ABSTRACT

Interactive computer systems do not provide users with the full benefits of interaction for program development. In general, the command/programming language distinction, carried over from non-interactive systems, separates the user from the program, with the program manipulated indirectly through the command language. This, coupled with the lack of interactive program configuration and testing, restricts unnecessarily program development. Programs are frozen by compilation and can only be configured statically through program text editing. Thus, program development requires a constant movement through explicitly invoked system programs to edit, compile and run test configurations.

"Interactive BCPL" attempts to overcome these restrictions by providing an integrated program development system for BCPL. The "command language" is an extension of BCPL (EBCPL) within which there is no difference in status between BCPL and system commands. Small EBCPL programs are entered from a terminal for immediate execution and may be used to construct and manipulate a test environment for a large BCPL development program. System commands are provided for the manipulation of development program text and to interface with the host computer system. "Interactive BCPL" is based around an interpreter which is compatible with the BCPL compiler and may also be used to test and configure pre-compiled BCPL subprograms.

I hereby declare that this thesis has been composed by myself; that the work of which it is a record has been done by myself; and that it has not been accepted in any previous application for any higher degree.

Gregory J. Michaelson

I hereby declare that the conditions of the Ordinance and Regulations for the degree of Master of Science (M.Sc.) at the University of St. Andrews have been fulfilled by the candidate, Gregory J. Michaelson.

A.J.T. Davie

INTERACTIVE BCPL

GREG MICHAELSON 1981

CONTENTS PAGE

0 INTRODUCTION

"Interactive BCPL" is intended to provide a flexible environment for modular program development. It is based around a BCPL interpreter executing an extension of BCPL (EBCPL). In addition to BCPL, EBCPL provides system and editing commands to manipulate a BCPL development program held in the system 'input' file. Within EBCPL, there is no distinction in status between system and BCPL commands: system commands may be used in the same context as BCPL commands.

The system provides direct execution of EBCPL programs entered from a terminal. These may be used to configure and test the BCPL development program. Each EBCPL program may access variables declared by preceding programs. Programs may also access external precompiled subprograms.

The system is written in BCPL and runs under the RAX operating system on the IBM 360/44.

This thesis consists of four chapters :-

1 MOTIVATION

- the limitations of current interactive systems for program development are discussed and suggestions are made for a system where the command language and programming language are integrated.

2 DESIGN

- the suggestions from the previous chapter are amplified to

provide a basis for an implementation of a BCPL program development system.

3 IMPLEMENTATION

- the implementation of the compiler, interpreter, file handling, editing and other system facilities are discussed.

4 CRITIQUE AND CONCLUSIONS

- the implementation is considered in the light of the motivation and design requirements; modifications and improvements are suggested.

APPENDIX 1 - The user guide to Interactive BCPL - contains a description of the facilities provided by and the use of the system. It also contains examples of system use.

Syntax is described using BNF with "option brackets"
i.e. [<construct>] => <construct> may be repeated 0 or more times.

I have tried to maintain production name consistency with the BCPL definition [*2].

I would like to thank Mr B.Mitchell for his assistance with modifications to the BCPL run-time interface, various Senior Honours students for trying out the system and my tutor, Mr A.J.T.Davie, for his help, encouragement and extraordinary patience.

The practical work was carried out in the academic year 1974/75 and was funded by an SRC grant.

1 MOTIVATION

Interactive facilities on small and multi-access computers are steadily displacing off-line batch and sequential systems. While they provide superior program development facilities they retain many features of off-line systems and fail to take full advantage of interaction. In particular, the distinction between the user programming language and the system command language is maintained.

Users communicate with interactive systems through simple command languages. These are used primarily to run system processes to carry out pre-determined sequences of operations on user files. Programs are manipulated by "remote control" through the command language which usually consists of single word mnemonics with file-name parameters and flags or keywords to indicate options. Typically, the system accepts only one command at a time or a macro call which executes a fixed sequence of commands. The only truly interactive facility on most systems is the editor, but this usually runs as an independent subsystem entered through a system command.

Special interactive languages have been developed e.g. BASIC, APL but these do not provide the power of high-level languages and are highly constrained by their implementations. Again, they run as independent subsystems within which the distinction between system and language commands is retained.

I regard this distinction between system and programming language as significant for the constraints it places on high level

language program development. Suppose a program is being developed by a modular approach. The problem is analysed top-down to identify a hierarchy of processes. Each process will be embodied in a subprogram. Ideally, for the development of each process at a particular hierarchical level the corresponding subprocesses are thoroughly tested and configured to form that process. Thus, development proceeds in a bottom-up manner, starting with the lowest level subprocesses which usually handle basic I/O and data structure manipulation.(Development could equally well be top-down with dummy subprogram calls inserted for undeveloped subprocesses at each level.) If the program has a simple tree structured process hierarchy then each process can be tested in isolation. For many programs, however, this approach is inadequate: there may be links between processes across a level or from a lower to a higher level. Thus, testing a particular process may necessitate the presence of other processes at the same or higher levels. The size or complexity of the program may preclude the simultaneous development of mutually dependent processes and this is often circumvented by simulating the effects of other processes by creating a suitable test environment. This involves either writing simple test programs to simulate the gross effects of the missing processes or hurling a large representative data set at the process under development. None the less, this approach enables structured and coherent program development and the hierarchical separation of relatively independent subprocesses eases program modification and development.

Using the development approach outlined above in a rigorous

fashion would suggest keeping distinct subprograms physically separate. This would lead to an initial proliferation of small files, test programs and system commands used for editing, compilation and execution. Consequently, it's more common to start subprocess development at a higher level with the corresponding lower level subprocesses held in a common file and configured in a single test program. While at first this reduces compilation and editing, as errors in several subprocesses may be identified and rectified at the same time, it may increase compilation and editing for the test program as it has to provide a more complex test environment. A test program substituting for dependent processes loses its value if its development does not involve less work than developing the dependent processes. In any case, as the complexity of the test program increases during program development, so does the time spent developing and debugging it.

On off-line systems, the value of this approach is reduced by the restricted user environment. With long turn-around times testing individual processes is inefficient. While file systems and editors are often available, trying to use them to debug programs is a frustrating, fiddly business as the user has no direct access to the file and cannot see immediately the effects of manipulations. In my experience, off-line system users tend to write and test a large section, (if not all) of a program at once and edit it manually off-line.

Interactive systems bring the advantages of direct access and fast response. The user knows about syntax or run-time errors

within minutes of compiling or running the program, and can promptly edit and rerun it. This gives greater continuity to program development but the system is still in between the user and the program. For interactive system users, the time spent in using system processes becomes more significant. A minor change to a program may involve calling the editor, compiler and linking loader before it can be re-run. These activities were necessary with off-line systems but the user didn't have to enter an idle wait state while each finished. An interactive call to a compiler may be relatively fast but 2 or 3 minutes is not long enough for independent purposeful activity beyond annoying other users. Furthermore, while interactive systems usually provide interactive program I/O, the program and its test environment are frozen by compilation and cannot be reconfigured when the program is running. This can be alleviated by steering the test environment through interactive I/O, by providing what is in effect a test command interpreter, but this again increases the size and complexity of the test program.

An interactive system should not only provide interactive communication with a running program but also the ability to configure it and its test environment interactively. The test environment should enable the the interactive declaration of test variables, routines and structures and the interactive running of commands to initialise and modify declared objects and to call and configure the development subprograms. Thus, the system should accept a sequence of test programs, each consisting of declarations and/or commands and able to access variables declared by preceding programs. This ability to develop

incrementally the test environment would result in the use of several small test programs rather than a single large one. Testing would consist of a series of small steps with the user's decisions about the form of each step being informed by the effects of the previous one and each step would have access to the results of preceding steps. The system should also provide system facilities for manipulating test program text and to interface with the host operating system.

Program development might then proceed in the following manner :-

i) interactively test the low level subprograms

ii) compile the low level subprograms and interactively test configurations of their code

iii) compile the configuration and proceed to the next level

iv) repeat the process until the program is complete

This description begs the question of the language implementation. I assume that the program will be run eventually as compiled code but that interpretive facilities may be more appropriate for some stages of program development. The relative merits of compilation and interpretation are discussed in chapter 2 section 5.

As a substantial part of program development is spent in program testing, I thought that this required facilities to interactively develop and modify the test environment. Providing a test environment involves writing a test program, usually in the same language as the development program: this suggested providing

interactive facilities for that language. The constant movement from editing to compilation to execution and the minor editing often required suggested the provision of editing facilities that did not involve entering a separate subsystem. During program development, system commands are implicitly intended for that program and this suggested system commands that did not require the program to be explicitly named. A system with these facilities would accept language or system commands without the need to enter distinct program or system states. This suggested giving all system commands the same status as language commands, with the additional possibilities of writing language routines that were in effect system macros e.g. generalised editing routines. These vague thoughts and their implications are spelled out in more detail in the next section.

2 DESIGN

1 Why BCPL?

While BCPL has many features which make it suitable for the sort of implementation I envisaged, I chose it primarily because I was interested in it. I first used BCPL as an undergraduate to implement a "buddy" storage allocation algorithm [*1]. I was impressed by its simplicity and power; it's an elegant and efficient language if treated with respect though somewnat prone to incomprehensible, catastropnic errors. When I came to St Andrews in 1974 BCPL was used as an undergraduate teaching language and to implement a variety of programming languages: this provided further stimulus.

BCPL is well documented elsewhere [*2]. Here I will consider those aspects I consider relevant.

The response time of an interactive system is, I think, the main factor determining its usefulness. An interactive language processor checks and executes programs in real-time. Most high-level languages provide a range of data types and constructs with specific operations and hence restrictions on their combination and use. These require appropriate checking and run time environments which can be complex (e.g. for structures requiring heap storage) and time consuming (e.g. for address calculations, run time type matching). BCPL is typeless and by comparison has a simple, generous syntax. The only BCPL type is the bit pattern which may be treated arbitrarily as a number, character, bit pattern or routine/function/vector/word/string address. A wide

range of operators are provided but they are not type specific. Thus, the onus is on the user to ensure that appropriate operations are carried out on objects: the language processor does not check subexpressions relative to the corresponding operator other than to establish precedence and to make sure that the requisite number of operands are present.

Similarly, while expressions are effectively typed by the operations they contain, all operations apply to all objects and so there are few restrictions on the construction or use of expressions. The exceptions are to prevent the use of meaningless addresses on the left hand side of assignments and the use of expressions containing variables in SWITCHON commands and the declarative initialisation of variables. The latter restriction enables preprocessing of such expressions by the compiler so that they can be replaced by constants. (This is not necessary: an intelligent compiler could recognise and optimise constant expressions as they occur and still generate code for other expressions.) This restriction also reduces BCPL's flexibility and increases checking - see 2.4.

The BCPL vector (VEC) can be identified as a distinct data structure but it is extremely simple, consisting of a word pointing at a word sequence constituting the vector proper. As a vector is inherently one-dimensional, run-time addressing only involves indirection and addition. There is no subscript checking as memory can be accessed arbitrarily through indirection on arithmetic expressions rather than solely through vector names. The user must explicitly construct complex structures from

vectors and ensure that addressing is correct.

BCPL has no heap storage: at run-time space is allocated on the stack, though programs can organise off-stack storage through address manipulation. This is a disadvantage for many applications, particularly the construction of linked data structures of variable size, but reduces implicit run-time storage overheads.

Thus in BCPL, checking and run-time maintenance are minimised and I thought this was advantageous for an interactive implementation.

The other BCPL feature I considered significant was the GLOBAL vector mechanism. This is a fixed memory area which may be accessed by independent program modules. Each GLOBAL vector location is numbered and variables associated with the same GLOBAL location number by GLOBAL declarations in different programs are mapped onto the same word in memory. This is used for module linkage, particularly to the BCPL standard library which contains I/O and utility routines. I thought this mechanism could be used to enable the interactive configuration of pre-compiled subprocesses, with the system mapping interactive GLOBAL variables onto the corresponding compiled GLOBAL vector locations. It could also be used for interactive access to the standard library.

2 Making BCPL interactive

In compiled BCPL, a program consists of a sequence of declarations. The program "proper" is embodied in a routine

associated with GLOBAL location 1 and initiated by the run time interface when the code is loaded and run. This will be referred to as the main module.

A routine declaration has the following form :-

LET <name>([<parameters>]) BE <body>

where

<body>::=$([<declaration part>;][<command list>] $)

For the main module, the <name>, usually START, is associated with GLOBAL location 1 and the <parameters> are used to pass it information when it is run.

This format could be retained for an interactive implementation with the system automatically running the most recent main module version, but this would be extremely inflexible. As suggested above, an interactive test environment requires facilities for incremental development, with the user running small programs consisting of declarations and/or commands. This suggested making the system appear as if the user were always inside the main module by retaining the form of a routine body for interactive programs :-

<interactive program>::=<declaration part>_¦
<command list>_¦
<declaration part>;<command list>_

Thus on first entering the system, the user would appear to be just to the right of the main module's opening "$(" and after

each interactive program had been run, to the right of the last <declaration> in that program. In this way, from the BCPL definition, each program could refer to variables declared by preceding programs :-

```
Initially => $(

Program 1 = <declaration part1>_

          => $( <declaration part1>

Program 2 = <command list2>_

          => $( <declaration part1>
                <command list2>

Program 3 = <declaration part3>
            <command list3>_

          => $( <declaration part1> - modified by
                                      <command list2>
                <declaration part3>
                <command list3>

Program 4 = <command list4>_

          => $( <declaration part1> - modified by
                                      <command list 2 & 3>
                <declaration part3> - modified by
                                      <command list3>
                <command list4>

 etc.
```

A BCPL variable is realised through the association of a name and a word. The word may point to a sequence of words, a structure made up of word sequences, or code for a routine. Memory space is allocated in a static area at compile time to STATIC variables, strings and routine code, and on a stack at run time to dynamic variables. Providing incremental facilities requires that names of and space allocated to variables declared by each program are available to subsequent programs.

From the BCPL definition, blocks may declare variables but they

are only in scope in the declaring and enclosed blocks. Thus, it is only necessary to retain names of variables declared at the outer level of each program. This requires a data structure containing the names and associated addresses of each program's outer level variables which is searched if a name reference is not satisfied by the current program's declarations. If all programs add information to this structure, with the information for inner blocks being removed on block exit, then when a program terminates, its outer level variable information may be left in place. The information for each program is added in a stack regime and this suggests organising the structure as a push down stack which is searched top-down when variable references are encountered. If all variable references are carried out through this structure, routines declared by preceding programs which refer to redeclared or reassigned variables will subsequently access the new version, avoiding recompilation of interdependent routines when one is changed.

The extent of space allocated to dynamic variables is the declaring and enclosed blocks, so it is only necessary to retain stack space allocated to outer level dynamic variables. When an interactive program terminates, its outer level dynamic space will be on top of the stack. If the stack top is not then reset to the stack base then this space will be available for subsequent programs, whose dynamic space will be above it. This is consistent with compiled BCPL, where modules loaded together share the stack.

The extent of static storage is the whole program and this

implies that all static space allocated to a program should be available to subsequent programs. In compiled BCPL, independently compiled modules have independent static areas. If interactive programs had static space allocated from a common static area then this allocation would again follow a stack regime. When each program terminated its static space would be "on top of" the static area and could be retained by not resetting the static area top to the static area base prior to the next program.

3 Language processor

The design of the language processor was largely determined by the desire to complete its implementation quickly and to expedite this by using all or some of the the existing standard machine independent BCPL compiler as a basis. Having made this pragmatic decision the fundamental design consideration was available space. Equally important but less determinant considerations were fast response, good diagnostic facilities and consistency with compiled BCPL.

The BCPL compiler is written in BCPL and this made it sensible to implement the whole system in BCPL. At the time of implementation, there was no provision for BCPL access to RAX files or for overlays and thus the whole system would be core resident. The BCPL compiler has 3 phases :-

i) syntax analysis - the context free program constructs are checked and a parse tree constructed

ii) OCODE generation - the tree is traversed, context sensitive

constructs are checked and OCODE - code for an abstract BCPL machine - is generated

iii) code generation - OCODE is translated into the machine code for the host computer

At first, it might seem that an interactive incremental compiler would provide the most efficient implementation. In general, compiling and running code for a program will be faster than interpreting the program. An interpreter will execute more code for each command than will be produced by compilation. Furthermore, the overheads for checking the program interpretively will at best be the same as for compilation. If, however, the program consists of a short simple sequence of commands then interpretation may be faster as checking and execution can be carried out in one phase without the need to link and load code. Interpretation becomes relatively slow for repetitive command execution, block entry and subprogram calls. In the system under consideration, I thought that test environment programs would generally be small and simple relative to the development program, with the development program consisting of a set of routine declarations. This suggested interpreting test programs but generating and running code for subprograms, blocks and repetitive command bodies [*3].

For several reasons, alas, the use of the whole compiler was discounted: the size of the BCPL compiler meant that little room would be left for an interpreter and system facilities; it is difficult to pinpoint source program errors from compiled code; considerable modification to the compiler would be required to

provide outer level interpretation; the run-time environment would have to be extended to enable incremental programming, in particular for reference by name to previously declared variables and for space reclamation. Another possibility was to interpret OCODE but this was ruled out for similar reasons.

I decided to use the syntax analyser to compile programs to parse trees and to provide a separate tree interpreter. The syntax analyser is relatively small and the changes required were minor. The parse trees are smaller than source text but may be used to reconstruct text. During interpretation, this could be used to trace execution and provide detailed error information. The RAX BCPL compiler and the GLOBAL vector mechanism enable the compilation of tested programs which can then be interactively configured. It would also be possible to save program trees for subsequent translation by the compiler OCODE stage.

The main disadvantage of this decision was the loss of OCODE stage checking, particularly for variable declarations and references. I did not want to modify the syntax analyser to carry out all such checking as this would have involved an effective re-write. I decided to add subpasses to carry out some checking (e.g. for correspondence between the left and right hand sides of assignments and dynamic declarations) but to carry out the rest in the interpreter. This meant that in many instances (e.g. recursive routine calls, repetitive blocks) variable checking would be carried every time the reference was encountered with attendant speed loss.

The design of the language processor is described in greater

detail in chapter 3.

4 Interactive and compiled BCPL

For the system to be of practical use, consistency with compiled BCPL was essential so that developed programs could later be compiled, and for communication with precompiled modules. Changes to BCPL should be avoided as far as possible and should be extensions or restrictions rather than revisions. Additional features should be clearly identifiable so they may be removed from developed programs before compilation.

I decided to implement full BCPL with only two major changes. As noted above, BCPL forbids the use of expressions containing variables in dynamic declarations and SWITCHON commands. While this enables efficient compilation it prevents dynamic VECtors and often results in dynamic declarations with dummy defining expressions followed by initialisation assignments. The checking for this restriction was carried out by the OCODE stage. Rather than transferring this to the interpreter, or adding a sub-pass to the syntax analyser I decided to drop the restriction to decrease checking and add flexibility.

The other major change was non-implementation of the GOTO command. The association of GOTO's with labels was carried out by the OCODE stage and would have been transferred to the interpreter. This would have involved active tree searching on encountering a GOTO for the corresponding labelled node or a prepass to pick up and declare labels. Although such checking is reduced by the BCPL restriction of GOTO use to the block

containing the label, I decided to omit GOTOs. For similar reasons, the SWITCHON command was restricted so that CASE commands must appear at the outer level of the SWITCHON body. This minimises tree searching for CASES during SWITCHON execution.

Additions to BCPL for system facilities are described in section 2.7.

Behavioural consistency was required for object representation and manipulation, and evaluation order. BCPL is well documented[*2] and this information was used to try and ensure that the implementation was consistent with compiled BCPL. Using BCPL as the implementation language also helped maintain consistency: many commands are interpreted by the corresponding command containing calls to interpretive routines so evaluation order is determined by the standard compiler.

e.g. <test command> ::= TEST <expression> THEN <command1>
ELSE <command2>

is interpreted by

```
TEST INTEXPR(E) THEN INTCOM(C1)
               ELSE INTCOM(C2)
```

where E - <expression>'s subtree
C1 - <command1>'s subtree
C2 - <command2>'s subtree
INTEXPR - interprets an <expression>
INTCOM - interprets a <command>

All interactive addresses are absolute and thus indistinguishable from those in compiled code, so, for example, structures built interactively may be passed to precompiled routines.

5 Development and test programs

As noted above, I distinguished between the program under development and the test programs. The development program would consist of a sequence of declarations; in particular, the routine declarations corresponding to the current level of subprocesses. It would be relatively large, might require substantial changes, would be run several times and would eventually be used outside the system. The test programs would consist of snort sequences of declarations and/or commands used to modify the test environment, structure and test subprocesses and control the system. They would not be used outside the system, would not be run very often and while individually small their total text might well exceed the development program's. Rather than retain text and provide editing facilities for all programs, I decided to do so only for the development program. This would minimise storage of unnecessary text and encourage the use of small test programs whose size would not preclude complete re-entry in the event of errors or changes. Test programs for repeated use could be embodied in routines and, if subject to much change, held as part of the development program.

All input to the system would be extended BCPL programs for implicit immediate execution. Development program text would initially be input and held without being run, under the control of an interactively entered command. Running the development

program, using another command, would declare the routines for subsequent interactive manipulation. Modifications necessitated in the light of interactive testing would be carried out on the text, which could then be re-run.

6 File and editing facilities

In the naive hope, shared by many aspiring research students, that my project might be used by other people I thought that consistency with RAX, the operating system running on the St Andrews' IBM 360/44 [*4], would make the system familiar and easy to use. Pragmatically, RAX facilities are easy to mimic and are oriented towards single program development.

In RAX, system commands operate on a nameless "input" file. The file may be filled with sequences of system commands, program texts and data which may then be executed. RAX files may be copied to/from the "input" file and the "input" file may be edited. Page (screen) editing facilities are provided for IBM 2260 terminals: lines of text are displayed on the screen for modification by cursor control. On CDC 713 terminals there is an extremely clumsy line editor with poor facilities for seeing the text during editing.

I decided to provide an "input" file for development program text with RAX like facilities to load it from a terminal or the RAX "input" file, insert text at the file bottom, display text, save text for subsequent RAX access and run the program. The lack of BCPL access to RAX files precluded more elaborate RAX file access.

The editing facilities I thought necessary were both line and text oriented. Program text is laid out on lines and it's common to have only one command on a line. This makes the line a convenient text fragment for user access. Lines are clearly delineated in text files by new-line symbols and this makes the line a convenient text fragment for the system to handle. This suggested providing commands to find, delete, replace or insert text after a specified line. As it may be necessary to examine text before modification, and this involves displaying lines of text on a screen, this suggested retaining page editing for the modification of displayed text. On the other hand, errors often occur which can be readily corrected in a textual context (e.g. miss-spelled or missing names) without line by line text examination. This suggested commands to find, delete, replace and insert a string after a specified string in context, without the need to enter a separate mode or examine text.

7 System commands and BCPL

There were some problems integrating system commands into BCPL. Either the routine call syntax could be used or new commands added. Routine call syntax is long-winded, requiring brackets round parameters and commas between them. Empty brackets must follow unparameterised routine calls to distinguish them from other references to the variable associated with the routine. I thought that commands should be simple to minimise potential errors and make them quick and easy to use. Routine call syntax would also involve either checking every routine call to see if it was a system command call or accessing system commands through the routine call mechanism. It seemed better to introduce new commands with simple syntax which could be easily identified during compilation.

BCPL has single word commands and this format was used for parameterless system commands. For parameterised commands conflicts arose. I wanted these commands to be general purpose (e.g. the edit 'replace' command could have a string or line-number as parameter) and to accept general BCPL expressions as parameters (e.g. to call a system command with a string manipulating expression as parameter). Unfortunately, BCPL's typelessness makes these aspirations contradictory. While it's possible to distinguish between numbers and strings declared explicitly, as the corresponding tree nodes have distinct tags, it isn't possible to tell whether an arbitrary integer value represents a string address or a number. Thus general purpose parameterised commands could not have arbitrary expressions as

parameters, as the parameter type, and hence the command requirement, would be indeterminate. This could have been circumvented by providing distinct command words for each case (e.g. to distinguish string replacement from line replacement) but I decided to retain generalised edit commands with explicit parameters to reduce the number of commands and the possibilities of classic BCPL blunders with system command calls.

For system commands, RAX command words were retained where appropriate and new single word commands were introduced. For the edit commands, where there was no equivalent RAX command word, I decided to introduce a new two letter word starting with 'E' (for Edit) and followed by a single letter denoting the corresponding activity. While I wanted short command words, single letter commands would reduce the number of single letter variables, which are often used for temporary storage and control. For command parameters, BCPL integers and strings were retained.

8 Space reclamation

In BCPL, space is allocated dynamically from the stack and statically from the static area. As noted above, the system was to provide incremental facilities with objects constructed by programs available to subsequent ones. Static space and stack space allocated to outer level dynamic variables were to be retained after each program terminated. Some means of space reclamation was required, as the repeated recompilation of development subprograms would gobble up stack and static area space.

Tne best facility would be automatic garbage collection. However, space is allocated to BCPL programs in varying quantities (e.g. tree nodes may have between 2 and 7 fields) and I thought that providing a general purpose storage allocation system was too substantial a task. Another possibility was reclamation of space associated with user specified variables but this would lead to fragmentation or require compaction.

The static space for each program is allocated sequentially on top of the static area and its outer level stack space is on top of the stack when it terminates. Thus, space for programs is allocated in a "stack" regime. This suggested reclaiming space for entire programs starting with the most recent and working backwards in the reverse order to that in which they were run. I decided to provide commands to do this and to display the outer level variables declared by each program, to inform reclamation.

9 Diagnostic facilities

A central requirement of a program development system is the provision of facilities to identify the nature and site of errors and, for run time errors, the preceding sequence of events. In my experience, diagnostic facilities provide either too little or too much of the wrong information, particularly for run time errors. A common user practise is to ignore the facilities provided by the system and to trace truant programs by using the language output facilities to provide information about changes to particular variables.

During compilation, errors should be notified by an intelligible

message which indicates the construct causing the error. The BCPL syntax analyser outputs the text enclosing the error site with a message. More precision is not possible as errors are usually detected during the parse of subsequent text. Some of the messages are rather cryptic e.g. they say that an error has occurred in a command without identifying the command. The error recovery is somewhat primitive, consisting of skipping through the input until an end-marker for the enclosing construct is found, and this tends to propagate errors. As I did not want to modify the analyser substantially I decided not to try to make analysis more intelligent but to make the messages more informative.

The BCPL compiler offers the options of printing text during syntax analysis and outputting the corresponding parse tree, and these were retained.

The OCODE generator checks context sensitive constructs and identifies the sites of errors by printing the corresponding subtrees. The subtrees have internal symbol values at the nodes and are of little use unless one knows the correspondence between symbol values and constructs. Some of the OCODE checking was to be transferred to the syntax analyser so for these errors the enclosing text would be printed. Where checking was to be carried out by the interpreter I had intended to identify error sites by reconstructing the corresponding text from the parse tree, but I did not have time to implement this. As a first stage, I modified the parse tree printer routine to output meaningful nodes and used the parse trees for some interpreter detected errors.

Compiled BCPL provides a backtrace mechanism which, after a run time error, outputs routines, functions and associated dynamic variables in reverse call order. This is useful for identifying the routine in which the error occurred and the call sequence leading to the error but is generally not specific enough. In particular, the backtrace cannot identify any routines called at the same level as and prior to the error routine. After a run time error, compiled BCPL also produces a map of the program area giving names and addresses of routines and functions and current values of GLOBAL variables but this information is hard to decipher and relate to the program. This problem is specific to BCPL: as integer values have several interpretations the user must be aware of absolute addresses associated with variables and structures to make sense of trace information. Programming in BCPL, however, does not require the use or the knowledge of absolute addresses as they may be manipulated implicitly through the use of address operations.

Some languages (e.g. RAX ALGOLW [*5]) provide forward tracing of assignments and conditionals but I think that they are not selective enough in the information they provide. The best solution is that provided by SASL[*6] where the user marks salient program points for tracing, but this would have required major changes to the syntax analyser to identify a trace marker within different constructs.

I decided to introduce a simple output command - the "skeletal write" command - which would print strings and expressions. This could be used for tracing by inserting appropriate instances

round required program points in the same way that language output routines are often used. It has the form :-

<skeletal write> ::= *<list>

<list> ::= <item>[,<item>]

<item> ::= <expression>|<string>

In BCPL, a <string> is an <expression> but here I have identified them as separate constructs. Output for user tracing should not only print numeric values but also messages to identify them. In compiled BCPL, an explicit <string> is evaluated to give the address of the corresponding character vector but because BCPL is typeless this address cannot be distinguished from any other numeric value. In a parse tree, however, the subtree for an explicit <string> is identified by the node marker. This can be picked up when a <skeletal write>'s <list> is scanned and the <string>'s characters output without evaluation. Any other <item> would be evaluated and the final value output as an integer.

This command has the advantages in comparison with the standard BCPL output routines of free format and simple syntax. The '*' cannot normally be used in this context in BCPL and provides a simple marker for editing location and deletion.

I decided to retain the BCPL backtrace and to modify it to print the calling expressions as well as routine names. I thought that being able to see a sequence of events as it occurs rather than retrospectively often helps locate errors so I decided to add a forward trace which would print the calling expression and the

formal and evaluated actual parameters for each routine call.

It often happens that a faulty program does not respond within an expected time period and the user wants to halt it for inspection and modification. Alas, RAX does not provide facilities for user programs to detect keyboard interrupts. The most common cause of non-response is a non terminating recursion or an infinite loop but determining whether a program contains these is recursively undecidable. However, in a finite implementation an infinite recursion will use all available stack space ,causing a system error and control can be returned to the user. Detecting an infinite loop is much harder so I decided to place a limit on the number of times the body of a repetitive command may be executed and to allow the user to change the limit.

In compiled BCPL, information may be passed to the main program module when it is run by providing a string parameter. This is used in the compiler to select options; the string consists of a sequence of character flags which are decoded. I decided to retain the parameter string for option selection and to provide new commands to turn options on and off. I retained character flags for options corresponding to compiled BCPL and introduced new ones for the additional options. As system commands would have the same status as BCPL commands the trace selection commands could be placed round desired program positions and thus restrict tracing to a particular area.

Finally a command is needed for recovery after a major system error. The space reclamation command may be used to selectively reclaim space after stack or static area overflow but if, for

example, a language processor work area like the symbol table is used up it would be necessary to reclaim all space and start again.

3 IMPLEMENTATION

3.0 Overview

The system is written in BCPL and runs with the standard BCPL library and a modified interface. It consists of a tight loop which repeatedly outputs the prompt 'OK', calls the compiler to input, parse and generate a tree for program text and then calls the interpreter to execute the tree if the program is error free.

Examples of system use and the effects of particular commands may be found in the user guide - Appendix 1.

The compiler, interpreter, system commands and interface are discussed in the following sections.

3.1 The Compiler

The compiler is a modified version of the syntax analyser from the BCPL compiler. The analyser builds a parse tree from a BCPL program utilising top-down one-step-look-ahead recursive descent, and checks context free constructs.

In the analyser, nodes for the program tree are allocated from a fixed area. Were this retained, node space would quickly be filled. An EBCPL program usually consists of one or more routine declarations, or a sequence of declarations and commands to establish a test environment and test the routines. Only routines and strings declared at the outer level may be accessed by subsequent programs: only their subtrees need be retained after the program has (successfully) run.

Consequently, the analyser was changed to allocate subtree space for each program from a re-usable area, except when a string or routine declaration was encountered. Space for strings and routine bodies is allocated from the static vector because they are static objects in BCPL. This space may be reclaimed by the user [3.3.3.2, 3.3.3.3]. All tree space allocated for a program is reclaimed if it contains syntax errors.

In the BCPL compiler, context sensitive constructs are checked by the OCODE generator. The analyser was changed to check equivalence between the left and right hand sides of multiple declarations and assignments by an additional sub-pass which climbs the corresponding subtrees. Other checks are carried out by the interpreter [3.2.4, 3.2.4.4, 3.2.5.6.1].

The analyser was extended to handle the EBCPL extensions and changes to BCPL. In particular, the lexical analyser was extended to recognise system command reserved words; the <program> routine was changed to recognise

```
<program>::=<declaration part>_¦
           <command list>_¦
           <declaration part>;<command list>_
```

as a valid EBCPL program; the <command> routine was changed to recognise system commands and to call routines to build the appropriate subtrees.

The parse tree printing routine was changed to output EBCPL reserved words instead of internal symbolic values. I had intended to use the parse trees to reconstruct program text but this was not implemented.

3.2 The Interpreter

3.2.0 Introduction

The interpreter climbs a program tree by recursive descent(!), executing the declarations and/or commands and manipulating the declaration table, stack and static vector.

The declaration table is a 3*N stack which holds:-

i) the details of outer level variables

ii) information about programs' space allocation to enable space reclamation

iii) dynamic linkage for command execution

iv) static linkage for routine and function execution

v) details of variables declared in blocks and routines

vi) return and escape information for leaving repeatable commands and routines

Pointers are maintained to the top and base of the current declaration table frame.

The stack holds words allocated to dynamic variables and error recovery information. A pointer is maintained to the stack top.

The static vector holds words allocated to STATIC variables, strings, TABLES and the subtrees for the bodies of routines and functions. Space is allocated from the static vector in a stack regime and a pointer is maintained to the next free static vector

word. This pointer is referred to as the static vector top.

3.2.1 Environments

Commands are interpreted in environments. An environment consists of :-

i) dynamic linkage to keep track of the routine/block entry sequence

ii) error recovery and break information

iii) the variables declared by the command

iv) static linkage for a routine or function call to keep track of the variables accessible from the routine or function

Physically, an environment consists of the current stack and declaration table frames. Pointers are maintained to the current declaration table frame top and base, and to the current stack frame top. A stack frame base is not maintained as the stack is never actively searched. The declaration table dynamic link is a pointer to the previous declaration table frame base. The stack dynamic link is a pointer to the previous stack frame top, where the error recovery information for the previous environment is held. The declaration table frame base points to the entry for the linkage, above the previous frame top.

A new environment is entered for the execution of a block, routine call, repeatable or SWITCHON command. This usually involves :-

i) saving error recovery information for
the previous environment on the stack

ii) saving the previous declaration table base
and stack top as dynamic links
on the declaration table

iii) remembering the type of command about to be executed

iv) updating the declaration table frame pointers and
the stack top pointer

v) updating the error recovery information

For a routine or function call, the dynamic stack link is saved in a system variable and the static link to the routine or function's static environment, which consists of details of the variables external to but accessible from the routine or function [3.2.4.4.4], is saved in the declaration table instead of the dynamic stack link.

On leaving an environment, the declaration table and stack pointers are restored from the dynamic links and the error recovery information is restored from the stack. On leaving a routine or function call environment, the stack top is restored from the system variable.

The outer environment consists of the declaration table entries and stack and static vector space for outer level variables declared by successive EBCPL programs. Above the declaration table entries for each program is reclamation linkage consisting of pointers to the stack and static vector tops when the program

terminated, and a pointer to the reclamation linkage for the preceding program. A pointer is maintained to the reclamation entry for the most recent program [3.3.3.3].

In diagrams in the following sections, the static vector top and reclamation link, and the reclamation entry pointer are not shown.

3.2.2 Program execution

<program> ::= <declaration part>_¦
<declaration part>;<command list>_¦
<command list>_

The <declaration part> (if any) is executed. The <command list> (if any) is executed. If the program contained any outer level declarations then the (new) current stack and static area tops, and the reclamation entry pointer are pushed onto the declaration table and the reclamation entry pointer is set to the new reclamation entry. This information may be used to reclaim space allocated to outer level variables [3.3.3.2, 3.3.3.3].

Fig. 1

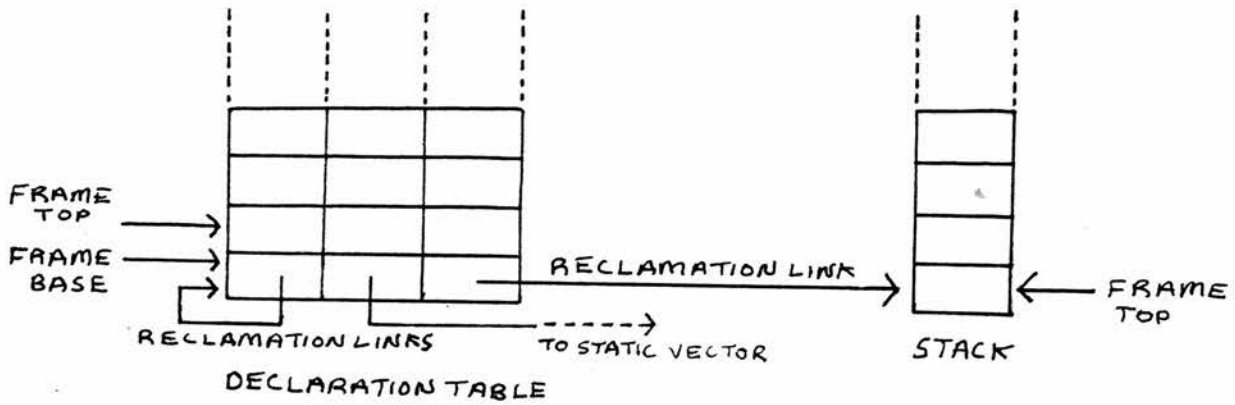

Figure 1 shows the declaration table and stack on entry to the system. The first entry in the declaration table is dummy reclamation linkage consisting of pointers to the stack base, static vector base (not shown) and to the entry itself. The reclamation entry pointer (not shown) points to this entry. The declaration table frame top points at the first free declaration

table entry. The stack frame top points at the first stack word.

Figure 2 shows the effect of the program :-

<program 1> => <declaration part 1>;<command list 1>_

Fig. 2.1

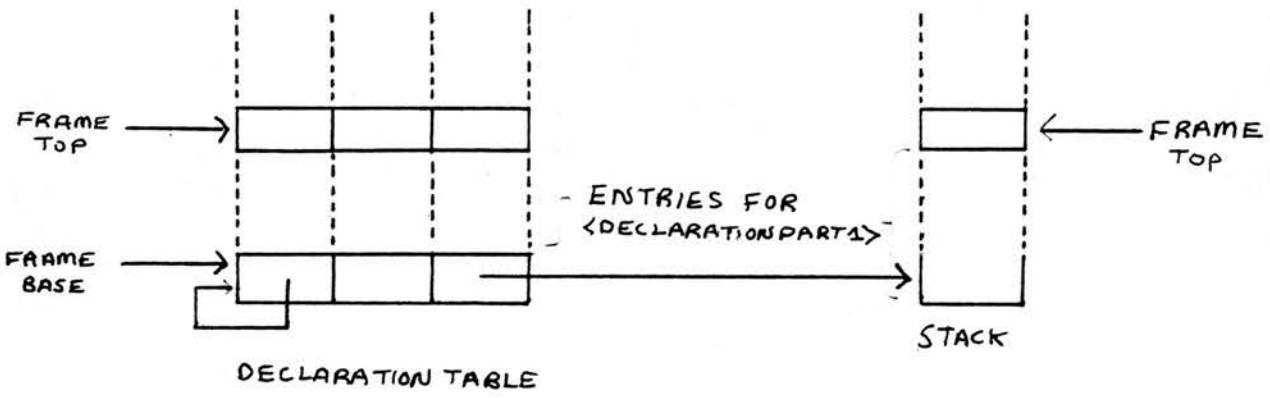

Figure 2.1 shows the declaration table and stack after the interpretation of <declaration part 1>. Between the declaration frame top and base are the entries for the declarations [3.2.4]. Below the stack top are the stack words allocated to the declarations.

Fig. 2.2

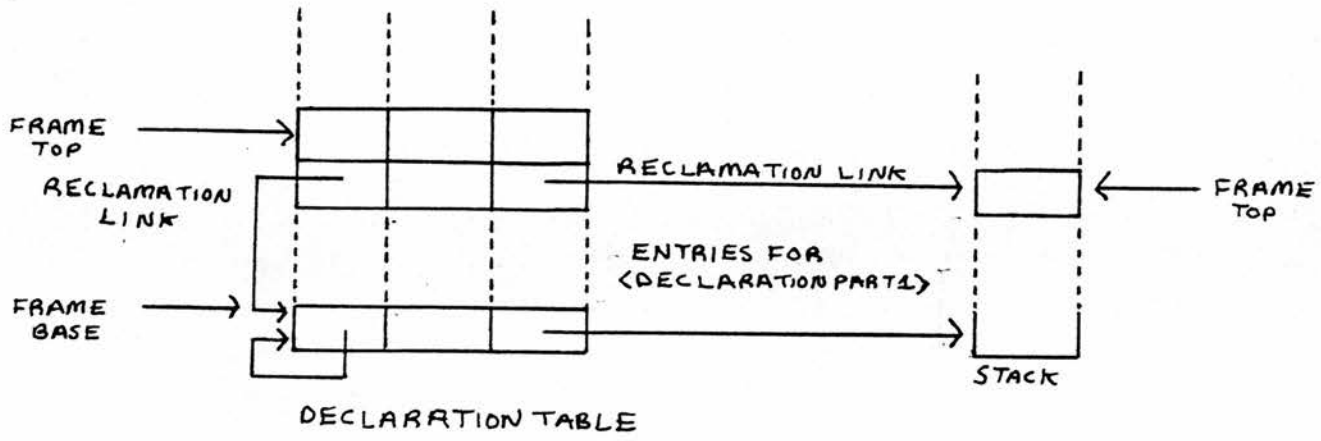

Figure 2.2 shows the declaration table and stack after <program 1> has terminated. Below the declaration table frame top is the reclamation entry with pointers to the previous entry and the stack and static vector (not shown) top. The reclamation entry pointer (not shown) points to the entry.

Figure 3 shows the effect of the program :-

<program 2> => <declaration part 2>;<command list 2>_

Fig. 3.1

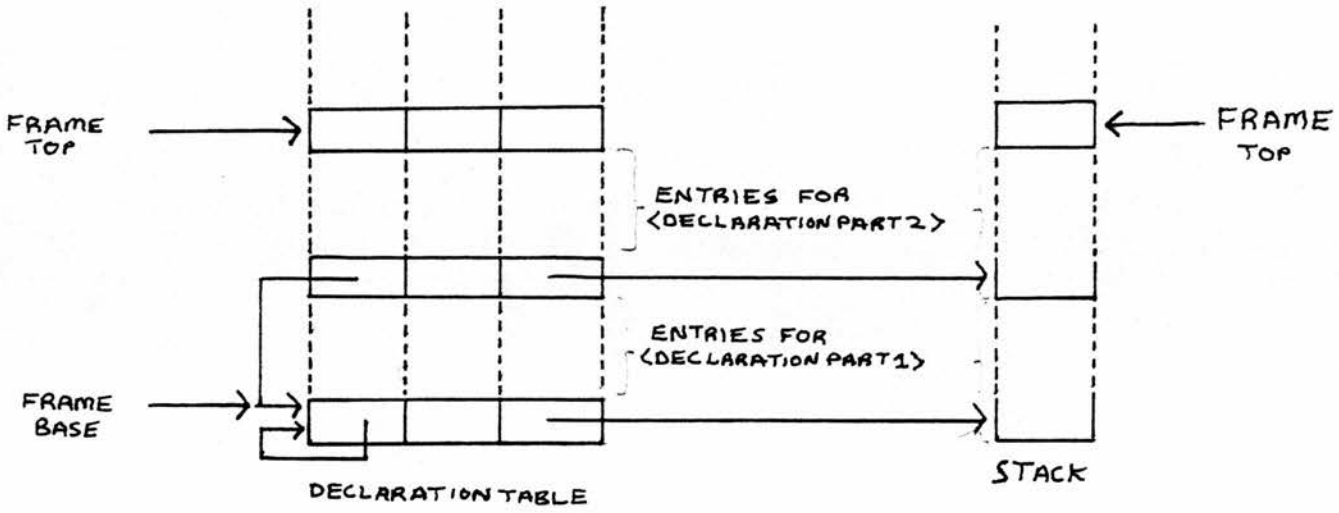

Figure 3.1 shows the declaration table and stack after the interpretation of <declaration part 2>.

Fig. 3.2

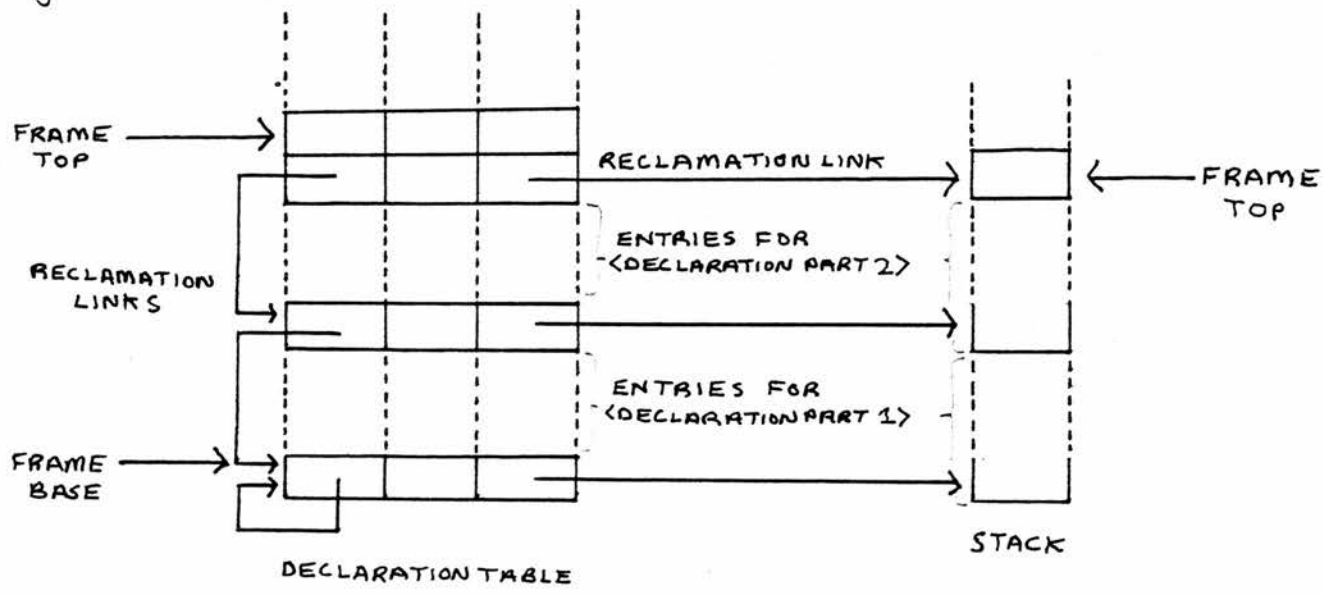

Figure 3.2 shows the declaration table and stack after <program 2> has terminated.

Note that EBCPL programs may not redeclare outer level variables created by previous programs. This would contradict the conception of successive EBCPL programs as successive additions of declarations to a single main module, as described in 2.2. The same effect may be achieved by re-assignment to outer level variables.

Note that the <program>s

<declaration part>;<command list>_

and

$(<declaration part>;<command list> $)_

will both declare outer level variables. The former leaves the variables declared by outer level declarations behind when it terminates.

e.g. LET A=1_

- declares A

The latter is a block which should not declare outer level variables: the <declaration part> is local and should be removed when the block terminates.

e.g. $(LET A=1 $)_

- should not declare A, but will...

This is because they have the same subtrees. Subtrees for blocks are not marked explicitly but consist of the subtree for the <declaration part> with the subtree for the <command list> in the last field of the node for the last <declaration>. In the OCODE stage of the compiler, if a <declaration> subtree is encountered

when a <command> is expected then a <block> is assumed.

In "interactive BCPL", the compiler consists of a call to the syntax analyser routine which expects to find a <block body>. This recognises :-

```
<block body> ::= <declaration part>¦
                 <command list>    ¦
                 <declaration part>;<command list>
```

In the last case it constructs a subtree as described above.

If the body is a single <command> consisting of :-

```
$( <block body> $)
```

then the routine skips '$(',calls itself, skips '$)' and returns the subtree as described above.

The point is that in BCPL, <declaration part>;<command list> can only appear in the context $(<blockbody> $). The EBCPL program $(<blockbody> $)_ should have been treated separately by my compiler.

3.2.3 Block execution

<block> ::= $(<declaration part>;<command list> $)

A new environment is entered, marked as a 'DEF(INITION)' entry point. The <declaration part> and <command list> are executed and the previous environment is restored.

Figure 4 shows successive stages in the execution of the program :-

```
<program 3> => <declaration part 3>
               $( <declaration part 4>
                  ...
               $)_
```

Fig. 4.1

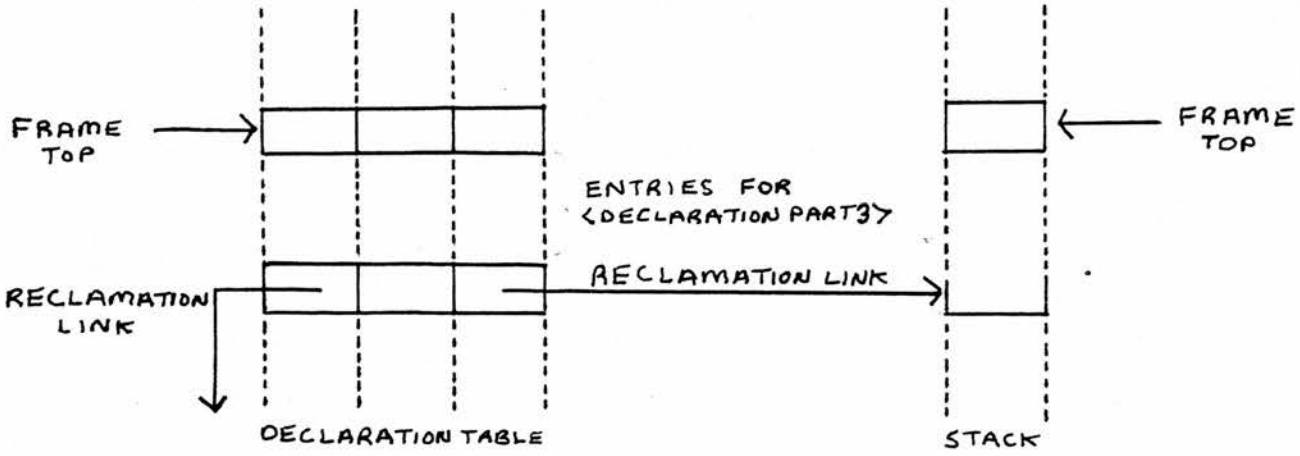

Figure 4.1 shows the declaration table and stack after the interpretation of <declaration part 3>.

Fig. 4.2

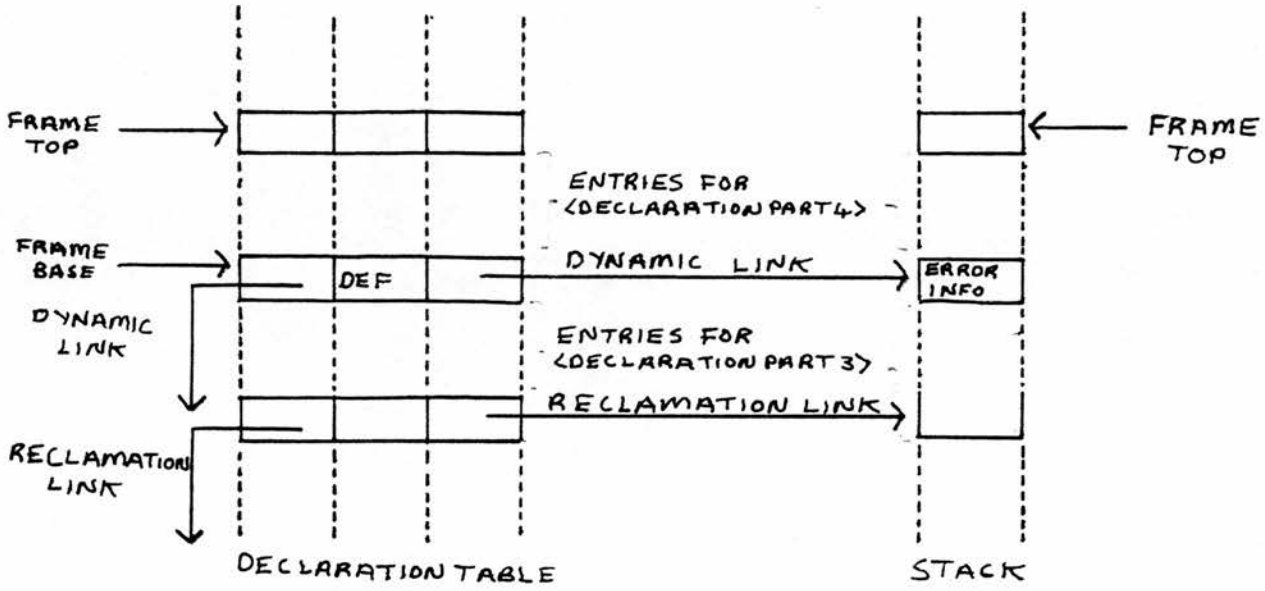

Figure 4.2 shows the declaration table and stack after the interpretation of <declaration part 4>. Above the declaration table entries for <declaration part 3> is the environment entry for the first block with dynamic links to the previous frame base and stack top. The stack dynamic link points to the error recovery information for the outer environment on the stack. The current frame base points at the environment entry. Above the entry are the entries for <declaration part 4>. Above the error recovery information on the stack is the space allocated to <declaration part 4>.

On leaving the first block, the error recovery information and the frame pointers for the outer level of the program are restored as shown in figure 4.1.

When the program terminates, a new reclamation entry is added to

the declaration table (not shown).

This implements block structured declarations because the declaration table is searched from top to bottom for names during expression evaluation. Thus the most recent declaration entry for a name will be encountered first

3.2.4 Declarations

3.2.4.0 Introduction

BCPL has 3 storage types for variables :-

i) Global - allocated space in global vector accessible from all program modules

ii) Static - allocated space in static vector accessible from declaring module

iii) Dynamic - allocated space on stack

The scope of global/static variables is the declaring and inner routines/ blocks and their extent is the whole program.

The scope and extent of dynamic variables is the declaring routine/block and inner blocks.

EBCPL also provides outer level storage: outer level dynamic variables are allocated static stack space so that they and their associated objects may be accessed by subsequent programs.

For all declarations, the current environment is searched for each declared name. If the name is found then the name is declared already.

Otherwise, the name, a storage type marker and the address allocated to the name (or a constant for a MANIFEST declaration) are pushed onto the declaration table. This will be referred to as "declared". The initial value for the variable, if any, is placed at the names's address.

Note that in all places where BCPL requires a <constant expression> (e.g. case expressions in SWITCHON commands, defining expressions in declarations) EBCPL will accept any <expression> allowing, for example, dynamic vectors.

3.2.4.1 STATIC declaration

<static declaration> ::= STATIC $(<name>=<expression> ... $)

The <expression> is evaluated and the next free static area word set to it. The <name>,'STATIC' type and static area word address are declared. If the declaration occurs in a routine/block then subsequent declarations will associate the <name> with the same static area location which will not be re-initialised.

Fig. 5

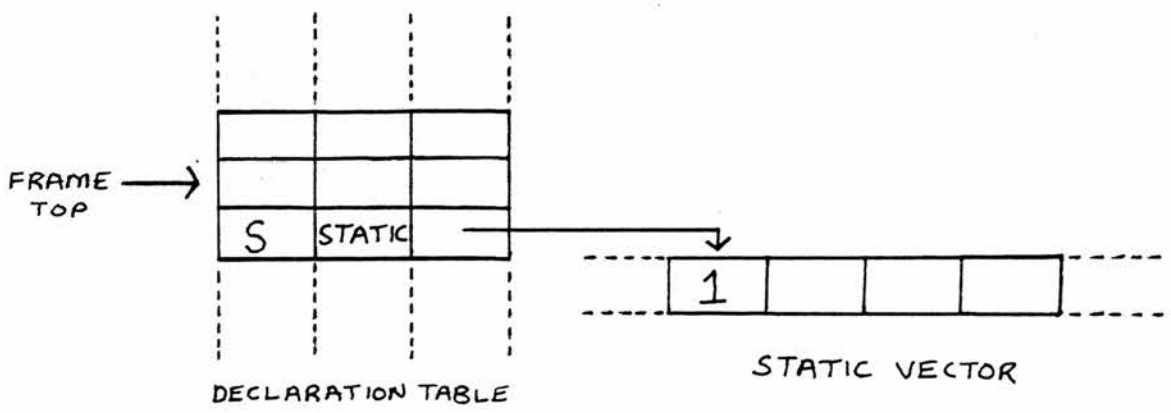

Figure 5 shows the effect of the declaration :-

STATIC $(S=1 $)_

The declaration table top is set to the name 'S', type 'STATIC' and the address of the next free static vector word, which is set to the initial value 1. The frame top is advanced.

3.2.4.2 GLOBAL declaration

<global declaration> ::= GLOBAL $(<name>:<integer> ... $)

Global declarations are mapped onto the global vector used by the system. This enables access to the BCPL library and pre-compiled program modules for interactive testing. It also proved extremely useful during system development as I was able to interactively manipulate system variables and routines.

The system uses global vector locations from 300 up . Attempts to access these outwith system development through global declarations are faulted. (By suitable manipulation of addresses it is still possible to access them, alas.) The <integer> is checked for illegal access and the <name>, 'GLOBAL' type and corresponding absolute global vector address are declared.

Fig·6

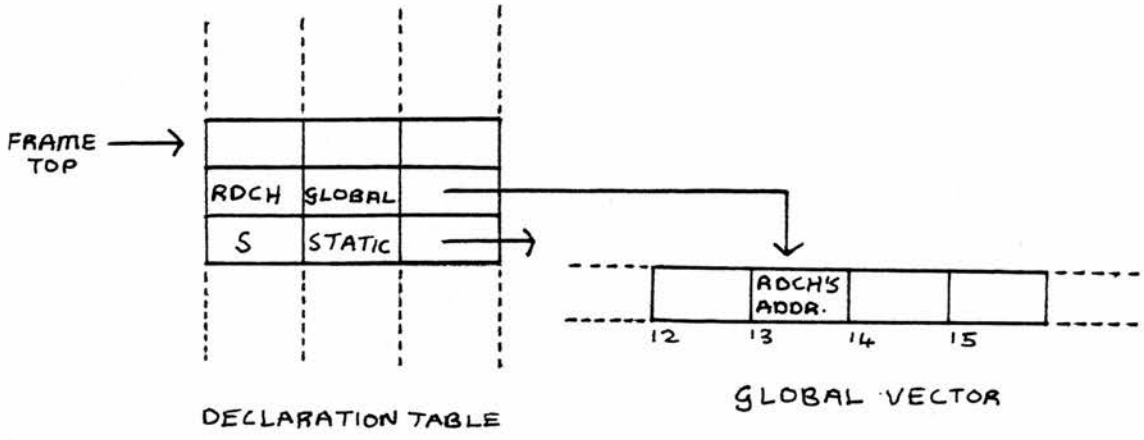

Figure 6 shows the effect of the declaration :-

GLOBAL $(RDCH:13 $)_

The declaration table top is set to the name 'RDCH', type

'GLOBAL' and the address of global vector location 13 which holds the address of the library routine for character input, RDCH. The frame top is advanced.

3.2.4.3 MANIFEST declaration

<manifest declaration> ::= MANIFEST $(<name>=<expression>...$)

The <name>,'MANIFEST' type and evaluated <expression> are declared.

Fig. 7

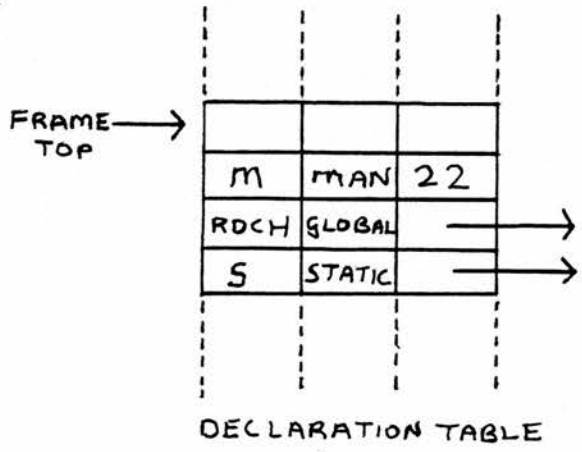

Figure 7 shows the effect of the declaration :-

```
MANIFEST $( M=22 $)_
```

The declaration table top is set to the name 'M', type 'MAN(IFEST)' and the initial value 22. The frame top is advanced.

3.2.4.4 Simultaneous declarations

<simultaneous declaration> ::= LET <definition>
[AND <definition>]

Each <definition> is executed serially as described below. This implements mutual recursion between routines as recursive references are picked up when a routine is executed rather than during compilation. Mutual recursion in the simultaneous declaration of dynamic variables is allowed by the replacement of <constant expression> by <expression> but forward references will be faulted as the <definition>s are processed serially and there will be no declaration table entry for the required variables.

3.2.4.4.1 Simple definition

<simple definition> ::= <name list>=<expression list>

Each <expression> in the <expression list> is evaluated and pushed onto the stack. For each <name> in the <name list>, the <name>, 'LET' storage type marker and corresponding stack address are declared.

Fig. 8

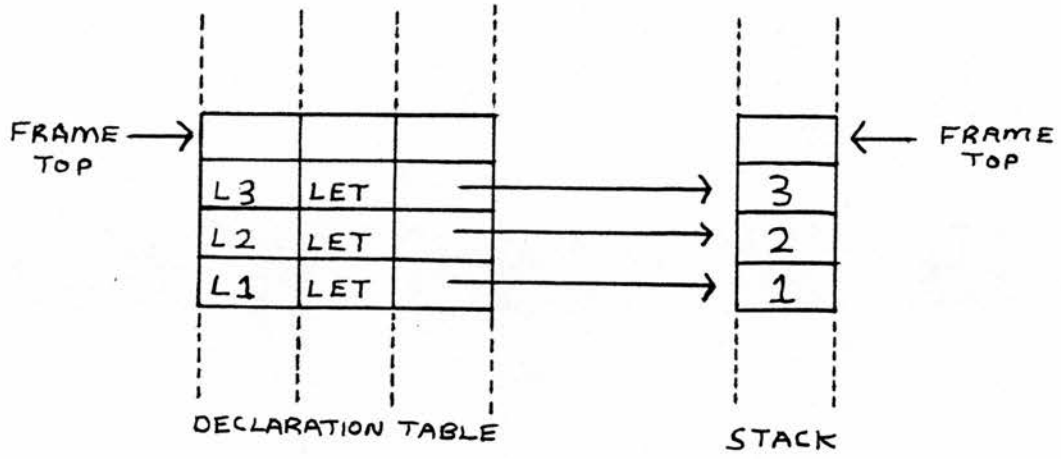

Figure 8 shows the effect of the declaration :-

LET L1,L2,L3=1,2,3_

The initial values are pushed onto the stack and the stack frame top is advanced. Entries for the variables are pushed onto the declaration table. Each consists of the name, type 'LET' and the corresponding stack address. The declaration table frame top is advanced.

3.2.4.4.2 VEC definition

<vector definition> ::= <name>=VEC <expression>

The <expression> is evaluated giving N (say) and N+2 words are allocated from the stack top. The first word is set to the address of the second. The <name>,'LET' type and the address of the first word are declared.

Fig. 9

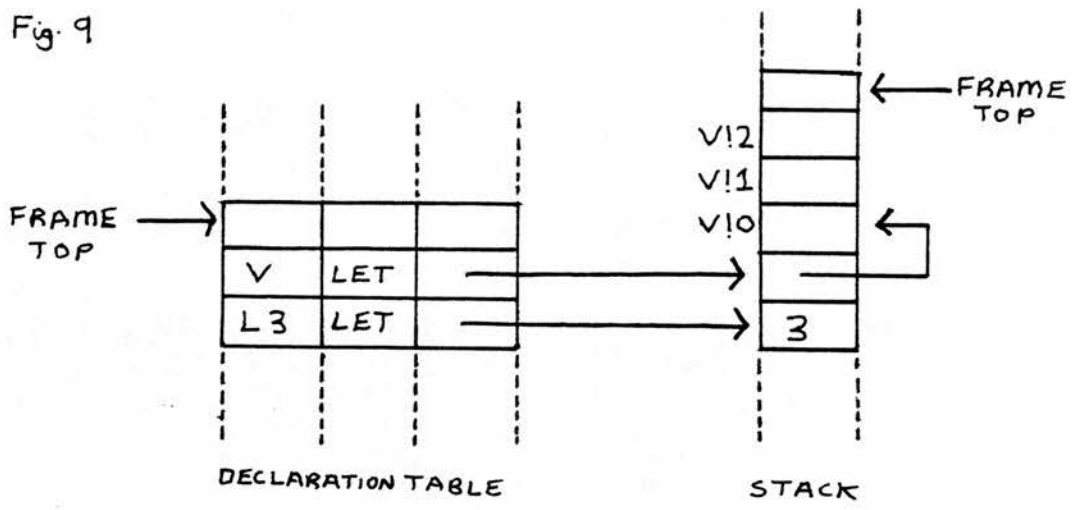

Figure 9 shows the effect of the declaration :-

LET V=VEC 2_

Four stack words are allocated and the stack frame top is advanced. The first allocated stack word is set to the address of the next. The declaration table top is set to the name 'V', type 'LET' and the address of the first allocated stack word. The declaration table frame top is advanced.

3.2.4.4.3 Routine and function definition

<routine definition> ::= <name>() BE <command>|
<name>(<name list>) BE <command>

<function definition> ::= <name>()=<expression>|
<name>(<name list>)=<expression>

If the <name> is already declared in any environment as type GLOBAL then the corresponding global vector location is set to the routine's subtree. Otherwise, a new static area word is set

to the subtree and the <name>, 'STATIC' type and static area word address are declared in the current environment.

If the routine is declared in a routine/block then subsequent declarations will associate the <name> with the same static area word.

Fig.10

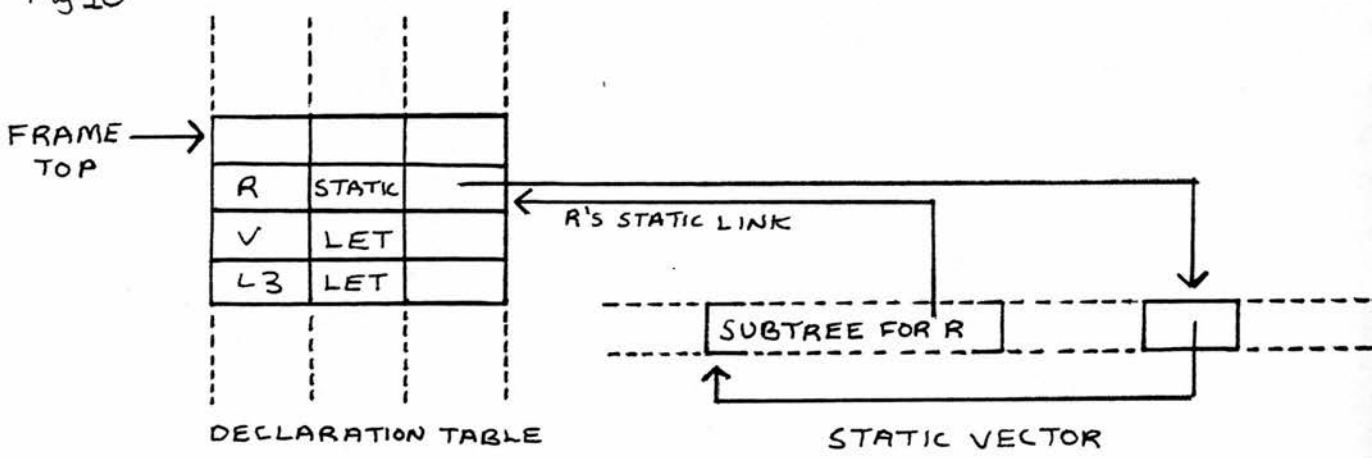

Figure 10 shows the effect of the declaration :-

```
LET R() BE WRITES("ROUTINE*N")_
```

The subtree for R's body has been built in the static vector. The declaration table top is set to the name 'R', type 'STATIC' and the address of the next free static vector word, which is set to the address of R's subtree. The frame top is advanced.

3.2.4.4.4 Static environments for routines

At the end of a simultaneous declaration containing routine definitions , the routines' static environment, consisting of global/static variables in scope when the routines were declared,

must be remembered for access when they are called.

In BCPL, routines have static extent and a routine's address has the same status as any other integer value. Hence, a routine declared by another routine may be accessed indirectly outwith the declaring routine's scope if its address has been passed out by a call to the declaring routine.

```
e.g.  LET ROUTINE=0
      LET R1() = VALOF
                 $( LET R2() BE $( ... $)
                    RESULTIS R2
                 $)
      ROUTINE:=R1() - sets ROUTINE to R2
      ROUTINE()     - calls R2
```

The declared routine may refer to static/global variables declared outside it: these variables may not be in scope when the routine is called, so when a routine is declared its static environment must be frozen.

```
e.g. LET R3()=VALOF
               $( STATIC $( COUNT=0 $)
                  LET R4() BE
                  $( COUNT:=COUNT+1
                     WRITEN(COUNT)
                  $)
                  RESULTIS R4
               $)

     ROUTINE:=R3() - sets ROUTINE to R4
     ROUTINE()     - calls R4 to increment and
                     print COUNT
```

Here, when R4 is called, COUNT is no longer in scope but R4 can still access it. Hence, R4's static environment must include COUNT.

In EBCPL, outer level static/globals are always in scope, so the environment need only contain globals/statics declared in intermediate routines/blocks. (An intelligent syntax analyser would identify the globals/statics the routines actually referred

to.) Furthermore, the static environment need only contain those globals/statics declared in the body of the declaring routine, as the declaring routine's static environment will contain those external to it.

```
e.g. LET R5()=VALOF
     $( STATIC $( S1=1 $)
        LET R6()=VALOF
        $( STATIC $( S2=2 $)
           LET R7() BE
           $( LET A,B=1,2
              ...
           $)
           ...
         $)
         ...
      $)
```

Here, R7 may refer to the local variables A and B, and to R7, S2, R6 and S1, and R6 may refer to R6 and S1.

R7's static environment need only include R7 and S2 as A and B will be declared when its body is executed and it can find R6 and S1 in R6's static environment.

Finally, all routines in a simultaneous declaration can share the same static environment. (In fact, all routines declared at the same level can share the same environment.)

```
e.g. LET R8()
     $( STATIC $( S3=0 $)
        LET R9() BE $( ... $)
        AND R10() BE $( ... $)
        AND R11() BE $( ... $)
        ...
      $)
```

Here, R9,R10 and R11 may all refer to S3, R9, R10 and R11 so they can share a static environment containing S3, R9, R10 and R11.

Notice that a routine's static environment must contain an entry for itself to enable recursion.

At the end of a simultaneous declaration containing routine

definitions, a copy of the static environment - consisting of a static table of the corresponding declaration table global/static entries - is made. The bottom of the static environment is set to the top of the previous static environment (if any) or the top of the outer level environment. The head of each routine's subtree is then set to the environment, forming a static link between the environments accessible within the routine. When a routine is called, its environment is "sown" into the declaration table to be searched when a name is referred to [3.2.5.6.1].

Figure 10 shows the effect of the declaration :-

```
LET R() BE WRITES("ROUTINE*N")_ - see 3.2.4.4.3.
```

Here, the simultaneous declaration declares a single subroutine associated with an outer level variable. Thus, R's static environment consists of itself and the preceding outer level variables. The static link in R's subtree is set to the address of R's declaration table entry.

Figure 11 show the effect of the declaration :-

```
LET R5()=VALOF
          $( STATIC $( S1=1 $)
             LET R6()=VALOF
                      $( STATIC $( S2=2 $)
                         LET R7() BE
                         $( LET A,B=1,2
                            ...
                         $)
                         RESULTIS R7
                      $)
                RESULTIS R6
          $)_
```

and subsequent calls to declare and return the subroutines within R5.

Fig 11.1

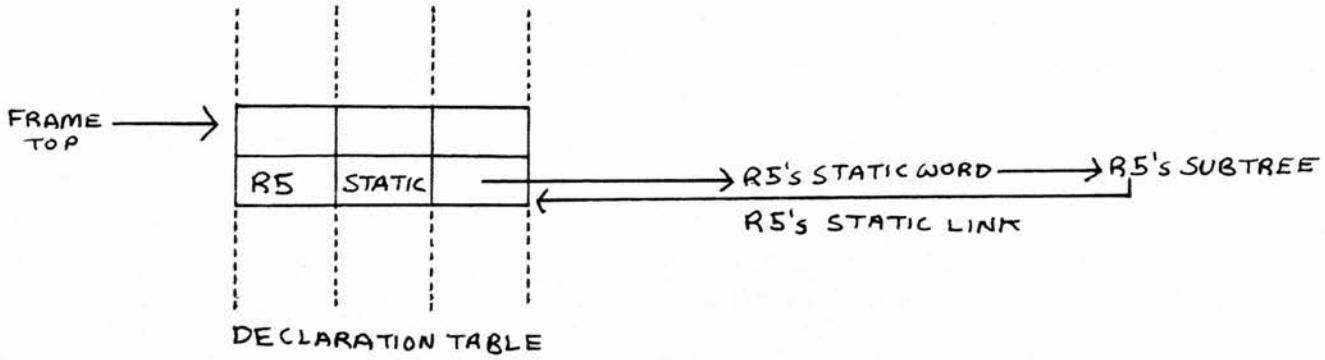

Figure 11.1 shows the effect of the initial declaration. The declaration table top is set to the name 'R5', type 'STATIC' and the address of the next free static vector word, which is set to R5's subtree. As in the previous example, R5 is an outer level subroutine so its' static environment is itself and the preceding outer level variables.

Fig 11.2

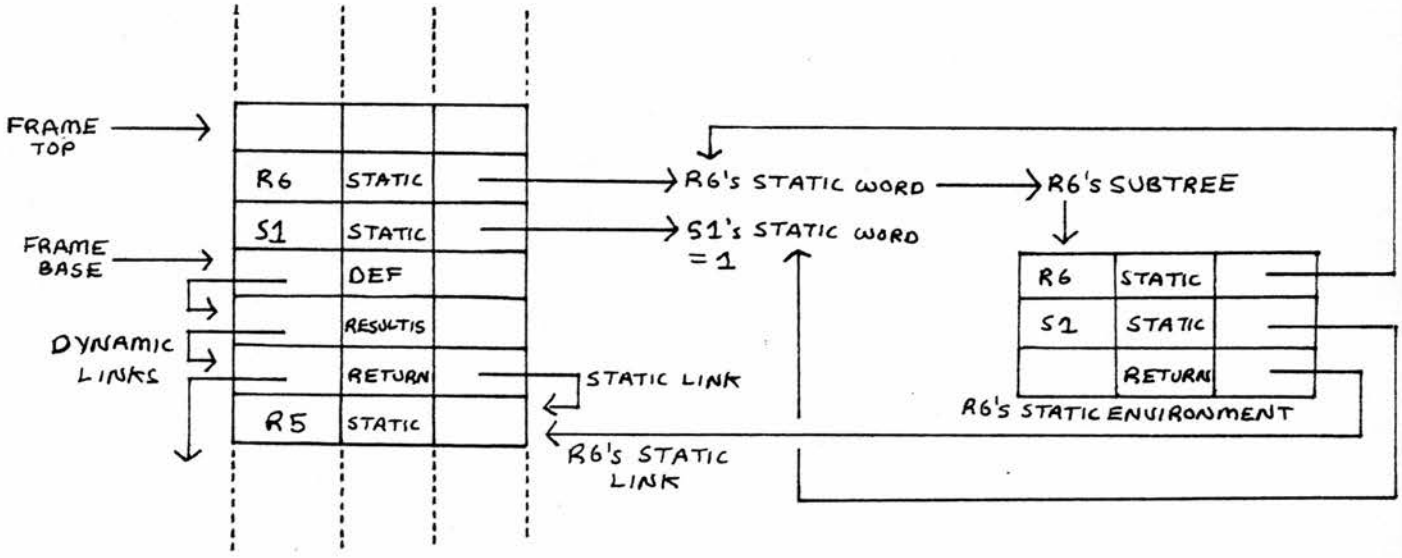

Figure 11.2 shows a call to R5 in :-

```
S:=R5()_
```

which declares R6. A new environment is entered, marked as type 'RETURN' with a dynamic link to the previous frame base and a static link to R5's static environment. R5's body is a VALOF expression [3.2.5.6.3] so a new environment is entered marked as type 'RESULTIS' with a dynamic link to the frame base for the call to R5. The dynamic link to the stack top is not shown. R5's body is a block body which declares variables so a new environment marked as type 'DEF' is entered [3.2.3]. R5 declares S1, with a new static word being allocated and set to the initial value 1, and R6, with a new static word being allocated and set to R6's subtree. At the end of the simultaneous declaration :-

```
LET R6()=VALOF...
```

a static environment is built with entries for S1 and R6. The static link is set to the previous static environment i.e. R5's static environment i.e.. R5 and preceding outer level variables. The static link in R6's subtree is set to the new static environment.

Fig. 11.3

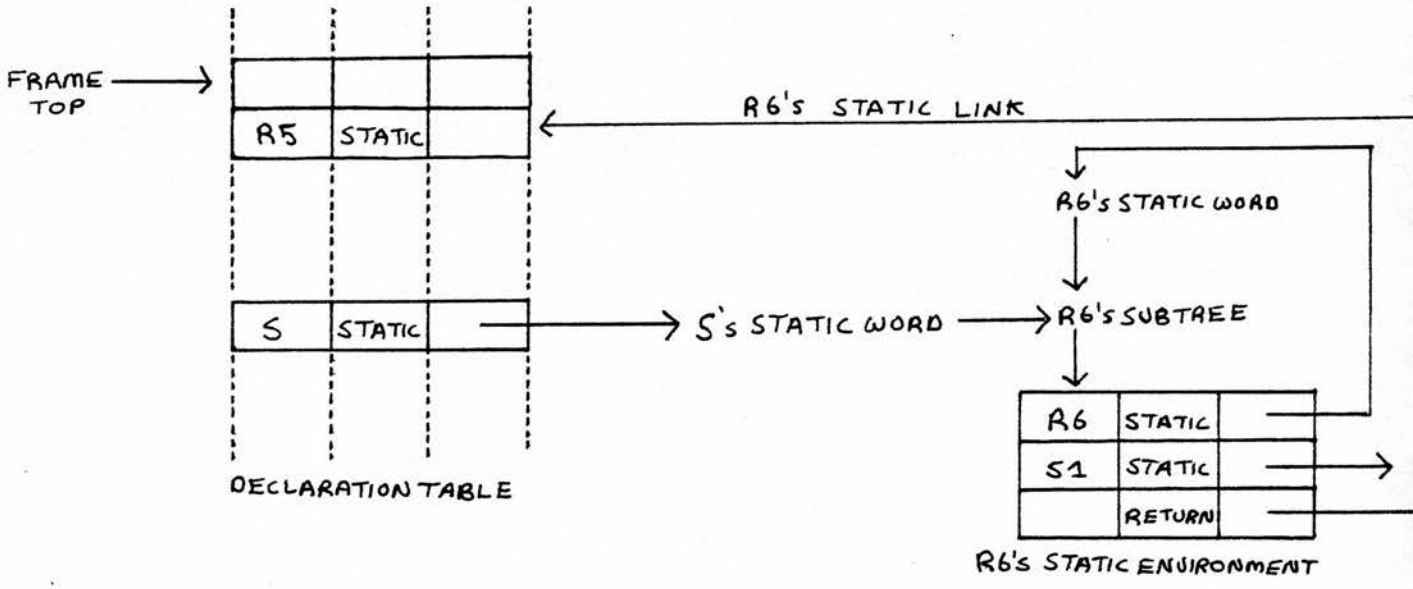

Figure 11.3 shows S set to R6 after termination of :-

S:=R5()_.

The frame top and bottom are reset on exit from R5 and S's static word is set to R6's subtree.

Fig·11·4

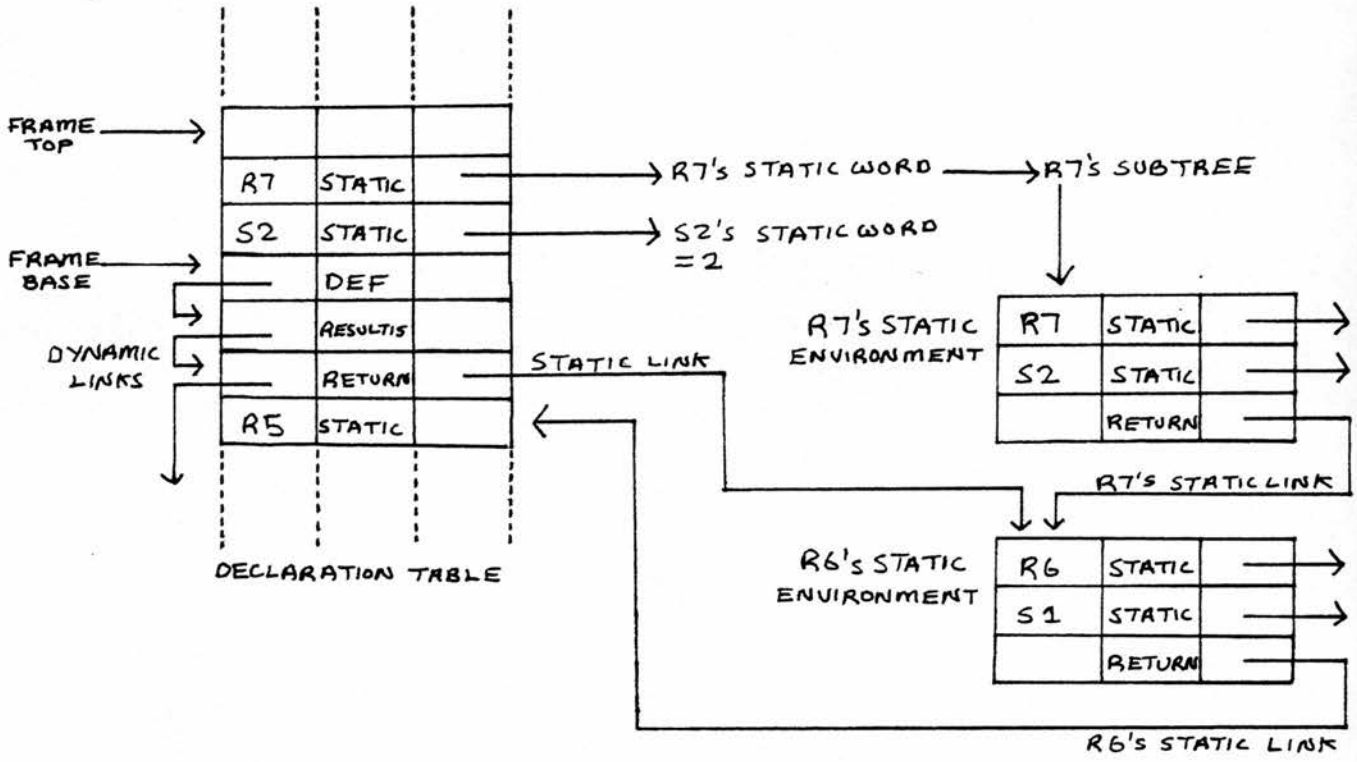

Figure 11.4 shows a call to S in :-

```
L1:=S()_
```

which calls R6, which declares R7. As before, new environments are entered for the call to R6 and the execution of its body. Here, the static link is set to R6's static environment. R6 declares S2 and R7. At the end of the simultaneous declaration :-

```
LET R7() BE...
```

a static environment is built with entries for S2 and R7. The static link in the new static environment is set to the previous static environment i.e. R6's. The static link in R7's subtree is set to the new static environment.

Fig. 11.5

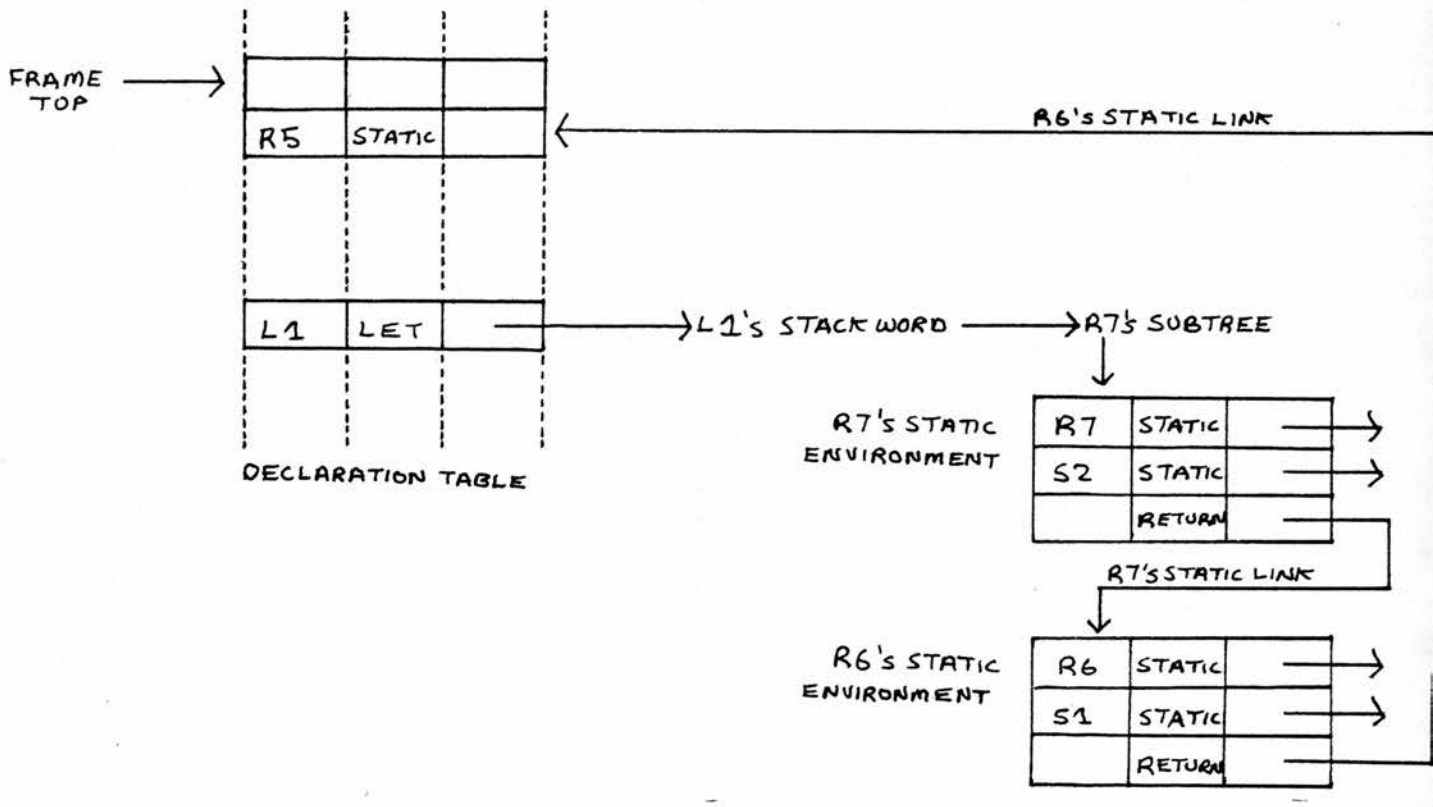

Figure 11.5 shows L1 set to R7 after termination of :-

```
L1:=S()_
```

The frame top and bottom are reset on exit from R6 and L1's stack word is set to R7's subtree.

Fig·11.6

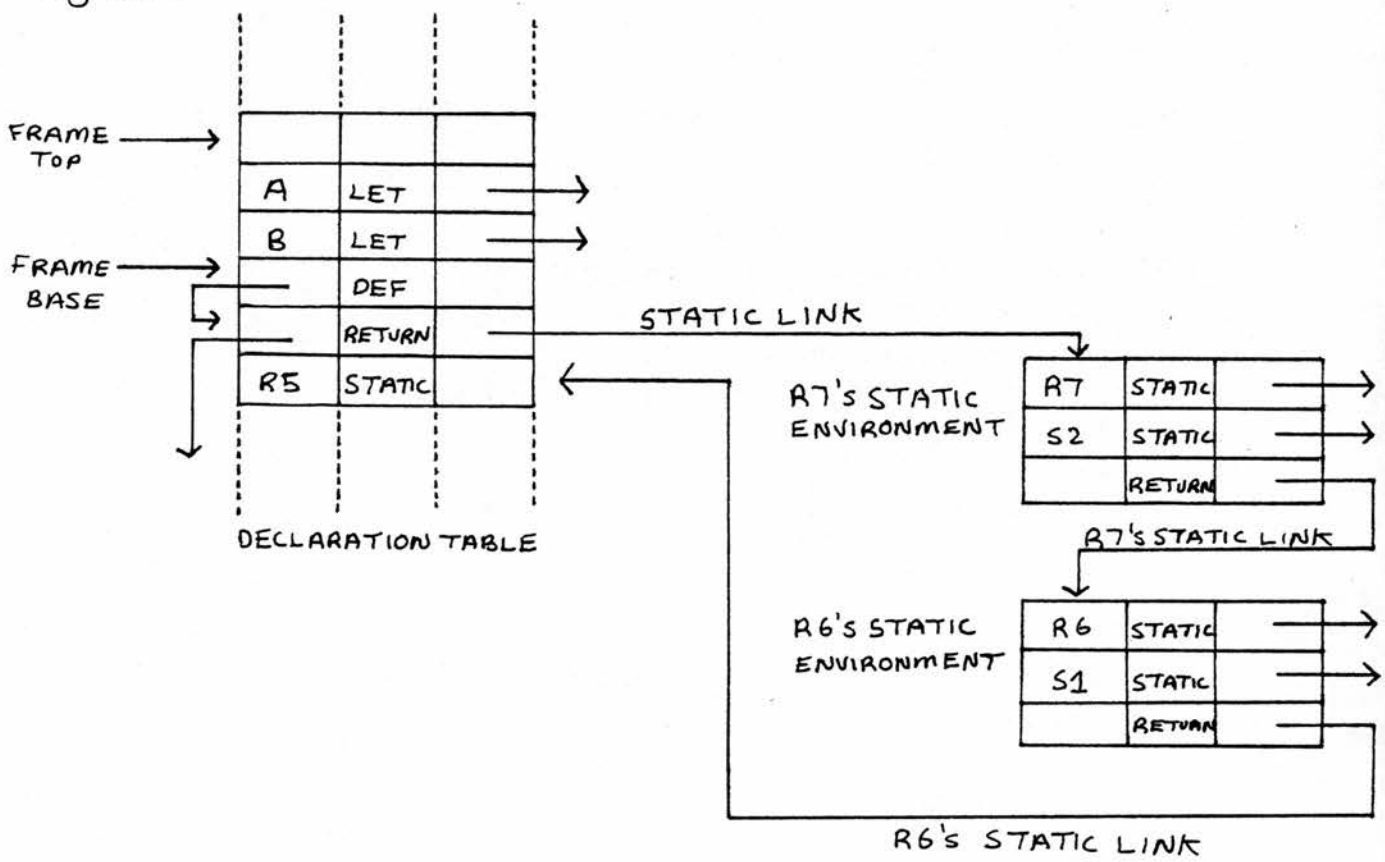

Figure 11.6 shows a call to R7 from :-

```
L1()_
```

A new environment is entered with the static link set to R7's static environment. A new environment is entered for R7's body which declares A and B.

3.2.5 Commands

3.2.5.1 Environments, breaks and error recovery information

Recovery labels are associated with various commands. Each recovery label follows the corresponding command interpretation routine call and marks a point to which control may be returned after an error or a break. Prior to executing these commands, the previous BCPL run-time stack pointer and recovery label are pushed onto the stack and pointers to the EBCPL stack frame base and declaration table frame top are pushed onto the declaration table, forming the dynamic link, along with a marker indicating the type of command. The marker is used to exit from loops, routines and SWITCHONs [3.2.5.5]. A copy is then made of the current stack pointer and recovery label. After executing the command, the previous stack pointer, recovery label, stack frame top and declaration table frame base are reset.

If an error occurs during command execution, the environment is used to restore control to the point marked by the recovery label. The previous environment is restored and the interpreter returns to the previous recovery label. This process continues until the top level of the interpreter is reached. If the user sets trace options [3.3.3.4] diagnostic routines may be called when the interpreter returns to each recovery label.

If a break occurs during command execution, environments are restored until a marker corresponding to the break command is found. Control is then returned to the point marked by the recovery label and execution continues because no error has been

flagged.

3.2.5.2 Repeatable commands

3.2.5.2.0 Introduction

For each repeatable command, a new environment is entered, marked as a 'BREAK' entry point. The command is executed and the previous environment is restored.

For breaks from repeatable commands see 3.2.5.5.

The number of times a repeatable command may be executed is set by default to 1000. This may be changed by the user [3.3.3.4].

3.2.5.2.1 WHILE, UNTIL, REPEAT and REPEATUNTIL

The commands are interpreted by the corresponding BCPL commands which call routines to evaluate the <expression> and loop body.

3.2.5.2.2 FOR

<for command>::=FOR <name>=<exp1> TO <exp2> DO <command>|
FOR <name>=<exp1> TO <exp2> BY <exp3>
DO <command>

The <name>, 'LET' type and stack top address are declared, and the evaluated initialisation <exp(ression)1> for the <name> is pushed onto the stack. The TO and BY <expression>s (if present) are evaluated.

An error occurs if the step length is 0. This is another attempt to avoid infinite loops through an inadvertent 0 step length from an evaluated 'BY' expression. If a FOR command with 0 step length

is required it may be made explicit through the use of an assignment to the control variable followed by a REPEAT command. The body is repeatedly executed; each time, the stack word associated with the <name> is incremented/decremented and execution stops when the TO value is passed.

3.2.5.2.3 Loop body execution

A new environment is entered, marked as a 'LOOP' entry point. The body is executed and the loop count incremented. An error occurs if the loop count exceeds the current permitted maximum. The old environment is restored.

For breaks from loops see 3.2.5.5.

3.2.5.3 Routine call

<routine call> ::= <expression>() |
<expression>(<parameter list>)

The <expression> is evaluated giving an address. If the address is that of an external compiled routine then the parameters are evaluated and the routine is called. Otherwise, an error occurs if the address is not that of a static vector word. A new unmarked environment is entered. The actual parameters are evaluated and pushed onto the stack. The formal parameters are declared as type 'LET'. Errors occur if there are too few actual parameters or if formal parameter names are multiply declared. While BCPL allows routine calls with variable numbers of parameters, there is no explicit mechanism for determining how many are present. If a routine is called with too many

parameters, this will merely waste stack space. If, however, it has too few parameters then formal parameters and local dynamic variables may be associated with the same stack locations which could give rise to peculiar results. The environment is marked as type 'RETURN' and set to point to the top of the routines static environment. The routine body is executed and the previous environment restored.

Implementing calls to external routines and functions was not easy. In BCPL, routines and functions may be called with variable numbers of parameters but there is no way of determining how many parameters a compiled routine was declared with and it is not easy for a routine to determine how many parameters it has been passed. Thus, the interpretation of calls to external routines involves determining how many actual parameters are present and then selecting a call corresponding to the number of actual parameters from a list. The list contains calls for zero to eight parameters which places a restriction on external routine calls.

For breaks from routines see 3.2.5.5.

3.2.5.4 SWITCHON

```
<switchon command> ::= SWITCHON <expression> INTO
                       $( CASE <expression>:<command>
                          ...
                          DEFAULT:<command>
                       $)
```

The SWITCHON <expression> is evaluated. Each CASE <expression> is evaluated. If its value matches that of the SWITCHON <expression> then the corresponding <command> will be executed. If no matching CASE <expression> is found then the DEFAULT <command> (if any) will be executed. A new environment marked as 'ENDCASE' is entered and the <command> is executed. This may include the execution of the subsequent CASE commands as the body of a SWITCHON is an undifferentiated command. At the end of the command, the previous environment is restored.

BCPL's generous syntax does not place any restraints on the textual positions of the CASE commands within the SWITCHON body. In implementing the SWITCHON I placed the restriction that the CASE commands must be at the outer level of the body. This reduces tree searching for CASEs and discourages unreadable programs.

Note that the implementation will accept any <expression> after "CASE".

For breaks from SWITCHONs see 3.2.5.5.

3.2.5.5 BREAK, LOOP, ENDCASE and RETURN

The previous environment is repeatedly restored until an environment entry with a marker corresponding to the <command> is found. Control is then returned to the recovery label where normal execution continues as no error has been flagged [3.2.5.1].

If for a BREAK,LOOP or ENDCASE a ´RETURN´ marker is found then an error occurs as an attempt has been made to exit from a non-existent repeatable command. Similarly, an error occurs if the outer level environment is reached and the corresponding marker has not been found.

Fig. 12

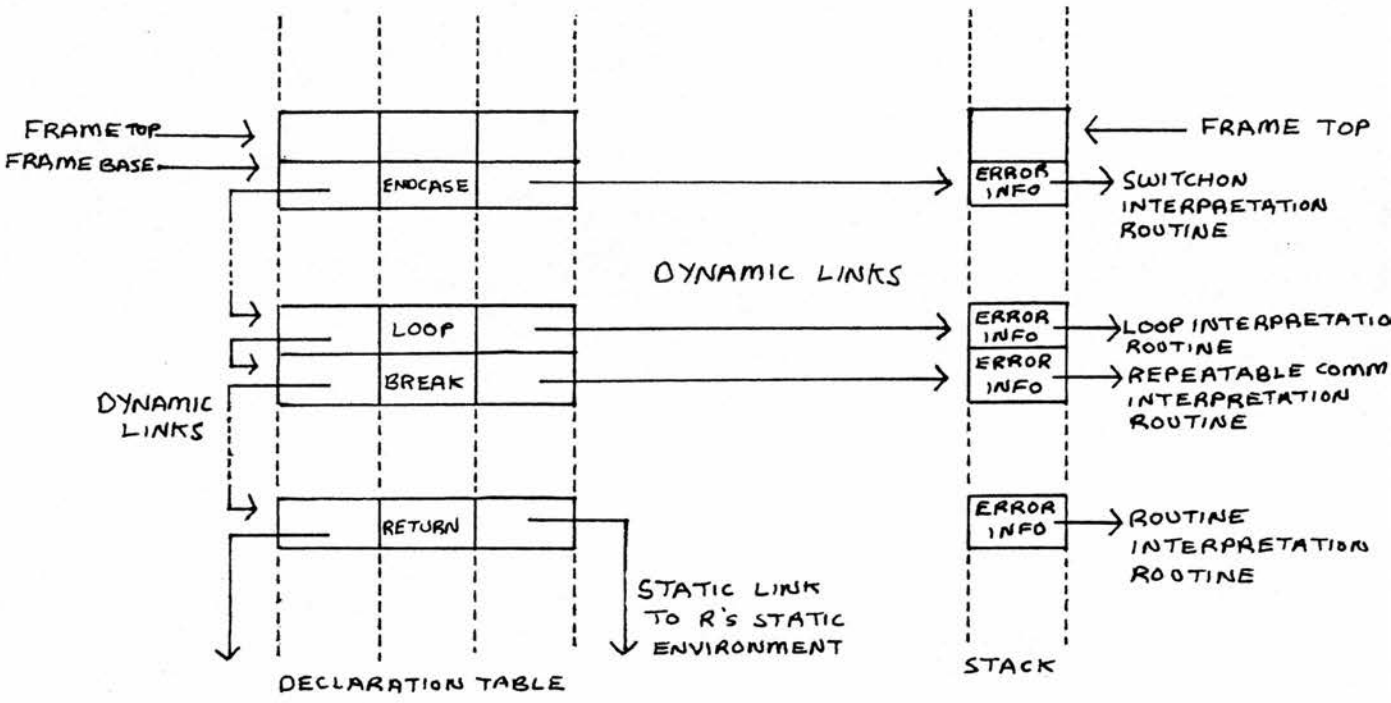

Figure 12 shows the declaration table and stack during the execution of a call to the routine :-

```
R()_
```

declared by :-

```
LET R() BE
$( ...
   WHILE ... DO
   $( ...
      SWITCHON ... INTO
      $( CASE ... :
         CASE ... : (*)
         ...
      $)
      ...
   $)
   ...
$)_
```

at the point marked (*).

The error information for the 'RETURN' environment, corresponding to the initial routine call entry, indicates the recovery address for the routine interpretation routine. If a RETURN is encountered during execution, control will be returned to this address following environment restoration. The routine interpretation routine will then return, in this example returning control to the program interpretation routine.

The error information for the 'BREAK' environment, corresponding to the execution of the WHILE command, indicates the recovery address for the repeatable command interpretation routine. If a BREAK is executed then control will be returned to this address following environment restoration. The repeatable command interpretation routine will then return and in this example, execution will continue at the point following the WHILE command.

The error information for the 'LOOP' environment, corresponding to the execution of the WHILE command body, indicates the recovery address for the loop body interpretation routine. If a LOOP is executed then control will be returned to this address following environment restoration. The loop body interpretation routine will then return control to the repeatable command interpretation routine which may, in this example, again call the loop body interpretation routine.

The error information for the 'ENDCASE' environment, corresponding to the execution of the CASE in the SWITCHON command, indicates the recovery address for the SWITCHON command interpretation routine. If an ENDCASE is encountered during execution, control will be returned to this address following

environment restoration. The SWITCHON command interpretation will return and execution will, in this example, continue at the point following the SWITCHON body.

3.2.5.6 Expressions

3.2.5.6.0 Introduction

Expressions are evaluated by a function that recursively evaluates the operands and uses them to evaluate the BCPL expressions corresponding to the operator.

In BCPL, a variable's address is called its L-value and its contents is called its R-value. Similarly, an expression may be evaluated in L-mode to return an address or in R-mode to return a value. Normally, an expression is evaluated in R-mode. The LV operator [3.2.5.6.7] forces L-mode evaluation, and expressions on the left of assignments [3.2.5.7] are implicitly evaluated in L-mode [*2].

Expressions evaluated in L-mode are checked to ensure that they are of type "left hand side" i.e. capable of returning a meaningful address. These are :-

i) <name>

return <name>'s address

ii) !<expression>

indirection - return address from <expression> evaluated in R-mode

iii) <expression>!<expression>

indexing - return address from sum of <expression>s, each evaluated in R-mode

iv) <expression1>-><expression2>,<expression3>

conditional - return appropriate <expression>
evaluated in L-mode

Here I have discussed only those expressions which I consider significant(!).

3.2.5.6.1 names

The environments are searched, starting with the current, following the static links, for the <name>. An error occurs if the <name> is not found. Otherwise, its type marker indicates where its value is held, and its address is used to extract the value. An error occurs if the <name> is referred to in a routine and is that of an outer level LET type variable. While such a variable's associated object has static extent, its name has the usual BCPL scope.

Figure 11 in 3.2.4.4.4 shows the creation of the static environment for R7 declared by the programs :-

```
LET R5()=VALOF
        $( STATIC $( S1=1 $)
           LET R6()=VALOF
                    $( STATIC $( S2=2 $)
                       LET R7() BE
                       $( LET A,B=1,2
                          ...
                       $)
                       RESULTIS R7
                    $)
           RESULTIS R6
        $)_
```

S:=R5()_ - called R5 to declare R6, create its static environment and set S to R6

L1:=S()_ - called R6 to declare R7, create its static environment and set L1 to R7

Figure 11.6 shows the effects of the program :-

L1()_

which called R7 which declared A and B.

Consider the evaluation of names in R7's body :-

i) any references to A or B in R7 will be satisfied in the current environment.

ii) any references to R7 or S2 will be satisfied in R7's static environment.

iii) any references to R6 or S1 will be satisfied in R6's static environment, indicated by the static link in the bottom of R7's static environment.

iv) any references to R5 or other outer level variables will be satisfied by the outer environment, indicated by the static link in the bottom of R6's static environment.

v) any references to other names will be faulted.

3.2.5.6.2 Function calls

Function calls are dealt with in much the same way as routine calls except that they return a value through a RESULTIS or BREAK command. An error occurs if a routine is called as a function.

3.2.5.6.3 VALOF - VALOF <command>

A new environment marked as 'RESULTIS' is entered, and the <command> is executed. If control returns to the point before the recovery label then an error occurs as a RESULTIS has not been executed within the <command>. Otherwise, the previous environment is restored and the value from the RESULTIS is returned.

Note that a BREAK, LOOP, ENDCASE or RETURN from a VALOF in a repetitive command, SWITCHON command or routine body will not be faulted as control will be returned to the interpretive routine for the enclosing command rather than for the VALOF.

3.2.5.6.4 RESULTIS - RESULTIS <expression>

The <expression> is evaluated, previous environments are restored until a 'RESULTIS' marker is found, and control is returned to the recovery label. An error occurs if no 'RESULTIS' marker is found.

3.2.5.6.5 Indirection - ! <expression>

The <expression> is evaluated in R-mode to give a value which is then used as the address from which the returned value is extracted.

3.2.5.6.6 Indexing - <expression1> ! <expression2>

The <expression>s are evaluated in R-mode and their sum is used as the address for the result.

3.2.5.6.7 LV - LV <expression>

The <expression> is evaluated in L-mode.

3.2.5.7 Assignment - <left hand side list>:=<expression list>

Each expression in the <left hand side list> is evaluated in L-mode (see above), to give an absolute address which is pushed onto the stack. Each expression in the <expression list> is evaluated in R-mode and pushed onto the stack. Each of the <expression list> values is assigned to the word addressed by the corresponding <left hand side list> value.

This is wrong! The assignment is not, as I mistakenly thought, analogous to the simultaneous declaration, but a shorthand for a sequence of assignments.

3.2.5.8 GOTO

The GOTO command was not implemented - see chapter 2 section 2.

3.2.5.9 Error handling

After a run time error, an error message is output and the interpreter restores environments until the outer level error recovery label is reached. The previous declaration table, stack and static area tops are restored, thus removing any outer level variables declared by the program and reclaiming all static space allocated to it i.e. for routines, strings, STATIC declarations and TABLES.

Note that if the program set previous outer level variables to

static objects declared within it, then subsequent attempts to access those objects will produce indeterminate effects, as they will be overwritten by subsequent programs.

3.3 System commands

3.3.1 Input file

3.3.1.1 Input file organisation

There was no provision for BCPL access to RAX files so the input file is held in core. It has space for 6000 characters e.g. 120*50 chars/line. The file is held as a circular buffer with pointers to the free space and current editing position (henceforth referred to as the "file pointer"). A record is kept of file free space and the number of the current edit line.

During editing, text is copied from the file pointer position by way of the edit buffer where it is manipulated, to the free space.

The edit buffer is circular with room for 300 characters e.g. 10*30 char/line. Pointers to the start of the edited buffer text and the buffer free space are maintained.

If a non-edit system command is called after an edit command, the edit buffer text and unedited text at the file pointer are copied to the free space and the file pointer is reset to the the start of the text. This will be referred to as "resetting the file".

During editing, the file pointer indicates the unedited text start. Note that there is no need to maintain a pointer to the start of the edited text because it always follows immediately the end of the unedited text. This is because after the initial input of text, the free space starts at the end of the text, and during editing text is copied to the free space.

Fig. 13.1

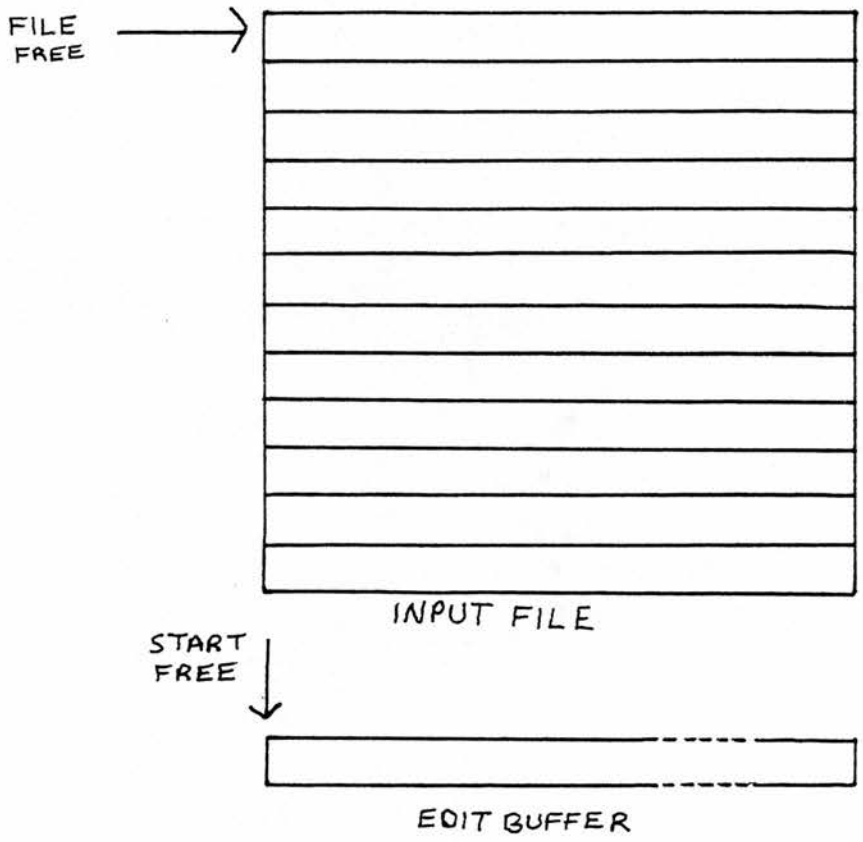

Figure 13.1 shows the input file and edit buffer on entry to the system. The file and free space pointers are set to the physical start of the file. Here, the file is shown with space for 12 lines of text. The edit buffer start and free space pointers are set to the physical start of the buffer.

Fig. 13.2

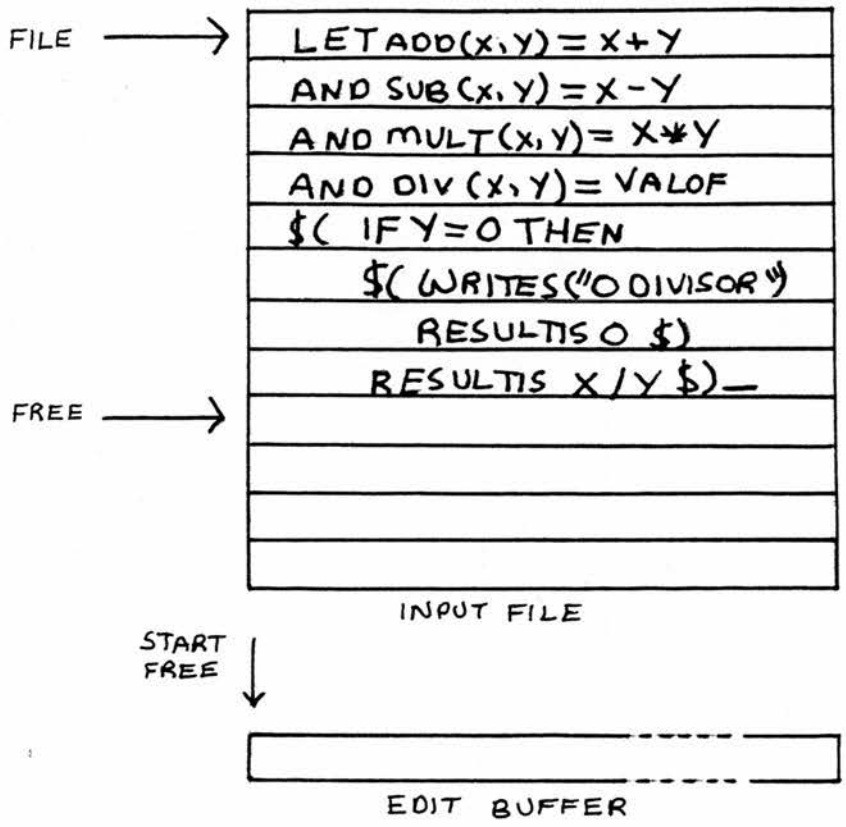

Figure 13.2 shows the input file and edit buffer after input of the text :-

```
LET ADD(X,Y)=X+Y
AND SUB(X,Y)=X-Y
AND MULT(X,Y)=X*Y
AND DIV(X,Y)=VALOF
$( IF Y=0 THEN
   $( WRITES("0 DIVISOR")
      RESULTIS 0 $)
   RESULTIS X/Y $)_
```

The file pointer indicates the start of the text and the free space pointer indicates the start of the free space immediately following the end of the text. The end of the text is marked by '_'.

Fig. 13.3

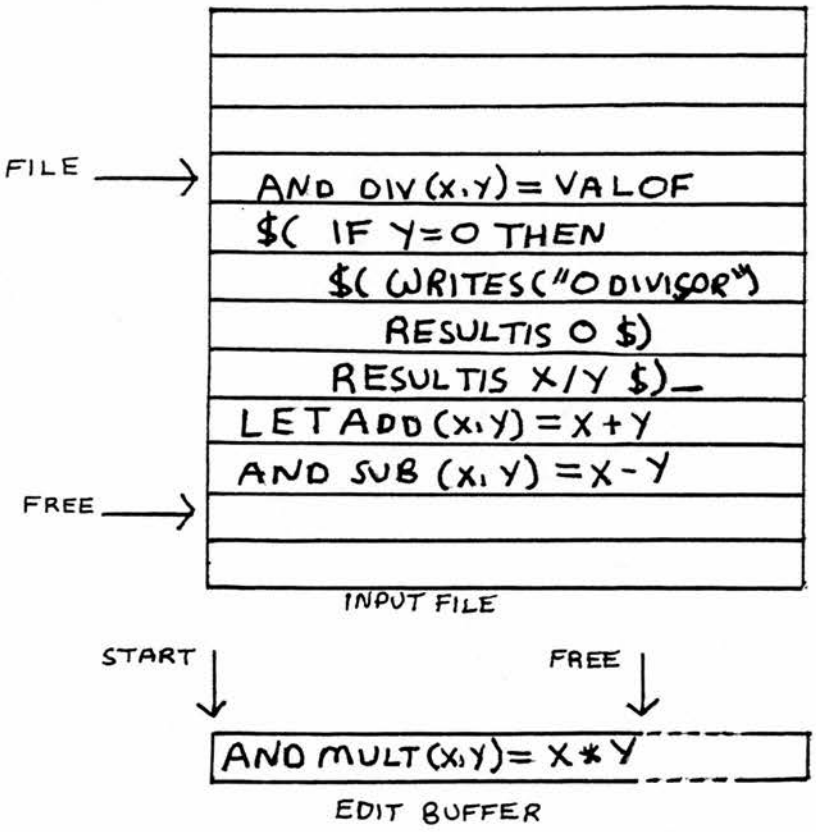

Figure 13.3 shows the input file and edit buffer during editing. The file pointer indicates the start of the unedited text, beginning with the unedited portion of the current line. The edit buffer start pointer indicates the start of the edited portion of the current line. The edit buffer free space pointer indicates the buffer free space at the end of the edited portion of the current line. The file free space pointer indicates the start of the free space following the edited text. Note that the edited text starts immediately after the end of the unedited text in the file.

Fig. 13.4

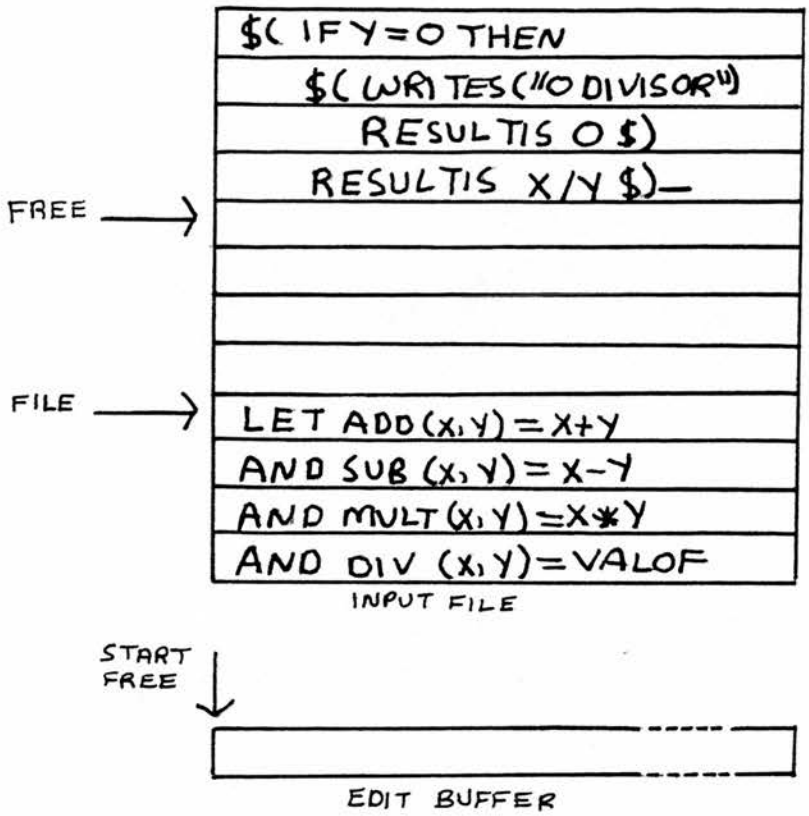

Figure 13.4 shows the input file and edit buffer after the file has been reset. The file pointer indicates the start of the text and the free space pointer indicates the free space immediately following the text. The buffer pointers are reset to the start of the buffer.

3.3.1.2 Input from terminal - INPUT

The file and free space pointers are reset to the file buffer absolute start address and the free space count is set to the file size. Characters are read from the terminal into the file free space until a '_' is found. An error occurs if the file is filled.

See figure 13.2.

3.3.1.3 Input from pre-supplied data - DATA

As INPUT, but characters are read from the data included when the system was loaded and run from RAX.

This provides the only means of loading program text prepared under RAX.

See figure 13.2.

3.3.1.4 Insert text from terminal - INSERT.

Characters are read in to the free space from the terminal until a '_' is encountered. An error occurs if the file is full/filled.

3.3.1.5 Write text to RAX file - SAVE

Each line of text in the file is packed into a vector, right filled with spaces, which is written to the RAX standard output file [*4]. This file may be saved in a RAX user file for subsequent manipulation under RAX.

3.3.1.6 List file on screen - DISPLAY

The file is displayed on the terminal screen as a sequence of numbered lines.

3.3.2 Editor

3.3.2.1 Editor organisation

Editing is line oriented with additional primitive context editing facilities. As noted above, text is passed from the file top through the edit buffer to the file bottom. During editing, the edited portion of the current line is in the edit buffer and the unedited portion is indicated by file pointer.

If during the execution of an edit command the end of the text is reached then the end of file message :-

```
.....E.O.F......
```

is output and the command terminates. This message is referred to as the "EOF message".

3.3.2.2 File reset command - ET

The file is reset [3.3.1.1]: future edit commands will operate from the text start.

See figure 13.4.

3.3.2.3 Find text command - EF "text"

Characters are brought into the buffer from the file and checked against the "text". At the end of the each line, it is rewritten to the free space. The command terminates with the "EOF message" if no match is found. Otherwise, the command terminates with the edit buffer free space pointer following the last matched "text" character in the current line.

Fig 13.5

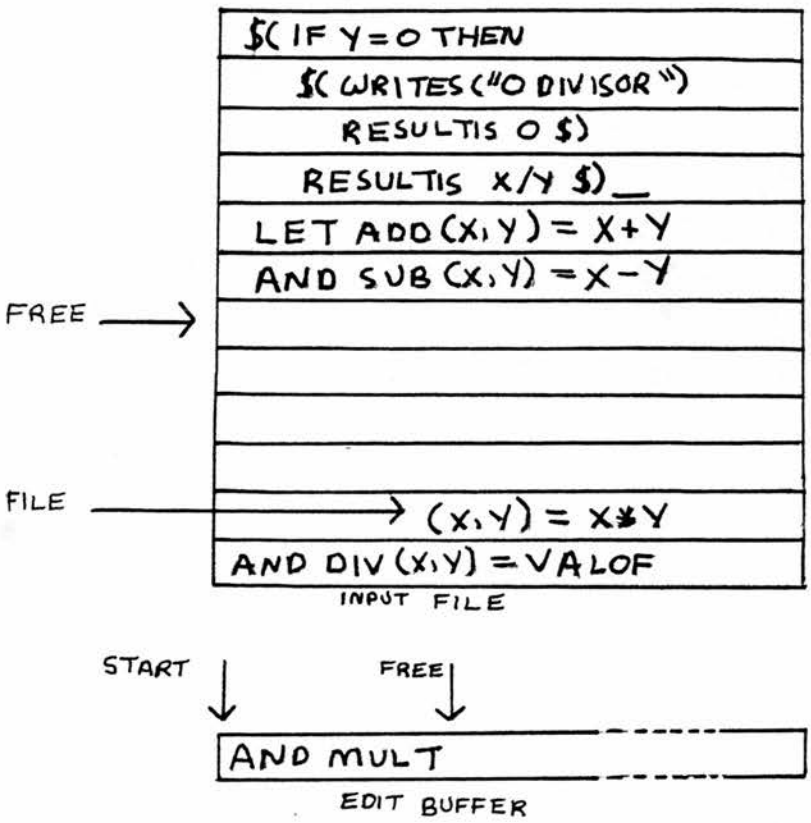

Figure 13.5 shows the input file and input buffer after the command :-

EF "MULT".

The edit buffer holds the line portion containing the first occurrence of "MULT". Subsequent edit commands will act on the rest of the line, indicated by the file pointer. Preceding lines have been copied to the file free space.

3.3.2.4 Move to line command - EM <line number>

If the <line number> precedes the current line number then the file is reset. Lines from the file are passed through the edit

buffer until the end of the preceding line is reached. Future edit commands will start from the beginning of the required line. An error occurs if the <line number> is negative. The command terminates with the "EOF message" if the <line number> exceeds that of the last line.

Fig.13.6

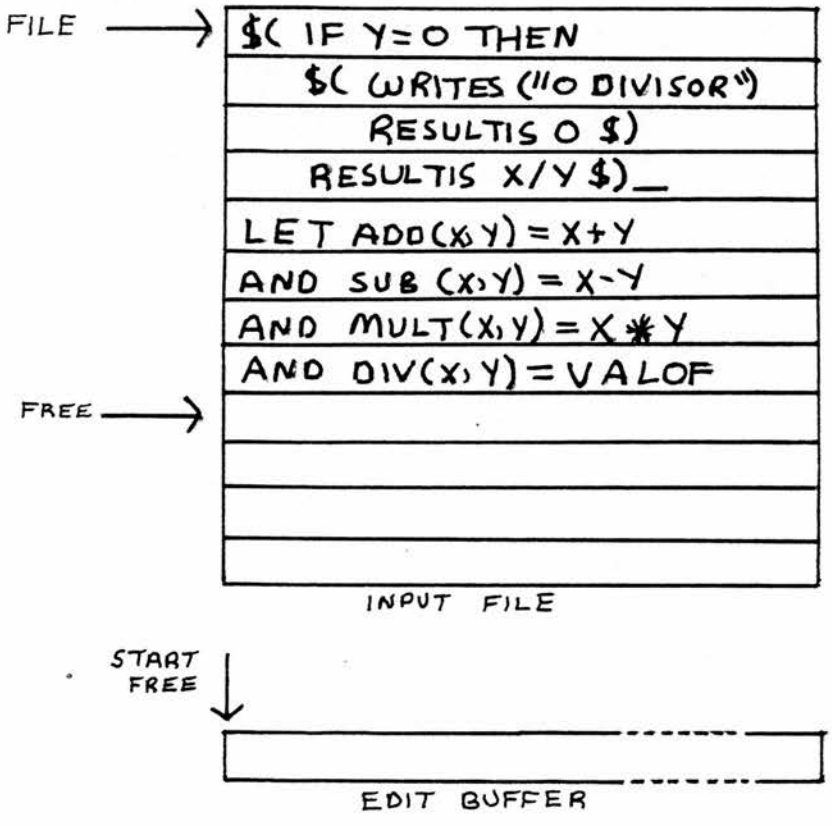

Figure 13.6 shows the input file and edit buffer after the command :-

EM 5.

The file pointer is now at the start of line 5 and subsequent edit commands will operate upon it. Preceding lines have been copied to the file free space.

3.3.2.5 Delete command - ED <line number> ¦ ED "text"

For <line number>: the line is found as for EM; the line is

brought into the buffer; the file free space count is increased by the line length and the edit buffer free space pointer is set to the edit buffer text start pointer so that the line will be ignored by future edit commands.

Fig·13·7

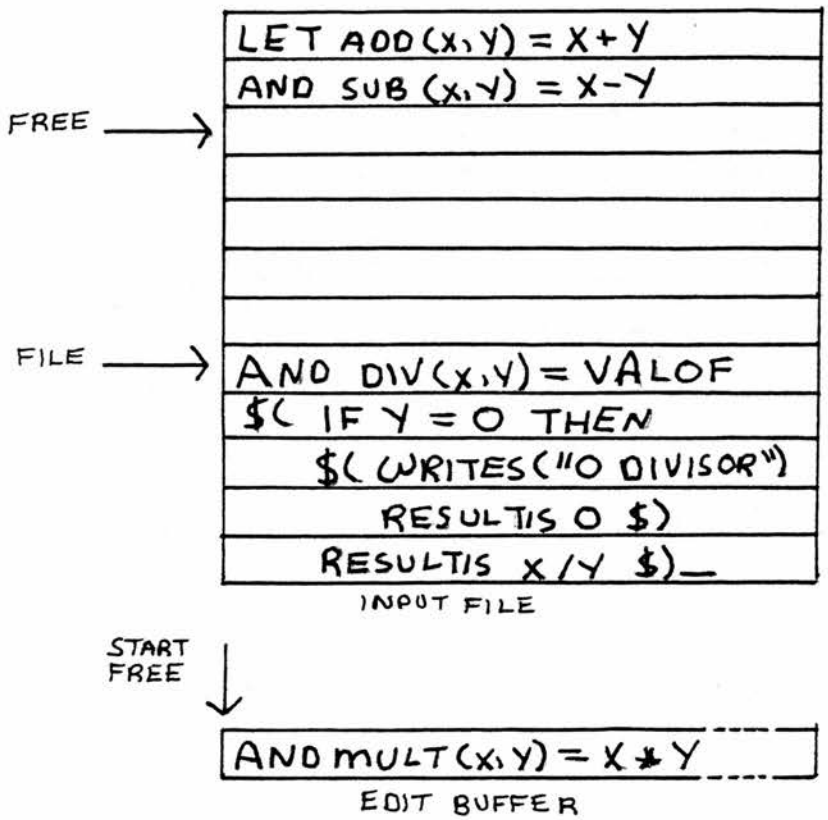

Figure 13.7 shows the input file and edit buffer after the command :-

ED 3.

Line 3 is in the buffer, but the buffer pointers are at its start so that subsequent text movements will over write it.

For "text": The "text" is found as for EF; the file free space count is increased by the "text" length and the edit buffer free space pointer is set to the start of the "text".

Fig. 13.8

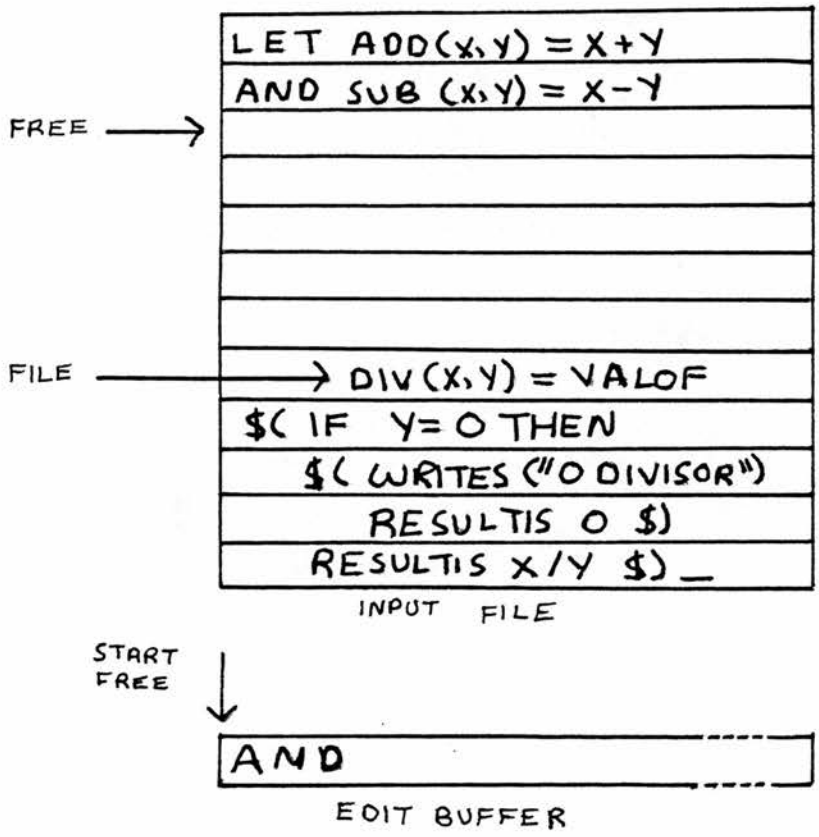

Figure 13.8 shows the input file and edit command after the command :-

ED "AND".

The start of the line containing the first occurrence of "AND" is in the edit buffer, but the buffer pointers are at its start so that subsequent text movements will over write it.

3.3.2.6 Insert command - EI "text" ¦

EI <line number>,"text" ¦

EI "text1","text2"

For "text": the text is inserted at the start of the edit buffer free space and the edit buffer free space pointer is set to the "text" end.

Fig.13.9

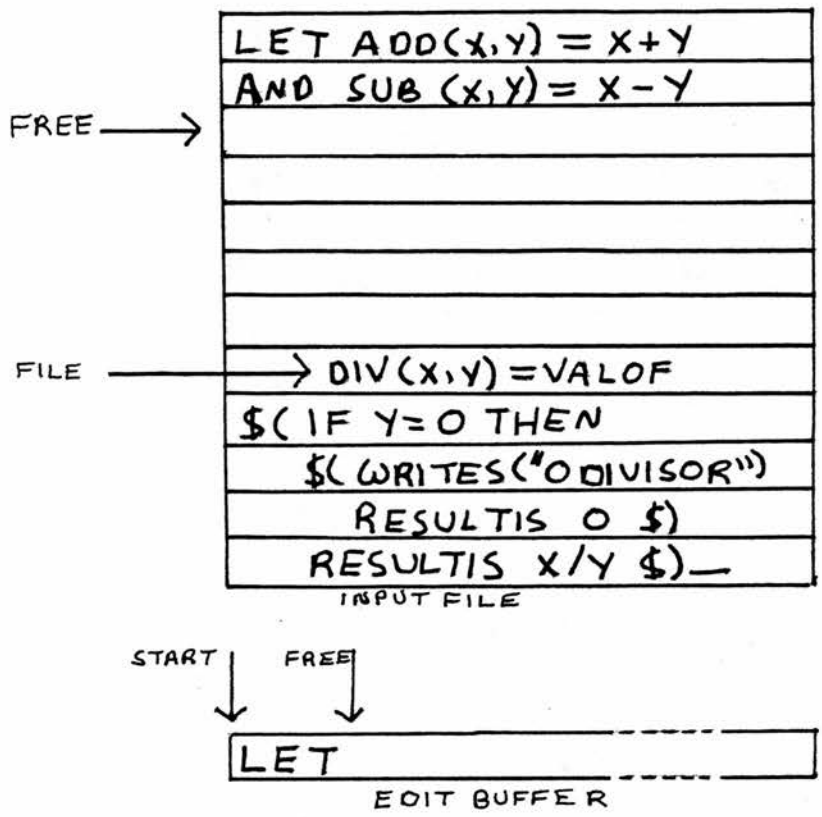

Figure 13.9 shows the input file and edit buffer after the command :-

EI "LET".

The text "LET" has been inserted at the buffer position indicated by the buffer free space pointer, overwriting the "AND" brought

into the buffer for deletion in the previous example.

For <line number> "text": the line is found as for EM and copied to the file free space; the "text" is copied to the edit buffer, terminated with a new-line symbol and copied to the file free space.

Fig.13.10

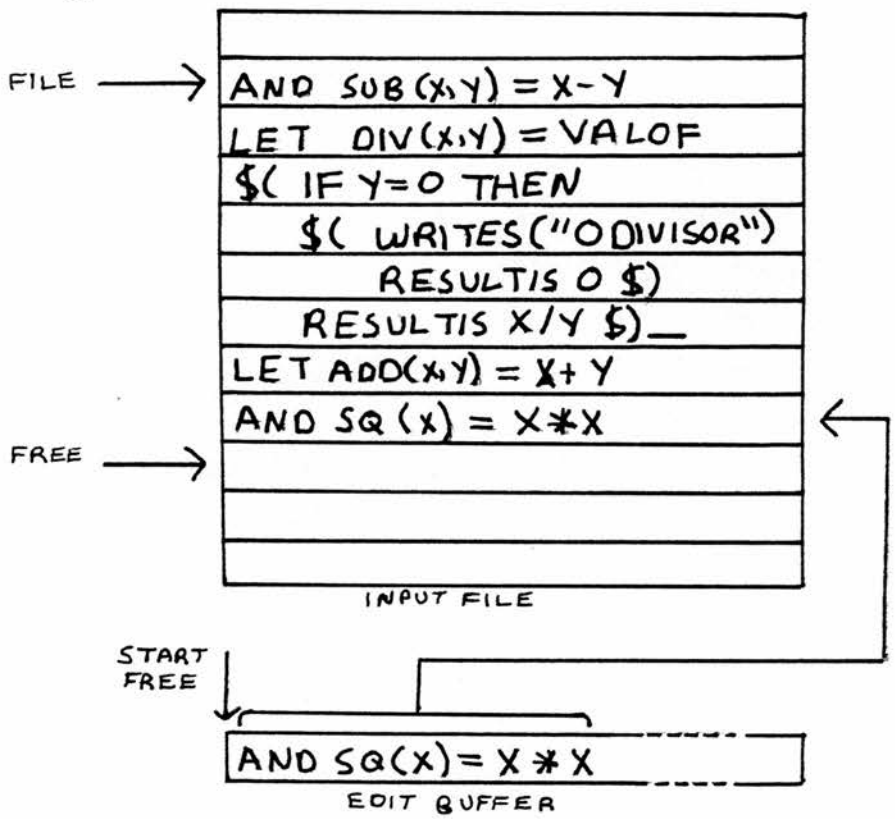

Figure 13.10 shows the input file and edit buffer after the command :-

```
EI 1,"AND SQ(X)=X*X".
```

Line 1 has been copied to the file free space, the text has been placed in the buffer and copied to the free space following the line.

For "text1" "text2": the "text1" is found as for EF and the "text2" is inserted as above.

Fig.13.11

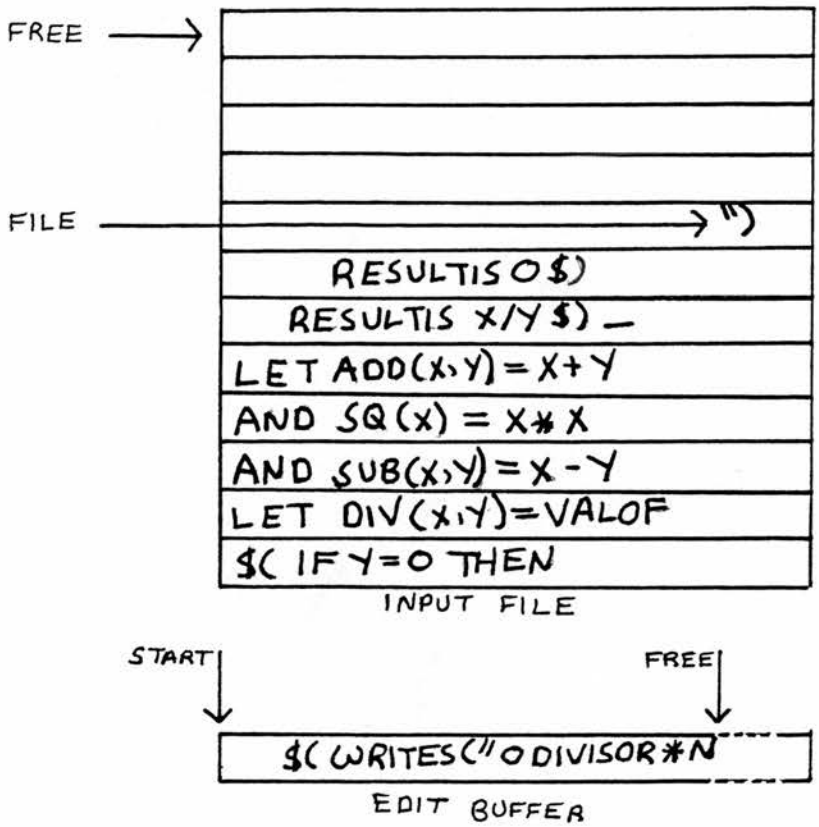

Figure 13.11 shows the input file and edit buffer after the command :-

```
EI "DIVISOR","*N"
```

The line portion containing "DIVISOR" has been brought into the buffer and "*N" has been appended to it.

3.3.2.7 Replace command - ER "text1","text2" |

ER <line number>,"text"

The "text1"/line is deleted as for ED and the "text2"/"text" inserted as for EI.

Fig. 13.12

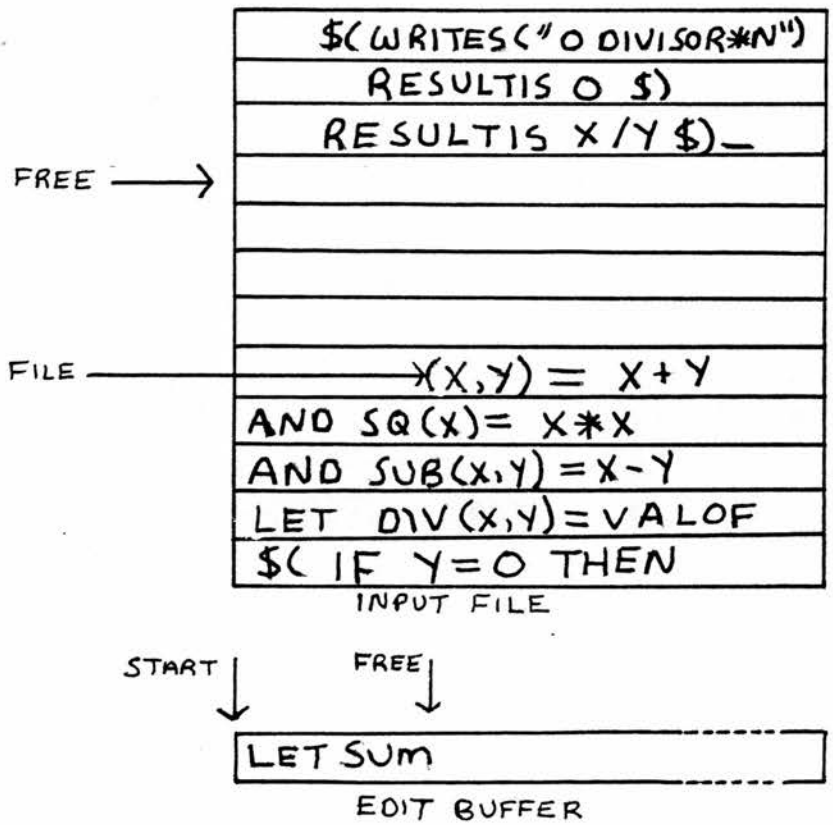

Figure 13.12 shows the input file and edit buffer after the command :-

ER "ADD","SUM".

The line portion containing "ADD" was brought into the buffer with the free space pointer positioned before the "ADD". The insertion of "SUM" has overwritten the "ADD".

Fig·13·13

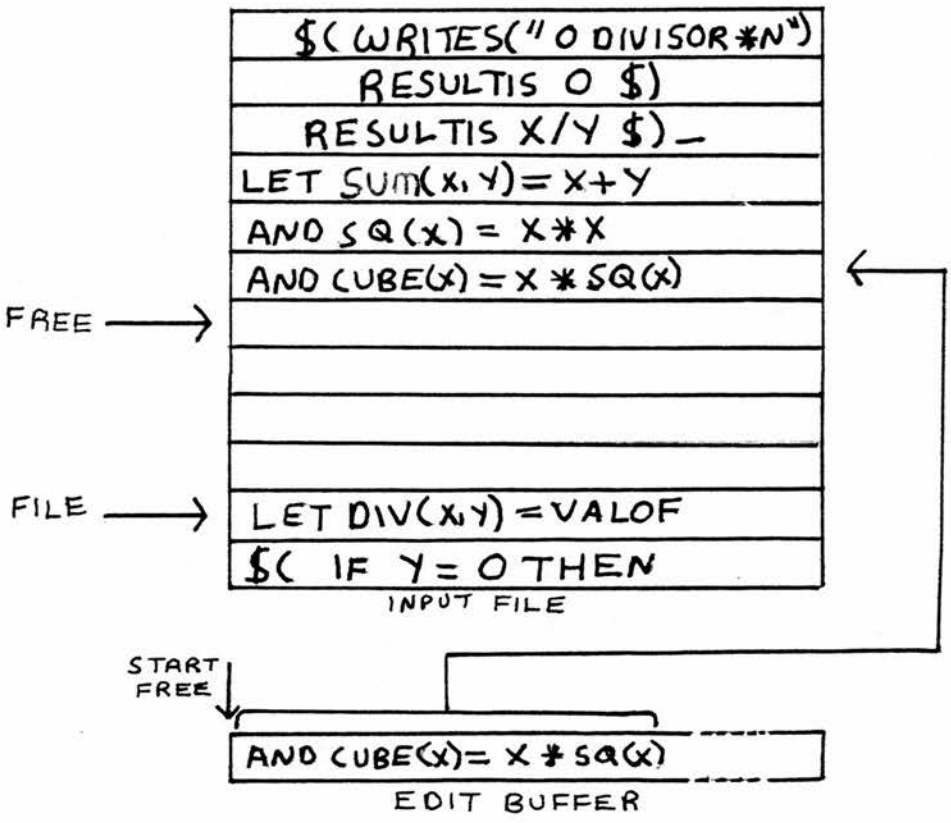

Figure 13.13 shows the input file and edit buffer after the command :-

```
ER 3,"AND CUBE(X)=X*SQ(X)".
```

Line 3 has been brought into the buffer and overwritten by the new line, which has copied back to the free space.

3.3.2.8 Page edit command - EP

Lines from the file are appended to the edit buffer text to bring the total number in the buffer to 4. These lines are displayed on the screen on numbered lines. Conversational input mode is altered [3.4] to enable re-input of the displayed text after modification. The user may change lines, indicating a modified line by placing a '#' at the end. Any text following a '#' will be deleted. When the user pushes 'return' each line is re-input to the edit buffer and scanned for a '#'. Lines starting with a '#' are deleted and those containing '#' are truncated. If the "page" is unchanged then the text is rewritten to the free space and page editing terminates. Otherwise, the "page" is re-displayed with new lines from the file inserted below to replace deleted lines.

e.g. Suppose that the program :-

```
ET
EP_
```

has been run. The system will reset the file, place the first four lines from the file in the edit buffer and display them :-

```
LET SUM(X,Y)=X+Y          001
AND SQ(X)=X*X             002
AND CUBE(X)=X*SQ(X)       003
LET DIV(X,Y)=VALOF        004
```

If the user changes line 1 :-

```
LET SUB(X,Y)=X-Y #        001
AND SQ(X)=X*X             002
AND CUBE(X)=X*SQ(X)       003
LET DIV(X,Y)=VALOF        004
```

placing a '#' at the end of the line and pushes 'RETURN', the system will re-display the lines :-

```
LET SUB(X,Y)=X-Y          001
AND SQ(X)=X*X             002
AND CUBE(X)=X+SQ(X)       003
```

```
   LET DIV(X,Y)=VALOF          004
```

If the user deletes line 3 by placing a '#' in column one :-

```
   LET SUB(X,Y)=X-Y            001
   AND SQ(X)=X*X               002
   #ND CUBE(X)=X*SQ(X)         003
   LET DIV(X,Y)=VALOF          004
```

and pushes 'RETURN', the system will remove line 3, append the next line from the file to the buffer and re-display the lines :-

```
   LET SUB(X,Y)=X-Y            001
   AND SQ(X)=X*X               002
   LET DIV(X,Y)=VALOF          003
   $( IF Y=0 THEN              004
```

If the user pushes 'RETURN', the four lines will be replaced in the file free space and future edit commands will operate from the start of line 5.

3.3.2.9 Edit repeat - <command> EREPEAT

As system commands have the same status as BCPL commands, the repeatable commands may be used to repeatedly execute edit commands. As there is no means for an EBCPL program to detect "End of file", the EREPEAT command was provided. It is processed in the same way as the REPEAT command but it halts when "End of file" is detected.

3.3.3 Execution and utilities

3.3.3.1 Execute input file program - RUN

The input file program is compiled and executed. Input is taken from the file instead of the terminal. If an error occurs, the input reverts to the terminal. Attempts to call RUN from an input file program are faulted as this can lead to infinite recursion.

3.3.3.2 Display outer level variables - DLIST

As noted above [3.2.2], outer level variables declared by each program are grouped together with details of the stack, static area and declaration table space allocated to them. The groups are numbered in increasing declaration temporal order i.e. most recently declared = group 1 etc. Each group is output in increasing order with the declaration order number below each group. Within each group, each variable name and associated address content are output.

3.3.3.3 Space reclamation - RESET [<integer>]

The stack and static space allocated to the last <integer> groups of outer level variables is reclaimed and the corresponding declaration table entries are removed. Since the space allocated to each group is linearly sequential within the stack/static area, this involves resetting the stack, declaration table and static area top pointers from the corresponding declaration table reclamation entries [3.2.1,3.2.2]. If no <integer> is present then the space

allocated to the most recent group is reclaimed.

Alas, any static space allocated to strings created by intermediate programs and assigned to non-reset outer level variables will also be reclaimed.

Fig. 14.1

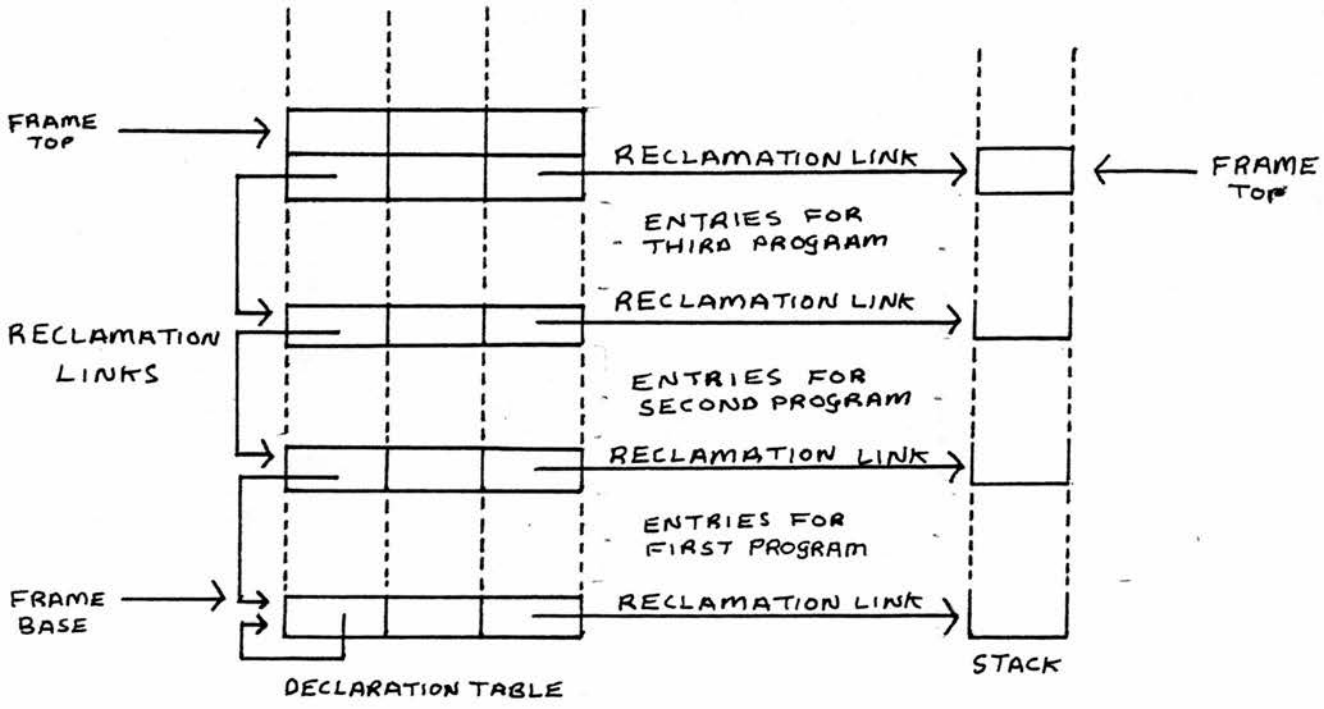

Figure 14.1 shows the stack and declaration table after three programs have declared outer level variables. Above the declaration table entries for each program's variable are the reclamation links to the previous reclamation entry and the static (not shown) and stack top pointer positions when the program terminated. The reclamation link pointer (not shown) is set to the most recent reclamation entry.

Fig. 14.2

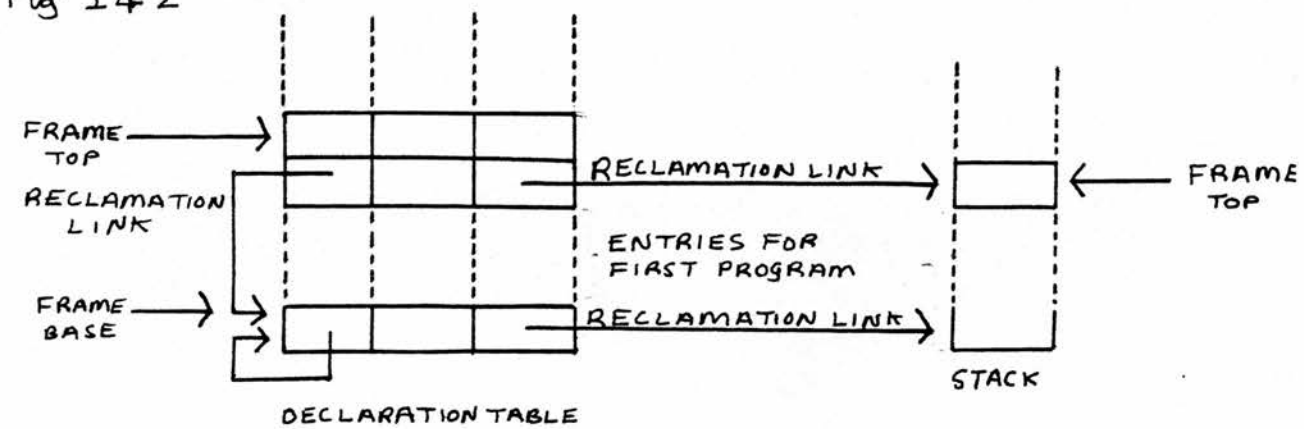

Figure 14.2 shows the effect of the command :-

RESET 2.

The reclamation links have been used to reset the declaration table and stack frame tops, and the static top pointer (not shown) to their positions after the first program terminated. The reclamation link pointer (not shown) indicates the entry for the first program.

Fig·14·3

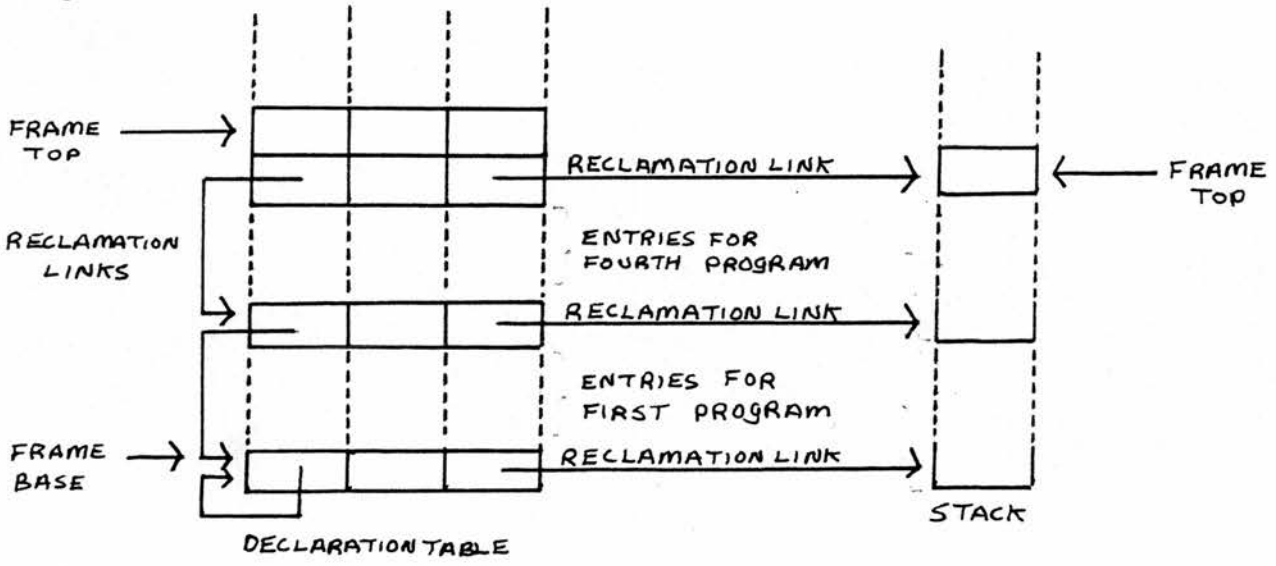

Figure 14.3 shows the stack and declaration table after a fourth program has been run. The declaration table entries for and the stack (and static) space allocated to the program over write those for the two preceding programs.

3.3.3.4 Traces

3.3.3.4.1 Options

The following options are available :-

i) Source listing - The compiler outputs the source programs with embedded error messages. This was retained from the BCPL compiler.

ii) Backtrace - After a run-time error, the routine and function calls are output in reverse calling order. This occurs as the interpreter returns to the routine call recovery label. The outer level variables declared by the program are output

with their associated values.

iii) Routine trace - For each routine/function call, the name of each formal parameter is output with its associated actual value.

iv) Timer - The execution time for each program, in 1/100ths of a second, is output when it terminates.

v) Tree - The parse tree for each program is output prior to execution. This was retained from the BCPL compiler.

vi) Loop count - The loop count [3.2.5.2] is reset to the required value.

Associated with each option is a (logical) variable. The variables are inspected at appropriate points in the interpreter, and action taken if required.

Originally arithmetic, conditional and declaration traces options were provided but these slowed the interpreter down and provided excessive information. The skeletal write command [3.3.3.5] enables user tracing of salient program points.

3.3.3.4.2 Selection flags

The following flags are used to select options :-

N - Source

B - Backtrace

I - Time

S - Routine

T - Tree

L<integer> - Loop count

3.3.3.4.3 Turn on options - ON ["flags"]

The option variables corresponding to the flags are set to TRUE. If no flags are present then all options are turned on and the loop count is set to the default value.

3.3.3.4.4 Turn off options - OFF ["flags"]

The option variables corresponding to the flags are set to FALSE. If no flags are present then all options are turned off and the loop count is set to the default value.

3.3.3.5 Skeletal write command

- * <list>

where

<list>::=<item>[,<list>]

<item>::=<string>|
<expression>

The <item>s are output sequentially, well spaced, on a new line. Each <string> is output as the corresponding character sequence. Each <expression> is evaluated in R-mode and output as an integer.

3.3.3.6 System restart command - RESTART

The stack, static area and declaration table top pointers are reset to the corresponding base pointers. All symbol table space is reclaimed.

3.3.3.7 System termination command - EXIT

Control is returned to RAX.

3.4 Interface

Routines written in BAL and PL360 were added to the interface to provide interactive I/O and a timer. The PL360 routines for basic I/O buffering were provided by Mr B.Mitchell. I modified the BAL routines for RAX input file input to accept terminal input and the BAL output routine to accept control characters for screen prompting and to re-input modified screen text.

The GLOBAL vector initialisation sequence was extended to pick up the new routines.

All this was accomplished with considerable help from Mr Mitchell, who wrote the timer routine.

4 CRITIQUE AND CONCLUSIONS

This project was over ambitious. The need to complete it within a year constrained design and implementation, resulting in pragmatic, ad-hoc decisions whose implications were not completely thought through.

Design and implementation were further hindered by the poor system and BCPL facilities provided by the RAX operating system. As noted in preceding chapters, there was limited store, no overlays, no file access, no terminal interrupt detection and no BCPL terminal I/O or control. Interactive I/O was effected by interface modification but providing file access or overlays would have constituted new projects in their own right!

While I think the use of a tree interpreter was correct, the use of the BCPL syntax analyser as the compiler was not. The desire to avoid major modifications led to the interpreter carrying out activities best dealt with at compile time; in particular unnecessary repetitive checking. It would have been better to have written a new BCPL to tree compiler which carried out static symbolic evaluation during the parse. This would enable compiler detection of multiple/ missing declarations and determination of relative addresses for dynamic variables and absolute addresses for STATIC variables. The address information could be added to the corresponding tree nodes avoiding table searching for variable references during interpretation. The standard BCPL compiler could still generate code from trees if the new information was placed in the latter fields of extended nodes, though this was not implemented in the present system.

Such checking would also provide extra error information for the whole program when it was first entered. At present, context sensitive errors in routines and functions are not detected until they are run.

The distinction between system and programming language has been pushed down a level but not removed. The effect of the test/development program distinction has been to make BCPL the system and the programming language with the development program still manipulated indirectly through the system language. The use of RAX as a model led to the system appearing to be BCPL independent while actually constrained by and oriented to BCPL.

It would have been better to have completely dropped the distinction: a system for modular program development should treat modules as the basic unit for manipulation and modification. In BCPL, modules are embodied in routines and in interactive BCPL those routines which are currently declared are accessible by name through the declaration table. No text need be held as a routine's text can be reconstructed from its tree. In fact, any declaration table name/object association can be reconstructed as a declaration and this could be used as a basis for editing and saving modules:-

e.g. EDIT <name>	-	reconstruct text for the declaration corresponding to the <name>s declaration table entry and add it to the bottom of the current reconstructed text
EDIT	-	reconstruct all declaration table entries as text
RUN	-	compile and execute reconstructed text

SAVE <name>	-	reconstruct and save text for this <name>s declaration table entry
SAVE	-	reconstruct all declaration table entries
CLEAR	-	delete all reconstructed text

The edit commands would still operate implicitly on the reconstructed text. As repeated re-entry of incorrect interactive programs proved somewhat frustrating, the text of the current program could be held on entry. This text would over-write the previous program's and be available for editing if it was incorrect. This would require double buffering of interactive program text to avoid required text being overwritten by the edit commands supposed to act on it.

The edit commands are too restricted for use in general purpose routines. They should also have BCPL variables as parameters at the expense of extra commands. Edit commands should return values or set implicit variables so that programs can identify the current line and detect the end of text.

Movement through the file is always from top to bottom and this is rather inflexible. In particular, it is not possible to move backwards through the file or to re-edit the edited portion of the current line. It may be necessary to make several changes to a line and strict left to right editing may not be convenient. Facilities should be provided for bi-directional movement. Line re-editing requires a change in approach: at present, all commands operate on the unedited portion of the file and so, for example, the replace command - ER - will ignore an occurrence of the text to be replaced if it is in the edited portion of the

current line.

The user must invoke "file reset" explicitly if an edit command reaches the end of the file while searching for text. Instead, the command might continue to search for the required text from the start of the file, and stop when the initial search position is reached once more.

As implemented, individual commands are not general enough. In particular, the delete command - ED - without a parameter, should delete the current line, and the replace command - ER - and the delete command - ED - should be combined. Insertion before as well as after specified text or a line should be provided.

System use showed that "blind" editing can be dangerous if the context is inadequately specified. An ambiguous context specification may result in an unintended change which will not be detected until the program is re-run. An additional option should be provided which would enable the display of the current line after each edit command. The absence of "display after edit" means that the page edit command must be used to display the current line, even if page editing is not required.

The page edit command is an exception to the full integration of editing commands: it is, in effect, a sub-system with two commands; "#" specifying "change and redisplay" and "return" specifying "end page edit". It is also implementation dependent.

The stack regime for space reclamation places an unnecessary

discipline on program development. All subprocesses should be equally accessible independently of the order in which they are declared. In particular, it should be possible to redeclare a subprocess without explicitly reclaiming space for it and the subprocesses declared after it, or redeclaring these subsequent subprocesses. The interpreter should recognise and allow redeclaration of outer level variables. This would require a different regime for static space organisation with either garbage collection or compaction.

Garbage collection would be costly as the number of consecutive static words that may be allocated to static objects ranges from 1 to the number required for the maximum string length (depending on the host computer - on the IBM 360 with 4 characters to a word the maximum is 64 words for 255 characters). The garbage collector would have to determine subtree node size from the node marker and, in effect, parse program trees to reclaim them. Space reclamation for strings is problematic as a string is not identifiable after evaluation and may have its character count reset or all its space used for some other purpose. This could be simplified if static space was allocated as list cells, with the compiler forming binary trees for programs and linked lists for strings but the latter would contradict the BCPL string specification. List cells would waste space , increase tree traversal time and lose compatibility with the BCPL compiler.

A compaction algorithm would have to relocate all references to compacted static objects and would have the same problem as a

garbage collector with strings.

Redeclaration of outer level dynamic variables is also not straightforward. If the stack was retained as a vector, redeclaration would result in fragmentation or loss of space.

e.g. LET A=VEC 3 - 5 words allocated

LET A=VEC 2 - redeclaration, only 4 of the original 5 words re-used

LET A=VEC 4 - redeclaration, 6 consecutive words required, original 5 words not used

This could be avoided by implementing the stack as a linked list but this would lead to increased stack access overheads as a relative stack address would no longer directly correspond to an absolute address found from an absolute stack base pointer. This also contradicts the BCPL specification of unrestricted addressing with the stack occupying a fixed linear space.

Environment entry for different commands is inefficient. In particular, a 'LOOP' environment is entered every time a loop body is executed and a new environments is entered for a block body in a routine or repeatable command. Only one environment should be required to handle both 'BREAK' and 'LOOP' in repeatable commands and this should be entered once, when the command execution begins. A new environment should not be entered for a blockbody at the start of a routine or repeatable command body.

While the skeletal write command makes tracing easier, user tagged tracing should be introduced to output the corresponding

program construct as text and information appropriate to the construct.

The absence of text reconstruction from trees greatly reduced the value of diagnostic output. The Anglicised tree nodes aid comprehension but are no substitute for source code.

APPENDIX 1

The User Guide to Interactive BCPL

Greg Michaelson July 1975 - revised July 1981

Notes

This guide assumes that you are familiar with :-

a) The BCPL Programming Manual - M.Richards

b) Introduction to the use of RAX with 2260's
- B.T.Mitchell CL/72/5

The following abbreviations are used in the programming examples :-

u - user input

p - program output

s - system output

System prompts are enclosed in quotes e.g. 'O.K.'

Each example assumes an understanding of the previous one.

INDEX

0 Introduction

Interactive BCPL is an interactive system for the development of BCPL programs, and is run under RAX from a 2260 terminal. The system command language is an extension of BCPL - EBCPL - and the system loops, reading EBCPL programs from the terminal and executing them interpretively. The texts of these programs are not held, and cannot be rerun or corrected, but system commands are provided for writing programs to the input file, and then running, editing, rerunning and saving them.

1 Programs

```
<EBCPL program> ::= <declaration part>_¦
                    <command list>_¦
                    <declaration part>;<command list>_

<declaration part> ::= <declaration>[;<declaration>]

<command list>     ::= <command>[;<command>]
```

You can find full details of these constructs in the BCPL programming manual.

1.1 Programs must be terminated by an underbar. The underbar is a reserved character, but '*_' may be used instead of '_' in strings.

1.2 System commands may appear in any position where BCPL commands are valid. System command names are reserved words, and attempts to redefine them will be faulted. You can find a complete list of system commands in Appendix 4.

1.3 The GOTO and labeled commands have not been implemented. Programs containing these will be syntactically checked but will not be run.

1.4 CASE and DEFAULT prefixes will be ignored unless the corresponding commands appear at the outer level of a block forming the body of a SWITCHON command.

2 Program entry from the terminal

The system prompt 'O.K.' indicates that the system is is waiting for you to enter an EBCPL program from the terminal. If the program extends over several pages, press 'SHIFT and ENTER' at the end of each one and enter the next page after the prompt 'ENTER DATA'. The program will be syntactically checked as it is read in, and if it contains errors, the system will not execute it. Otherwise, the program will be executed until it terminates or an interpretive error is discovered.

3 Outer level variables

Variables declared in the outermost declaration part of a program remain declared after that program successfully terminates. They may then be referred to by subsequent programs. Such variables are called "outer level variables". Outer level LET variables (and MANIFEST identifiers) are allocated static space, but retain their usual scope properties.

```
e.g. s: 'O.K.'

     u: LET A,B,C=1,2,3_

     s: 'O.K.' - A,B & C are now available for subsequent access

     u: A:=B+C_

     s: 'O.K.' - A is now set to 5

     u: LET V=VEC 5
        FOR I=0 TO 5 DO V!I:=I*I_

     s: 'O.K.' - V is now available for subsequent access
                 and set to 0 1 4 9 16 25
```

Outer level variables cannot be redeclared.

```
e.g. u: LET A=123_

     s: 3 NAME DECLARED ALREADY - A
        'O.K.'
```

If a program contains an interpretive error, any space allocated to outer level variables as a result of the program's partial interpretation is reclaimed, and the corresponding identifiers are deleted. Outer level variables declared by the preceding programs are not affected.

4 Skeletal write command ::= * <expression list>

The skeletal write command may be used for simple program output. The expression list is printed and automatically terminated with a new-line.

```
e.g. u: *A,B,C_

     p: 5 2 3

     s: 'O.K.'

     u: FOR I=0 TO 5 DO *I,"SQUARED =",V!I

     p: 0 SQUARED = 0

        1 SQUARED = 1

        2 SQUARED = 4

        3 SQUARED = 9

        4 SQUARED = 16

        5 SQUARED = 25

     s: 'O.K.'
```

This command may also be used to trace a program, by inserting appropriate calls to it in the requisite positions in the program.

5 Routines and functions

All routines and functions are allocated static code space. If they are declared in the outermost declaration part of a program, the variables they are associated with are declared as outer level variables and allocated a static word which is set to the code address. They may then be called in subsequent programs and in the routines and functions they declare.

```
e.g. u: LET FAC(N)=N=0->1,N*FAC(N-1)_

     s: 'O.K.' - FAC is now available for subsequent access

     u: FOR I=0 TO 4 DO *FAC(I)_

     p: 1

        1

        2

        6

        24

     s: 'O.K.'

     u: LET BINEXP(X,Y)=FAC(X)/(FAC(X-Y)*FAC(Y))_

     s: 'O.K.'

     u: *BINEXP(3,0),BINEXP(3,1),BINEXP(3,2),BINEXP(3,3)_

     p: 1 3 3 1

     s: 'O.K.'
```

If routines and functions are declared in inner blocks and linked to an outer level variable they may be called in subsequent programs through that outer level variable.

```
e.g. u: STATIC $( CUBE=0 $)
        IF CUBE=0 THEN
        $( LET FN(I)=I*I*I
           CUBE:=FN
        $)
        FOR I=1 TO 4 DO *CUBE(I)_

     p: 1

        8

        27

        64

     s: 'O.K.' - CUBE holds FN's address and
                 may be used to call FN
```

If a program or a routine or function declared and called within it contains an interpretive error, any space allocated to routines, functions and associated outer level variables declared by the program is reclaimed and the corresponding identifiers are deleted. Routines, functions and outer level variables declared by preceding programs are not affected - see 3.

Routines and functions are not checked for interpretive errors until they are called.

6 Outer level variables listing command ::= DLIST

The DLIST command is used to list the current outer level variables and their values in the reverse order to that in which they were declared.

```
e.g. u: DLIST_

     s: ...0...

        CUBE  5260

        ...1...

        BINEXP  5240

        ...2...

        FAC  5219

        ...3...

        V  7452

        ...4...

        C  3

        B  2

        A  5

        ...5...
```

The numbered lines delineate the variables declared by each program.

7 Space reclamation command ::= RESET <number> ¦ RESET

The RESET command is used in conjunction with the DLIST command to selectively reclaim storage space. A call to RESET <number> will reclaim the space allocated to the variables above the line numbered <number> in the list output by the DLIST command. Space allocated to routines, functions, strings and TABLES declared in the corresponding programs will also be reclaimed.

```
e.g. u: RESET 3
        DLIST_

     s: ...0...

        V  7452

        ...1...

        C  3

        B  2

        A  5

        ...2...

        'O.K.'
```

A call to RESET is equivalent to a call to RESET 1.

8 Expressions in declarations

In BCPL, only constant expressions may appear on the right hand side of GLOBAL, MANIFEST, STATIC and VEC declarations, and in CASE and TABLE expressions. This restriction does not apply to EBCPL, enabling the use of dynamic vectors in routines and inner blocks - see the example in 14.

9 Blocks

Blocks must be preceded or followed by a command if they are to be used to allocate temporary space to local variables.

```
e.g. u: LET FRED=1
        *FRED
        $( LET FRED=2
           *FRED
           $( LET FRED=3
              *FRED
           $)
           *FRED
        $)
        *FRED_

     p: 1

        2

        3

        2

        1

     s: 'O.K.'
```

Programs of the form :-

$(<declaration part>;<command list> $)

or

<declaration part>;$(<declaration part>;<command list> $)

will be treated as if the section brackets were not present.

```
e.g. u: $( LET A=69
         *A
       $)_

     s: 3 NAME DECLARED ALREADY - A

        'O.K.'

     u: LET X=1
        $( LET X=2
         *X
        $)_

     s: 3 NAME DECLARED ALREADY - X

        'O.K.'

     u: $( LET X=7
         *X
        $)_

     p: 7

     s: 'O.K.' - X is now available for subsequent use.
```

10 Repeatable commands

For any one execution of a repeatable command, the number of times its body may be executed is limited to avoid infinite loops. This number is set to 1000 on entry to the system, and may be changed through the use of the system option L - see 16.

11 Global declarations

Global locations above 300 are used by the system and attempts to declare them will be faulted. Please avoid assignments to addresses which are equivalent to system global locations as they will thoroughly confuse the system.

12 Global clashes

A warning will be output in the event of a global clash but the program causing it will still be executed.

```
e.g. u: GLOBAL $( FISH:172 $)
        FISH:=123_

     s: 'O.K.'

     u: GLOBAL $( CRAB:172 $)
        CRAB:=321_

     s: WARNING - GLOBAL 172 DECLARED ALREADY - FISH
        'O.K.' - global 172 is now set to 321 and
                 may be accessed through FISH or CRAB
```

13 The basic library and I/O

The BCPL library is loaded with the system and library routines and functions may be accessed through the global vector. Appendix 2 contains a full list of the basic library routines and functions.

RDCONV - global 80 - and READN - global 70 - may be used to read characters and numbers respectively from the terminal. Data will be requested by the RAX prompt 'ENTER DATA'.

```
e.g. u: GLOBAL $( READN:70 $)
        LET MAX=0
        FOR I=1 TO 5 DO
        $( LET TEMP=READN()
           IF TEMP>MAX THEN
            MAX:=TEMP
        $)
        *"MAX =",MAX_

     s: 'ENTER DATA'

     u: 1 3 5 2 4

     p: MAX = 5

     s: 'O.K.'
```

Excess input will be picked up by the system and faulted as invalid program text. Please ensure that your programs do not include calls to these functions in non-terminating repeatable commands.

RDCH - global 13 - may be used to read characters from text included or inserted below /DATA when the system was loaded. See Appendix 1.

WRITEVEC - global 24 - may be used to write a vector of 80 characters to the RAX SYSOUT file to be saved in your own RAX file - see 23.

All other library output routines will write to the terminal. If the last line of your program output is not terminated with a new-line it may be lost or mixed up with system output.

14 Using your own compiled routines

User routines which have been compiled with global linkages may be included with the system when it is loaded - see Appendix 1 - and accessed through the global vector. The system will be extremely confused by compiled user routines which are linked to global vector locations between 1 and 81 or above 300.

e.g. to test routine SORT(V,N) which sorts the N elements of vector V. SORT has been compiled, linked to global 200 and included with the system at load time.

```
u: GLOBAL $( SORT:200 $)
   $( LET N=READN()
      IF N=0 THEN BREAK
      $( LET V=VEC N
         FOR I=1 TO N DO V!I:=READN()
         FOR I=1 TO N DO *I,V!I
         SORT(V,N)
         FOR I=1 TO N DO *I,V!I
      $)
   $) REPEAT_

s: 'ENTER DATA'

  u: 3 9 14 7

p: 1  9

   2  14

   3  7

   1  7

   2  9

   3  14

s: 'ENTER DATA'

u: 6 6 5 7 3 1 2
```

```
p: 1 6
   2 5
   3 7
   4 3
   5 1
   6 2
   1 1
   2 2
   3 3
   4 5
   5 6
   6 7
s: 'ENTER DATA'
u: 0
s: 'O.K.'
```

Care should be taken when using compiled routines. If they contain run time errors they may crash the system.

15 Errors

If a program contains a syntactic error, the system will print the line number, the error message and the last 64 characters read during syntactic analysis. The rest of the program will be checked for syntax errors but it will not be executed and any space allocated to functions, routines and strings declared by the program will be reclaimed.

If a program contains an interpretive error, the system will print the error message and terminate the program's execution. Any space allocated to routines, functions, strings, TABLEs and outer level variables will be reclaimed and the corresponding identifiers will be deleted. You should re-initialise any outer level variables which have been modified during the program's partial interpretation. See Appendix 3 for hints on trouble shooting.

16 System options

System options may be turned on and off with the ON and OFF commands.

<ON command> ::= ON "<option list>" | ON

<OFF command> ::= OFF "<option list>" | OFF

<option list> ::= <option code>[<option list>]

<option code> ::= N - list the program's text during compilation

T - print the program's tree and 'vecsize'

D - declaration trace - for each variable declared, list its storage type, name and initial value

S - routine and function trace - for each routine or function call, list the calling expression and the parameter list

B - backtrace - after an interpretive error list all variables and their values in the reverse order to that in which they were declared, and the calling expression and address of each routine or function currently being executed, in reverse call order

I - print the program's execution time in 1/100 ths of a second - the timer is extremely unreliable

L<number> - set the number of times the body of each repeatable command may be executed, for any one execution of that command - default = 1000

The options are all turned off, and the L option is set to the default value, on entry to the system and after a restart.

An ON command without options will turn on all options, and set the L option to the default value.

An OFF command without options will turn off all options, and set the L option to the default value.

Options remain on until they are turned off.

17 Program development

Programs entered from the terminal for immediate execution cannot be corrected, rerun or saved. The system provides a RAX like input file which can hold 6000 characters of program text i.e. 120 lines of 50 characters. A program which is run from the input file will be executed in the same way as a program entered from the terminal, and you can use the system commands to edit, rerun and save it.

You can call the INPUT command to fill the input file from the terminal, or the DATA command to fill the input file from a RAX file which was included with the system when it was loaded. The program should be written as a sequence of routine and function declarations so that it is compatible with the BCPL compiler.

You can then call the RUN command to execute the program. If it contains syntactic errors you can correct it with with the edit commands and rerun it. When it successfully terminates, the routines and functions will be declared at the outer level and can be tested by short EBCPL programs.

If the routines and functions contain interpretive errors you can correct them, reclaim the space allocated to the program and rerun it.

When you are satisfied with the program you can call the SAVE command to write the program to the RAX SYSOUT file to be saved in your own RAX file and compiled at your leisure.

18 Input command ::= INPUT

The input file is initialised and filled with program text, terminated by an underbar, entered from the terminal. The system will request the first and subsequent pages of the text with the prompt 'ENTER DATA'. A single INPUT command, terminated by an underbar, may be immediately followed by the first page of text. When the file is full, a warning message will be output.

```
e.g. u: INPUT_
        LET ADD(X,Y)=X+Y
        AND SUB(X,Y)=X-Y
        AND MULT(X,Y)=X*Y
        AND DIV(X<Y)=VALOF
             TEST Y=0 THEN
             $( WRITES("ZERO DIVISOR *N")
                RESULTIS 0 $)
             ELSE
                RESULTIS X/Y_

     s: 'O.K.'
```

19 Data command ::= DATA

The input file is initialised and filled with program text which has been included or inserted below /DATA when the system was loaded - see Appendix 1.

The text need not be terminated by an underbar.

20 Insert command ::= INSERT

Program text terminated by an underbar, from the terminal, is inserted below the last line of text in the input file - see 18.

21 Display command ::= DISPLAY

The input file is listed on the terminal screen with each line numbered.

22 Run command ::= RUN

The program from the input file is executed as if it had been entered from the terminal.

```
e.g. u: RUN
        *ADD(1,2),SUB(1,2),MULT(3,4),DIV(10,2),DIV(15,0)_

     p: 3  -1  12  5
        ZERO DIVISOR
        0

     s: 'O.K.' - ADD, SUB, MULT & DIV are now
                 available for subsequent use.
```

23 The edit commands

The following commands provide you with facilities for editing the input file without necessarily inspecting it or entering a special edit mode. You are not advised to use these commands in development programs.

The input file is manipulated sequentially from top to bottom. When the end of the file is reached a warning is output

i.e. .....E.O.F.....

Thenceforth, any attempt to go beyond the end of the file is faulted.

The system maintains an edit file pointer which is set to the start of the first line on entry to the system and reset after calls to non edit system commands other than the skeletal write command.

24 Top command ::= ET

The (edit) file pointer is reset to the start of the first line.

25 Move command ::= EM <(line) number>

The file pointer is set to the start of the specified line.

26 Find command ::= EF <string>

The file pointer is set to the end of the start of the specified string.

27a Delete command ::= ED <(line) number>

The specified line is deleted and the file pointer is set to the start of the next line.

27b Delete command ::= ED <string>

The next occurrence of the specified string is deleted and the file pointer is set to the position it occupied.

28a Insert command ::= EI <string>

The string is inserted at the position indicated by the file pointer which is then set to the end of the string.

28b Insert command ::= EI <(line) number>,<string>

The string, terminated by a new line symbol, is inserted after the specified line. The file pointer is set to the start of the next line.

28c Insert command ::= EI <string>,<string>

The second string is inserted after the next occurrence of the first string. The file pointer is set to the end of the second string.

```
EI "s1","s2" => EF "s1"; EI "s2"
```

29a Replace command ::= ER <(line) number>,<string>

The specified line is replaced by the string, which is terminated with a new-line symbol. The file pointer is set to the start of the next line.

```
ER n,"s" => ED n; EI "s*N"
```

29b Replace command ::= ER <string>,<string>

The next occurrence of the first string is replaced by the second string. The file pointer is set to the end of the second string.

```
ER "s1","s2" => ED "s1"; EI "s2"
```

30 Edit repeat command ::= <edit command list> EREPEAT

<edit command list> ::= <edit command> |
$(<edit commands> $)

<edit commands> ::= <edit command>[;<edit commands>]

The edit command(s) are repeatedly obeyed until the end of the file is reached. The warning message is then output and the command terminates.

31 Page edit command ::= EP

The line containing the file pointer and the next three lines are output to the terminal screen. The file pointer is not printed.

Move the cursor along the lines and make the requisite changes. Terminate each line which you change with a '#'. Characters after the '#' are discarded, and a line may be completely deleted by placing the '#' in column 1. The line numbers start in column 75 and the '#' should not be placed in or after this column.

After inspecting or modifying the page, press 'SHIFT & ENTER'. If the page has not been modified it will be replaced in the file and the file pointer will be set to the start of the next line. If the page has been modified, it will be reoutput to the terminal screen. A line from the file will be added at the bottom of the page to make up for each line that is deleted.

e.g. u: EP_

```
s: LET ADD(X,Y)=X+Y          0001
   LET SUB(X,Y)=X=Y          0002
   LET MULT(X,Y)=X*Y         0003
   LET DIV(X,Y)=VALOF        0004

u: ( making change to line 1 and deleting line 2 )

   LET SUM(X,Y)=X+Y#         0001
   #ET SUB(X,Y)=X=Y          0002
   LET MULT(X,Y)=X*Y         0003
   LET DIV(X,Y)=VALOF        0004

s: LET SUM(X,Y)=X+Y          0001
   LET MULT(X,Y)=X*Y         0002
   LET DIV(X,Y)=VALOF        0003
        TEST Y=0 THEN        0004

u: 'RETURN'

s: 'O.K.' - updated lines replaced in file
          - editing continues from line 5
```

```
32    Edit command examples

u: ER "AND","LET" EREPEAT
   DISPLAY_

s: .....E.O.F.....

   1 LET SUM(X,Y)=X+Y
   2 LET MULT(X,Y)=X*Y
   3 LET DIV(X,Y)=VALOF
   4       TEST Y=0 THEN
   5      $( WRITES("ZERO DIVISOR *N")
   6         RESULTIS 0 $)
   7       ELSE
   8         RESULTIS X/Y
   'O.K.'

u: ED 3
   ED 4
   ED 5
   ED 6
   ED 7
   ED 8
   ET
   EF "MULT"
   FOR I=1 TO 2 DO ER "Y","FROG"
   DISPLAY_

s: 1  LET SUM(X,Y)=X+Y
   2  LET MULT(X,FROG)=X*FROG
   'O.K.'

u: EI 2,"LET HELLO() BE WRITES(*"HELLO*N*")"
   DISPLAY_


s: 1  LET SUM(X,Y)=X+Y
   2  LET MULT(X,FROG)=X*FROG
   3  LET HELLO() BE WRITES("HELLO*N")
   'O.K.'
```

```
u: EF "HELLO"
   EI "O","O!!! "
   DISPLAY_

s: 1  LET SUM(X,Y)=X+Y
   2  LET HELLO() BE WRITES("HELLO!!!*N")
   3  LET MULT(X,FROG)=X*FROG
   'O.K.'

u: EP EREPEAT_

s: - the whole file is listed in page edit mode

   'O.K.'

u: $( EF <string>;EP $) EREPEAT_

s: every occurrence of the string is listed,
   in context, in page edit mode

   'O.K.'
```

33 Restart command ::= RESTART

The RESTART command will reclaim all space allocated since the system was loaded or last restarted, delete all outer level variables and turn off all system options. The input file will not be affected.

34 Exit command ::=

```
e.g. u: EXIT_

   RAX: 'READY'
```

35 Compiling programs developed with the system

Before trying to compile a program developed with the system you should :-

a) remove any calls to Interactive BCPL system commands

b) replace any non-constant expressions in places where they are not permitted in BCPL

c) if the program contains a <command list>, rewrite it as a routine declaration

d) if the program refers to outer level variables, insert the corresponding declarations at the top of the program

36 Save command ::= SAVE

The SAVE command is used to write the input file to the RAX SYSOUT file which can then be written to your own file file using the RAX /SAVE command.

```
e.g. u: SAVE
        EXIT_

   RAX: 'READY'

     u: /SAVE name(lock),SV

   RAX: 'READY' - tne input file is now in the named RAX file
```

APPENDIX 1 - Starting the system

Run the following job under RAX from a 2260 terminal :-

```
/INPUT

/JOB GO,T= your maximum time

/INCLUDE GMGIB1

[/INCLUDE your file] - your own compiled routines
                       to be accessed through the
                       global vector - see 14

/INCLUDE GMGIB2

[/INCLUDE your file] - a) BCPL text to be written to
                          the input file - see 19

                       or

                       b) data - see 13
```

The system will write :-

```
BCPL BCPL BCPL BCPL BCPL BCPL BCPL BCPL BCPL BCPL BCPL BCPL BCPL
BCPL BCPL BCPL BCPL BCPL
```

Press 'SHIFT & ENTER' and the system will prompt :-

'O.K.'

APPENDIX 2 - Basic library routines and functions

A '*' indicates that the routine or function is the same as that in the basic library for compiled BCPL. Details may be found in the BCPL programming manual.

A 'N' indicates that the routine or function should not be used with the system.

GLOBAL	NAME	NOTES
3	ABORT	*N
4	BACKTRACE	*N
13	RDCH	*
14	WRCH	*
24	WRITEVEC	*
28	TIME	returns the time of day in 1/150ths of a second - unreliable
31	LEVEL	*N
32	LONGJUMP	*N
40	APTOVEC	* should only be used with machine code programs
43	SPACE	SPACE(N) writes N spaces
60	WRITES	*
62	WRITEN	*
63	NEWLINE	*
64	NEWPAGE	*
65	WRITEO	*
66	PACKSTRING	*
67	UNPACKSTRING	*
68	WRITED	*
69	WRITEARG	*
70	READN	reads and returns a number from the terminal
74	WRITEX	*
75	WRITEHEX	*
76	WRITEF	*
77	WRITEOCT	*
78	MAPSTORE	*N
79	PRMPT	the sequence :- PRMPT('+');PRMPT(char1)...PRMPT('*N') will send a prompt to the terminal - see global 81
80	RDCONV	reads and returns a character from the terminal
81	PROMPT	PROMPT(<string>) will prompt the string to the terminal. If this routine is not followed by READN or RDCONV, the system will wait for 'SHIFT & ENTER' after the prompt has appeared before continuing

APPENDIX 3 Troubleshooting

a) RAX will not load or execute the system

This should only occur if you are trying to use the system to test compiled routines and functions, and there is not enough room in memory for them - see G.M.

b) the system crashes

Please see G.M. with a listing of any system output, and the details of what you were trying to do.

c) 'ENTER DATA' repeatedly requested

i) You have forgotten to terminate a program or input to the input file with an underbar

ii) Your program contains a nonterminating conversational read loop. This is an unrecoverable state - enter '/CANCEL', sighing.

d) 'SHIFT & ENTER' sticks

This may occur as a result of fiddling while watching

'EXECUTION IN PROGRESS, WAIT FOR MORE OUTPUT'

If no other users are affected, wait and the terminal will clear itself.

e) System error messages

i) PROG TOO BIG

The <command list> in your program is too big, and you should embody more of it in routines and functions.

ii) STATIC STORAGE EXCEEDED

There is no more static space for routines, functions, strings, TABLES and STATICS.

iii) STACK OVERFLOW

There is no more stack space for parameters, LET and VEC variables, and MANIFEST identifiers. You may have routine or function calls that are nested too deeply.

iv) DECL TABLE OVERFLOW

The identifier declaration table is full.

For errors ii) to iv) you can selectively reclaim space with the RESET command or reclaim it all with the RESTART command.

v) TOO MANY NAMES

There is no more space for identifiers name strings and you must use the RESTART command to restart the system.

APPENDIX 4 - System Commands

NAME	DETAILS
DATA	- fill input file from /DATA - see 19, APP 1
DISPLAY	- display input file on screen - see 21
DLIST	- list outer level variables - see 6
ED <(line) number>	- delete line - see 28a
ED <string>	- delete string - see 28b
EF <string>	- find string - see 27
EI <(line) number>,<string>	- insert string after line - see 29b
EI <string>	- insert string - see 29a
EI <string>,<string>	- insert string after string - see 29c
EM <(line) number>	- move to line - see 26
ER <(line) number>,<string>	- replace line with string - see 30a
ER <string>,<string>	- replace string with string - see 30b
EREPEAT	- edit repeat command - see 31
ET	- move to file top - see 25
EXIT	- leave the system - see 35
INPUT	- fill input file from terminal - see 18
INSERT	- insert text to input file from terminal - see 20
OFF	- turn off options - see 16
ON	- turn on options - see 16
RESET[<number>]	- reclaim space - see 7
RESTART	- restart the system - see 34
RUN	- run program in input file - see 22
SAVE	- write input file to SYSOUT - see 23
* <expression list>	- skeletal write - write expression

list to screen
- see 4

ACKNOWLEDGEMENTS

I would like to thank Tony Davie, Dave Turner and Bruce Mitchell for their help and encouragement.

Criticisms and suggestions for improvements will be gratefully received.

APPENDIX 2

BIBLIOGRAPHY

*1 Comparative study of programming languages course
- Prof. R.A.Brooker, Dept. of Computer Science,
University of Essex 1972/73

*2 BCPL - the language and its compiler
Richards & Whitby-Strevens C.U.P. 1979
This volume is based on the original BCPL papers
which appeared between 1969 and 1971

*3 The IMP interpreter running under EMAS uses this technique
- private conversation with P.Robertson, Dept. of Computer
Science, University of Edinburgh 1974

*4 Introduction to the use of RAX with 2260s
B.Mitchell St Andrews University Computer Laboratory
CL/72/5 1973

The use of RAX using CDC 713 Video Terminals or
ASR 33 Teletype Terminals
J.R.Stapleton St Andrews University Computer Laboratory
CL/73/4 1973

*5 ALGOLW Language Reference Manual
St Andrews University Computer Laboratory
CL/72/8

*6 SASL Language Manual
D.A.Turner Dept. of Computational Science,