 | |
| Peripherals | | 92 |
| PERM | 100 | |
| Permanent input/output streams | 37 | 45 |
| Permissions | 101 | |
| PINOS | 97 | |
| PInterrupt | 75 | 98 |
| Post-mortem arrangements | 11 | 18 |
| POWERON | 95 | |
| PPTR | 96 | |
| PPtr | | 26 |
| PrepareforRun | 2 | 9 |
| PRIME | 99 | |
| PRIMEt | 99 | |
| Primitive disc routines | 54 | 67 |
| Printer | 120 | 92 |
| PrinterReport | 127 | 143 |
| Private disc routines | 118 | 134 |
| PrivateStack | | 26 |
| Procedure pointer | | 26 |
| Prog | | 8 |
| Program failure | 11 | 18 |
| Programming Language | | 2 |
| Programs - loading | 5 | 11 |
| Programs - running | 2 | 8 |
| PTR | 99 | |
| Punch | | 92 |
| PutBack | 28 | 35 |
| PutBack vector constants | 99 | |
| PWC | 96 | |
| QUERYt | 99 | |
| QuickValFSF | 83 | 105 |
| QUIETEND | 94 | 92 |
| RBLOCK | 95 | 112 |
| RDRDEVSUM | 95 | |
| READABS | 94 | 91 |
| ReadBackwards | 50 | 62 |
| READBFFSIZE | 98 | |
| READBLOCK | 98 | |
| ReadBuff | 44 | 53 |
| Reader (see also II:5.4) | | 92 |
| Reader stream constants | 98 | |
| ReaderDev | | 53 |
| READERLEFTTORIGHT | 95 | 92 |
| ReaderOffLine | 45 | 57 |
| REASONFORINTERRUPT | 95 | 111 |
| Reasons for interrupt | 95 | |
| References | | 144 |
| ReleaseNonSystemGlobals | 79 | 103 |
| Remote consoles | | 93 |
| REP | 96 | |
| ReportBlocks | 16 | 22 |
| ReportCallTrace | 15 | 22 |
| ReportFreeStoreState | 15 | 22 |
| ReportMessage | 84 | 105 |
| ReportS | 31 | 41 |
| ReportStream | | 107 |
| ReportWords | 16 | 23 |
| RESET | 97 | |
| Reset | 30 | 38 |
| ResetBFPT | 44 | 56 |
| ResetEP | 60 | 77 |
| ResetIF | 59 | 76 |
| ResetLP | 126 | 142 |
| ResetOP | 63 | 81 |

| | Txt | Con |
|---|---|---|
| ResetPB | 29 | 36 |
| RESETSTATE | 97 | |
| ResetState | 30 | 39 |
| ResetStateTT | 39 | 48 |
| ResetTele | 42 | 50 |
| ResetTT | 38 | 47 |
| ResetW | 49 | 61 |
| RestartClock | 52 | 65 |
| RestoreFreeStore | 27 | 33 |
| RESTRICTED | 101 | |
| RETURN | 99 | |
| Return | 19 | 24 |
| ReturnChain | 61 | 77 |
| ReturnDiscBlock | 57 | 73 |
| ReturnDiscVector | | 3 |
| RETURNt | 99 | |
| ReturnVec | 24 | 30 |
| ReturnWord | 26 | 32 |
| ROUTINE | 97 | |
| RPRE | 96 | 6 |
| RSIZE | 96 | |
| RUBOUTt | 99 | |
| Run | 2 | 9 |
| Run-block constants | 96 | |
| RUNOUT | 99 | |
| RUNOUTt | 99 | |
| RYGiveUp | 26 | 32 |
| SAME | 100 | |
| SCAN | 102 | |
| SectId | 5 | 12 |
| SEG | 94 | 91 |
| Segmentation of the system for compilation | 103 | 115 |
| SEIFPTR | 94 | 92 |
| SERIAL | 100 | |
| Set-up | 75 | 97 |
| SetDateandTime | 81 | 103 |
| SetGlobals | 10 | 17 |
| SetLabels | 9 | 15 |
| SetStackBase | 79 | 102 |
| Setting up files | 109 | 127 |
| SetUpDiscPage | 76 | 100 |
| SetUpDummyExecStructure | 13 | 21 |
| SetUpExecConsole | 86 | 107 |
| SetUpFS | 76 | 99 |
| SetUpParityTable | 85 | 106 |
| SetUpPMStacks | 77 | 101 |
| SetUpReader | 88 | 108 |
| SetUpRunBlock | 78 | 102 |
| SetUpStreams | 85 | 106 |
| SetUpSundryItems | 77 | 101 |
| SetUpTimeofDayClock | 78 | 101 |
| SHARPt | 99 | |
| SIZE | 97 | |
| Size (of numeral) | 33 | 42 |
| Size (of string) | 115 | 132 |
| SLASHt | 99 | |
| SLOWBLOCK | 98 | |
| Source | 30 | 38 |
| SOURCE | 97 | |
| Special functions in WIC | 72 | 89 |
| Stack | | 11 |
| Stack element | 100 | |
| Stack pointer | | 27 |
| Stacks, private | | 19 |
| Standard contents of TRANSFER block elements | 94 | |
| StandardErrorFn | 126 | 143 |
| StandardGiveUp | 12 | 20 |
| StandardPM | 15 | 22 |
| START | 100 | |

| | Txt | Com |
|---|---|---|
| State | 30 | 39 |
| STATE | 97 | |
| StateBFPT | 45 | 57 |
| StateTT | 39 | 48 |
| STATUS | 101 | |
| STOPCH | 94 | 91 |
| Storage allocation (core) | 23 | 29 |
| Storage allocation (disc) | 56 | 70 |
| StoreCode | 72 | 90 |
| STR | 97 | |
| STREAM | 99 | |
| Stream elements used by InitiateTransfer | 97 | |
| Stream primitives | 30 | 38 |
| Stream vector constants | 97 | |
| STREAMABLE | 101 | |
| StreamError | 30 | 40 |
| Streams from files | 58 | 74 |
| Streams (see II:4.1, II:5, II:7, III:2, VIII:6) | | |
| String | 108 | 126 |
| Style | | 4 |
| SUMCH | 100 | |
| Sumcheck | 73 | 94 |
| SUMCHINHIBITED | 95 | 111 |
| System constructing | 75 | 123 |
| System entry-point | 75 | 98 |
| System housekeeping | | 120 |

| | Txt | Com |
|---|---|---|
| System index | | 119 |
| System set-up | 75 | 97 |
| SystemIndex | | 119 |
| TBLESIZE | 98 | |
| Teletype | 37 | 45 |
| Teletype character code | 99 | |
| Teletype stream constants | 98 | |
| TerminateRun | 3 | 9 |
| Text of the system | 1 | 8 |
| THISFILE | 100 | |
| TIME | 95 | |
| TimeofDay | 51 | 64 |
| TIMEOFDAYCLOCK | 95 | 112 |
| TRANSFER | 94 | 91 |
| TransferIn | 74 | 96 |
| TransferInC | 74 | 96 |
| TransferOut | 74 | 96 |
| TrapPageThrow | 125 | 141 |
| Traps (see Permissions) | | |
| TryAgain | 45 | 57 |
| TryDiadAgain | 50 | 61 |
| TT | 86 | 107 |
| TTREAD | 95 | 92 |
| TTWRITE | 95 | 92 |
| TurnPage (AddVectoFile) | 119 | 135 |
| TurnPage (OuttoFile) | 63 | 80 |
| TYPE | 100 | |
| Types of file | 101 | |
| UB | 96 | |

| | Txt | Com |
|---|---|---|
| ULMASK | 99 | |
| UNDEFINED | 94 | |
| Unload | 8 | 14 |
| UNRESTRICTED | 101 | |
| UPARROWt | 100 | |
| Update | 66 | 82 |
| UpdateDiscVectorElement | | 3 |
| UpdateHead | 67 | 83 |
| UpdatePermission | 111 | 129 |
| User | | 125 |
| User details | | 124 |
| Validation | 83 | 105 |
| Vector allocation | 23 | 29 |
| VectorfromFile | 116 | 133 |
| VectortoFile | 116 | 133 |
| VTABt | 100 | |
| Wait | 53 | 66 |
| WIC | 72 | 89 |
| WordsfromDiads | 47 | 60 |
| Write | 35 | 43 |
| WriteAddr | 35 | 43 |
| WRITEBLOCK | 98 | |
| WriteByte | 35 | 43 |
| WriteN | 35 | 43 |
| WriteO | 35 | 43 |
| WriteS | 31 | 41 |
| Written IC | 72 | 89 |
| Wrong | 82 | 104 |
| XOFFt | 100 | |