A Pascal Programming Environment

I declare that this thesis is a report of my own original work, and that no part of it has been previously accepted or presented for the award of any degree or diploma by any University, and that to the best of my knowledge no material previously published or written by another person is included, except where due acknowledgement is given.

A Thesis submitted for the degree of
Master of Science
in the
Australian National University

G. Huston
Australian National University
June, 1982

GEOFFREY IAN HUSTON
I declare that this thesis is a report of my own original work, and that no part of it has been previously accepted or presented for the award of any degree or diploma by any University, and that to the best of my knowledge no material previously published or written by another person is included, except where due acknowledgement is given.

G. Huston
Australian National University
June, 1983.
# Table of Contents

Acknowledgements .......................................................... 1  
Abstract ............................................................................. 2  
1. Introduction ................................................................... 3  
   1.1 The Program Development Environment ....................... 3  
   1.2 Software Tools for the Pascal Environment ................... 5  
   1.3 Implementation Aspects ............................................ 7  
2. The User Level Environment - A Survey ........................... 8  
   2.1 A Description of the User Environment ....................... 9  
   2.2 Conventional User Environments ................................ 11  
      2.2.1 The File Store ................................................... 11  
      2.2.2 The Command Language .................................... 13  
      2.2.3 Common Limitations of Conventional Environments .... 17  
   2.3 Integrated Program Development Systems - A Survey ..... 18  
      2.3.1 The Interlisp Environment ..................................... 18  
      2.3.2 The UNIX Environment ....................................... 19  
      2.3.3 Ada Directions .................................................. 21  
      2.3.4 Pascal Environments ........................................... 22  
   2.4 A Pascal Program Development Environment ................. 26  
3. The Pascal Workbench .................................................... 31  
   3.1 The Role of the Workbench ......................................... 31  
   3.2 Design Goals of the Workbench Environment ................. 34  
      3.2.1 Files - The Basis of Commands .............................. 34  
      3.2.2 The Display Process .......................................... 35  
      3.2.3 The Command Set ............................................. 39  
      3.2.4 Advanced Command Features - The Macro Facility ...... 41  
         3.2.4.1 Terminal Function Keys .................................. 42  
         3.2.4.2 Local Micro System ...................................... 42  
         3.2.4.3 Front End Devices ........................................ 43  
         3.2.4.4 Macro Command Features at the Command Level .... 44
4.3.3.4 Algorithms and Maintenance 81
4.3.3.5 Text Accessibility 82
4.3.3.6 String Locate 82
4.3.3.7 User Priority 82
4.3.3.8 Allowance for the User 83

4.4 A Case Study 83

4.4.1 Memory Management 83
4.4.1.1 The Edit Buffer Memory Management Scheme 85
4.4.1.2 Linked Lines 86
4.4.1.3 A Paged Management Strategy 88
4.4.1.4 A Multiple Buffer Gap Strategy 91
4.4.2 The Editor Command Language 94
4.4.2.1 Printable Character Commands 94
4.4.2.2 The Escape Key 95
4.4.2.3 Command Prefixing 95
4.4.2.4 Arguments to Commands 96
4.4.2.5 Aborting and Undoing commands 97
4.4.2.6 Key Bindings 98
4.4.2.7 Summary 98

4.4.3 The Display Controller 98

5. The Pascal Language Environment 106

5.1 Introduction 106
5.2 Errors, Compilation and Editing 106
5.2.1 Compiler Modifications 111
5.2.2 Editor Modifications 113
5.2.3 The Resultant Editor Environment 114
5.3 Editing Pascal Text 115
5.3.1 The User View of a Pascal Text Editor 118
5.3.2 A Case Study - Implementation of a Pascal Editor 122
5.3.2.1 Introduction 122
5.3.2.2 Statements and Templates 122
5.3.2.3 Expressions and Parameter Lists 126
5.3.2.4 Variables, Constants, Labels and Data Types 128
5.3.2.5 The Display Driver 130
5.3.2.6 Implementation Issues 132
5.4 Debugging Tools 134
5.4.1 User Requirements of a Debug Environment 136
5.4.2 Implementation of a Pascal Debug System 142
5.4.3 A Case Study: A Pascal Debug System 145
5.4.3.1 The DECSys-tem-10 Pascal Debug System 146
5.4.3.2 The Pascal-M System 147
5.4.3.3 Implementation of DECSys-tem-10 Pascal Debug 148
5.4.3.4 Implementation of Pascal Debug System 153
6. Further Developments 159
6.1 Portability of the Workbench System 159
6.2 Extensions to the Workbench 160
6.3 The Pascal Language Sub-System 161
7. References 164
I. The Pascal Abstract Machine 179
1.1 Basics 179
1.2 Operation of the Pascal Abstract Machine 181
II. User Guide for the Pascal Workbench 184
III. User Guide for the FRED Text Editor 189
III.1 Invoking FRED 189
III.2 The FRED Display During Editing 190
III.3 Input and Output File Conditions 190
III.4 FRED Commands 191
IV. The User Subsystem Hardware 201
IV.1 The Keyboard 201
IV.2 Other Input Devices 201
IV.3 Display Devices – The Terminal Model 202
IV.4 Memory-Mapped Display 203
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Model of the Interactive Environment</td>
<td>4</td>
</tr>
<tr>
<td>2-1</td>
<td>The User Level Abstract Machine</td>
<td>10</td>
</tr>
<tr>
<td>2-2</td>
<td>APSE organisation</td>
<td>23</td>
</tr>
<tr>
<td>3-1</td>
<td>File Store Structure</td>
<td>36</td>
</tr>
<tr>
<td>3-2</td>
<td>Workbench Screen Format</td>
<td>37</td>
</tr>
<tr>
<td>3-3</td>
<td>The Virtual Macro Command Processor</td>
<td>41</td>
</tr>
<tr>
<td>3-4</td>
<td>Parallel Pipe Organisation</td>
<td>47</td>
</tr>
<tr>
<td>3-5</td>
<td>Internal Organisation of the Workbench</td>
<td>50</td>
</tr>
<tr>
<td>3-6</td>
<td>Sub-Procees Management and I/O Redirection</td>
<td>55</td>
</tr>
<tr>
<td>3-7</td>
<td>Process Management Organisation</td>
<td>56</td>
</tr>
<tr>
<td>4-1</td>
<td>The Editing Abstract Machine</td>
<td>62</td>
</tr>
<tr>
<td>4-2</td>
<td>Typical Internal Organisation of a Text Editor</td>
<td>63</td>
</tr>
<tr>
<td>4-3</td>
<td>Star System Window Format</td>
<td>73</td>
</tr>
<tr>
<td>4-4</td>
<td>The Buffer Gap Scheme</td>
<td>85</td>
</tr>
<tr>
<td>5-1</td>
<td>The Binding of the Editor and Compiler</td>
<td>110</td>
</tr>
<tr>
<td>5-2</td>
<td>The Structure of Error Tags in the Editor</td>
<td>110</td>
</tr>
<tr>
<td>5-3</td>
<td>Implementation View of Editor/Compiler Interface</td>
<td>114</td>
</tr>
<tr>
<td>5-4</td>
<td>Statement Template Structure</td>
<td>123</td>
</tr>
<tr>
<td>5-5</td>
<td>Pascal Editor Screen Display</td>
<td>131</td>
</tr>
<tr>
<td>5-6</td>
<td>Pascal Expression Editor Screen Display</td>
<td>132</td>
</tr>
<tr>
<td>5-7</td>
<td>Error trapping on the DECSystem-10</td>
<td>150</td>
</tr>
<tr>
<td>5-8</td>
<td>Internal Debug System Organisation on the DECSystem-10</td>
<td>152</td>
</tr>
<tr>
<td>5-9</td>
<td>Screen Display for DEBUG System</td>
<td>154</td>
</tr>
<tr>
<td>5-10</td>
<td>Internal Organisation of Screen Debug System</td>
<td>158</td>
</tr>
</tbody>
</table>
Figure 6-1: Internal Organisation of a Pascal Language System

Figure A-1: The Initial State of the Pascal Machine
Acknowledgements

I would like to thank my supervisor, Dr Brian Molinari, for his interest and guidance throughout this project. I am particularly grateful to him for his careful reading of drafts of this thesis, and useful discussions on this subject.

I would also like to express my sincere appreciation of the constant support and encouragement of Michele Huston, particularly through the latter stages of this work.

I am also indebted to the Computer Science Department of the Australian National University, and the large number of students who used, and thoroughly tested, all parts of this software on the Faculties Computer Unit system.
Abstract

This thesis addresses the topic of software tools involved in a programming environment for the development of Pascal programs.

It examines the major components of such an environment, surveying existing software tools where applicable, and presenting an implementation of a portable environment, designed to meet such a goal. The major components of such an environment, a command interpreter, a text editor and the Pascal processor subsystem are examined in detail, presenting the major design goals of each component, existing software, and the major aspects of the implementation of each component.

The thesis also demonstrates the practicability of linking a number of specialized tools into a complete user environment, and points to the benefits that result from creating such highly specialized environments, both for novice, and more expert users. The implementation also indicates that this environment may be constructed on a host system by modifying a number of existing tools, and linking these to a small set of implemented tools. Furthermore, it is indicated that this may be achieved using very few host system dependent constructs, making the complete environment readily portable.
1. Introduction

A common user view of an interactive computing system is that of a number of independent software processors, bound loosely together by an operating system. The operating system maintains a set of files, and manages all devices connected to the system. The set of processors forms the set of tools available to the user, to manipulate these files and devices. There is, as a result of this distinction between these individual processors and the distinction between the operating system and the processors available on the system, no general commonality of style of interaction with the user. The resultant variety of environments, created by each of these processors is a major criticism of current computing environments.

This thesis addresses this criticism, and will explore an approach of providing a set of processors, or tools, with each member of this set exploiting a common style of interaction, both with the user and each other. The resultant environment created by such a set of tools, a Pascal program development environment, will be examined here, together with the relevant issues raised by the implementation of such an environment.

Consider the schematic model of an interactive computing system, represented in Figure 1-1. Here, the user, by means of an input device, can issue directives to an active process on a computing system, and, using an output device, the active process can display information to the user.

1.1 The Program Development Environment

The interactive environment under examination here is the Pascal program development environment. The specific components of this environment which will be addressed here are the interactive command component of the operating system, the area of program language implementation with respect to the interactive environment (the compiler interface to the user and the interactive debug system of the language system), and program text preparation, and the wider associated issues of
The thesis will include a survey of existing software, where appropriate, and will also discuss the major issues involved in the design and implementation of such software support tools, including case studies of implementation of these tools where applicable.

In terms of the user level, such an environment will be used for the creation, development and testing of programs. This encompasses entering an algorithm as text (written in a programming language), and the use of the computer to manipulate this text, compile the program into a machine executable form, and the debugging of this program.

The tools considered here are only part of a wider range of software tools for the support of the complete life cycle of a program [Wass81]. The program development stage encompasses some of the later aspects of design, the area of implementation, and much of the debugging and testing of software.

Rather than attempt to implement a general program development environment, which, considering the specific goals for each tool in relation to the variety of languages which would be supported, is an impossible task, the Pascal language has been used as the supported language here. The reasons for this choice are, firstly, those relating to block structured text editing.
languages in general:

- The concepts of structured program development can be mirrored in the text of the program in a consistent fashion.

- The ease by which such concepts as delayed specification of blocks of text, and the testing of partial programs can be achieved.

- The visibility of the environment of a block, by the importing of the environment of the immediately enclosing block.

However, given only such considerations, then ALGOL, Simula 67, C and Pascal are all widely used languages based on such a block structure. Pascal has been chosen from this set for the reasons:

- The language is based on "fundamental concepts clearly and naturally reflected by the language" [JenW78], such that the program code shows much of the program structure in a clean and concise fashion.

- Pascal is sufficiently complex to illustrate many of the issues faced when addressing other block structured languages, including both control flow and data definition structuring.

- Pascal is a language with a wide acceptance, and an active user community. It has extended throughout both large computing systems and micro processor systems.

1.2 Software Tools for the Pascal Environment

Given this choice of Pascal as the target language, the major components of the interactive environment can now be defined. These are:

- a command language processor,

- a program editor and
Chapter 2 presents a broad survey of program development environments. Subsequent chapters present, in more detail, the aspects of each component of the complete environment, including, as case studies, descriptions of the implementation of each component of the environment.

The 'idle' state of an interactive environment is normally at the command input level, where the system is waiting for the user to specify a command to the command interpreter. This interpreter includes commands to perform basic file management tasks, or invoke a processor.

This interface between the command level, and the user, is examined in Chapter 3. This chapter presents a case study of an interactive command interpreter implementation. Such an approach was adopted to avoid defining a complete operating system, or rewriting the command interpreter of an existing operating system. This presents an operating system interface which illustrates the practicability of providing a portable front end that makes minimal assumptions about the host operating system.

The issues surrounding text editing are examined in Chapter 4. This chapter presents a survey of text editors, and a detailed case study of the implementation of a screen editor.

The area of the Pascal language processor is a wide topic, and is presented in Chapter 5. Certainly the compiler is the major consideration within this. It is not within the scope of this thesis to define variants of the common Pascal compiler model [Wirt71b], nor to define another Pascal compiler, however this thesis does address those aspects of the Pascal compiler which are relevant here. Also in this chapter the area of program debugging tools is presented, and also a specialized editing tool for entering and editing only Pascal text.

The aim of this environment is to provide a uniform interface between the user and each of the component software processors. This
includes the effort to make the information displayed by each of the components of this system conform to a common style, and also, to make the format of the user input to these programs also conform to a common style.

Over many years there has been proposed a significant case for the human engineering of computer systems. This is not an aspect that can be a component of the input/output interface of the system [DIGI82]. The design of the user interface is a critical aspect of the system's usability. Over the years there has been a significant amount of literature on the importance of designing systems that are easy to use.

1.3 Implementation Aspects

All tools for this thesis were developed on the range of DECSystem-10 computer systems under the Digital TOPS-10 operating system [DIGI82]. Much of the early implementation was performed on the KA-10 model, an early DEC-10 with limited memory. Later versions of this software were developed using KL-1091 hardware, a more recent, and considerably faster processor.

Portability of the system has been a consideration here, and some investigation of this aspect is described in Chapter 6. The target system for this exercise was a micro processor system, using an 8 bit micro processor, running under the Digital Research operating system CP/M [DIGR82].

The tools described in this thesis make use of specific hardware and operating system functions provided by the host system. The case studies will note the areas where this is necessary, and the form of interaction of the software tool with the host system.
2. The User Level Environment - A Survey

Over many years there has been proposed a significant case for the human engineering of computer systems. This is not an aspect that can be tailored onto an existing system, as a component of the input/output process, but an aspect that must be integrated into the development of the system from the design process onward, with the goal of a system that makes the user more productive in the use of the computer system, with less effort expended to achieve a particular task.

However it must be remembered that the state of knowledge about the criteria of a 'good' system from a human engineering point of view is rather primitive, and there are very few definitive guidelines other than the limited experience of the software author and intuition [Sil.H81]. There are a number of very general guidelines that can be considered as major aspects of human design, but more specific criteria are not so easily defined.

In the course of an interactive session the user must be continually aware of the status of the file store, the level of interaction with the system (which active processor is running, and the state of this processor), and the consequences of previous actions specified by the user, and possible actions that may be specified, all in relation to the task which the user wishes to accomplish on the system. Of direct benefit here is the concept of integrating many of the tasks which the user must perform in an interactive session into a common environment.

This chapter will examine a number of significant developments in the area of program development environments, both for the Pascal language, and other programming languages, and more general environments. The areas of each system under specific attention are the style of interaction with the user, including the command language used, and the form of output from the system, and the tasks that can be accomplished
within each system.

2.1 A Description of the User Environment

As this thesis is concerned with both a user environment, and the details of implementation of a number of processors, a tool for describing both the resultant user environment and the details of implementation is necessary. The adopted base is by using a hierarchy of abstract, or virtual, machines.

A description of such an entity consists of a description of the data structures used by the machine, together with an algorithm for the operation of the abstract machine, or "the transformations of these data structures that occur during program execution" [Moli83]. Such a description can be formalized by using a programming language to describe this algorithm and the associated data structures, and, here, Pascal is appropriate where a rigorous definition of the abstract machine is required, but, in most cases a less formal textual description will be used.

The advantages of this style of description are that of a single descriptive tool, to describe both the user environment, and the issues of implementation. The abstract machine model may include other abstract machines as components, and such a hierarchy of abstract machines will provide a multi-level description of a single process, from the user level of the process, through the operating system level, down to a hardware level description of the process. This view includes, for executing Pascal language programs, the view of the Pascal abstract machine: an abstract machine that can directly execute Pascal language text, which is a component of this abstract machine hierarchy (a brief presentation of the aspects of such a machine is presented in Appendix I of this thesis).

Thus a user level description of a process can be achieved by a description of the operational details of the interactive abstract machine, as indicated in Figure 2-1 (from [Moli83]).
The Basic Machine Subsystem Algorithm is:

REPEAT
  send prompt to display;
  receive command from keyboard;
  interpret command;
  send response to display;
UNTIL false.

The gross structure of the machine store can be viewed as:
- a file store and
- associated file pointers to impose a user visible structure on
  this file store (e.g. as in Figure 3-1).

This provides a logical framework for discussion of command
languages, and discussion of file store structures. It must be noted that, as
the main object of manipulation of the command language is the file store,
the structure of the file store has very direct repercussions on the structure of the command language.

Throughout this thesis, the structure of the user subsystem is assumed to be a video terminal device, connected by a serial link to a computing system, as this is the most common form of interaction with a computing system. The reader is referred to Appendix IV of this thesis for a more detailed examination of the range of hardware used in this area.

2.2 Conventional User Environments

The components of the machine subsystem of the user level abstract machine (as outlined in the preceding section) are the processor and a store. The processor implements the command and response level of the user environment, and the store models the file store area.

2.2.1 The File Store

The main object of manipulation of this command language is the file store, so that the structure of the file store has very direct repercussions on the structure of the command language. File store subsystems can take a very wide variety of forms, and, while a detailed enumeration of the types of file store structures exhibited in current systems will not be attempted here, a number of general observations about the file store structure can be made.

Most systems implement a delineation of the file store into two types of files: directory and non-directory files. Non-directory files may contain any type of data other than directory information, and may be internally structured on the basis of a further categorization of non-directory files into text and binary data files, or exhibit no internal structure. Such files are used to contain text or binary data, and are commonly accessed by the user by binding a textual name to the file. Directory files are used by the system to manage the file store, and commonly contain sufficient
information to allow the system to associate a name (entered by the user) with a particular file within the file store. Such directory files cannot be explicitly updated or altered by the user, but are updated by the system to reflect any changes made to the state of the file store as a result of the interpretation of user commands. Thus the complete set of directory files describes the state of the file store.

One method of categorization of conventional file stores can be based on whether the system attempts to centralize all directory information into a single directory (a 'flat' file structure), or allows the user the ability to create files which are themselves directories, and subsequently allow the user to make entries into this directory, corresponding to new files that have entries only within this user file directory (a 'hierarchical', or 'tree-structured' file structure). In the second case, a unique specification of a file must include the ordered list of directory files as well as the file name, whereas, in the first case, all file names have entries in the single system directory, so that the file name is always sufficient specification. A third form of file structuring implements a limited form of hierarchical file structuring, where each file has a directory of internal 'elements' of the file, where these elements of the file may only contain data. Here a unique file specification of a file is the file name and element name.

Although there is no implicit limitation on the method of interpretation of the contents of a file (whether as text, binary data, executable program or so on) at this level, most conventional file systems impose such limitations, and store additional information (usually within the file directory) as to the 'type' of a non-directory file. This may take the form of an 'extension' to the file name, using mnemonics to indicate the type of the file (e.g. the Digital TOPS-10 file system and the UCSD Pascal file system) or may take the form of 'internal' directory sub-fields, initialized by the system when a new entry is made in the file directory (e.g. UNIVAC EXEC-8 file system). This type information is usually visible at the user level, and is used by the command interpreter to check whether a particular
operation can be performed on a specified file (e.g. to ensure that only 'text' files may be edited, and 'executable' files invoked for execution and so on).

2.2.2 The Command Language

Conventional command languages can be regarded as the poor relation of programming languages, which have received scant literature attention, and "there are few examples to which a designer could point with pride" [Beec80]. Despite numerous types of command languages that have been developed, there has been little success in the efforts of standardization in this area, and the Codasyl COSCL [Harr80] and ANSI X3H1 [FrMS80] efforts are still far from completion, and further still from any widespread acceptance.

There are two major aspects of the command language that are of particular importance: the capabilities of the command language, in terms of the actions that may be invoked by the user, and the characteristics of the user interface so defined by the language and responses generated. (In this section, only those aspects of a command language that have a direct bearing on the interactive environment will be examined)

To use a definition of Beech [Beec80], a command language provides for the invocation of:

- User defined programs

- "Utilities" (used here in the sense of programs, provided with the system because of their general usefulness, rather than any intrinsic difference between this set of programs and the user defined programs).

- System functions which are not replacable by normal programs, since they are concerned with control of programs, such as program interruption, resumption and resource allocation.
However, there is little observable difference between the first two classes of commands, and a command language should be able to remove any user level distinction between such programs, so that there is a definite case for simplification of this model. Such simplification recognizes that, at the command level, the user may invoke execution of a program by specifying the name of the program (specifying any parameters for the program that may be required), or the user may interrupt a running program, or resume execution of an interrupted program. This provides a small, but adequate, set of facilities that a command language can implement.

When addressing the second component of command languages, the characteristics of the user interface, there are a number of design criteria that are relevant to command languages which most current conventional user environments reflect, either in part, or in full:

- The command language should provide access to all necessary facilities required by the user when performing a particular task. The language should not make some functions difficult, or impossible to invoke.

- The responses generated by the command interpreter should be structured in such a fashion that avoids confusion about system environments. This is an area which is difficult to define accurately. The extreme bounds of this area are: using a different prompt character to define a different system environment, and on the other extreme, a long, verbose, and repeated prompt string. A command language environment should be able to reach a compromise between these two extremes, using responses which are short, but informative.

- The command language should implement a consistent method for the user to specify parameters to commands. When specifying a command, which may use a number of parameters
(to change, in some fashion, the execution of the command) the user should not have to enter all these parameters at each use of the command. Each command needs to have two sets of parameters: required parameters which must be specified (usually file names and such) and optional parameters, which assume default values used when such parameters are not specified by the user. Where a command has a number of such parameters, there needs to be a common method of matching the values entered by the user to internal command parameters. Simple ordering is not sufficient here, as a command language should be tolerant of parameter specification in any order by the user.

- Have a simple syntax with few formal rules. This implies that command strings should be short, requiring only a keyword, and a set of optional parameters. More elaborate forms of command syntax tend to cause significant problems to users, and pose the additional problem of correct syntax at the user level. It is an unnecessary, and also tedious, process, often prone to errors, to require the user to enter a command in such a fashion that requires unnecessary verbiage. Commands such as:

Run <program name> using <file name> and <file name> with output to <file name>

can be more simply expressed using a command of the form:

<program name> <file name> <file name> =<file name>

- Avoid non-alphabetic characters assuming overwhelming significance in the command language. Where non-alphabetic characters are used, these should be used in a sense comparable with their 'usual' meaning. Thus, a '+' character can be used in the sense of concatenating, or adding two objects, and commas as separators of object names, and so on.
Here an entered command should have some easily discernable textual relationship to the desired action to be performed. Use of 'special' characters within the command to have significant side-effects in a form which is neither internally consistent, nor in any way discernable from the text of the command is an undesirable feature of a command language.

- Allow simple repetition of common command sequences. Thus command sequence should be able to be specified by the user, and later invoked as a single command.

- Allow for definition of user commands. This area is an expansion of the previous item in this list. This includes the ability to create a new command using sequences of commands from the existing command repertoire. An additional refinement of this concept is to allow commands which alter the capabilities of sequencing of these command recipes, from a simple sequence to include conditional constructs (the IF construct), loop constructs (the WHILE construct), and parameterization of command invocation. As well as the capability to define new commands using existing commands, the user should also have the capability to create new commands based on invocation of existing user programs.

- Inform the user of commands which result in error conditions. There are many ways by which error conditions may arise on a system. The user interface, in conjunction with the operating system should have the capability to trap and perform limited diagnosis on such error conditions, and provide a diagnostic message as to the type of error encountered, and, if possible, the reason why the error was generated. As well as this diagnostic, the system should always return the user to a defined state (preferably command input level) when such errors occur.
2.2.3 Common Limitations of Conventional User Environments

Operating systems, and especially operating systems for large computing systems, are generally designed with a desire to be 'all things to all men'. The user environment of such operating systems strive in their design for general applicability, where the form of interaction, while not suited in particular to any type of user, can make most operations possible, if not always simple or easy to invoke. The user environment can suffer many weaknesses, particularly in the command language so defined. The most common problems that many basic command languages share in this respect are:

- A limited set of commands and command formats

- Non-extensibility of the command language. Here it is meant the ability to bind a name to a command sequence of user commands to the system, and invoke the command sequence by the new command so created.

- The impossibility for the user to initiate concurrent processors

Some systems have attempted to solve a number, or all of these basic complaints. The TOPS-10 solution to the second problem here, for example, is to introduce two more command processors, 'COMPILE' and 'MIC'. This allows a limited form of command extension, but the form of invoking such commands is inconsistent with the remainder of the command set, so that these facilities are not part of the environment proper, but are the result of grafting separate processors with the normal command language interpreter.

2.3 Integrated Program Development Systems - A Survey
2.3.1 The Interlisp Environment

The first of these environments is that of the Interlisp system [TeiM81]. This is a language based (Lisp) environment where all support tools are written in Lisp. The development of such an environment can be traced to the JOSS system of the early 1960's, and include APL, Basic and MUMPS as major developments in this area. The user of such a system sees an integrated environment in which a single, unified setting provides all of the necessary operating system functions, access to utility functions and supports editing, execution and debugging of programs written in the supported language.

Much of the motivation for the Interlisp system is in the implementation of an experimental programming environment, a style described by Sandewall as structured growth: "An initial program with a pure and simple structure is written, tested and then allowed to grow by increasing the ambition of its modules. The process continues recursively as each module is rewritten. The growth can occur both 'horizontally' through the addition of more facilities, and 'vertically' through a deepening of existing facilities and making them more powerful in some sense" [Sand78].

The facilities provided by Interlisp include a file package to integrate the file sub-system into the normal Lisp environment of Lisp data objects, so that file manipulation may be performed by the user within the Lisp environment. One of the more impressive features of this system is the DWIM facility, invoked whenever an error is detected by the basic system. DWIM (Do What I Mean) is an error corrector, applying mainly to spelling correction, but also for atom to list conversion. Thus the facility provides a tool for silent, useful changes to the user input, and also to supply reasonable default values to missing input parameters where necessary. The Programmer's Assistant is a tool that alters the conception of the system from a passive executive processor to an active intermediary [Teit72]. The assistant records the user input, a description of the side effects of the operation, and the result of the operation. In conjunction with this, the
input can be redone, edited and redone, or even undone [Tei81].

The major user view of this system is of successful integration of the programmer's tools, including a Lisp structure editor and dynamic debug tool into an environment where there is no visible context switching between activities. Included with this is a smooth transition between the system supplied facilities and user defined functions, so that the system allows for graceful and consistent extension and modification of the user environment.

2.3.2 The UNIX Environment

The UNIX environment [Rit78, Ker81, DoM80] is an operating system in its own right, developed to produce a timeshared system with a large set of user tools. The feature of UNIX is a consistent design philosophy and uniform file format within both the operating system and the associated software tools. The aim of UNIX is not to support a single development methodology for program development, but supply a wide range of tools, from which the user may select a set of tools to build a desired development support environment. This reflects a desire to produce system software which can actively support a variety of user tasks.

The file system is a conventional hierarchy of files, based on a tree structure. Each interior node of the tree is a directory of files, and each terminal node is either a file or a directory. The complete file system is based on a single ROOT directory, and all users work within a defined current directory. There is little distinction between directories and other files, with the exception that only the system may write into directory files. Any file can be accessed by the user by using an extended file name consisting of an optional path (ordered list of directory names) and the file name, with the path either being relative to the current directory, or being a full path specification (using the ROOT directory explicitly). The other major feature of the file system is that a file is regarded as a sequence of bytes, with no other implicit internal structure, regardless of the characteristics of the device on which the file resides. Thus, as there is no
visible user buffering, all bytes of a file are accessible in any order.

The program interface is also maintained as a simple set of entry points into the operating system, using only seven distinct calls to open the file, read or write to the file and randomly access the file. The other feature of the UNIX system is that all input/output devices are regarded as files, with appropriate operating system defined driver routines to map from the byte view of the file to a device dependent view of I/O. These devices may be directory devices (such as disks or tapes) or non-directory devices, and may be either read or write only devices. Such considerations are managed by the operating system.

The user interface is kept as simple as possible. The major characteristic of this interface is the shell program, a UNIX utility that implements the user interface by interpreting user commands. The most common form of command is simply the name of the program to run, with any necessary arguments appended to the program name. If this program cannot be located within the user directory, the system directory '/commands' is scanned. This implicitly removes many of the distinctions between system and user utility programs. The shell of UNIX supports a wide variety of command language operations (and, to be quite accurate, is not a single program, but now a family of programs) and is, in many ways, a complete language in itself, manipulating processes and their sequencing as the basic data objects of the language.

The next major feature of the user interface is the potential for input and output redirection. Here the user can specify a file containing the input for the program rather than the terminal, and also a file to contain the output, and, in the case of output files, specifying either rewriting the file, or appending to the file.

This last feature produces the possibility of using the output from one program as the input to another program, a concept central to UNIX, and an important point of programming style within UNIX. This is
supported by the use of 'pipes'. A pipe is a list of commands, where the output from an element of the pipe is directed as input to the next element of the pipe. This is implemented using parallel processors, with an invisible buffering of data between the elements of the pipe. Thus many applications can be assembled with little direct programming effort, using only the supplied utilities and the pipe concept.

The shell of UNIX is regarded as a program within UNIX, so that command files can be invoked by calling the shell program and specifying input to be read from the command file, rather than the terminal. This supports the use of parameters, shell variables, and basic control flow constructs within the command file. Much of UNIX can be looked as the result of making particular logical decisions about the file store, and then applying the consequences of this structure to the command language.

The major aspects of UNIX, relevant to the topic of software tools, are the rich environment of specific tools which can be easily combined to achieve tasks with little direct programming effort. This concept has already spread far beyond UNIX, and is a major feature of many recent operating systems.

2.3.3 Ada Directions

With the release of the Stoneman document [DOD80] has come a commitment of the Ada language to provide a comprehensive set of tools aimed at a complete life cycle support for Ada programs. This set of tools, collectively termed the Ada Programming Support Environment (APSE) is aimed at defining an open-ended environment, where the support tools can be modified or extended at any time by the user. This environment supports program libraries, configuration for particular host systems and, as a major feature, portability of both programs and tools across systems, creating, in effect, a portable environment across systems.

The mechanism to implement such an environment is based on a
three level environment. The first level, the Kernel (or KAPSE) provides a set of primitive operators, such as I/O facilities, and a set of program-callable routines to allow the user to perform simple dynamic debugging, one program to invoke a second, and, of course, the Ada runtime support environment for a program. The interface of the KAPSE to an Ada program is, therefore, aimed toward a machine independent interface. The KAPSE maps the host operating system functions into a form which is host independent, and callable from an Ada program.

The second level of the program support environment is the minimal environment, or MAPSE. This defines a minimal set of support tools which form the basis of any APSE. Thus the KAPSE maps the characteristics of the host system into a standard format, and can be viewed as a level which defines a virtual operating system interface for the MAPSE. MAPSE tools include an Ada editor, compiler, database manager and a command language interpreter and the intertool interfaces [SFGT81].

The complete APSE is the MAPSE together with a set of user-supplied tools, defined within the interface of the MAPSE. Thus there exists within this environment the ability for the user to tailor the MAPSE level to produce an APSE which can actively support the programming methodology desired by the user. This relationship among the various levels is shown in Figure 2-2.

It must be stressed that these tools are, at the time of writing, still at the design phase, so a more detailed appraisal of the APSE approach is not yet possible.

2.3.1 Pascal Environments

There have been a significant number of developments in the area of a user level environment directed towards Pascal program development. This section will examine a number of such implementations.

The Pascal Assistant [SiLH81] is an integrated environment for Pascal
APSE

<table>
<thead>
<tr>
<th>USER Interface</th>
<th>User Defined Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAPSE</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>Link</td>
</tr>
<tr>
<td></td>
<td>Editor</td>
</tr>
<tr>
<td>Interpreter</td>
<td>Compiler</td>
</tr>
<tr>
<td>Loader</td>
<td>Data</td>
</tr>
<tr>
<td>KAPSE</td>
<td>Manager</td>
</tr>
</tbody>
</table>

HOST OPERATING SYSTEM LEVEL

Figure 2-2: APSE organization

program development aimed at novice users, supporting file management, text editing, program formatting, program compilation and execution of programs within a single environment. Like the APSE, this is also at the design stage, although a number of the components have been completed at the time of writing the reference article (1981).

Unlike the Interlisp environment, which is not a system which aids novice users (and could be fairly described as an 'expert' system), and the APSE, where in the absence of information to the contrary, also appears to be an 'expert' system, the Pascal Assistant is predominantly a novice system. There is, however, a very fine border between a system which is only used by novice users, and is seen as a stepping stone to using other, more 'expert' systems, and a system which can provide the necessary support for novice users, but can also provide a satisfactory for the more complex demands of more experienced users. This is an area which the Pascal Assistant, with its very small command set, has failed to include within the design. The problems lies in the level of assumptions made about the user environment. Producing an environment where a number of far-reaching assumptions influence much of the resultant environment creates
a system which is prone to be too inflexible for more experienced users.

A similar system to the Pascal Assistant is the Conversational Pascal System (COPAS) [AtkNS81]. Again, this is a system aimed toward novice users, incorporating a simple, line based, editor and an interface to a compiler. Also, this system does not achieve the goal of providing a system that can be used by novice and expert users alike, as the system is certainly biased towards usage by novice users. The system provides a small, and relatively simple interface to the user (a total of 15 distinct commands, 10 of which refer to the text editor), but encounters problems by making too many simplifying assumptions about the environment, and, from these assumptions, producing a command set which is too inflexible. The system has not exploited any of the possibilities in providing a tighter binding between the editor and compiler, allowing only simple invocation of the compiler, specifying only a text file containing the program. Thus a useful environment must be more flexible in the command set, and allow a greater set of options when selecting a module for execution.

Another aspect that needs to be addressed by a Pascal support system is the potential size of Pascal program texts, and the attendant problems in manipulating such large programs by such a system. One approach to these problems is to support program modularity in program definition, where the support procedures and functions of a program are stored, and edited, and compiled separately from the main program, using an environment descriptor file to aid module compilation. These separate modules are linked to the main program using the results of compilation (the binary relocatable files) to produce the resultant execution object [OIR'T81, WaiC81]. These systems commonly include the ability to define an environment which is common to a number of modules, to allow all modules of a system to share access to a common run-time environment.

The UCSD Pascal system [Over80] is a relatively complete Pascal program development environment, aimed specifically at micro computer systems, but also capable of being configured on much larger systems. The
UCSD Pascal system differs from the other systems mentioned here in so far as the system is a complete operating system, rather than a language system grafted onto as host operating system, and incorporates all operating system functions for a single user micro system. The components of the user level environment include a text editor, Pascal compiler, link-loader, P-Code assembler and a file manager. All these processors are invoked from the UCSD Pascal command interpreter.

This command interpreter is a multi-level menu driven system, with the addition of sub-menus to extend the set of commands. Thus, at any stage of execution within UCSD Pascal, the top line of the screen shows on line of applicable command options at this point of execution. This sub-menu system is extended to further sub-menus within the text editor. Command format is simple menu selection, using the first letter of any item of the current menu.

As a complete environment, incorporating both program development and execution, UCSD Pascal is a relatively successful system. However it is relevant to note that the menu style of commands are directed more at novice users, and can become somewhat problematical for more experienced users. To keep the menus brief, the command options at any point are limited, forcing some commands to enter sub-menus, where a menu selection must be further elaborated before command execution. Other minor problems with the system include the artificial distinction between program and text data files made by the editor, making it very difficult to edit a file which has been created by a Pascal program.

A more advanced, and somewhat richer environment for Pascal program development is the PASES environment [ShEA81]. This environment is centered around a structure editor, capable of editing the syntactic structure of a program. The environment also includes a text editor, parser (to produce the equivalent syntactic structure of a text object), an interpreter (interpreting the syntax tree, compiler, and a command processor. This is a more expert system than the other Pascal
program development systems presented here, with considerably more power and flexibility than UCSD Pascal or the COPAS system. The major point of interest in this system is an aspect that will be covered in more detail in Chapter 5 of this thesis, that of an internal representation of a program text as a syntax based structure. The editor is a bi-modal editor, allowing the user to either perform simple operations on the syntax structure, or to specify text, which is parsed, and, if the parse is successful, enter the result of the parse into the syntax structure. Existing structure can be altered by elaborating the structure into text, and then editing, and parsing and reinserting this text back into the full program representation. The environment is not a complete operational environment, as only editing, debugging (by interpretation of the syntax structure) and compilation can be performed within the system. Other system operations, such as file management, are performed on the host system using the host operating system command interpreter.

The system does not have the internal consistency of Interlisp or UNIX, and the structure editor relies quite heavily on a text editor to perform much of the actual manipulation of text.

2.4 A Pascal Program Development Environment

To examine the user level environment of a Pascal program development environment, it is necessary to enumerate the classes of tasks which are performed by a user in developing programs.

- Creation of the program text. The means by which this can be done are varied, using textual input from the keyboard, or using program fragments which are already stored in the system and integrating these fragments into a complete program. The major tool used here is a text editor, which may be either a general purpose text editor, or an editor designed for use specifically to edit Pascal programs. This area also encompasses such aspects as formatting the text, using
tools such as 'prettyprinters' [HeuL77, Pete77, Mohi78, Mike81].

- Parsing of the program, to detect any errors of syntax in the program (this step may be performed within a suitable Pascal program editor, or during compilation of the program).

- Compilation of the program, to produce an executable program. This commonly requires also the ability to link other modules to the program, to form an internally complete executable program.

- Testing of the program (including the aspect of debugging), with the aim of satisfying the user as to the robustness of the program. Debugging also requires particular attention here, as the process can be far more effective when using a debug tool which can not only allow the user direct control over execution of the program, but also allow the user to simultaneously interrogate the state of execution at a sufficiently abstract level that directly relates to the original Pascal program text.

- To set up the resultant program as a production program (or as a new command utility) within the environment.

From the material presented in this chapter, it is possible to present a number of essential characteristics of such an environment. Unlike some of the systems surveyed in the second section of this chapter, the environment is intended to be a complete environment, in so far as all functions which need to be invoked by the user can be invoked without the necessity of leaving the environment to use the host operating command language.

The capabilities of the environment include:

- Invocation of the Pascal language processor. This includes compilation, and execution of programs, together with a debug tool.
- The ability to create Pascal programs, documentary text and data files. Thus, the environment must include an editor to accomplish this, or even a number of more specialized editors to accomplish particular classes of editing (here a distinction may be drawn between editing text and editing a Pascal program).

- Invocation of file management and other necessary system utilities. This includes the basic operations of file deletion, creation, renaming and printing of files, and will also include operations that are derived from the host computer system. The rationale for this ability is to allow the user to perform such operations without having to leave the environment.

- The ability to tailor this environment to suit particular needs and requirements. This is a wide area, encompassing additions to the command set (using command sequences and invocation of user programs), and access to the operational parameters of the environment, to allow the user the ability to tailor the environment to reflect the user's expertise and the desired style of interaction with the system.

- The final capability of the environment lies within the definition (and use) of a consistent interface between all operational modules which implement the complete environment. Such an interface must allow a consistent method of transfer of information between such modules, and also the ability of one module to invoke another. This is aimed at producing, at the user level, a rapid, and consistent style of changes in operational modes of the system, such as the mode change from editor to compiler, and from compiler to the debug system, for example.

The style of dialog between the user and this system include those
points made in Section 2.2, when examining command languages. There are
two major, additional, aspects that are relevant to such a system.

Firstly the system should not be aimed at either novice users or more
expert users, but should be suitable for both types of users, as well as the
areas of expertise between these two extremes. As well as the ability of the
user to alter operational parameters of the system to suit a particular level
of expertise with the system, there is also the important requirement that
the basic set of facilities of the command language (simple program
invocation and program interruption) should be a complete and sufficient set
of commands to allow the user to use the system effectively.

Secondly the system must be capable of providing assistance to the
user at any point of execution. This assistance should include the ability to
list all applicable commands at this point of execution, and also to obtain
brief descriptions of these commands in terms of the command syntax,
necessary command parameters (if any), and the actions that these
commands invoke. Also the user must be able to access more detailed
information on the system, so that the system itself can provide quick help
on command options, background documentation on specific aspects of the
system, and also provide a tutorial-styled introductory document on the
system for novice users. This area of assistance to the user does however
extend beyond documentation issues. The system should also be capable of
trapping error conditions, and as well as the ability to produce understandable diagnostic messages, should also be able to perform error
recovery to bring the user back to the main command level after any form
of error condition. Also the system should be capable of providing
assistance in the area of inadvertent catastrophic command errors. It is not
envisioned that this encompass the ability to 'undo' the effects of every
command, but the file deletion command, and the editor text deletion
command, at a minimum, should be commands which (if specified before the
next deletion command) should be able to be 'undone', and the deleted
object restored, and also the common practice of the editor creating a
'backup' of the edit file before commencing editing operations is one with considerable merit.

0.1 The Role of the Workspace

The implementation of a complete system incorporating these design goals will be presented in succeeding chapters.
3. The Pascal Workbench

3.1 The Role of the Workbench

The preceding chapters have examined a number of interactive programming environments. This chapter will examine the implementation of a programming environment which is aimed towards supporting those features required by programmers (both novice and more expert users) for Pascal language support. The major element of this environment is the command interpreter. This command interpreter is the 'default' active process within this environment, entered by the termination of any other process in this environment.

With regard to an interactive program development environment, there are a number of specific problems shared by many user environments:

- Poor presentation of information to the user. Most systems present no information on the terminal screen when the user re-enters the system command level, with the exception of a prompt character or short text string to indicate the user is at command level. Such lack of information is inappropriate to novice users of a computer system, and can also be problematical to more experienced users of a system. The system should be capable of presenting more 'default' information to the user at this level, selectable by a display command.

- The command syntax suffers from unreasonable terseness. A very common form of command syntax is:

  `<Command Verb> [<>Modifier>*]`

The command verb is one of a set of recognised commands, and the modifier is a set of file names, action switches and
secondary verbs required to select non-default action conditions, often specified by an obtuse system of single letters and '/ characters. Very few systems have such features as a modifier memory, where previously used modifiers for a command can be recalled and edited when using that command, or a closely related command, again.

- State conditions of the previously running program are usually discarded, so that most processors are not re-entrant after the users enters the command entry level of the system.

- Many systems do not use some form of file type information to apply common sense rules of applicability of certain commands relating to files. The weakness lies in most approaches to file type information, which is left to the user to define, typically in the form of a file name extension, and, as a result of this, command processors rarely check such an extension when operating on a specified file.

- Lack of basic command names to apply to 'macro' actions and tests, such as a 'RUN X' command being bound to a process that compiles file X according to the processor specified by the file type information, then link-loading the resultant object file, and then executing this core image, with the additional proviso that the compilation will only occur if the source file was created, or modified at a later time than the related object file creation time, and the link-loading will only occur if there is no more recent executable file than the related object file.

It is these problems, either in toto or in part, in command language interpreters which have prompted the development of this Pascal Workbench. Ideally this Workbench is a command processor whose command syntax, and user interface is consistent with that of the editors and Pascal processing environment presented in later chapters. The goal
here is to create an environment such that a user does not have to use the system command level directly: the user can be continually be presented with information which is relevant to the possible command options at that point in execution, and that such a system can be grafted onto a number of different operating system with few alterations in the resultant interactive environment.

To implement such a processor implies either an operating system written specifically for such ends, or the use of a command processor that sits 'on top of' the operating system, providing similar functionality, but within the parameters of the above interactive environment.

Certainly the UNIX operating system can be seen as the result of developing a programmer's workbench by writing a complete system, and the later portability of the system to a large number of radically different hardware configurations is a fitting tribute to this design. However the design and implementation of a complete operating system is inappropriate within the confines of the research for this thesis, and the ability to create the desired environment within existing operating systems reinforces this decision. Thus, the method of implementation chosen here is that of an application program which effectively 'hides' the normal command level of the host system. This program must, therefore, be automatically invoked whenever the user process would normally enter command level. This method of implementation allows this application program to implement a different environment to that of the default command environment, being able to define both a new command language and a different type of response to commands.

The following sections will detail the desired user interface, and also detail the interface to the existing operating system required for this command language interpreter.
3.2 Design Goals of the Workbench Environment

3.2.1 Files - The Basis of Commands

The Workbench is a file-based processor, which maintains a 'current' file description of the current user workfile, and entered commands are assumed to be either file-independent commands, or commands which will operate on the current file. As the file organization is one of the main bases of the Workbench, it should be described first.

The Workbench assumes that the operating system can maintain a set of file directories for each user of the system. This directory for each user is a set of specifications of all files on the system owned by this user. The contents of this directory are the working set of the controller, and all such files can be accessed by the user with equal ease. Each entry in the directory is a file name which consists of the user defined label and a file type label as an extension to this name. The extension provides a mechanism for the workbench to make assumptions about the contents of each file, whether text, program code or binary files and so on. These file types are based on the DECSYSTEM-10 method of file type labels, which uses a mnemonic form of type labels. The file types used, and the corresponding type labels are:

- PAS - Pascal source code
- REL - relocatable binary file, used as output from the compiler and input to a link loader.
- EXE - executable binary file
- COM - a textual command file
- DOC, TXT, DAT - text files used as documentation, data files and such. Any unrecognised extension can also treated as a text file.
- SFD - project directory file

- DPF - debug program file, a binary file used by the Pascal debug system, created by compilation of a program.

Thus a file name 'PROG1.PAS' is assumed to be a Pascal source file, and PROG1.REL, the corresponding compilation binary file.

The Workbench uses a file store organised as a hierarchy of file directories, where a file may be a directory of a further set of files. Operating systems may support many levels of such sub-directories, and this is also the case for the Workbench. Such directory files are termed project files (see Figure 3-1).

The other necessary file construct is the library directory. The library directory is the default directory which is searched by the operating system when a referenced file is not located within the user area.

At any stage of execution the Workbench must maintain the specifications of the current filename, and it's type, derived from the file extension, the project which owns the file (the project, or sequence of projects by which it is defined), and the current library directory.

The resultant tree structure of files so defined is described in Figure 3-1.

3.2.2 The Display Process

The Workbench display is ideally one which can be adapted to suit the expertise and requirements of the user, and, furthermore, such alterations in the display process can be specified by the user. The basic level of display outputs a relatively complete set of information to the screen, whereas higher levels display somewhat different information, and assume the user is more familiar with the command set and and command semantics of the Workbench.
The basic, or novice level uses a split-screen approach. The upper half of the screen is used for a display of a section of the file directory within the current project, windowing the directory so that the current file entry is always displayed, displaying file names and types. The lower half for the display of applicable commands to the current file. Higher levels of display delete the list of applicable commands from the screen, expanding the space for the file list as a consequence, and also increase the number and scope of available commands.

At the basic level of display, the format of the screen is as shown in Figure 3-2.
Thus, the Workbench maintains a large amount of information on the screen as a default display. The penalty of such a large output overhead is avoided to a large extent by allowing many commands to be interpreted without any alteration to the screen, with the exception of the message space at the base of the screen.

The file window space (10 lines) is scrolled independently of the remainder of the screen, and includes a pointer positioned at the current file. The cursor up, down, left and right keys have a direct effect on the pointer position (and hence the current file). As the display is a multicolun m display the left and right keys toggle the current file column, and the up and down keys move the pointer vertically within the column, scrolling this region as necessary. Also each file has an index number, so that a single number as a command will position the pointer at the corresponding file. This area is maintained as a window on the current

<table>
<thead>
<tr>
<th>SCREEN CONTENTS</th>
<th>SCREEN POSEITION</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROJECT : _____</td>
<td>line 1</td>
</tr>
<tr>
<td>LIBRARY : _____</td>
<td>line 3</td>
</tr>
<tr>
<td>window into directory, displaying file names</td>
<td>line 12</td>
</tr>
<tr>
<td>10 line window using multicolumn display</td>
<td>line 14</td>
</tr>
<tr>
<td>An -&gt; symbol points to the current file</td>
<td>line 15</td>
</tr>
<tr>
<td>COMMAND : user command entered here</td>
<td>line 19</td>
</tr>
<tr>
<td>space for list of available commands</td>
<td>line 20</td>
</tr>
<tr>
<td>space for short messages</td>
<td>line 24</td>
</tr>
</tbody>
</table>

Figure 3-2: Workbench Screen Format
project file directory, where the window always surrounds the current file. The rationale for such a display is that the Workbench maintains for the user a window on the state of the project area, and the concept of a current file is used in recognition of the fact that many of the commands within this environment operate on a single file. Use of a current file eliminates the need to continually specify a file name on each command entered.

The command list is a comprehensive list of all commands which are accepted in respect of the current file type. Thus, for example, executable files may be loaded and executed, but not printed. As the pointer is moved, commands within this area are either inserted, or the corresponding field blanked out on the screen. Commands are displayed on the screen as verbs, in a multicolumn format, each command placed at a fixed location on the screen. The command entered can be entered as a minimal abbreviation of the command verb, and verbs have been chosen such the abbreviation of all common commands is one letter.

The second level of display removes the command area from the screen, and also the file type field, replacing it with more information on the file storage (displaying more files on the screen, and adding a file size display field for each file). The file window on the screen is extended by 5 lines and is split into two columns, and the command prompt line is displaced accordingly on the screen.

The display process is a process which is an interruptable process, interrupted by either the user entering a command, or a system event requiring servicing (in the case of a multitasking controller this class of events includes subtask activity). Display always commences with the current file name, then the list of applicable commands is filled on the screen, followed by the remainder of the file window and then the project and library fields, and, lastly, the field headings.

In the case of commands which are serviced within the Workbench
controller (pointer movement and basic file operations), or commands which will produce less than 4 lines of output, the screen is not blanked during the execution of the process, and any output produced by the command execution is displayed on the base of the screen. This produces the user view of many of the common commands being executed within the Workbench, and also reduces the terminal output traffic substantially. Thus, in common with the editor, the Workbench places a high priority on user input, interrupting any screen operations to service user input immediately. The Workbench also recognizes the high cost of creating a full screen display in terms of output traffic, and attempts to interpret as many commands as possible without destroying the screen image.

3.2.3 The Command Set

The criteria of the command set is similar in many respects to that of the editor: a concise, and consistent set of commands, able to be invoked with a minimum of verbiage and unnecessary modifiers.

This leads to either a menu style of command specification or one which uses mnemonics for commands. The menu system has many benefits when regarding the demands of more novice users, but suffers from a number of problems when considering the demands of more experienced users. The menu is often unnecessary, particularly for common command sequences, and the delay in creating the menu display is less than acceptable for such users, and the problems faced in creating a menu for a large command set often force the decision of multi-level menus, with their attendant drawbacks. A mnemonic style of commands often suffers from the problem for novice users of remembering the appropriate key word for an action, upon which the mnemonic command is based. This command style also breaks down as the command set increases. A path between these styles is to display a menu of command keywords, and allowing the user to specify a command by entering any abbreviation of the command, and also allow the menu display to be selectable at any time by the user.
Commands fall logically into three main sections:

1. Commands referring to the control and system interface, rather than the local file system. This includes such commands as logout, interrogation of system information, and system utilities (send messages etc).

2. Commands referring to the local file system, but without reference to a current file. Such commands include commands to create a new file, list the library files, change the project or library directory, or to copy a file into the local project area.

3. Commands which implicitly reference the current file, such as file deletion, execute, edit, debug and such.

For the first set of commands, the system interface and control commands, many of these require a modifier, or set of modifiers, so such commands, when invoked, display the previous set of arguments that were given, which may either be selected or overwritten, or augmented, with a new set of modifiers. Commands which fall into this category are commands to obtain on-line help (about either the controller or help from the system help facility), and commands to interrogate system information (such as disk free space).

With the second set of commands, there is no such requirement for a modifier memory for the commands, as the same command would rarely be repeated with the same set of modifiers. The commands which fall into this class are the create new file command, list, or read from the library directory, set the current project or library directory.

The third set of commands need only the current file as an argument, so that, again, no modifier memory is necessary. These commands include delete, edit, run (compile, link and execute), debug (as for run, but with dynamic debugging enabled, if supported by the relevant language processor), print, format, copy and rename.
This completes the basic level of commands. If the user selects a higher level of the controller, the ability to create new commands from sequences of current commands, or to pipeline commands as a subtask is then available. The command syntax differs here according to the facility used, but consistency is maintained with the lower level command syntax and style.

These more advanced features of the command interpreter will be presented in greater detail in the following sections.

3.2.4 Advanced Command Features - The Macro Facility

A macro facility at an interactive level is one which can provide significant benefits to the user. At a basic level, such a facility can be viewed as simple textual substitution: where a commonly used command string can be placed in the processor's input buffer when the user enters a simple command. This assumes the existence of a virtual process which is positioned between the user input device and the input buffer to the active process, as shown in Figure 3-3.

![Diagram of the Virtual Macro Command Processor]

Figure 3-3: The Virtual Macro Command Processor

The advantages of such a facility lie in the ability to specify long, and commonly used input sequences by a smaller number of keystrokes on
the user terminal. Such a simple tool can be further enhanced by the use of parameterization. Here the macro is invoked as before, but additional text fragments may also be entered by the user. These fragments, or parameters, are placed into the macro string where specified by the macro text and the resultant text is then sent to the processor as input.

This tool can be further enhanced to allow repetition of a substring of the macro text, conditional expressions, labelling of points within the macro (with associated GOTO commands), or even a recursive ability of the macro processor to itself invoke further macros. Together with an ability of the macro scanner to synchronize the feeding of input text to the processor with the processor input calls, the macro scanner can effectively generate lengthy sequences of commands. As can be seen, such enhancements increase the complexity of the macro scanner, and effectively change the view of macro objects into that of programs in themselves, written in the macro language.

There are a number of methods of providing such a 'macro' input facility on computer systems, and this section will some common tools.

3.2.4.1 Terminal Function Keys

This is a terminal level facility, where the terminal is programmed to send a specified sequence of characters when a function key is pressed. The programming of this key may be a firmware operation of the terminal resident micro processor, or may be specified by the user, or loaded by the computer system. The facility is one of simple text substitution, where the resultant input stream has the function key replaced by the macro string.

3.2.4.2 Local Micro System

This configuration uses a stand-alone micro processor system that connects to both the terminal and the computer system. This device normally acts as a 'pipe' for data flow between the terminal and the computer system, but can also provide additional facilities. The micro can perform more complex data scanning than the terminal controller, and can
provide a better macro command facility, control of additional devices connected to the micro system, such as a printer, and can allow the user access to a local file system as well as access to the remote computer system.

In this fashion the micro can implement a more versatile local command language, and data scanning facilities, such as terminal emulation, the ability to use the contents of a local file as a command stream for the remote computer system, or to create a local file using the output from the remote computer system.

Such a system can, through the input scanner, implement a macro command language, that can allow full use of the local file system, both as a store of commands and as a place to store remote system output. The major problem faced by this configuration is that there is no flow of data control information between the two systems. The remote computer system cannot indicate to the micro when more input is requested, and when input generation must pause, and similarly for the micro receiving output from the remote system. Certainly single character pause/no-pause control characters (XON/XOFF protocol) can be used to implement this control information, but such protocols are very sensitive to timing problems, and rely critically on buffering both within the micro and the remote computer system.

3.2.4.3 Front End Devices

A more robust method of synchronization of data flow between the macro processor and the computer system is to move this function 'closer' to the computer system, and the use of 'front-end' computers as intelligent device controllers is a common feature of many mainframe configurations. These front end devices, as well as performing basic device control functions, creating a conceptual data 'pipe' between the user terminal and the computer system, can also perform some limited form of macro input. The Honeywell CP6 system is an illustration of such a system, where the
front end computer can perform a form of terminal function key functionality, but with the ability to allow the user of the computer system to bind any input key, or short input sequence, to any specified text string. Here, as there is a considerable flow of control information between the computer system and the front end device, problems of synchronization of data flow are not apparent to the user.

The range of facilities provided by such a configuration are limited by the inability of the front end to access the user file area, so that a more sophisticated macro processor, using command files is not possible from the front end controller.

3.2.4.4 Macro Command Features at the Command Level

The three approaches outlined above possess a degree of uniformity in their approach, in so far as this ability can be invoked at any time by the user, regardless of the state of the active process which the user is running. The macro input facility can be used as a command at the command level of the interactive environment, invoked with the name of the user file containing the macro definition, and any parameters needed.

An example of such a facility is the DEC-10 'MIC' [DIGI82] facility, a processor which can be regarded as a separate asynchronous process to the user process. The command language includes conditional constructs using an IF directive and basic looping using labelled points in the command file (together with a GOTO command), and also local variables and parameter substitution. Because of the position of the 'MIC' processor within the operating system, this process can implement significantly more functions than a front-end or micro configuration. Retrieval of job status information, and limited retrieval on the success, or otherwise, of actions specified by the resultant input stream are possible within 'MIC', together with the ability to return control of input back to the user, and the user also being able to resume this suspended input processor. The major weakness of this approach lies in two areas: The inability to invoke this
processor from within a process, as 'MIC' can only be invoked at the command level, and the inability of 'MIC' to provide a closer interface with an active process, in terms of the ability to interrogate the state of the active user process.

However this solution provides a uniform approach to macro input, where the form of macro input is constant within the interactive environment, not dependent on the particular active process.

3.2.1.5 Processor Based Macro Input

The final method is for each processor to implement a form of macro input. This style is very common within editors, allowing the user to specify edit procedures, and then invoke such procedures in a fashion consistent with the editing environment, creating a macro language for editing within the editor itself. As mentioned previously, TECO [BBN73] is a good example of such directions within editors. The major advantage of such a tight binding between the processor and macro input is the ability for the macro language to interrogate the state of the active process, and the ability to create more powerful macro commands, where the range of conditional tests within the macro language is considerably greater than any of the above methods.

3.2.4.6 Macro Input and the Pascal Workbench

In designing the Pascal Workbench, the decision on the form of macro input, and the means by which such a form of macro input may be specified requires a compromise between a number of goals of this form of input. Providing a macro facility within each processor is not a sufficient solution, as the user then loses the ability to create a command file that calls a number of processors, but creating a more general macro facility then implies a smaller set of capabilities within command files, and also a greater overhead in processing such macro input.

Thus the desire for consistency among all the tools of the interactive environment is a more important consideration than the ability to use
processor specific macro commands, so that, from a user point of view, a single macro facility is a better solution here. However the special requirements of the editor environment must be recognised, and the ability to ‘package’ a sequence of commands specified during the edit session, and execute these commands within the editor is a significant tool of the editor environment. The broader class of macro command files, and invoking macros at the command level is presented with the implementation of the command language interpreter in Section 3.3 of this chapter.

3.2.5 Advanced Command Features - Multitasking and Command Pipes

One of the most powerful features of a user interface is the ability of the user to redirect the output of a command to a file, rather than the terminal. This feature is particularly useful when a command generates lengthy output, enabling output to be generated at a higher speed, and, subsequently, the editor can be used on this output file to view relevant sections. Similar benefits can be obtained from file based input, where redirection allows the user to specify an input file, used in place of user input to the program. Apart from the aspects of increased speed of execution using such a facility, there are a number of other benefits that may be realized.

The first relates to ‘background’ processes, or tasks. This ability allows the user to issue a command, with associated input and output files, and, as there is no further interaction with the user until task completion, this task need not remain in control of the interactive terminal. This leads to the concept of a multitasking environment, where there is a single interactive task, in communication with the user, with the ability to spawn background tasks, which are executed with I/O redirection in a non-interactive mode, such tasks being removed on completion. The user has little direct control over such tasks, apart from the ability to interrupt the task, and issue either a terminate or resume command.
The second benefit from this feature of I/O redirection lies in the concepts of a chain of processes, each process using the output from the previous process, and generating input for the next process. At a basic level this implies each process in turn taking the input file, and producing a new output file, to be used by the succeeding element. Using a multitasking environment, the I/O redirection could be effected between running processes, such that a true parallel pipe could be built, with operating system control over the concurrent access to the common buffer areas. Such an arrangement is shown in Figure 3-4.

![Parallel Pipe Organization](#)

**Figure 3-4: Parallel Pipe Organization**

Such a model allows a view of processes which is not exclusively that of complete entities, but with the additional ability for such processes to be discrete elements of a larger user-defined process. The underlying difference lies in the assumption that the output from any process may be used as input to some unspecified subsequent process. Many of the applications of this methodology of interfacing processes are described in 'Software Tools' [KerP76, KerP81], and in articles on the UNIX operating system [KerM81, RitT78].

### 3.3 Implementation Issues of the Pascal Workbench

This section presents an examination of a Pascal Workbench, created as part of research into material for this thesis. The system has been implemented on a Digital DECSystem-10 running the TOPS-10 operating
system [DIGI82].

3.3.1 The Interactive Environment of the Workbench

The Workbench is designed to be a resident assistant to the interactive user, aiming to present relevant information to the user, and to provide active assistance during an interactive session.

The most visible change in the user environment when using the Workbench is the display produced whenever the Workbench environment is resumed. This display (Figure 3-2) is intended to display to the user the current state of the user area within the host system. This information is based on the user file store state and current directory settings, and may be abbreviated in order to accommodate the display within the terminal display capacity.

The user has a certain amount of control over this display, in terms of the type of files presented within the file window, and the level of assumed expertise of the user. At higher levels of display, the command menu window is removed from the display, enlarging the file window. As this display takes a considerable period of time to produce, the display generator can be interrupted at any time by user input.

The next major alteration in the environment of the Workbench is the command language. In common with many command languages, the Workbench command language is verb-based. However any of the command menu verbs can be specified by using any abbreviation of the word, down to a single letter. Also, commands assume that if only a single file would normally be specified as an argument to the command, then no file specification is entered by the user: the current file is assumed (as described in Section 3.2). Thus, for example the commands 'EDIT', 'ED', or 'E' all invoke the editor to edit the current file. If additional arguments are necessary for the command, then either these can be specified within the command line. If the command interpreter detects missing arguments
for a command, then the Workbench will issue a prompt specifying the type of argument that must be entered for each missing argument, displaying any pertinent information necessary to enter the missing argument.

The Workbench also maintains a table of applicability of commands to certain file types, so that some commands are bound to specific file types, and certain commands are disallowed for some file types. Thus, for example, binary file types may not be edited or printed. Also some generic commands, such as the 'RUN' command, while being applicable to a number of file types, will invoke the applicable compiler to compile the program, and link the resultant object file with the appropriate runtime libraries necessary to execute the compiled code. This command also illustrates the ability for a single command to be able to generate a number of actions, not directly visible to the user. Here, the 'RUN' command causes compilation (if necessary), followed by link-loading, and execution of the resultant code.

As well as these commands, the Workbench also supports a number of different command types. A prefix of the <file> character indicates that the following word is to be interpreted as a file name. This command causes a search for this file name, stopping when either an executable file type, or a command file type, of that name is located. This search is performed within the user area, then the library area, then the common area, and lastly in the system area. If an executable file type is located, then the Workbench will invoke this program. If the file is a command file, then the Workbench will perform an implicit redirection of input, so that the file contents will be interpreted by the Workbench command interpreter. In this case additional parameters specified on the command line will be interpreted as parameters to either the executing program, or parameters to the command file.

A prefix of the <background> character directs the command to the background execution environment, to be executed as a background task.
A complete command description is provided in Appendix II of this thesis for the Workbench.

3.3.2 Internal Organization of the Workbench

Before presenting detailed descriptions of the various functional modules that create the Workbench environment, it is necessary to outline the major modules of this processor, and present a brief outline of their interaction.

Internally, the Workbench is organised using three asynchronous processors: a display controller, to manage the screen display, a command interpreter, and a background task module. The internal arrangement of these modules is shown in Figure 3-5.

![Figure 3-5: Internal Organization of the Workbench](image)

In terms of internal synchronization of these processes, the command interpreter is passive until user input is available. If this input completes a command, then the command interpreter will remain active until the command is complete. The background task process remains passive until activated by either the command interpreter, or by an information packet received from the background task. This process will return information to both the command interpreter and the display processor, and then will
revert to a passive state. The display processor is initially active, and will remain so until the display is complete. The task may be reactivated by either the command interpreter, or by the background task process. If more than one process is in the active state, then there is an internal scheduler implicitly defined by a priority placed on the processors, such the command interpreter is the highest priority, followed by the background task process, and then the display process.

3.3.3 The Display Process

Once the entry actions of the Workbench are completed, this display process is activated. The task of this process is to maintain an accurate picture of the file directory within the current project, and, if selected, to display a list of commands which are applicable to the current file and with respect to the current state of the Workbench environment.

As this process may be interrupted by user input, the display process must display the most relevant information for the user first. Thus the order of the display process is to first clear the screen, display the current file name, then the list of applicable commands for that particular file, followed by the remainder of the file list, then the constant commands (logout, help and such), and, lastly, the screen headers. As this display is being produced, the display process maintains an internal description of the state of the screen display, so that interruption and later resumption of this process will result in the display process continuing to build the display from the interruption point.

The display uses an internal virtual terminal to generate the display control codes, and a configuration table to map these codes into terminal control strings for a particular terminal. The process also uses the command interpreter’s internal table describing the file store state, and also the command interpreter’s table which maps a file type into a set of applicable commands. The display process can also use a user defined filter when displaying the file state, so that certain file types can be masked out
of the display (This feature is used for novice computer users, to mask out the display of binary files from the display. Deletion of a file also implies deletion of any binary files of the same name in this case).

Thus the display process manages all output to the terminal generated by the Workbench, ensuring integrity of the display at all times.

3.3.4 The Command Interpreter

When an input character is entered by the user, the display process and the background task process are suspended (if active), and the command interpreter is activated to process the character. If the character is an in-line edit character (character rubout, line rubout, word rubout), then this edit function is performed on the current command line. If the character is a 'normal' character, then it is added to the current command line, and in either case, the command interpreter passivates, and any previously active process is resumed. If the character is a command terminator, then the command is interpreted. The first action is to determine if the command falls within the set of masked out commands (according to the current file type). If not then there are a number of possible actions taken by the Workbench in order to interpret the command:

1. Internal Commands

This set of commands are the most direct to interpret, implemented within the Workbench. Such commands include file operations (rename, copy, delete, change current file, project, library or common area and such), print commands, help and Workbench status setting commands. These commands are interpreted either by a corresponding change in the internal state of the Workbench, or can be executed by using program level calls to operating system functions. If such a command affects the displayed environment, the display process is notified of such changes on resumption of the
display process, so that the screen may be updated appropriately.

2. External Commands

This set of commands are interpreted by the facility (provided by the operating system) that allows the Workbench to invoke a utility program, and ensure that the Workbench is invoked on termination of the utility. Thus the utility program is invoked, and a return made to the Workbench without any visible change of environments at the user level. This mechanism is used to implement a variant of the print command, and to invoke a number of other utilities. These commands are implemented by a state save of the Workbench into a core-based temporary file (TMPCOR files on the TOPS10 system [Digi82]), followed by an invocation of the utility program. On termination of the utility, the Workbench is invoked by the operating system, which, as it's first action, restores the state of the Workbench from the temporary file. This restoration allows the display processor to retrieve the current state of the screen, and the command processor can now complete command interpretation as if the command had been accomplished internally.

3. External Processor Calls

This set of commands are interpreted by clearing the display (so that the change of environment is clearly visible to the user), and performing a processor call (or calls) to utility programs. These programs execute at a level which is visible to the user, and include calls to the editor, language compilers, link-loader, system utilities and also invocation of user-defined executable files. Such processors may interact with the user,
or simply display information during execution. On termination, the Workbench environment is resumed. Again the internal state of the Workbench is retrieved, but now the display process is re-initialized when invoked internally.

4. Command File Invocation

Command which cause a command file to be used as input effectively 'freeze' the keyboard for the duration of the command file interpretation. All Workbench output in this mode is also abbreviated, so the only the current command, and any direct changes to the display are output. The command interpreter reads successive lines from the command file, substituting any parameters specified in the invoke command as appropriate. User control over this process is limited to interruption, followed either by resumption or abort of this process. Termination of the command file causes the normal mode to be resumed.

5. Background Task Commands

These commands take the form of a processor call, and input and output file names (if necessary). This command form can also include pipe, where a number of processors are specified as a processor pipe. The background task is called by the command interpreter to handle this command, returning a simple status indicator to the command interpreter on whether the call was successfully initiated.

3.3.5 The Background Task Controller

The background task controller is implemented using a number of separate processors within the operating system level of the system. When considering the implementation of such a feature, a number of aspects of
the host operating system are important considerations here. In relation to
the TOPS10 operating system, a job cannot perform its own redirection of
input and output from the default device (the terminal) to specified files.
This function must be performed by a managing process, which acts as a
terminal to a sub-process, using input and output files as shown in Figure
3-6.

![Figure 3-6: Sub-Process Management and I/O Redirection](image)

This process management provision could be included within the
Workbench process itself except for one major consideration: while the
sub-process is active the managing process must also remain active, and
cannot invoke other processors. Thus, to allow the Workbench the ability
for the user to specify further commands while the sub-process is active
implies that the task of sub process management must be performed by a
separate process. Communication between the Workbench (in particular the
background task module) and the process manager uses the operating system
construct of inter-process communication packets (IPC packets). This allows
asynchronous communication between the Workbench and the process
manager. Thus, if the process manager sends a IPC packet to the
Workbench process, if the process is not currently active then the packet
will be read once the user returns to the Workbench state. These packets
allow the Workbench to instruct the manager to initialize a sub process,
specify input and output files, and also specify the commands to be directed
to the sub-process. The process manager informs the Workbench of completion of a sub-process again using these packets.

In a timesharing environment such a solution may appear expensive, using three processes to serve a single user, but there are a number of refinements to this design to avoid this problem. Firstly a process manager may control a larger number of separate sub-processes simultaneously. Secondly, such a process manager need not serve a single user, but can accept commands from a number of users, and serve all such users. Thirdly, using input queues for process initiation commands allows the manager to have a number of pending sub-process requests while all sub-process 'slots' are active. In many ways this is a parallel of a batching system, but with additional features of status inquiry, user abort control, and command pipes. This organization is shown in Figure 3-7.

![Diagram showing Process Management Organization](image)

**Figure 3-7: Process Management Organization**

### 3.3.6 Host Operating System Interface

To graft a new user interface on to an existing operating system certainly requires a certain amount of flexibility from the original system. The desired goal is to supply a new user interface to the system, by utilizing the existing program, or system call interface to the system to best advantage, so that much of the functionality of the Workbench may be
implemented without having to invoke utility processors.

The initial facility required from the system is the ability to specify a program to execute whenever a user program exits, either normally or abnormally. Thus, whenever the user job enters a state which would normally carry through to the interactive command processor of the operating system, the system loads, and commences execution, of the specified program instead. Without this facility the user would be forced to continually invoke the Workbench whenever the system reached the command processor state.

The next facility required is the ability for an active process to specify another program to execute, and then use the operating system to load, and commence the execution of the second program. Again this is a critical facility for the implementation of the Workbench, allowing the Workbench to invoke utility programs.

For the display processor to function efficiently there is the need to interrogate the system to ascertain whether the user has typed a character waiting to be read by the program. This is the interrupt condition of the display processor, and causes the Workbench to switch states to the command parser. Also single character terminal I/O facilities are a necessary component of this system to manage the screen effectively.

The program also makes use of a temporary file to save the state of the Workbench across process calls, so that the re-entry to this controller places the user at exactly the same state (or as close as is possible) as when the process was called. It is advantageous in terms of speed to use a temporary memory-based file, if the operating system supports such a construct, but disk-based files will only imply slightly slower entry into the Workbench controller.

For the features of a Library directory and the use of grouping files under Project names, there must be a comparable construct within the
operating system. Within TOPS-10 the library is implemented by use of a
job's search list, and the projects by the use of SFDs and PATH calls to
TOPS-10 [DIGI82]. It is outside the scope of the Workbench to create such
constructs without any operating system support.

This chapter is an examination of the role of the editor in the user
interaction with the operating system. The necessary constructs to support background task commands have
been detailed in the previous section. It must be noted here that this task
could be made internally much simpler, and the background commands
more ambitious if the operating supported the concept of 'forking' a process
into two user processes, and if the operating system supported the concept
of I/O redirection. Thus background tasks could be created by simply
forking the current process, and setting up the I/O redirection necessary for
the background component of the fork.

Any text editor design is the result of balancing the desirable goals
of such a tool and the applicable constraints of the environment of
operation, and is attempt to position the editor optimally between such
goals and constraints. The following sections in this chapter will contain
numerous goals and constraints of interactive text editors and will present
as a case study, an implementation of a general purpose text editor,
developed for this thesis as a component of the Pascal Workbench. The
next chapter will also present an editor, in this case a structure editor for
Pascal, as a component of the Pascal language environment.
4. Interactive Editing Tools

4.1 Introduction

This chapter is an examination of the role of the editor in the user environment, and also an examination of text editing and existing editor processors.

The primary goal of an editor is to edit text. As secondary goals, the editor should provide an efficient environment to achieve this, and accomplish this without extravagance of system resources.

The rationale for text editing software in an interactive environment is to allow the user to specify changes to be made to a stored text object in 'conversational' fashion. It is therefore essential that this interaction between the user and the editor be the major component of editor design. As Kernighan and Plauger point out, "concern for human engineering must dominate the design" [KerP78]. This inevitably effects all aspects of the editor design, from the command language definition to the choice of internal data structures, and this aspect of design cannot be stressed too heavily. As well as a concise and easily learnt command set, this includes aspects such as the structure of the information displayed to the user.

Any text editor design is the result of a study of the desirable goals of such a tool, and the applicable constraints of the environment of operation, and is attempt to position the editor optimally between such goals and constraints. The following sections in this chapter will examine common goals and constraints of interactive text editing, and will present, as a case study, an implementation of a general purpose text editor, developed for this thesis as a component of the Pascal Workbench. The next chapter will also present an editor, in this case a structure editor for Pascal, as a component of the Pascal language environment.
4.2 A Brief Survey of Text Editing Strategies

As understanding of a particular problem domain matures, largely as a result of experience, the tools involved become more specialized, moving from a small number of general tools to a wide variety of specialized tools. This observation is certainly relevant to a study of text editing, where, since the emergence of the initial interactive editors of the 1960s, there has been introduced a wide variety of text editing tools, matched by an equal quantity of literature on the subject. The directions of development of text editing are widely divergent, but there is a discernable trend toward editors which have been developed for a particular application area, rather than all-purpose editing tools.

4.2.1 The Structure of Text

An editor is a user tool to allow the creation and revision of a text object by the user. To obtain a clearer view of the definition of a text object, it is necessary to examine the attributes and possible internal structure of text in an editing system. Such an examination is useful to categorize such structuring of text, and can be used as a method of classification of editors themselves.

It is appropriate at the outset of this survey, to define in a more rigorous sense the definition of a 'text' data object. Here some elementary ideas from formal language theory are appropriate, where an editor may be considered as a tool to edit elements of a language \( L \), generated according to some grammar. Elements of this language \( L \) are said to have a structure according to their canonical parse tree. Based on grammars of increasing complexity, there are editing tools of increasing complexity.

The most simple grammar of this language \( L \) is:

\[
G_1 \quad L = \{\text{character}\}^* \text{eof}
\]

(To avoid a tedious enumeration of the special cases that arise from different character sets, the ascii character set will be assumed in all
examples and definitions, except where explicitly stated. There is no loss of generality of the arguments presented here by such an assumption.

The 'eof' symbol denotes the end of the character stream.

This grammar places no inherent structure on the text object, but the component characters may be either elements of the 'printable' subset of the ascii set, or elements of the subset of 'control' characters.

A second level of complexity of such a grammar is the explicit use of 'line breaks' in the language:

\[
G_2 \quad L = \{\text{LINE}\}^* \text{eof} \\
\text{LINE} = \{\text{character}\}^* \text{eoln}
\]

This grammar places an inherent structure on text, in a manner which corresponds to a simple display device, as a sequence of print lines.

This view of text as line structure plus printable characters can be further structured by the introduction of 'page' structuring, where a page mark in a text object will cause a display device to generate a page break:

\[
G_3 \quad L = \{\text{PAGE}\}^* \text{eof} \\
\text{PAGE} = \{\text{LINE}\}^* \text{eop} \\
\text{LINE} = \{\text{character}\}^* \text{eoln}
\]

These definitions have successively defined one-level, two-level and three-level textual structuring. As more structure is placed on the definition of the object there is a loss in generality of the resultant class of such objects, but this loss can also be viewed as simplification of the definition, where text is divided into two components, structure defining characters and printable characters.

These views of the structure of text are by no means inclusive, and, for particular applications, a more complex description of the edited text is necessary. Thus a 'Pascal oriented' editor is simply a tool to edit a language generated by a grammar which is further along in this hierarchy of grammars presented here.
4.2.2 Editing - The User View

The user level abstract machine (as presented in Figure 1-2) can be used to indicate the user view of the editing process. An editing abstract machine, a refinement of this user level machine, is shown in Figure 4-1.

![Diagram of the Editing Abstract Machine]

Figure 4-1: The Editing Abstract Machine

The user view of editing is that of a command-response dialog, to position the current component pointer within the text object, and then to specify edit operations to modify the text element referenced by this component pointer.

The current component pointer of this machine is determined largely by the grammar used by the editor to parse the text object, and this current component pointer can be seen as a pointer to an element of that grammar within the parse of the text. Thus, in the basic grammar presented above (G.1), this pointer is a pointer to a character within the text, and in the second level grammar (G.2), this would point to a line of
text, and so on.

Thus, the resultant user view of the text object is based on this notion of the parse tree of the text, and a visible pointer into this tree. The commands generated by the user affect the current component pointer, to position the current component pointer to a desired element within the text object, or to replace, or modify the referenced element in some fashion. The response is typically the current contents of the referenced text element, so that the user is constantly presented with immediate verification of each command's actions.

More detailed aspects of this editing model will be presented in later sections of this chapter.

4.2.3 Editing - The System View.

A typical editor internal organization is that of Figure 4-2. Here the main functional modules of an editor are outlined. This figure can be viewed as an implementation of the machine subsystem indicated in Figure 4-1.

![Diagram of Text Editor Internal Organization]

**Figure 4-2: Typical Internal Organization of a Text Editor**
The command processor accepts commands from the input device, and then, on parsing these commands, will send control to either the command interpreter, or directly to the display controller. The commands that are generated by the command processor are either an internal version of the user command, or implicit commands ("side-effects" of the user command). This module is also the logical controller of all other modules within the editor. Thus, if the display controller, the command interpreter, and the file subsystem are all viewed as asynchronous processes, then the command processor arbitrates between these processes.

The display controller accesses the internal text object, and current edit position, to rebuild the display. This process is resumed by the command processor, and may be halted by the command processor on specific conditions (user entered a subsequent command). This process may use a model of the output device to produce a display, or a virtual terminal, if such a concept is supported by the operating system. The process may also hold a data structure reflecting the current state of the display, or any other information necessary to update the display with a balance of I/O traffic and CPU time. This module may also be used to control multiple windows, or similar advanced display features.

The command interpreter can affect the exit text (by a write operation), and access the text only through the edit text pointer, and can also direct the file subsystem to transfer a block of text from the text area into the file store area. The actions of the interpreter are determined by the command processor either by a number of entry points into this module, or the interpreter can perform a parse of the internal token string generated by the command processor.

The current component is a pointer into the edit text object. The interpreter can move this pointer, or use it to read or write a text element at the current position, or to delete or insert an element at this position.

The edit text is the internal data structure of the text object. This
structure may reside in internal store, or may, by virtue of an internal, or system-defined paging system reside on both disk and internal store. If a paging system is implemented within the editor, this would reside within the command interpreter, and paging would be determined by movement of the current edit position to a location within the edit text which is currently residing on disk.

The file subsystem is independent of this paging operation, and is a module which can write a specified subtext to a file, or insert a file contents, or section thereof, into the edit text. This process is under explicit control of the command interpreter, and implements the interface between the editor and the file management section of the host operating system. and ViPens (ViPS) are examples of such editors.

The configuration of the system is also a consideration here. There are commonly two types of configuration: firstly the timesharing environment, where the editor, to produce a responsive environment, is constrained by memory requirements, cpu resources any a very low bandwidth to the user display device. The second configuration type is the stand-alone environment, generally based on a micro processor system. Here the parameters of the system may vary from an 8 bit micro-processor to a more powerful personal workstation using a fast 16 bit micro processor. Less common is the distributed configuration, where, with support of a high bandwidth communications subsystem, a stand-alone workstation can perform editing on a local buffer, and the file store resides on a remote disk, accessed through the communications system.

4.2.4 Editors - A survey

This section will initially give an outline of the developments within the area of interactive text editing, and then examine in more detail the differing conceptual models of text editing.

The 1960's produced the first interactive editing systems, providing
systems support for the manual process of editing card images for a batch system. The text was viewed as a sequence a card images (of 80 characters), and each image, or record, was marked by a sequence number. An edit transaction consisted of a sequence number and the replacement record. Alteration within a record was an early improvement to this style of editing.

By the mid 1960's, these editors had become a little more sophisticated, and the basis for current line editors was established. Sequence, or line numbers, were still a feature of these editors, but as well as replacement, and in-line alteration, the user could also move a group of lines within the text object to a new location. The IBM editor ATS [IBM70], and VIPcom [VIP69] are examples of such editors.

A major improvement, by allowing variable length records, and thus, records of up to 500 characters was a feature of QED [Deul67], but the style of editing, and the conceptual model of text was significantly altered by the character stream editors of the early 1970's. FRESS and TECO [BBN73] are major examples of editors where the distinction between text and line breaks were removed.

At the same period the common acceptance of the interactive terminal prompted development of editors which could fully exploit this hardware improvement, and TVEDIT [Toll65], THOR [MDFA67] and IDA are examples of early development in this area.

As the applications for editing increased in scope, editing tools became more specialized during the 1970's, and syntax directed editing (EMILY [Hans71] is one such early development) and structure editors were developed to meet more specialized requirements of the edit environment, and word processing editors (Wordstar [MICR81], for example) is one such result of this development.

This period has also seen the beginnings of research into parameters
of human-system interaction, ergonomic design and usability of editing systems [CaMN80 and references therein], and many recent editors show the results of such research.

UNIX has also had a considerable influence over the area of interactive editing, and the concepts of the text toolbox, and the method of interaction with the user [KerM81], are now extending far beyond the original UNIX environment.

Hardware improvements have also enabled significant departures from the interactive model of computing in the 1970's, and current developments with personal workstations will produce a substantial change in format and style of interactive computing for timeshared systems. The Bravo editor [Lamp78], and the concepts used in the Smalltalk-80 environment [[Kr81, Inga81, Tesl81], with associated interactive devices of such workstations are leaders in this trend of interactive environments.

There are a number of current activities in the area of standardization of editing languages, text editing structures, and text formatting languages. ANSI (ANSI-X3J6 and X3V1) and ISO (ISO/TC 97/SC 5/EGCLPT) are both active in this area, but is seems probable, given the wide diversity of development in this area, both past and current, that such standards will be retrospective, referring to established procedures of text manipulation rather than more recent and innovative methodologies.

The following sections will cover major points of interest in currently available editors. In an attempt to categorize these editors, it is a significant point that the view of the text object presented to the user during an edit session (the edit 'window') in effect determines the user's view of the entire text object: the conceptual view of text. In this fashion, editors which present 'lines' of text to the user enforce a view of fixed physical line boundaries within the text, whereas editors using a character-based window can show that these line boundaries are simply ascii control characters, and are functionally no different from any other ascii
character in terms of editing the text. This differing view can affect the ease with which certain edit operations can be accomplished by the user. For this reason, the format of the text window is a reasonable guide to categorization of these editors.

4.2.4.1 Line Editing

The line editors assume that a line object is defined as a sequence of printable (i.e. non-control) characters (usually this sequence is upper bounded to between 80 and 500 characters) followed by a line break. This line break is not accessible to the user, and the defining characters which cause the line break are therefore not part of the text which can be edited normally. Line breaks can usually be manipulated (insertion, deletion, movement) but in a manner inconsistent with similar operations on the actual contents of the line. The text object is defined as a sequence of line objects, corresponding to the virtual card image of the batch model.

There are a number of compelling reasons to favour this style of viewing the text, the major one being that this model of text covers a wide range of applications.

Movement of the current component through the text is by units of lines, where the complete line is displayed to the user upon the result of applying an operation. Insertion and deletion of lines are relatively easy and consistent here, but alteration of text within the line must either be performed by entering a character based edit mode, or by substring specification, which does not sit easily on the line structure.

The user can be presented with a display of the affected line after each edit operation, and therefore can gain a better view of the context of in-line character alterations, but the still lacks an adequate display of textual context for line-based operations. Attempts to display such context do not fit into the main structure of the editor.

Commonly, a distinction is made between edit and input modes,
where in edit mode only editing commands are accepted, and input mode, where all input is placed into the text object with no alteration. Mode change from input to edit is by an empty line. Digital's SOS editor [DIG178] used this modal approach to create 7 separate modes, with elaborate methods to allow the user to switch between modes. The Univac 1100 series editor, ED, feature a similar approach, with an extensive command set allowing the user to define a private set of 'macro' editing procedures. The UNIX editor, ed [Kern78a, Kern78b] is also a bi-modal editor, with input and edit modes, yet here the editor produces no response to commands. The command to change mode from input to edit is simply '!', which causes many problems: "If ed seems to be ignoring you, type an extra line with just "!" on it. You may find you've added some garbage lines, which you'll have to take out later" [Kern78a].

In summary, this editing style has encountered significant criticism, such as the comment by Meyrowitz and van Dam on line editors: "We not not, however, advocate the continued production or use of these editors" [MeyD82].

4.2.1.2 Character Stream Editors

This approach views a text object as a string of characters, and the editor maintains a pointer to the current character window. There are 4 atomic commands in this approach, moving the window to the 'left' or 'right', and inserting or deleting a character at the window position.

This basic approach is well documented by Bourne [Bour71]. More useful text operations can be defined in terms of these four operations, and any useful editor must incorporate these more complex text operations (such as moving words) into the command set of the editor. With a small number of added capabilities, a very large super-structure can be built on these four operations, which produces an internally consistent and remarkably powerful editor. TECO [BBN73] is a definitive example of this approach, although there are many more specialized editors using a similar
Some problems do arise in this style of editing by allowing the user to see too much of the file. A good example of this is the line boundary, which is commonly defined as the ascii sequence <CR LF>. These editors will allow this text (which is more commonly defined as textual structure rather than text itself) to be edited in the same style as any other character sequence, with the potential of creating problems for the novice user. As the full character set is considered editable text this implies, at the command language level, arguments to commands which use the somewhat clumsy, but necessary 'literal' character specification. The TECO language, for example, does not provide a clean interface to the user, and the resultant syntax of the language can be described as "cryptic" [MeyD82] and complexity of the interface can at best be described as unhelpful and frustrating.

However this is a style of editing which is unequalled in versatility. The simplicity of the basic form of editing and the consistency of the character-edit approach combine to produce an editing tool which can perform a large variety of edit functions. TECO has become more a language used to implement other editors rather than a user level edit language itself (EMACS [Fins80] is an example here).

4.2.4.3 Screen Editors

A better strategy here is to use the video display as a window on the file. This approach was adopted in 1967 by the IDA editor, and TVEDIT, among the first successful screen editors, and this edit strategy was quickly adopted by a number of editors, including the Yale E editor, the Rand RE editor, and NED [Bil67], bb [ReL81], PEN [BaTW81], Z [Wood81], sds [Fras79]. The common model for these later editors can be traced to Iron and Djorup [IroD72]. The screen is a constant 'window' into the text object, and the position of the cursor within the screen is mirrored by the position of the current edit component within the editor's text object.
Thus, by moving the cursor within the screen, and changing the text displayed on the screen, such alterations are mirrored by the editor on the text object.

This is a very compelling method to edit text for a number of reasons. Firstly the size of this window on the file is usually quite large (24 lines of 80 characters), and this window is a window on the larger two dimensional plane of text. Secondly the core command set is small, and uses the keyboard in a logical fashion such that any printable character is placed in the file at the cursor position and any control function on the keyboard echoes that function on the screen and the edited text. A current edit position is always visible as the cursor on the screen, eliminating the positioning commands necessary within line based editors to perform in-line manipulations.

As an interactive software tool the screen editor is well suited to the interactive environment. The editor attempts to display a large amount of contextual information on the screen to provide the user with relevant information feedback for all edit operations. The display and the command parser can be tailored to suit the characteristics of the terminal, such the editor can implement an environment where commands can be entered easily, and the results displayed immediately.

These screen editors do, however, require more facilities from the host system than the classes of editors mentioned above. There is an implicit assumption in this design that transactions between the host system and the terminal are character-based. Thus when a key is pressed on the keyboard, the character, or character sequence is transmitted immediately to the host system and hence to the editor task with no form of echoing of the character by either the terminal or the host system and no alterations to the character sequence. This also assumes full bi-directional transparency of character transmission between the edit task and the user terminal. Such facilities are not a standard feature of many computer system or networks, due to the very high overheads of a character-based terminal device.
controller when compared to a record-based terminal handler.

This is not the only model for such interaction, however. An alternative is to provide local editing at the terminal level, and a mechanism to move 'screens' of text between the terminal and the host computer system. IBM XEDIT [IBM80] is such an approach, where 3270 terminals support changes to a local screen image, and also the ability to send to, and receive from, the host system, screen-sized buffers of text.

4.2.4.4 Memory-Mapped Display Editors

Xerox PARC's Bravo editor [Lamp78] is one of the first editor/formatters based on the display of soft-fonted text on high resolution display devices. The Star editor [SlKH82] is a later refinement of this approach.

The conceptual model is one of typeset text, using the hardware of a personal workstation. The method of interaction strongly favours the mouse device, and the screen display is based on a window, the components of the window being both textual (typeset text) and reserved sections of the window, which perform specific movement actions (see Figure 4-3).

Here the text display is maintained as a facsimile of the final copy. Thus multiple fonts, graphics capabilities, proportionally spaced fonts are all editable tools for producing the target document. Moving the display consists of moving the cursor (via the mouse) into a movement box, then pressing the mouse button. In Figure 4-3, the arrow areas scroll the screen in the indicated direction, the P retrieves the previous page, and the N the next page. The — and — areas scroll sideways to make either the left or right margin visible.

A similar style of editing is used by the PERQ system [Love82], with the notable difference that the PERQ editor does not encompass formatting or typesetting operations, so that this is a more conventional editor. Movement of the display is by a scroll bar (where the position of the
display relative to either extreme of the text can be set. Editing involves selecting an extent of text either a character, word, line, or number of lines (using the mouse device to perform this operation). This selected text can be manipulated by the use of pop up menus, which can delete, move, replace or otherwise, the selected text. Most of the editing is performed by the mouse device, on the principle that the screen is divided into a number of active and inactive areas. Each active area defines the actions performed when the mouse is positioned in this area, and the button is pressed. The style of editing is exceptionally fast and the resultant edit environment, is simple, consistent and quite powerful.

4.2.4.5 Structure and Syntax Editors

NLS, developed in the early 1960’s at SRI, later renamed to AUGMENT, was the first of these types of editors. This editor used a rather elaborate text structure, using text nodes, branches (a sequence of nodes), a plex (a sequence of all the nodes of a parent node), and a group,
a subset of a plex. Imposing a structure on the text object, makes certain manipulation operations far easier to specify, and in many cases achieve for the editor, but such structuring also can make some quite simple operations almost impossible to specify. There have been many variants of this approach to a tree-based structure of text, such as XS-1 [BurN80] and a number of more interesting derivatives, such as Fraser's 's' [Fras80]. 's' is a generalized editor 'front end', and, under UNIX, allow such features as editing a file directory, mirrored by comparable actions taken on the user's file store area.

Structure editors, and in particular the tree structured editors, have lead to the natural development of editing the tree structure corresponding to the syntactic structure of a program language. EMILY [Hans71] uses this approach, where the screen is divided into text, menu and message areas. The menu displays all possible substitutions, if the cursor is positioned on a non-terminal syntactic symbol of the language, and menu selection inserts that syntactic entity into the program at that location, substituting for the non-terminal symbol. However, as the editor possesses no form of text entry, all program detail must be specified using only non-terminal symbol expansion. This approach certainly maintains a continual syntactic correctness of the program at all stages of creation, but is quite awkward to use when entering complex expressions and similar entities, and the allowance of a textual input mode is an improvement on this model.

An editor using only syntactic structure can only maintain syntactic correctness of a program, and semantic issues cannot be covered unless the editor also includes some form of parser as well as the syntactic structure of the language. These issues will be covered in more detail in Chapter 5, where editing program text, and Pascal text in particular, will be studied.

Editors using this approach are the MENTOR editor [DHKL79], CAPS [WiTD76], Interlisp [TeiM81] and the Cornell Program Synthesiser [TeiR81].
4.3 Design Goals of a Text Editor

4.3.1 Overview

When considering potential goals of an editor design, the area of application is an inescapable consideration. Text held, and manipulated, on a computing system can take many differing forms, and the internal structure of the text will match, in some sense, the area of application. As the domain of text to be edited is made more restrictive, it is possible to make specific design goals for the editor, based on this restricted text domain. This section will focus on non-specific, or general-purpose, text editor (the line, character and screen editors as described above).

It is a major design goal that the edit language can meet the requirements of more specialized commands. For example, a language which can manipulate a number of non-text files within the command language. Thus a language which is a

There are two major areas of consideration in defining design goals for an editor. The first is the format of the user interface, and secondly is the performance criteria of the editor, imposed by consideration of the characteristics of the host processing system.

4.3.2 User Interface Design Goals

A text editor can be viewed as a transaction processor, in so far as, during an edit session, the user keys in edit commands, and is given some form of feedback on the result of the command before entering subsequent directives.

From this consideration there are two major aspects of the user interface that assume major importance for text editing:

- A structured and concise edit command language

- A well structured method of command response

4.3.2.1 The Command Language

As a general characterization, edit command languages are simple, context-free languages, where the actions dictated by a particular command are constant throughout the edit session, and are unaffected by any
previous, or subsequent commands.

The most elementary operation of an editor is to either insert a specified character at a particular position, or the delete an existing character, again at a specified position. Most of the editor's actions are built from these insert, delete and positional commands.

It must be emphasised that these commands are basic edit commands, and are at a level very near to the actual mode of operation of the editor. To make use of an editor more efficient, it is necessary to support a much broader set of edit commands.

It is a major design goal that that the edit language can meet the differing criteria of use from such a range of users, from novice to more expert users. For the novice user is a demand for a small set of commands with clearly defined simple actions. As a user becomes more experienced with an editing tool comes the demand both for a range of more specialized commands (‘macro’ commands in a sense, which can encapsulate a number of basic commands within the single command), and extensibility of the command set to enable the user to define command sequences to perform particular editing actions.

The second design goal in the edit command language is that of consistency within the command language. Thus a language which is a mixture of textual command verbs, and single control character commands can only cause considerable confusion to the user, and significantly increase the time taken for the user to become familiar with the command set. If the language includes the ability of specifying commands by single characters, then this design goal becomes quite critical. The use of mnemonics is a possibility here (e.g. U for cursor up 1 line, D for down, etc.), where the user can become familiar with a relatively small working set of commands in a short period of familiarization with the editor. However this method does have many problems (such as D for down, delete character, or delete line ?) and requires careful design. A somewhat
different approach is to use the position of the keys within the keypad to associate actions with keystrokes, though again this may present problems.

Also within this goal is the necessity of a consistent form of specifying arguments to commands. To introduce any 'power' in the command language, and keep the number of distinct commands to a minimum, it is necessary to parameterize certain functions. As a simple example, the 'move down' (D) command is made more powerful if the user can specify the number of lines to move the cursor down. This introduces the command $nD$ ($n$ is any positive integer), and the special case $D$, which is the equivalent to $1D$. Other commands with similar functions must also take similar forms of parameterization. Continuing the example, this would produce the commands $nU$ (move up lines), $nR$ (cursor right $n$ characters) and $nL$ (cursor left). In the general case in this exemplary language, if any command can be 'multiplied' by an integer the general form of the command would be $n<\text{command}>$.

The third design goal is that the command language be constructed from a small set of simple syntax rules. Although this may be considered to be implicit in the second design goal here, it is of sufficient importance to state as a separate design goal of the command language. This also implies a design goal of avoiding verbosity in the command language. The user's motivation, and hence the user's attention is focused on the alterations to be performed on the text object, rather than the correctness or otherwise of the commands needed to produce the desired result. Complex command strings tend to induce higher rates of command errors by the user.

Relating to this aspect, and, as a fourth design goal, the complexity of the operation invoked by the command should be related to the command syntax, so that simple, and more common commands can be easily, and directly invoked. For a novice user, having mastered the style of such commands, more complex commands may then be easily learnt and used. In this fashion, the user need not remember a large set of disjoint
commands and formats, but only a basic command set, and the syntax of more complex commands.

The fifth design goal relates to the extensibility of a command language. Given the wide range of text objects that a 'general' text editor would be used to process, and the diversity of operations that the editor would be called to perform, it is an impossible task to define an edit command language which allows all such operations, and still retain some brevity and internal consistency of the language. The language needs some form of extensibility, which would normally take the form of allowing the user to bind a name to a defined sequence of edit commands, and internally substitute that command sequence whenever that name is used. In this fashion a user can 'tailor' the edit command set to suit the requirements of the operations to be performed on the text object. Of particular attention here, is the attendant aspect of such extensibility, where, for such a facility to have reasonable applicability, these must be commands which refer to sequencing of 'normal' commands. As a basic example here, this refers to commands of the form REPEAT <command sequence> UNTIL <condition satisfied> or similar. The more general, and more powerful such facilities are made, there is a corresponding penalty in the complexity of the resultant edit language.

4.3.2.2 Command Response

This section deals with the design criteria which govern the method by which the editor shows the user the results of the user commands.

The first major goal of the command response is to provide a response which details, as completely as possible, the effects of the command on the edit text object.

The second goal of such a response system is to structure the response in a fashion which is consistent with the internal structure imposed on the text by the editor. Thus, for example, a line editor should use a response system of displaying the currently referenced text line, and not a
sub-section of this line.

The third goal is that such response should be as economical as possible, so that the user does not have to wait for a long response to complete before entering the next command. Such economies can take the form of utilizing the functionality of the display device to perform much of the response, when using a screen editor. With reference to screen editors this implies responses based on incremental update of the screen display rather than continual redisplay.

The last design goal relates to the structuring of a long response (such as updating a screen display in a screen editor). The user should be presented with the most relevant information first. Continuing the example of a screen editor, this implies that the area of the screen surrounding the current cursor position (the current edit component pointer position) should be updated first, followed by more distant areas of the screen.

All these goals relate to the speed of use of the editor, where the response of the editor is designed to present the user with accurate response to commands, and structure this response so that the most relevant information can be presented as quickly as possible. On system configurations where the display device is connected via a low speed character channel to the editor, the response of the editor has the potential of becoming a major bottleneck, so that careful design of the response subsystem of the editor can imply significant increase in the effective speed of the editor.

4.3.3 Performance Goals

This section will examine a number of relevant points concerning both the performance of the editor implementation and its interface and relation to the host operating system. The host operating system assumed here is a normal interactive, timeshared operating system found on most large computing systems.
4.3.3.1 Text size

Firstly, if there is to be a limit on the size of files which can be edited, this upper bound should be quite large. Without resorting to a heavy reliance on disk-based data structures there must, obviously, be some incremental penalty of memory usage for increasing the size of the edit text, and that this incremental penalty will, at some stage, be bounded by an operating system imposed size limit. What is a desirable goal here, is not that such a limit should not exist on the size of text files to be edited, but that this limit should be quite liberal on the size of the text.

4.3.3.2 Disk File Space

Some operating systems automatically provide a level of file security on the system by providing back cycles of every file, so that if a file is altered, the state of the file before such alteration occurred is stored as a back cycle to the altered file. As this is not a common feature on operating systems, it is usual for the editor to provide a similar feature by manual backup. Here, when the user commands a normal exit from the editor, the input file is renamed, so that the file type is changed to indicate a backup file, and the edit text is saved under the original input name. Such a strategy may lead to problems with disk space. If the editor is using a temporary file, then this file holds a copy of the input file, plus any edit changes specified during this session. Upon normal exit from the editor, the editor may opt to copy this temporary file to a new file, then delete any existing backup file, and the temporary file, and then rename the input file to a backup name and rename the new file to the input name. In such a sequence the editor may use disk space equal to 4 times the size of the file, and double the normal disk space requirements (current and backup). Such a large temporary disk usage will cause problems when disk space is limited.

4.3.3.3 Memory Size and System Interface

Memory size of the editor is critical within a number of parameters:
the upper size limits of the editor are often bounded by the size of physical memory (on a mini or micro stand-alone system), or a more nebulous factor of optimal behaviour under the scheduler and/or page fault handler of a timesharing operating system (a memory size sufficiently large to avoid being swapped out in many cases, but small enough to allow being swapped back in quickly). Also memory expansion or contraction should be very rare operations, opting for a small number of larger operations rather than a large, and continuous number of smaller requests. The technique of localising code, and mapping code and data wherever possible should be used to avoid unnecessary page faults on paging systems. Thus here is a very difficult balance to strike here between the more obvious size/speed relationships of the editing algorithms used, the paging rates, the memory size and I/O occurrences, balancing the behaviour of the editor under the operating system behaviour. Certainly it is a valid point that an efficient program which is handled at a low priority by the host system's scheduler will be significantly worse from the user point of view than a less efficient program which is treated more favorably by the scheduler.

Thus, tailoring an editor's characteristics to perform well under a particular operating system's scheduler may return much larger benefits than tuning with respect to machine efficiency only. In the case of sharable code on a timeshared system, again there is the tradeoff between this sharability, so that if the sharable segment is swapped out of core, all editor users are delayed, and the other situation of having multiple, paged copies of the editor code. To effect a reasonable solution here requires more than a cursory effort when tailoring an editor to a particular operating system, and the potential performance improvements are quite significant.

4.3.3.4 Algorithms and Maintenance

As is the case with many similar heavily used system components, the somewhat conflicting tenants of software engineering apply: of both the need for algorithms which can offer the best efficiency, and a maintenance of program clarity and style. An editor will typically have a very long
system life, and as a result of that, maintenance effort of a heavily used editor will far exceed initial development effort. For this reason, the effort of maintenance, in both error detection and alteration must be reduced by a goal of clear and well structured code when developing the editor, overriding in many cases alternative code segments which are efficient at the cost of obscurity and maintainability.

4.3.3.5 Text Accessibility

The above points relate to the system impact and maintenance of the editor. There are also a number of equally important performance goals relating to the usability of the editor. The first of these relates to the internal organization of the editor's edit window. During editing all parts of the edit text should be accessible, to be examined by the user in any order, and with no 'undue' penalty for large movements of the text window in either direction. Single pass editors, which allow the user to edit each segment of the text in sequence only are not a desirable goal of editing, nor is an editor which must perform a full file copy to reach the first line of text, nor an editor which must use an internal move to line 1 as a component of the 'move up 1 line' function. Certainly a user can conceptualize, and readily tolerate a delay to movement (probably related logarithmically to the distance to move), but such delays must not be longer simply because the movement is towards the start of the text rather than the end of text.

4.3.3.6 String Locate

A text movement based on locating a target string should not be significantly longer than a movement based on line numbers or similar. Owing to the difficulty of implementing this goal, it is often the case that variations on the basic search (such as wild characters) are not implemented [Dav82].

4.3.3.7 User Priority

To provide a tool which is well tailored for use, it is necessary that
the user be allowed to interrupt any editor process that is currently active, and that that process may be resumed if the user so wishes. There are two types of this interruption: interruption of a display process by entering a command character, in which case the editor should either execute the command or resume the display process, and interruption via a break character, in which case the editor should suspend active execution of the current user command, and offer the user the choice of either resuming the editor and the current command, resuming the editor at the user command level, or aborting the edit session with no updating of files.

4.3.3.8 Allowance for the User

This implies that the editor should make some allowance in the design for the fact that inappropriate commands will be entered quite frequently, and often such mistaken commands will be delete commands. Thus the editor should save in a local buffer the contents of the last delete command (whatever the span of text) so that, if the delete was a mistaken command, the text can be recalled back into the edit text. Similarly the quit exit, which would throw away the results of the edit session would need the user to send confirmation of the command before the command is effected.

4.4 A Case Study

This section will examine in considerable detail an implementation of a general purpose text editor, developed as a component of the Pascal Workbench for this thesis. The section will cover implementation aspects of text editors, outlining alternative strategies where applicable, and the reasons why particular approaches were adopted. The editor has been designed within the goals given in previous sections of this chapter, and this section illustrates the issues that arise from this approach.

4.4.1 Memory Management

A text editor, at the basic level is capable of two fundamental
operations: insertion of a character at a given position in the text, and deletion of a character in the text. The implication of either of these basic operations is that, logically, the remainder of the data structure must be rewritten at some stage following the operation to allow for the expansion (or contraction) of the text object at that point. Much of the design of the editor data structure must be devoted to a means of efficient circumvention of this extremely high manipulation overhead.

This section will examine a number of applicable methods of data management. At the outset of this section there are two points which must be stressed. First the data structures being examined here are data structures for a general text editing system. More specialized editors can take advantage of a number of assumptions about the structure of the text to be edited and can be optimized within such assumptions. There are very few assumptions that may be made about the structure of all text objects on a computing system. It also must be stressed that any solution to this problem is somewhat reliant on the type of hardware and operating system under which the editor is implemented. Solutions for a paged virtual memory system are not necessarily identical to those of a non-paging environment, so that the characteristics of the target system are a contributory factor in the choice of appropriate data structures.

Design of the editor data structures can fall, broadly, into two major classifications: firstly the core-based buffer concept, where the complete edit text, or paged sections of text are read into a core data structure, and upon termination, or at an edit checkpoint, the core image is written out (logically ordered) to a disk file. The second class of structures are disk-based structures, where a small core buffer is used, and a randomly accessible 'scratchpad' file is utilized to store a copy of the edit text (in some possibly non-sequential ordering). Using this method there is continual traffic between the core buffer and the I/O file.
4.4.1.1 The Edit Buffer Memory Management Scheme

The first core-based structure to be examined is the buffer gap method. The edit text is stored in core a two contiguous streams of text with a (possibly null) gap between the two streams. This gap is notionally 'unused' and is invisible to the user. Character insertion is performed by physically shuffling all text between the current gap position and the point of insertion, so that, in effect, the gap is moved to the insertion point. Insertion is then accomplished by reducing the gap size and updating the associated index system into the buffer to note this. Deletion involves a similar operation, and correspondingly increases the gap size.

Within this system, unused gaps may appear at either the start or end of the buffer. To make the complete structure consistent, a rather elaborate set of indices must be constructed into the buffer structure, and a three level coordinate system is necessary (user level, gap level, storage level). All user level accesses are converted to a gap level access.

As an example, Figure 4-4 illustrates a buffer containing the word "ELABORATE".

![Buffer Gap Scheme Diagram](image)

**Figure 4-4: The Buffer Gap Scheme**

This buffer is 13 characters long, and the gap is positioned between the "B" and "O", and the current pointer position is between the "A" and the "B".

The top level of numbers is the text level coordinate numbering.
The values run from 0 to the end of the buffer. The gap is invisible to this numbering level. Note also that the numbering system numbers the inter-character gaps, not the actual characters themselves. The current pointer is positioned using this numbering level.

The second level is the buffer numbering level, displayed immediately below the text. Here the positions between the characters are labeled, and the labeling refers to the buffer storage system. All internal arithmetic is performed with reference to this system.

The third level is the storage level, where the underlying character storage cells are numbered according to the storage locations used.

To read the character at a text level position, the position is converted into a buffer level position by comparing the position with the gap start position, and if greater, then the gap length is added to the position to produce a buffer level position.

To insert or delete a character is an internal change in the internal value of GS or GE, if the point P is at either GS or GE. Otherwise the gap will have to be moved to P, by performing a physical shift of the characters between P and GS or GE. Obviously it is possible to create a sequence of moves and insert/delete commands which induce poor behaviour when the gap has to move relatively large distances through the buffer. It is also worth noting that, unless the size of the buffer is allowed to expand dynamically, there is a finite upper limit to the size of the edit text. There is an obvious trade-off between buffer size (and the resultant edit workspace size) and the speed of the basic insert and delete commands.

4.4.1.2 Linked Lines

An alternate structure is the result of implying some textual structure to the edit text. The text can be conceived as a doubly linked list of lines, where each line has some defined maximal size, and is allocated a multiple of some constant number of bytes in core.
As a Pascal data structure this would correspond to the declaration:

```pascal
Line = PACKED RECORD
  NextLine, PreviousLine : ^Line ;
  Length : 0..Maxlength ;
  Allocated : 0..Maxallocation ;
  Marks : ^Marklist ;
  Text : ^AllocatedText ;
END ;
```

- The `NextLine` and `PreviousLine` fields are used to implement the doubly linked list.

- The `Length` field is the actual length of the characters string.

- The `Allocated` field is the number of bytes storage that have been allocated to this line. This is a multiple of some fixed allocation unit.

- If the editor supports the ability to 'mark' sections in the text, then the line header will hold a pointer to the marks corresponding to this line.

- The final field is a pointer to the allocated storage for the text of the line.

The line can now be treated as a unit entity, so that allocation and freeing of single characters is performed within the line. Whenever this operation causes the line to increase over the allocated size, or decrease by this constant number of bytes, then a larger allocation is requested, the text is then copied into this area, and the old allocated space is freed. This design is similar in many ways to a garbage collection system, with fixed multiple length allocation and freeing ensuring that compaction is not a major (or frequent) problem, and fragmentation can be avoided in all but worst case behaviour. Referencing a position in the text then needs a vector pair (line pointer, offset within line).
4.1.1.3 A Paged Management Strategy

Comparing these two schema yield the result that the buffer structure is cheaper on memory requirements than the line structure, as the overheads of unused memory are smaller, but the buffer structure requires a greater amount of processor overhead than the line structure. In a virtual memory environment, these factors favour the buffer structure for most applications, but at the cost of more cpu intensive manipulation routines. The line structure will perform well in an environment where the complete structure can be contained in physical memory.

These two structures (and other core-based structures) break down badly when confronted with editing an extremely large file. The solutions involved here are either to insist on a single pass editing structure, where buffers are 'yanked' (to use a TECO term) from the input file into the core buffer in sequence, edited and then written in sequence on to an output file (by a subsequent 'yank' or an explicit write buffer command). To simulate full window movement on the edit text, a three file system (utilizing random I/O) can be used. When moving 'backwards' through the text, buffers are read from the output file in reverse order, and placed on a third file. When moving 'forwards' through the file, input is taken from the third file (in reverse order) and placed on the output file. Thus the third file (the scratch file) is used as a disk-based buffer stack. Neither of these solutions to the large file problem can be easily adapted to meet the requirements of the linked line approach.

Both these algorithms assume, in order to produce an efficient runtime structure within a multi-user interactive environment, that either the buffer size is tolerably small in terms of total system resources, or that only part of the buffer need reside in physical core at any time (paged virtual memory systems). This assumption may not always be the case. In a non-paged, multi-user environment, neither of these two strategies has a suitably small overhead in memory requirement. Thus the task of the editor data structures is, in such cases, to be able to, in effect, to simulate
the effect of a paged virtual memory environment, so that suitably large files can be edited within the multi-user environment. It is also worth noting here that even in a virtual memory management system, editing a large file using the buffer gap approach can produce an unacceptably large number of page faults if the gap needs to be moved over a large section of the text.

The buffer/file structures presented above are certainly applicable here, but they suffer a major weakness when trying to move the edit window to either extreme position of the text. In the three file structure, this movement from 'bottom' to 'top' of the text will result in a full file copy operation being carried out, and will also imply the overhead of having three copies of the file on disk during editing. The two file structure does not run the same disk overheads, but a 'backward' movement of the edit window requires the remainder of the file to be copied to the output file, and then the output file assigned to input, followed by a read from the start of the text to the new window position. Again this is a very expensive, and time consuming operation.

It is possible in such an environment, for the editor to build its own virtual address space, and operate its own paged memory management environment. Thus, the editor reserves space for a number of core buffers, and runs a page replacement scheme onto a scratch I/O file. The original text object is initially copied into core, and into the scratch file, in an appropriate format, and the file is segmented into a number of buffers (with the buffer size being equal or a multiple of the disk I/O unit record size). As long as the editor also maintains appropriate information about each buffer in core, then the editor performance can be comparable to a paged virtual memory system.

Addressing the type of page replacement algorithm to be used in such a situation, a least recently used buffer replacement system will produce reasonable results. There are a number of minor problems with such a structure: it is possible to contrive a sequence of movements and simple
operations that cause 2 units of disk I/O (a write and a read) for each movement, and also the complete edit text needs to be copied twice, on entry to the scratch I/O file, and on exit to the output file. Despite these problems, this is a structure which is quite appropriate for editing both large and small files. For small files, the text will fit completely in core, and no paging need be used, and for larger files, the paging system will ensure that the file will be edited within a fixed memory size of the editor.

Such a structure can be further improved, in terms of access speed, by adding a number of refinements to the systems outlined above, to speed the basic insert and delete operations.

Firstly the linked line system will not be directly applicable to this memory management strategy. A doubly linked list cannot be stored on both disc and core memory without major modifications to the list structure, and the pointer system would need to be replaced by some method of keyed data file.

To explain the modifications in detail, it is first necessary to examine a simplified view of the structure. If the editor window is a structured object, some number of lines long, then if this window is held as a linked list of lines, the editor can benefit from faster insert and delete operations within lines, as described previously. Movement of the window can be seen as a simple operation involving two stacks. If the input and output files are viewed as two stacks of lines, the 'forward' movement of the window entails 'pushing' the head of the list onto the output stack, and then making the old head the new tail of the list, and 'popping' a new last line from the input stack. 'Backward' movement of the window is accomplished similarly. Inserting new blank lines, or deleting lines from the window also involve similar push and pop operations on the two stacks. A push operation also involves compacting the line to it's minimal size, and a pop also implies an expansion of the line to a multiple of a constant number of bytes.
This structure is further modified by refining the input and output line stacks. The stack can be regarded as a sequence of buffers, with the top element of each stack being a partially filled buffer. The push operation can be defined by:
- Check for room in buffer for line
- If no room then
  Close buffer
  Open new buffer
- Store line in buffer

The set of closed buffers on the two stacks need not reside in core. Either all closed buffers can reside on disk, or these buffers can be managed by a paging simulator, similar to that described above. If core space is a critical factor, then the first choice is forced, but if there is enough physical core space, the second option will perform better for local movements of the edit window. To allow large movements of the edit window, instead of the window 'moving' through the text, it is faster to close the window (by pushing all window lines into a buffer), locate the buffer containing the new start line of the window, and then re-open the window at that point.

There is one major criticism of this data structure, which relates to the lack of uniformity in internal representation of the edit text. There is an expense associated with each push and pop from the stacks. Thus the strategy will perform very badly when performing a locate operation, or any other command requiring a scan through the text in either direction. Before each line is scanned, the window must be moved (by pushing a line onto the stack and popping the next line into the window), requiring an unnecessary overhead. Certainly strategies can be developed to circumvent the problem, but this still remains a rather expensive approach.

4.4.1.4 A Multiple Buffer Gap Strategy

Rather than attempting to mold the line and buffer gap approaches together, it is more efficient to use a variant of the buffer gap approach. Each unit of paging is a buffer with its own gap. This structure can be used in lines that can be described as...
specified as a doubly linked list of buffer descriptors, with each actual buffer residing either in core space or in the swap file. In Pascal this data structure is:

```pascal
BufferPointer = ^BufferDescriptor;
BufferRecord = Packed Record
  NextBuffer, PreviousBuffer : BufferPointer;
  BufferEnd, GapStart, GapEnd,
  Chars : Word;
  Modified : Boolean;
  DiskBlock : 0..Maxblock;
  CorePage : ^CoreRecord;
  End;
```

- `NextBuffer` and `PreviousBuffer` are used to implement a doubly linked list of these descriptor records.

- `BufferEnd`, `GapStart`, `GapEnd` and `Chars` are used in manipulating the buffer, and refer to the end of the buffer, the extent of the buffer gap, and the number of characters in this buffer.

- `Modified` is a boolean flag referring to any modifications which have been made to the buffer since it was swapped in. This saves writing out the buffer when the buffer is swapped out.

- `DiskBlock` is the disk area of the swap file which holds the buffer contents while the buffer is swapped out. This number is unchanged for the life of the buffer.

- If the buffer contents have been swapped into core, `CorePage` will point to the core entry describing the page frame into which the buffer has been placed, and if swapped out this field will be a null pointer.

To describe each of the page frames, an array of core records is used. In Pascal this can be described as:
CoreTable = Array [1..Maxframes] of CoreRecord;
CoreRecord = Packed Record
  Use no : 0..Use max;
  Freeze : Boolean;
  BufferRec : BufferPointer;
  Body : ?Block;
End;

Block = Packed Record
    ControlWords : Array [1..CtrlLength] of Integer;
    Bytes : Packed Array [0..Bufferlength] of Char;
End;

- CoreTable describes each page frame by a CoreRecord.

- The fields of the CoreRecord are:

  Use no, a field which records the time at which the buffer
  was created, or swapped in, and is used to implement the
  least recently used paging algorithm.

  Freeze is set if the buffer is the current buffer,
  inhibiting this buffer from being swapped out.

  BufferRec points to the corresponding BufferRecord if
  this frame is in use.

  Body points to the actual page frame of byte storage.

If a buffer is emptied, the first three fields of the corresponding
CoreRecord are zeroed, the DiskBlock number placed on a list of free
blocks, and the space used by the BufferRecord is garbage collected.

To allocate a new buffer, either a free page slot is located, or a
buffer must be swapped out. The system first attempts to swap out the
oldest non-modified non-frozen buffer (no disk I/O necessary), or failing this,
the oldest non-frozen buffer is swapped out. Space for the BufferRecord is
claimed from the free space list, and the fields initialized.

It is important to note here that these base level algorithms would be implemented in a low-level language, commonly at assembler level, rather than Pascal. As these routines would be called very frequently when executing a user command or updating the screen, the runtime efficiency of these algorithms is an important consideration.

4.1.2 The Editor Command Language

The main command loop of the editor can be written in Pascal as:

```
WHILE NOT terminate DO
  BEGIN
    getcommand(command);
    execute(command);
  END ;
```

Algorithm (4.1)

This loop places as few restrictions on the user interface as possible, and assumes that screen refresh is an interruptable process callable from within the procedure 'getcommand'. The command syntax implemented by this procedure is:

```
[<command prefix>] <command> [<command arguments>]
```

using no introducer or terminator symbols for a command. Most single characters that can be entered from the ascii keyboard are viewed as complete commands. This section will not provide a comprehensive description of all the commands implemented by this editor (the reader is referred to Appendix III of this thesis for such a description), but will highlight the major issues raised by the command language.

4.1.2.1 Printable Character Commands

All the ascii printable characters are considered as edit commands, which cause the character itself to be entered into the edit text at the current edit (or cursor) position, and the edit position then shifted 1 position to the 'right' (or toward the end of the text), while remaining on the current line of text. There are two ways in which such characters may
be entered, and this is reflected in the two modes of operation of the editor (a bi-modal approach was used here to allow better versatility of commands with the minimum of keystrokes). The first mode, the 'Edit Mode', uses overtyping to enter the new character, replacing the character in the cursor position with the entered character. The second mode, 'Insert Mode', logically extends the text (from the cursor position to the end of text position) by 1 character, and then inserts the entered character in the created space in the text.

4.1.2.2 The Escape Key

Of the ascii character set, the remaining 32 characters (excluding the null character from this set) are the control characters and the delete key. Many of these characters are bound to functions which govern movement of the cursor position and the editor attempts to preserve the 'normal' meaning ascribed to such characters. Thus, the Line feed key moves the cursor to the same horizontal position in the following line, the Return key to the first column of the following line and so on. The editor also attempts to preserve the meanings of the character codes generated by the arrow keys of the user terminal, by binding the movement functions to the character codes generated by each of these keys. Certainly it is possible to provide a complete set of editing commands within this set of control characters [Ewina83], but the resultant command set will be unnecessarily restrictive. The method used here is to use one character, (the escape character) to cause a rebinding of the characters, so that a further 127 possible commands can be specified with the second character of this two character command sequence. However, to reduce the likelihood of user errors in specifying commands, this set is reduced by eliminating any case dependency of the second character (Thus the command <ESC> A can also be specified as <ESC> a and so on).

4.1.2.3 Command Prefixing

Prefixing a command is often used to modify the following command by specifying a repeat count for the command. Thus the command
<delete character> will delete the following 12 characters from the text (at the cursor position). This prefix value may be a positive or negative integer value, so that the command -12 <delete character> will delete the preceding 12 characters from the text.

To preserve consistency of commands, the escape character is used to indicate a prefixing of a command, so that complete command using prefixing is, for example, the key sequence: <ESC>-2<page move command> to move the edit position 2 pages toward the start of the text. While entering the numeric value, the backspace and delete keys can be used to edit the number. This additional feature introduces the need for an optional prefix terminator, necessary in the case of specifying a prefix to the backspace command, the delete command, and the set of printable character commands: \{+, -, 0..9, space\}, and optional in the case of a prefix for any other command. Thus to specify 9 backspaces, the command sequence is: <Esc>9<Space><Backspace>.

This prefix command also has options to extend the repeat of the command over a number of lines. Thus the command: <Esc>9*3<Delete> will delete 9 characters on this line, and on the succeeding 2 lines, at the current horizontal cursor position.

Prefix values may also be interpreted in other ways by commands. Thus the command to move the cursor to a particular line number uses the prefix value as this line number, making the command: <Esc>12<goto line> indicate a movement to line 12.

4.1.2.4 Arguments to Commands

Arguments to commands are used to modify the behaviour of commands in a manner specific to the particular command. Unlike the prefix command, where the prefixing is optional for all commands, the use of arguments is necessary for a subset of edit commands. These arguments are commonly text values, interpreted as strings, file names, buffer names and such. All commands which require such arguments will normally
prompt the user for a value, indicating the type of value required, and
default value that may be selected. This default value may be either the
value used in the previous call on this edit function (or a closely related
function), or may be a constant default value. This default may always be
selected by the `<Esc>` key, or a new value is specified by a text string,
terminated with the return key, and can be edited using the backspace and
delete keys). The text value is taken literally, with the exception of control
characters. These may be specified into the text string using the literal
character specifier: `<Ctrl-^>` char

Some examples of using arguments are:

*locate>*<text>*
*Replace text>*<text1>*<text2>*
*Get from file>*<file name>*
*Place in buffer>*<buffer name>*

4.4.2.5 Aborting and Undoing commands

When a command is executed, the user has no direct control over the
editor's actions until command execution has completed, and the editor is
ready to accept the next command. There is, however, an exception to
this, which allows the user to interrupt the current command, aborting
execution of the current command, and returning immediately to user
command input level. This will not attempt to perform a complete restore
of state of the editor to that existing before the aborted command was
executed, but will perform the minimum possible actions to revert back to
the normal command input level. The abort command can be issued at
any time, so that if this command is issued while a command is being
entered, this partial command will be discarded.

As well as being able to abort execution of a command, there is also
a limited ability to 'Undo' a command. This refers to delete text
commands, and the text deleted by the most recent delete command can be
re-inserted back into the text using the `<undo>` command. This is
implemented by placing all deleted text into a delete buffer rather than
discarding this text.
4.4.2.6 Key Bindings

As mentioned in previous sections of this chapter, it is an almost impossible task to define a consistent and complete methodology of mapping single characters, and two character Escape sequences, into the editor command set. Any method chosen, whether based on mnemonics or topology of the keyboard will not define a complete mapping, and the implementor must make a number of arbitrary decisions when performing this task. These decisions can be overridden by the user, by use of an editor parameter file. This file can specify an alternate mapping of characters to edit functions, specifying a complete mapping, or specifying a smaller number of reassignments. In this way the user can specify a command set to suit the style of editing preferred by the user, or to make the editor look, at the command level, similar to an editor which the user is familiar with.

4.4.2.7 Summary

It is not intended to provide a complete command description in this section of the thesis. The reader is referred to Appendix III of this thesis, the User Guide to this editor for such a description of the command language. This section has briefly touched on some of the main issues raised by this command language, in relation to considerations raised in previous sections of this chapter.

4.4.3 The Display Controller

The display controller is responsible for maintaining an accurate picture as possible of the relevant section of the edit text on the user terminal screen at all times during the edit session. It has the additional requirement of performing this task with the minimum possible delay at the terminal. Also, the display process must be integrated into the editor in such a fashion that the user can interrupt the display process with a new command to be executed.
In general, the state of the display will only change by minimal amounts with the execution of each command. Thus the display process must be an incremental process, updating the display on the screen to reflect changes to the edit text, rather than continually redisplaying the complete screen. When considering the main control loop of the editor (as described in Algorithm 4.1), the display process is integrated into the procedure 'getcommand'. The internal details of this procedure can be given in more detail to illustrate this structure in Algorithm 4.2.

PROCEDURE getcommand(VAR command : editcommand);
BEGIN
WHILE NOT commandfinished DO
BEGIN
IF (NOT inputwaiting) AND (NOT displaycomplete) THEN
  display ;
  IF NOT inputwaiting THEN waitforinput ;
  READ(inputchar) ;
  process(inputchar) ;
END ;
END ;

PROCEDURE display ;
BEGIN
WHILE (NOT inputwaiting) AND (NOT displaycomplete) DO
BEGIN
  updatenextitem ;
  IF lastline THEN displaycomplete := TRUE ;
END ;
END ;

... Algorithm (4.2)

Top Level Display Algorithm

Thus the display algorithm constantly checks to see if input is waiting while updating the screen (breaking down the display process into a number of smaller, faster steps), and will abort the display if the user enters a complete command.

A major problem when designing a driver that must perform quite complex operations on a display device, is how to specify these operations in an efficient manner to the device. There is no standardization of terminal hardware as to what operations are possible on a display device, nor any
standardization on the command sequences that must be generated by the editor to invoke these operations.

For this reason any display driver must be designed for a particular terminal type, or contain sections of code which will allow a number of different types of terminal hardware, or use a library of terminal drivers within the editor to map an internal virtual terminal to the user's terminal type. Such approaches tend to be cumbersome, and exact a penalty of generality or efficiency, but there is no clear route around this problem. Here, the internal virtual terminal model has been adopted. The basic set of operations that may be performed on this model are:

GETCHAR(C) input a single character from the terminal. This assumes that the operating system does not echo this character.

PUTCHAR(C) output the single character to the terminal.

MOVE(X,Y) move the cursor on the screen to position (x,y), using direct cursor addressing if possible.

CLEAR This command takes two forms. CLEAR-TO-END-OF-LINE, which erases the display from the cursor position to the end of the line, and CLEAR-TO-END-OF-SCREEN, which erases from the cursor position to the end of the screen.

The display process can also more advanced terminal control calls if these are supported by the terminal:

INSERT(text) Inserts text into the screen at the cursor position, pushing existing text to the right on the current line to accommodate the new text.

DELETE(n) Deletes the next n characters from the current line, moving the remaining text to the left to remove the resultant gap.

INSERTLINES(n) Inserts new (blank) lines into the screen, scrolling the screen
down to accommodate these lines.

DELETELINES(n)

Deletes lines from the screen, scrolling the remainder of the text up.

SCROLLREGION(y1,y2)

Define the 'active' portion of the display to lie between lines y1 and y2. All other text on the screen is frozen.

There are two basic types of redisplay. The first is that the display controller is given a log of the changes to the screen performed by each user command, and performs each of these actions in sequence. This is prone to considerable problems, especially when the display process lags behind the user, and the effects of the most recent command may take some time to be displayed. Thus the user may have to pause continually to allow the display to 'catch up'.

The second approach is to use a copy of the current screen, and the display process scans the edit buffer, applying any changes to the screen display (and the internal screen copy) to reflect the current state of the edit buffer. This has an advantage of being interruptable at any point, and then, when resumed, can continue this update process (even though the edit buffer may have been altered). If the redisplay starts with the line which contains the current edit position, and then updates, in turn, a line above and then below, then this process continues until the screen display is complete, or the user enters a command. This algorithm is relatively simple, and is presented in Algorithm 4.3.

BEGIN

frame : (*frame returns a data structure BUFFER, which is a copy of the edit text, which contains all text which needs to be displayed. This procedure uses four cutting parameters, the top and bottom of the display area, and the left and right columns of the display. It is held as a single text object, with 'newline' characters to denote end of lines *)
(* initialize variables*)
complete = FALSE;
botrow = cursorrow;
toprow = cursorrow;
loc = i;

(* start at cursor row *)
row = cursorrow;
WHILE NOT complete DO
BEGIN
col = 1;
WHILE col <= screenwidth DO
BEGIN
(* process a single line *)
IF buffer[loc] = newline THEN
BEGIN
IF screen[col, row] <> ' ' THEN
BEGIN
(* not at end of text so clear remainder of line *)
move(col, row);
clear-to-end-of-line;
FOR i = col TO screenwidth DO
screen[i, row] = '';
END
END
ELSE
BEGIN
buffer[loc] == screen[row, col] THEN
BEGIN
(* characters disagree - update screen character *)
move(col, row);
putchar(buffer[loc]);
screen[col, row] = buffer[loc];
END
loc = loc + 1;
END
END

(* check if interruption pending *)
IF inputwaiting THEN complete := TRUE

(* select next line to update *)
ELSE IF (row > cursorrow) AND (toprow > 1) THEN
BEGIN
toprow = toprow - 1;
row = toprow;
END
ELSE IF (botrow < screendepth) THEN
BEGIN
botrow = botrow + 1;
row = botrow;
END
ELSE complete := true;
END
A Simple Screen Update Algorithm

The actions of this algorithm are quite straightforward, using a copy of the relevant section of the current edit text to match to a copy of the screen state. The disadvantages of this algorithm are that the algorithm is prone to redisplay large sections of the screen whenever characters or lines are inserted or deleted or if the screen is scrolled over the text by a single line (which may be unnecessary if the terminal hardware supports comparable operations). This is quite problematical if the user is simply scrolling through the text, and will imply unnecessary output traffic overheads on editing.

The solution to this problem lies in using more processor time to calculate a more optimal, and hence significantly shorter, output string which will perform this update operation on a particular terminal.

One method of doing this is to perform a complete pattern matching operation on the current screen display and the relevant edit text subsection, finding a match which has the minimal difference between these two data objects. This difference information is then encoded into a set of inserts, delete, overwrite and clear calls to the terminal, and then the resultant string is then sent to the terminal [Gos81]. This task may become very processor intensive, as the algorithm calls for a selection of a match between the two objects which results in a minimal difference between the two text objects, and the text objects will contain approximately 1920 characters in each.

A modification of this algorithm uses the basic structure of the Algorithm 4.4, where each line is updated as a single entity. If each line within the screen display is given an identifier, and each line in the edit text within the the current screen span is also assigned this identifier (using a unique identifier for each line). The algorithm first compares the lists of line identifiers, and when a mismatch occurs, searches for either the result
of a line deletion (where, for example, the screen line identifier sequence "1, 2, 3, 4" is matched against the edit text sequence "1, 4") or a line insertion (where, for example, the screen sequence "1, 2, 3, 4" is matched against the edit text sequence "1, 2, 11, 12, 3, 4"). The display process can then use a sequence of insert and delete line calls to restore a complete line match. The display process then examines the contents of each line, again attempting to find a match of characters which presents the minimal difference between the two text objects. This process is now somewhat more efficient, as the text objects now contain 80 characters, rather than 1920. This difference can then be translated into a sequence of insert and delete character operations, combined with direct overwriting of text on the screen.

This line comparison can be tuned to the specific characteristics of the system, using more processor time to find the optimal match of the existing screen display to the new line contents, or using a faster algorithm which may result in less efficient use of the terminal. A solution which performs well under most conditions is given (at the top level of refinement) in Algorithm 4.4.

- By a string comparison find the start string of the two lines which are common. Thus for lines "this is" and "this new is" this start string is "this "
- If this string is the complete line then no further work need be done, and the procedure can exit at this point.
- By a string comparison from the end of the line find the common end string. Thus, in the above example, this will match the string "is"
- If the common end string is of zero length then clear the screen line from the end of the common start string, write any remaining text from the buffer line and exit at this point.
- If the buffer line is longer than the screen line, then insert spaces into the screen (at the end of the common start string), and rewrite the portion of the line lying between the common start and end strings. In the example case this will insert the text "new " into the display after "this ".


5. The Pascal Language Environment

- If the buffer line is shorter than the screen line, then delete text from the screen (at the end of the common start string), and rewrite the portion of the screen lying between the two common strings.

Algorithm (4.4)

Top-Level Line Display Algorithm

This algorithm has been implemented in the display driver in the editor implementation.

There are a number of other details that are relevant here to the display algorithm. Tab characters in the file are expanded to the equivalent number of spaces on display (this is one area of inconsistency here, where the display does not mirror the text being edited: there is, however, an edit parameter which causes tab characters to be displayed as a Ctrl-l). Other control characters (except the line break sequence (Ctrl-M Ctrl-J)) are displayed as printable characters in reverse video on the screen (thus Ctrl-K would be displayed as a reverse video "K").

The last section will address the area of dynamic debugging of a program, and will present an implementation of a better user interface to this editing tool, with a user interface which is consistent with the remainder of tools presented in this thesis.

6.2 Errors, Compilation and Editing

Compilation of Pascal programs is very commonly based on the structure of the Pascal 78 Compiler [And78]. This is a single pass compiler so that the process of parsing the text, checking syntax, determining the syntactic structure of the program, and the semantic structure of the program (illegal assignment, sequence errors and so on) is performed together with code generation for the object program. It is this compilation process
5. The Pascal Language Environment

5.1 Introduction

This chapter will examine a number of software tools which relate specifically to the Pascal language environment. It must be stressed here that it is not the aim to design and implement another Pascal compiler, but rather to explore the possibilities of grafting a different user environment onto an existing Pascal compiler and runtime support system.

Section 5.2 addresses the interface between the editor and the compiler, and will present a method of binding the parsing phase of the compiler to a text editor. The result of this is a user edit environment where parse information generated by a compilation may be gracefully merged with the edit text object.

Section 5.3 will extend this introduction of language considerations into the edit environment, addressing the design and implementation of an editor which allows the user to edit the syntactic structure of the text, and will explore the implications of such an editing tool.

The last section will address the area of dynamic debugging of a program, and will present an implementation of a better user interface to this debug tool, with a user interface which is consistent with the remainder of tools presented in this thesis.

5.2 Errors, Compilation and Editing

Compilation of Pascal programs is very commonly based on the structure of the Pascal P4 Compiler [AmN176]. This is a single pass compiler so that the process of parsing the text, to detect errors in both the syntactic structure of the program and the semantic structure of the program (illegal assignment, subrange errors and such), is performed together with code generation for the object program. It is this compilation process
which is inevitably the final arbiter of syntactic errors in the source code.

There is much potential to be gained by the ability to correct such errors in a consistent fashion, and, further, to edit the program text, within the compilation environment, and this is a great virtue of languages such as Basic, for all it's other shortcomings.

To realize such a goal of a compiler, bound to some processor which can edit errors in the source code, implies significant alterations to the structure of the compiler. In a single pass, recursive descent compiler, simple immediate errors, such as missing delimiters, and misspelled identifier names, can be corrected by the user at the point of detection within this compilation environment, and, with a small amount of internal juggling, the compiler may be resumed at the original error point. However, for more complex, or insidious errors, where the compiler can only detect the presence of an error condition at some stage after the actual program error, the problem becomes more complex. Firstly, the user would need to have access to more information than a simple diagnostic message to locate the actual error location, and secondly, resumption of the compilation would imply an ability to 'roll-back' the compiler to resume at some previous location within the text.

Although the solution does exist to allow the language to be compiled in separate modules, as, for example, in ADA, this is not a valid route when compiling Pascal text, as the facilities for external declaration commonly refer only to procedures and functions, not to complete static environments. Thus, this section will continue to consider 'monolithic' compilation of Pascal text.

To allow the user some vehicle of interaction with the compiler has considerable benefit in the process of program development. Providing access to the identifier information, type information, and syntactic structure of the program allows the user to compare his preconceptions of the program and language with that of the compiler. This is an excellent
environment for correcting such errors in a program. Certainly errors can be corrected more effectively by access to detailed information concerning the cause of the error, and is particularly useful when a deeper misconception about the user's view of the language is involved.

When addressing the problem of access to information generated by the compiler, and the ability to "roll-back" a compilation to a specified point, the single pass compiler is hopelessly inadequate as a basis for such an extension. A recursive descent parser generates code statement by statement. Once a statement has been compiled, the chain of recursive exits implicitly throws away the syntactic information provided by the parse. More importantly the syntactic structure of the program at any point is not held in an accessible data structure, but in the execution state of the compiler itself, and, particularly, in the stack of procedure calls at that point in the compilation. Thus, although the compiler can create, as either a file or internal data structure, the complete mapping of program text into syntactic structure, such information is not available during the actual compilation.

Given such restraints on the compiler operation, the alternatives here to provide a better access to the compilation environment are either to implement a compiler whose mode of operation is radically different from the P4 template, or to modify an existing compiler to provide more information which can be subsequently used by an editor. The latter alternative will be examined here, as the practice of producing a number of compilers for the same language, each to achieve a particular aim is one which is counter productive. The problems of such an approach, where each compiler may contain small, but highly critical, differences in language semantics far outweigh any advantages gained.

The role of a compiler is twofold:

1. To parse the program and inform the user of the presence of any syntactic errors, and (limited) information as to the
probable position and cause of each error.

2. To produce, in object code, an equivalent version of the program text.

It is the first role which is of interest here, as it is this information which needs to be available during a subsequent edit session.

This section will address alternatives to the most common method of displaying compilation errors to the user: on the terminal as the compiler detects such errors, and within a listing file, displaying the line in which the error condition was generated, and the position within the line, together with a diagnostic message on the cause of the error. These methods are not adequate for many users as either one has to make a record such errors as they are displayed on the screen, or print the list file, and use this file when editing the text to remove such errors.

The UCSD system [Over80] offers an improvement on this method by, at each compilation error, allowing the user the option to edit the error, and if taken, the editor is automatically invoked, the error diagnostic displayed, and the editor position is set at the error point. The major disadvantage is that this can only be performed on a single error. When entering the editor, the positions of any previous errors in the text, and their diagnostics, are inaccessible.

A better method is the ability to invoke the editor with some form of internal option set. This option would cause the editor to effectively 'merge' the relevant output from the compilation with the text object. The aim of such a merged text is to allow the user access to this compilation information within the edit environment. As a result of this the user can position the display at the point of a compilation error, and then request the diagnostic of the compiler to be displayed. This is shown in Figure 5-1.

Hence, instead of producing a listing file, the compiler produces information concerning the location and type of each error in the program
text. The type information of the error takes a form that allows the editor to reproduce the relevant diagnostic text for that error type when required. Thus, conceptually, the editor then sets up a group of tags in the edit text, with each tag containing the appropriate error number and error type. This is shown in Figure 5-2.

Figure 5-2: The Structure of Error Tags in the Editor

To locate the next compiler error, the editor retrieves the next tag in sequence from the current position. The diagnostic can be reconstructed from the tag's error type. To remove an error, the tag entry is deleted.
from the set of editor tags.

The editor commands which need to be implemented in such a system are:

**Locate Error Tag**

This command will, by default, locate the next error in the program text, moving the display window to surround this error position. The diagnostic and error number will appear at the base of the screen, the editor will set the current error position to this tag position and then will enter normal edit mode. This command may also be used to locate a specific error: by adding a prefix to the command (e.g. `<number>` Locate Error Tag) or a relative error position to the current opened error (e.g. `<signed number>` Locate Error Tag). If a number of 0 is entered the editor will re-position the display to the current error. If no further errors exist the editor will clear the current error display.

**Remove Error Tag**

This command will delete the tag entry for the current error (the absolute numbering of errors will remain unchanged however). The diagnostic for this error will be deleted from the screen, and the editor will revert to normal edit mode.

To achieve such a user level tool implies some modifications to both the compiler and the editor. The structure of such changes to both programs are outlined in the following sections.

### 5.2.1 Compiler Modifications

The modifications to the compiler to achieve this aim are neither extensive, nor do they change in any way the compiler defined language
semantics. These alterations have been made to the DECSystem-10 Pascal [KNH83], and take the following form:

1. An additional output file is created by the compilation. This file essentially an error file, contains a list of error positions within the text object, and, for each error, a code indicating the type of error encountered. This file may be defined using a Pascal data descriptor:

```pascal
errorfile = FILE OF RECORD
    linenumber, colnumber, diagnostic : INTEGER;
END;
```

2. As the compiler parses each line, it already stores the position (as a column number) of each error encountered, together with the number of the diagnostic message. The compiler also holds the line number of the current line as an integer variable. When a new line is requested, the current line is ejected to a list file, together with any errors on that line, giving the position and diagnostic of the error. The modification simply entails the writing of an additional record to the error file for each error.

3. The listing file is not created by default, and the terminal will not, by default, display the lines containing errors, and the error diagnostics. Also all code generation is modified such that code generation may be inhibited (the default action is to generate such code). All these actions are reset from the defaults by compiler call options.

A second file is also maintained as a component of the Pascal system. This is a file of records where each record contains the diagnostic code and the text of that diagnostic. This is an implementation device to reduce the size of the compiler generated error file, and to reduce the editor overheads in storing diagnostics within the editor.
5.2.2 Editor Modifications

The editor described in Chapter 4 has been used to implement the editor component of this environment. The features of the editor that are used to implement this facility are as follows:

1. This editor already supports the concept of a 'tag'. A tag is conceptually a marker that the user can insert into the text at any location, and can, at a later stage, move the edit position back to this tag position. To allow a number of tags within a single text object, each tag is identified by a user-defined numeric label. These tags are used to implement the merge of the compiler error information with the edit text. This is achieved by extending these tags to include an additional field, which will be used to store the diagnostic code of the error detected at that position. The error file is read with the text file, and is mapped into the tag structure, using the error number as the identifier of each mark. The editor also opens the diagnostic text file in read only mode, and uses random positioning to speed access of the required diagnostic text for an error code.

2. Two commands are used to implement the locate error tag and remove error tag commands, <Esc> Ctrl-L and <Esc> Ctrl-R respectively. The locate error tag command may be prefixed by a signed number for relative positioning, and an unsigned number for absolute error positioning.

3. To provide a tighter binding between the compiler and editor, the editor may invoke the compiler directly by a Parse command (Esc) Ctrl-P). This command performs a context save of the editor state, and directly calls the compiler to compile the edit text. In this case the editor invokes a set of compiler options that includes suppressing normal code
generation (the rationale for this decision is that the compiler is being used only as a parsing tool, and code generation is an unnecessary, and somewhat expensive side-effect of the parse operation). The editor also uses the operating system to call the editor once the compilation is complete, (this is a Digital TOPS-10 facility), and the editor, once invoked performs a state restore. As the compiler performs no output to the display, the editor does not redraw the display on reentry, but displays a status message on the base line of the screen, giving the number of errors.

5.2.3 The Resultant Editor Environment

The result of such a call structure is that the user view of this process is that of the editor performing the compilation as a background process, and furthermore, the user job never leaves the editor environment. However the editor 'freezes' the screen and keyboard while the parse is active. The 'freezing' of the keyboard is not quite accurate, as the user can abort the compilation, but in such a case the editor will still be resumed by the operating system. In this situation, the status message on resumption will show that the parse was aborted. The resultant user view of compilation is shown in Figure 5-3. Again, this interception of the abort command to force the editor to resume execution is an operating system construct.

![Diagram](image)

**Figure 5-3: Implementation View of Editor / Compiler Interface**
From the user view, the parse command is a means of introducing additional information on the syntactic structure and correctness of the program into the editor environment, while still maintaining a view of the program, and the means of manipulating that program, which is still essentially textual rather than syntax-based. These modifications do not alter the conceptual view of the structure of the text object, but allow the editor to 'import' additional information relating to the equivalent syntactic structure of the text, and in particular, in relation to errors in this equivalent structure.

5.3 Editing Pascal Text

The preceding section outlined the necessary steps to make the compiler a 'tool' of the editor, enabling a tighter binding of the compiler and editor to create a more productive program development environment. This section will examine the implications of changing the conceptual view of the editor, to that of an editor which can edit a structured object, where the structure is based on the syntactic structure of the programming language. The aim is to implement an editor environment based on the programming language in which the textual object is written.

The use of such an editor, suitable only in the limited environment of program development holds much potential in improving the efficiency of program development, and, by encouraging a 'top-down' style of program development can improve the quality of such programs by the tailored edit environment.

In tight connection with this are the principles of structured program development. The editor should be able to create a developmental environment where the structure of the computation (can) be reflected in the structure of the program [DaDI72]. The means of ensuring this kind of textual structuring in a program is by a series of refinements on the original, top level algorithm. In such a fashion a large program can be
constructed in an orderly manner, making one refinement decision at a time, while still maintaining a manageable conceptual size of the program, analogous to a hierarchical structuring of abstract machines, where each level of the structure can be examined for correctness independently of the considerations of the upper and lower levels [Wirt71].

When relating this form of program structure to a general purpose text editor, obvious weaknesses are present in the editor that obscure the program structure. It seems reasonable that the user should be able to view a program as a succession of refinements, rather than a stream of characters, and that each level may be viewed, and tested within the context of higher levels of refinement, but without the necessity of definition of lower levels of refinement [Wat71]. Thus the formidable task of debugging and validating a single, large program entity can be circumvented by an editing tool which reflects this organization of the program.

This section will examine an implementation of an editor which offers such support. On a general level this implies that the editor can place sufficient structure on a piece of program text such that the code corresponding to that particular level of refinement can be edited within the context of the outer levels of the program. Thus the text of the program at each level can be displayed and manipulated without affecting outer levels of the program structure. In this fashion the editor implements an environment where hierarchical program structure is a major consideration.

This implies more than simple automatic formatting of program text. In order for the editor to create such an interactive environment, the editor must have some 'understanding' of the program language, in terms of the language syntax, so that the structure of the object being edited, and the range and style of commands are guided by the language itself. Thus the editor needs, as an integral component of the editor, a language parser, capable of parsing both program fragments as well as complete programs.

There is still considerable argument over the most appropriate method
of editing text which contains obvious structural overtones, such as program
text. Structure editors are seen as being too restrictive and also profligate
consumers of computer resources, and the same task can be accomplished at
lower cost with many of the features of a 'structure' editor in a 'normal'
editing environment [Wood81]. The reverse side of this argument is that
the structure editor "saves typing" [TeiR81], producing much of the formal
text of program language syntax with fewer keystrokes, "eliminating the
mundane" [TeiR81]. A structure editor also de-emphasises the simple
textual considerations of editing programs, and places a much heavier
emphasis on the structure of the resultant code, producing a user
environment which is tuned towards the considerations of top-down
algorithm design.

Certainly editors using a dual approach, where both text and tree
structure are interchangeable views of the text object [Wate82, Cohe82], are
heavier consumers of computer resources than either of the two single
approaches. However many of the recent structure editors do use some
elements of the textual approach (the Cornell Program Synthesiser [TeiR81]
uses text editing to enter expressions for example), but there is still
considerable interest in producing an efficient environment which preserves a
complete structural environment.

The Interlisp editor [Sand78], used to manipulate the textual
representation of LISP lists is an example of such directions in program text
editing. LISP lists are compound structures where each element of the list
may be either an atomic symbol, or a list itself. There are two types of
movement through the list : 'vertical' movement, where a list at the
current level can be opened for editing, or the current level is closed, and
the upper level list is opened and 'horizontal' movement, where the current
component window of the editor is moved within the current list. The
object within the edit window (either a list or an atomic symbol) can be
deleted, or replaced by a LISP expression. The display is a display based
on this two dimensional structure, and uses two parameters, depth and
width: depth indicates to what level list components will be displayed in full, and the width is the number of list components (including and succeeding the current window object) to be displayed. The major advantage of this editor, situated within this interpretive environment, is that LISP expressions can be evaluated within the editor. Thus dubious expressions can be evaluated, and altered as necessary, without leaving the editor environment. Such a powerful tool is a side-effect of the LISP abstract machine, where the evaluation function and the edit function are simply LISP functions themselves, and mutual calls are possible.

The other major aspect of this editor is the structuring of the editor, both in terms of the edit display and the edit commands, are based on the structure of LISP lists. This provides a view of the LISP object in both textual detail and LISP structure.

The use of an interactive editor within an interpreted language environment is an efficient and necessary attribute of an interpreted language abstract machine. It allows efficient alteration of the runtime environment, where the user can use the interpreter and editor in close correspondence. This provides a quite complete program development tool.

This compelling concept is difficult to realize when using a normally compiled language. Although it is possible to construct systems which allow alteration of program text within the runtime environment, this implies a significant complexity of both the compiler and the runtime environment [Brow76]. The alternative is to create an editing environment which utilizes as much of the language environment as possible.

5.3.1 The User View of a Pascal Text Editor

In order to define the user level facilities of such an editor an examination of the language structure is necessary, and, more importantly, an examination of the general principles of program creation using a block structured language such as Pascal. The most powerful such principle is
that of stepwise refinement. This method stresses the gradual development of a program by the progressive refinement of program and data structures in parallel. Here programming decisions are by the progressive decomposition of programming decisions into a set of sub-goals, applicable at each level of program refinement. In this fashion the program is created not sequentially, but in a style which reflects the steps taken in refining the initial algorithm. This reflects in a final program whose overall structure is conceptually simple, and directed towards the program goal.

In terms of editing this implies that the user must be able to define the code corresponding to initial levels of program refinement, and use the editor to 'complete' the program with respect to closure of structure of such a partial program, so that the intermediate level may be compiled, and, as far as possible, tested. With the Pascal language this requirement falls naturally within the structure of the language, where outer blocks match the initial levels of program refinement, and inner blocks correspond to successive levels of program refinement. In practice, this requires that subprograms can be specified only by their headings, with an empty body.

When viewing the program text, the editor should be capable of masking out both the inner and outer levels of program structure, allowing a view of a single level of program structure. In this fashion the environment created by the editor is one which stresses, and is based on these successive levels of program refinement.

The most useful structure to represent such syntax structure is a tree, where each interior node corresponds to a sub-program or compound statement, and each terminal node corresponds to a simple statement. This nodes of such a tree do need a finer level of structure corresponding to expressions, and also an additional structure corresponding to declarations within each sub-program. The tree allows natural movement of the display within the text object, where movement of the current statement window corresponds to opening the next node in sequence from the current node (with a common parent node), and movement to outer or inner levels
corresponds to opening up either the parent node or a sub node of the current node. When defining a level of program structure, the user need only define the positions where further refinement of the program is necessary, and the editor can create a corresponding node in that position. When displaying code to the user, parameters for width and depth of the display are very useful here.

As this tree structure is direct representation the syntactic structure of the text, the result of such a structure is that the editor attempts to maintain a text object which is syntactically correct during the edit session. Thus commands which manipulate text within statements can be guided by the syntax of the program language, and the edit commands allow clearly defined operations on this parse tree. This creates an editing environment which mirrors the strategy of hierarchical program refinement and the actual structure of the language, in the style of editing text based on the notion of a generative grammar of the text object as presented in Chapter 4.

The Cornell Program Synthesiser [TieR81] is worth a brief examination at this point. This is a program text editor which has not only encompassed the syntactic issues of the program language, but has also included an interpreter within this environment, so that the language system has been imported almost completely into the editor environment. The basic data unit of this editor is the template, which has a direct correspondence to a non-terminal entity of the syntactic definition of the language. Each template is a pre-defined, pre-formatted pattern, holding textual attributes, output directives, interpreter directives, and placeholders. A placeholder identifies non-terminal objects within the corresponding syntax production rule for the non-terminal symbol under consideration, having as attributes the the allowable types of templates which can be substituted for this placeholder, and the position of this insertion within the 'owning' template. Thus the basic editing structure can be viewed as an enhanced syntax tree, where each node of the tree contains both textual and syntactic
information for the system.

When editing a program object a full screen display is utilized, expanding each template into its formatted textual form (using the internal output directives). The edit position, indicated by the cursor position within the display, can only be positioned at the start of a template, or at the position of an unelaborated placeholder. Hence movement of the edit position within this display can be seen as movement within the corresponding syntax tree, and the editor has analogous commands: 'Left' and 'Right' correspond to infix traversal of the tree, and 'Up' and 'Down' move the current edit position to nodes at the same depth of the tree, with the added proviso that such movement will preserve the same parent node. 'Diagonal' movement is movement to the current parent node.

Editing of the program is based on the operations of insertion, deletion and cut and paste. Insertion can either expand a placeholder by inserting a new template at that position, or, if appropriate, can insert an expression at the placeholder by entering the text of the expression. Template insertion can only be performed if the type of template matches one of the set of template types defined by the placeholder. Text insertion only refers to expressions, and is parsed only when the text has been fully entered. The parse is performed for internal syntactic correctness, and also that the type of the resultant expression is compatible with that defined by the placeholder. Any errors detected in the parse are flagged, and the text can be edited at a later stage in the edit session. Deletion can affect only complete text objects or templates, and the template or text is removed and placed into a delete buffer (this delete buffer can be access to implement the 'undelete facility'). Cut and Paste allows the user to store a template in one of a number of template buffers, possibly removing the template from the program. This buffer can be reinserted into the text at a different point in the program. Selection of a template for deletion or cut operation implies selection of both the template contents at this level, and also any templates defined within this template.
5.3.2 A Case Study - Implementation of a Pascal Editor

5.3.2.1 Introduction

This section will examine the major issues of the design and implementation of a Pascal structure editor. This section will not cover many of the aspects of editing that have been covered in the previous chapter, particularly in relation to the choice of edit command style, and the screen display algorithms. The major changes in structure editor as opposed to a general purpose text editor lie in the internal data structure used to represent the object being edited, and the radically different command semantics that are necessary in such an editor.

Considering the major issues involved in editing Pascal programs, there are three major aspects which deserve attention in the implementation:

1. The management (and means of user manipulation) of the structure of statements within the program, both within the main body of the program, and the body of any declared procedures and functions.

2. The management of the resultant symbolic information relating to user defined identifiers, data types (both pre-defined and user defined data types), and the context of use of such identifiers (particularly relating to assignment statements, procedure calls and expressions).

3. The ability of the system to maintain an accurate parse level description of the program, particularly in relation to any errors of syntax within the program, and the ability to provide a diagnosis of such errors.

5.3.2.2 Statements and Templates

The body of a Pascal program (or subprogram) can be viewed as a tree structure, where each node of the tree corresponds to a statement within the program, and subnodes correspond to 'inner' statements of the
parent node. The main 'BEGIN - END' compound statement can be regarded as the 'root' node of the tree, and all the top level statements of the tree can be regarded as sub-nodes of this parent node.

This concept is mirrored within the editor by the use of templates. Templates are hierarchically structured objects, where the outer level of structuring corresponds to broad types of objects within the program, and successive levels provide a more specific instantiation of the object, and the lowest levels provide a complete object description. This section addresses the statement class of templates. This group of templates can be represented by the structure as shown in Figure 5-1.

```
STATEMENT CLASS

ASSIGNMENT PROC COMPOUND IF CASE FOR WITH GOTO
STMT STMT STMT STMT STMT STMT

WHILE REPEAT
STMT STMT
```

**Figure 5-1: Statement Template Structure**

Each template contains a number of attributes, corresponding to the type of object being represented. These attributes are the fixed text of the object, the type and position within the text of subtrees (or templates) that this template can 'own', the type and position of optional subtrees within this template, display information used when displaying this template (or generating a textual equivalent of the complete structure) and the type of the template.

For example, the STATEMENT CLASS template contains an optional
marker to indicate a label template position, and a marker to indicate the substructure:

```
{(le)<comment>(nl)}{<Label> }{(lm)<statement> (nl)
```

Here the braces indicate an optional component of a comment preceeding the statement, an optional label (with a fixed text component of a ':'), followed by the template information from the template subclass. Encoded formatting information is enclosed in brackets, and the '(le)' symbol indicates that the comment begins on the left edge of the screen, the '(nl)' symbol indicates a new line, and the '(lm)' symbol indicates the current left margin (indicating that the label should, if possible, lie to the left of the current left margin).

An **IF STMT** template has the following structure:

```
IF <boolean expr> THEN (sm+3)(nl)<statement> 
{ELSE (sm+3)(ln)<statement>(sm-3)}
```

Indicating a fixed text component of 'IF ' and 'THEN ', insertion points for a boolean expression and statements, and the option of an ELSE clause within this statement. When instantiated, the `<statement>` and `<boolean expr>` entities are replaced by pointers to corresponding templates.

Thus the main body of the program is held within a single template, corresponding to a PROGRAM template.

Edit movement within this statement structure is by the arrow keys on the terminal. The UP and DOWN keys move within the same level of a subtree, moving the current template pointer across nodes of the tree with a common parent. On the screen (where the generated text of the template is displayed) this corresponds to vertical movement of the edit cursor, to the start of statements at the same syntactic level of the program. The LEFT and RIGHT keys open up the template, and, if the internal template positions have been instantiated, moves the current template position to the 'owned' template. If this is a position marker, the current template is positioned at this marker position. Thus the LEFT and RIGHT keys
perform a complete traversal of the tree structure of templates, while the
UP and DOWN keys perform movement only at the current depth of a
subtree. Other movements commands are: a diagonal move, moving to the
parent template of the current template, and a home move, to the root
node of the program tree structure.

If the edit position is at a statement template insertion position, then
the user can specify that a particular type of statement template can be
used to instantiate that position. Thus an Insert IF STMT command can
instantiate a <statement> template, and so on.

Statements may also be deleted by deleting a template. In this case,
the complete subtree owned by the statement template is also deleted from
the file. All such deletions are placed into a 'delete' buffer, and the most
recent deletion may be re-inserted into the structure as an alternate form of
instantiating a <statement> template. The parent template of the deleted
section replaces a pointer to the subtree with a template insertion descriptor
field again, and the screen display is updated to reflect this deletion. This
operation is generalized to allow a subtree to be copied (or copied and
deleted) into a named buffer, for later re-insertion. Such buffers are
regarded as templates, possessing the template type corresponding to the
type of statement at the root of the removed subtree. These buffers can
also be used to provide a library of program fragments for the programmer,
as the re-insertion of a buffer is, like a template insertion, a copying
process, which leaves the buffer unchanged.

The 'display options' command opens the optional component of the
current template, displaying this on the screen. The user can then move to
within the optional component of the template, and instantiate the template
descriptors within. A complementary command, 'hide options' will close the
optional component of a template, but only when none of the template
descriptors have instantiated to pointers to subtrees. A special case of this
command is within the COMPOUND, REPEAT and CASE STMT
templates. As an example, the compound statement template has the
internal structure:

\[
\text{BEGIN}(\text{n}l)\langle\text{statement}\rangle[(\text{un}), (\text{n}l)\langle\text{statement}\rangle]\text{END}
\]

(The 'un' is an 'undo' display command, negating the newline directive at the end of the previous \langle\text{statement}\rangle template)

Here, the 'display option' will include the optional component into the body of the template, but will also leave the section enclosed by the square brackets within the template, for later inclusion in a similar fashion. to result in the altered template:

\[
\text{BEGIN}(\text{n}l)\langle\text{statement}\rangle(\text{un}), (\text{n}l)\langle\text{statement}\rangle[(\text{un}), (\text{n}l)\langle\text{statement}\rangle]\text{END}
\]

In summary, the editor provides a small, but sufficient set of user commands to manipulate these statement templates:

- Movement of the current edit component (Up, Down, Left, Right, Diagonal, Home).
- Expansion of a template placeholder into a template itself by the Expand \langle\text{template type}\rangle command.
- Deletion of a template (and it's associated subtree structure) by the Delete command, and a similar Copy to Buffer command for a template.
- Retrieving the contents of a buffer (or the delete buffer) by the Get from Buffer command.
- The Display Options and Hide Options commands.

5.3.2.3 Expressions and Parameter Lists

Expressions, and associated actual parameter lists for procedure and function calls, are not handled using the template structure as presented for statements. Expressions can assume quite intricate levels of complexity, for
which a template structure would be inappropriate at the user level. For these reasons, expressions are instantiated by the user entering the text of the expression, which is then entered into the template as text. The expression is then parsed, and, if the parse encounters no errors, the resultant parse subtree is placed into the program structure. If errors are encountered, the text is marked, and displayed accordingly. The user may, at any later stage, re-open this text for further character-based editing to remove such parse errors. At this stage, the diagnostics of the erroneous parse are also displayed. Such parsing is completed in an environment where the variables used within the expression may not have been declared at this point. Such use of undeclared variables is noted as an error at this point.

Successfully parsed text can also be reopened for editing, which causes a text object to be generated from the expression tree, and this text is then edited, and, later, re-parsed.

Such a 'text mode' of the editor uses a split screen approach, using a 5 line text field, a five line parse error field, and the lower half of the screen used to display the normal program text display, centered around the position in the text where the expression will be placed (See Figure 5-6 for a representation of the screen display in such a situation). The cursor is positioned within the text field, and this upper section of the screen becomes a display for a simple screen text editor (allowing cursor movement, character insert and delete, overtype and erasing). Once the user issues the exit command, the text field contents are then parsed. The positions of parse errors are indicated in the text field in reverse video, and diagnostics are indicated in the parse error field. At this point, the user may continue editing this text, or may leave the text editor mode, which extends the display in the lower half of the screen to become the complete display again and inserting the text at the expression position. If an existing expression is re-opened for editing, the text is initially parsed, and the screen set up accordingly. In this way, the editor attempts to supply as much
information as possible to the user when entering expression texts, including
the context of the expression within the program, and any parse errors
within the expression.

5.3.2.4 Variables, Constants, Labels and Data Types

As can be seen from the above description, the only restriction placed
on the user when defining a program is that the style of definition is one of
a top down approach, where inner levels of code are defined by elaborating
non-textual template items of outer levels of code. To maintain this style
of the program development environment through to the area of identifier
and data structure definition, identifiers need not be defined before their
actual use within expressions, assignment statements or procedure calls.

Thus, for each identifier in a program, the editor needs to maintain
somewhat more associated information about the identifier than a compiler,
as the use of the identifier may precede declaration of an identifier. For
each environment in the program (procedure, function or main program
body) the editor uses an associated list of identifiers, where each entry
(corresponding to each identifier) contains a list of pointers to the positions
in the text where the identifier has been accessed, and also a pointer to the
declaration of the identifier in the text (this declaration may be within the
current environment, or within enclosing environments: the editor uses a
system of locality where changes to the identifier declaration structure have
minimal impact on these list of identifiers).

When an identifier is used within the program, the local environment
list is consulted, and, if an entry for this identifier is located, an additional
'use' item is added to the associated usage list for that identifier.

Entering a declaration for a previously undeclared identifier will cause
a re-parsing of all expressions which access this identifier (as determined by
the complete usage list of this environment (and all inner environments). If
such expression can then be successfully parsed with the additional identifier
information, then the expression is then stored as the equivalent parse tree,
and the expression parse error information is removed. Note that declaration of a variable in this context is taken as a complete definition. If the type of a variable depends on an, as yet undefined, type name, then the variable definition is incomplete, and no expression parsing will be attempted.

The display editor uses a terminal driver which assumes all of the screen is not used.

The internal storage structures are modelled on those of the Pascal compiler [KNH83], and will not be described here. The only major addition here is the extensive internal cross reference of identifiers to enable the editor to correctly identify all the implications of entering a declaration of an identifier, or altering an existing declarations.

To declare identifiers (non-procedure and non-function identifiers), the editor creates a display similar to that used in creating and editing expressions, using a split screen and a character-based editor to enter and edit identifier declarations. The difference here is that the text area of the screen is extended to 16 lines (as Pascal declarations may extend over a larger area than 5 formatted lines of text, particularly in the case of record declarations), and the base section of the screen (used to display the context of the expression on expression editing) displays only the name of the environment where the declaration is being made, so is reduced to a single line field.

There are three additional commands that refer specifically to user-defined identifiers in a Pascal program:

- Display the declaration text for a specified identifier, which will, if the identifier has been declared, display the text of the declaration, allowing the user to consult the declared identifier list while editing statements, expressions, and other identifier declarations.

- List all identifiers which have been used within the program, but have no current declaration, i.e. list all undefined identifiers.
- Move the edit position to the next (or previous) construct which access a specified identifier (analogous to a 'locate' command in more general text editors).

5.3.2.5 The Display Driver

The display driver uses a terminal driver which assumes all of the basic terminal operations as described in Section 4.4.3. The display driver displays a template by interpreting all format directives embedded in each template, displaying all the fixed text components of the template, and displaying the remainder of the template according to a number of display rules:

1. All expressions are displayed as text, using reverse video if the expression has not been successfully parsed, normal video otherwise.

2. All uninstantiated placeholders in a template are displayed in reverse video, displaying the text of the placeholder type as text.

3. All optional components of a template are only displayed in response to a specific user 'display optional' command.

4. Comments within a template are not displayed only if the user has given a 'hide comment' command.

5. Instantiated placeholders are only displayed to a single depth.

A further inner level is displayed as a single line: '...' at the current indentation on the screen.

These rules are best shown by an example of the display during editing, as shown in Figure 5-5.

As indicated in the preceding text, the display is changed if the user is editing an expression, and the screen display is then set up as a multi-windowed screen. Again, this is best shown by example, and the
PROCEDURE DISPLAYLEVEL 1

frame := false;
botrow := cursorrow;
toprow := cursorrow;
loc := 1;
row := cursorrow;
WHILE not complete DO
    BEGIN
        col := 1;
        WHILE col <= screenwidth DO
            IF inputwaiting THEN
                ELSE IF (row > cursorrow) AND (toprow > 1) THEN
                    ELSE IF (botrow < screeendepth) THEN
                        ELSE
                            END

The cursor is at the start of the fifth line, and all items which would be displayed in reverse video are indicated here in **bold face**.

The program fragment is from Algorithm 4.3.

**Figure 5-5: Pascal Editor Screen Display**

The aspects of setting up such a display, and methods of efficient update of this display have been examined in Chapter 4 of this thesis, and therefore will not be covered here.
The cursor is at the start of the second line, and all items which would be displayed in reverse video are indicated here in **bold face**.

The program fragment is from Algorithm 4.3.

**Figure 5-6: Pascal Expression Editor Screen Display**

5.3.2.6 Implementation Issues

This system has been implemented in both Pascal, and for the more commonly called routines, assembly language.

The internal structure of the program is held as an internal data object to the editor program. This structure is a dynamic data object, which must expand, or contract in response to user commands. For this reason, the system uses its own NEW procedure to handle the allocation of dynamic variables. This procedure normally attempts to service requests for dynamic storage from the free area of dynamic store. If this fails, then
NEW consults a list of free space which is fragmented within currently allocated space, and attempts to service the request on a first fit algorithm. If this fails, the program requests additional space from the operating system, and services the request from this space. Associated with the modified NEW call is a DISPOSE call, which enters previously allocated space onto the free list.

Buffers, used to hold both subtrees and complete programs, are maintained as files on the system. As this structure contains pointers, the method of reading and writing such buffers also deserves attention here. The tree is written out in by writing out the tree type information first, and then writing out the tree in post order. As a node of the tree is written out, the first word of the node in store is changed into a description of the position within the file where the node is written. As subsequent nodes are written, a pointer field within a node is written out as the value stored within the first word of the referenced object. Once the complete subtree is scanned, the tree is scanned a second time, again in post order, this time explicitly DISPOSEing of each node as it is scanned. Reading in a file opens a temporary copy of the file for random access and updating. As each node is read in, and allocated a store address by the NEW call, the first word of the node in the file is altered to this store address. As subsequent nodes are read in, the value of pointer fields are resolved by accessing the referenced node in the file, and retrieving the new store address. In this way, the contents of buffers can be saved across edit sessions, and the state of the program can also be stored on file using this method.

It is worthwhile noting that all the data area of the program is held in the memory store, with no attempt to introduce any form of editor paging. While this tends to induce significant memory demands while editing quite small Pascal programs, this decision has been taken to dramatically simplify implementation of the editor.
5.4 Debugging Tools

When designing and implementing a program, it is unusual for a programmer to plan ahead for possible errors, and in doing so, plan a debug strategy together with the program. It is more often the case that the user is confronted with a program which, although the program is syntactically correct, produces some erroneous results, or creates a runtime error condition. This creates the arduous task of isolating the conditions which caused the errors, and mapping this information back to the source program, to establish the program conditions at the point of the error, and then inserting source level changes in the code.

For interpreted languages this task is relatively direct, as the runtime environment can be interrogated at a level which is the same as the program language level, and all machine level considerations can be effectively 'hidden' from the user. This is not the case for compiled languages, as a second level has been introduced by the compiler. Thus, if an error has been generated in the compiled code, then the user must establish the cause of the error within the compiled (machine level) code, and then map this information back to the source code level, in order to establish a particular fault in the source code.

The most simple tool available to the user, when attempting to debug a program in a compiled environment, and also the most tedious, is to insert into the source code a series of output statements, to inform the user that:

- Control has passed through this point.

- The values of particular variables at this point in execution coincide (or not) with the user's expectations.

The major advantage of this system is that inquiries of the runtime environment can be phrased within the program environment at the symbolic level, and the resultant output will also be at the symbolic level,
as the inquiry will be compiled at the same time as the program. However this method contains many shortcomings from the user view:

- It is very time consuming both to insert and remove such 'sentinel' statements.
- Each run of the program will not guarantee that the user will be any closer to establishing the exact cause of the error.
- It is not easily capable of isolating errors which occur from incorrect assumptions about the state of the runtime environment of the program.

- The possibility of the user generating an error within such a sentinel also exists, and such an error is remarkably difficult to isolate.

This leads to a number of relevant observations about debug systems for compiled languages. Firstly the debug system must address the problem raised by compilation of relating machine level errors to program language level considerations, and, more broadly, having the ability to effectively 'decompile' the machine level state of a program into a state description at the program language level. Secondly the system must be capable of isolating any segment of the program for detailed examination by the debug system with little user effort. Thirdly, the debug system must be able to present information on the state of the program execution which allows the user to effectively isolate error causing conditions within the program.

This section will examine debug systems for high level programming languages in the light of these observations, with particular attention to the PASCAL language. The interactive user requirements of a debug system will be examined, followed by a look at implementation strategies for debug systems. This section concludes with a case study of a PASCAL debug system designed for interactive use.
5.4.1 User Requirements of a Debug Environment

It is often the case that, when designing a debug system for a language, the design is built from the implementor's point of view rather than structuring the information that needs to be presented to the user, and designing the system from that criteria. The most effective method of structuring this information is by using the notion of abstract machines, and, from the user view, augmenting a language abstract machine with appropriate reporting facilities. In this fashion the user can examine the state of the machine, using queries in a format consistent with the structures of the abstract machine, and allowing reporting of the state of the abstract machine while executing a program. This feature provides:

- Pedagogic features for novice programmers as to how this abstract machine functions, analogous to the operational semantics of the program.

- An inspection feature for the general user to allow understanding of various aspects of a particular algorithm.

- Interrogation features to allow incorrect features to be debugged.

- Reporting features, relating to machine abort or exception conditions (array index out of range, divide by zero and such).

The question that this section will address is: how best to implement a user view of an abstract machine? This section will address high level, compiled languages, with particular attention to Pascal.

Compiled language implementations tend to provide a 'black box' view of the abstract machine defined by the language. This, input is given to the executing program, and output is displayed to the user, but the internal arrangement of the abstract machine is inaccessible to the user. The program cannot be halted during execution, and the state of the abstract machine cannot be examined interactively at the abstract machine
level.

An excellent attempt to allow the user to view this abstract machine state has been implemented by Satterwaite for Algol-W on an IBM system [Sat72]. This system is a post-mortem dump system, where, on error termination, a very complete set of symbolic information, relating to the state of the abstract machine is dumped on the output file. This includes the values of all variables, statement execution counts, the state of the activation record stack, and an attempt to isolate the error to a source level construct. The major problem with such a tool, is that it is a policy of enumeration rather than selection. The user is presented with a plethora of information about the state of the abstract machine, only some of which is directly relevant to the user's inquiry about the program state.

This problem is one which is common among many debug systems. In general, the state of the abstract language machine is very large. The requirement of the debug process is a selective report of the state of this machine. This involves either specifying some filter to be applied to the abstract machine state report to produce a selective report, or the ability interactively select areas of the abstract machine state for inspection [Bro873].

Any such abstract language machine state analyzer must be capable of drawing quite fine distinctions regarding the abstract level at which errors occur. If the operating system is allowed to trap and handle machine level error conditions then the resultant diagnostic is inevitably a brief message to the user, describing the class of error, and the machine location at which the error was generated. Thus messages such as:

'ARITHMETIC OVERFLOW AT USER PC 400243'

are generated from the operating system error routine. For this type of diagnostic to have any meaning to the user, then the user must have access to the mapping, generated by the compiler, between the abstract programming language machine, and the actual machine level. If the error
is handled by a specific routine for that language then that routine must
have access to such a mapping, in order to relate the error condition and
location at the machine level to a corresponding construct at the
programming language level. If such access is possible, then diagnostics such as:

\[ \text{EXPRESSION: } x+2 \times x \text{ GENERATED ARITHMETIC OVERFLOW} \]
\[ \text{IN STATEMENT (LINE 17): } r := x+2 \times x \]
giving the variables and value which generated the overflow, can be
generated from the error analysis routine.

A more appropriate method is to allow the examination of the state
of the abstract machine to be user driven. Thus, when referring to a
language such as Pascal, output directives such as 'display the internal
status of Procedure X' are more appropriate here. But post-mortem
dumps are only of limited use in debugging a program. The user still is
faced with a backtrack problem, such that, knowing the immediate state of
the abstract machine which induced the error, the user than has to deduce
which conditions caused such a state to be induced. A post-mortem dump
can offer very little help in such a task.

It is more appropriate to allow the executing program to maintain a
continual description of the state of the abstract machine, or to generate a
routine which, given a program description, and a machine level state, can
generate a description of the state of the abstract machine at that point.
Given such a facility, the executing program can then be 'interruptable' at
any state of execution, and the state of the machine can be examined
interactively by the user.

Given that such a tool exists, so that the program, when executing,
can be interrupted at any point, and an abstract level description of the
state of execution can be generated at that point, the question then arises
as to what form of interface should be provided to allow the user to access
the state of the abstract machine. This question will be addressed with
specific reference to the Pascal abstract machine, although the comments
made here are applicable in general to most high level languages.

The basic tools of examination of the state of the Pascal abstract machine are:

- A display of the current statement being executed.

- A display of the variable identifiers, and their attributes (value and structure), which are being accessed.

- A display of the context, or the environment within which the statement is being executed.

To take this to a more basic level, the system must be able to display the structure and value of any variable which is 'visible' at that point. The system must be able to display the static chain of procedure calls, defining the state of the static environment, and also be able to display the dynamic chain of procedure calls at that point, defining the dynamic environment. The system must also be able to display the source code of the program, in such a fashion that the source code is matched against the procedure call stack.

Thus the user must have access to debug commands which:

- Display the current source code construct, and also to display any section of the source code text.

- Display the attributes of any variable, where the attributes are variable type (i.e. structure) and variable composite value.

- Display the static and dynamic environment of execution.

There must also be a means of defining 'safe' interruption points. At the abstract machine level there are a set of indivisible constructs which are mapped by the compiler to a sequence of machine instructions. Thus at the machine level, abstract indivisible constructs can be considered as divisible sequences. To ensure that the remapping from the machine state
to the abstract machine state is accurate, the machine level can only be interruptable at those points which correspond to interruptable states in the abstract machine. These commands can be considered as 'breakpoints' in the execution. Thus the user must have access to two types of breakpoint commands:

1. Commands which manipulate the state of any of the breakpoints. (listing all such points, or setting or resetting a particular breakpoint)

2. A command which allows resumption of program execution, to be suspended when execution 'encounters' a set breakpoint.

The format of the output of such display commands also plays a major role in how the user can view and manipulate the abstract execution state. Associated with such a debug system must be a complete description of the associated abstract machine, including the operational semantics of such a machine. In Pascal, the basis of such an abstract machine is that of activation records (ARs), where an AR is created for each active program, procedure or function instance within the abstract machine. The state of the abstract machine can be viewed, at the top level, as a stack of such ARs, together with a heap area for pointer (or dynamic) variables (the reader is referred to Appendix I for a more complete description of a Pascal abstract machine). The components of each activation record are:

- The environment pointers (static and dynamic) to preceeding ARs on the stack.

- A copy of (or pointer to) the corresponding code of the AR.

- A local sequence pointer to within the code of the AR.

- The name, structure, value and type of all parameters to the AR.

- The name, structure and value of all variables local to that
A display of the state of the abstract machine could be a graphic representation of the current stack of ARs. Thus the display could show a stack of 'boxes', with internal attributes corresponding to the components of the AR. On a normal video terminal such a display is not practical to build and maintain. The constraints on the number, and size of the display would imply that only a few ARs could be displayed on the screen at any time. A high resolution memory-mapped display is the appropriate output device for such a requirement. The activation records can be regarded as 'closed boxes' on the display, and the screen would, by default, simply show the top level of the ARs, without any internal detail. The user could have access to commands which 'open' any closed AR, and display the internal components of the AR.

If each possible breakpoint included a routine which, when execution 'passed' the breakpoint, updated the current display then it is possible to display to the user the complete state of the abstract machine during execution. Such a complete access tool certainly has areas of application, but it suffers from the same problem as the post-mortem dump, of presenting too much information to the user. The overheads from the user view of examining closely a small section of the execution of the program are quite high, and although this would be an excellent teaching tool for novice programmers, experienced users would not appreciate the additional costs involved.

An effective tool to answer such criticisms must be capable of detailed display of particular areas of code, but also capable of effectively 'ignoring' other sections which the user currently regards as correct in some sense. Thus the user must be able to specify to the debug system which sections of code are to be displayed in detail at the abstract level, and must also be able to change this focus of interest during execution.

Summarizing the main points of this section, from the user view, a
debug system must be able to effectively hide all aspects of the machine level from the user, and must be capable of answering queries as to the state of execution of a program, where the query and answer are phrased at the abstract level of execution. Information presented to the user must be structured in such a fashion that only information relevant to the user query is displayed, and the display must be as concise as possible.

5.4.2 Implementation of a Pascal Debug System

As was stressed in the previous section, the goal of any debug system is to provide the user with an effective tool to allow access to the abstract, language defined, runtime environment of a program. This section will examine a number of methods to allow the user to access this abstract environment.

There are many debug strategies available to a user when attempting to debug a program. One such strategy involves pre-processing a program to produce a program which has sentinels inserted into a modified source code (a distinction is drawn here between user-defined sentinels as described in the previous section, and 'automatically' inserted sentinels and procedures in a modified program by a pre-processor). The form of the sentinels inserted here are of the form of a procedure call to an inserted debug procedure, which implements a limited runtime inquiry mechanism. This modified program is then compiled and executed normally. The resultant execution can then output, at selected points, a very limited view of the execution environment. The debug procedure (invoked by the sentinel) can only access those variables which are within the current static scoping of the program at that point, and non-local variables are inaccessible. This pre-processing can create a relatively sophisticated view of the user program as the complexity of the pre-processor increases. In effect this adds another abstract level to that of the language system, where a user program is first augmented textually by the pre-processor, where specific constructs in the program are expanded to full modules. The result is a program at the language level, which can then be compiled and executed as before. This
abstract level can include many of the requirements of an effective debug system. Execution of the user program is halted at defined points, and the values of simple variables can be displayed to the user [Moli80].

Although this will provide a good level of debug facilities, this strategy cannot adequately handle the case where the program, when compiled, generates machine level errors. Also the system does lack a certain amount of flexibility. Because the debug routines are inserted in the program before pre-processing and compilation, to change the area of the program under examination, the program must be edited, pre-processed, compiled and loaded. On a multi-user interactive system this may be a prohibitive overhead for the user.

However, as the debug system is placed effectively 'within' the program environment before compilation., there is no requirement for this debug system to be able to 'decompile' the resultant machine level state of the program. This results in the major compelling aspect of this type of debug implementation: the portability of the system. As the pre-processor is written in the same language as the remainder of the system, all that is necessary to transport this system to a different computer system is the existence of a compiler on the target machine. Against this is the aspect of compilation and execution efficiency. There is a significant overhead in the increase in the size and complexity of the user program. To decrease these overheads of the debug tool requires portability to be sacrificed. Thus debug constructs can be added by the compiler rather than by a pre-processor. This has the effect of making the translation from the debug abstract level to the program language level invisible to the user, but also involves machine specific considerations to be added to the compiler to produce sufficient information for later processing by a debug system.

Once the decision has been taken to modify the compiler, there arises the choice of which direction such modifications should take. The Pascal language has been implemented on many machines using a two step process: initially transporting a P-Code compiler onto the target machine, and
writing a machine-specific P-Code interpreter, and then writing a machine
specific compiler for the target machine, based on the P-Code interpreter
[AhoU72, Wirt71b]. This intermediate language, the P-Code defines a
abstract machine which is very 'close' in some sense to the Pascal abstract
machine. Although the source code is not present in the P-Code, the
language of the P-Code is a language based on the activation record
implementation of Pascal.

Thus it is quite feasible to modify an existing P-Code interpreter to
produce a display of the state of the abstract machine at any stage. The
information that is missing from the existing interpreter is the mapping from
Pascal to P-Code, which is relatively direct, and the mapping between
reserved locations in the interpreter data space and variable names and
structures. If the interpreter is given access to the compiler's mapping
function for a particular program, then the interpreter can produce a
symbolic 'snapshot' of the abstract state of execution at any breakpoint in
the P-Code.

The disadvantages of such a strategy are that the overheads of the
interpreter are present in executing the P-Code. Thus the object program
is limited as to the size, and the number and range of system calls that are
possible in the object program (the range and number of file operations are
commonly sacrificed when using an interpreter for Pascal).

Interpretation is not the only possible solution to the problem of
maintaining, or being capable of generating, a abstract machine description.
It is possible to effectively 'decompile' the state of the machine level
program and produce an equivalent abstract machine state description. The
technique relies on dynamic code alteration by the debug system, and
employing the use of 'boundary markers' in the machine level execution
stack. Certainly such a strategy is specific to a particular compiler on a
single machine, but has a number of advantages to the interpreter solution.
The complete language system is more efficient on the particular machine.
The compiler can perform some measure of code optimization, based on the
characteristics of the machine level. The debug system does not alter in any way the code generated by the compiler, except by inserting a thread of 'null' instructions (linked by the address field of the instruction) throughout the compiled code. Thus the semantics of a program are unaltered by specifying to the compiler that the program is to be run in 'Debug mode'. As the system uses activation record 'markers' on the machine level stack, it is possible to decode the stack as a series of activation records.

The additional information needed by the debug runtime system is: a means of direct access to the text of the object program. This needn't be held in core, but if this text is stored on disk, a method of random access must be provided to allow direct access to a specified text segment. Also the system requires access to a copy of the compiler generated symbol table. This must be a core based structure, as the symbol table is commonly organised as a tree, or array of trees. Thus, if a compiler-based debug system is chosen, there is little effect on the execution speed of the compiled program, but there are penalties in the size of the executing program (the symbol table in core) and penalties on the speed of the debug routines. The advantages of this method is that the language semantics of execution with debug and execution without debug are identical.

5.4.3 A Case Study : A Pascal Debug System

This section describes the implementation of a Pascal debug system. This system was not implemented directly from a standard compiler, but was implemented by taking two existing systems, one of which contains an excellent user interface, the other containing a good set of primitive debug operators, and building a better user interface from this set of primitives. The two systems are, firstly, the implementation of Pascal-M, implemented for a VAX VMS system [BaAH80], (transported to the DEC-10 system by the author), and secondly, the University of Hamburg implementation of Pascal for the DEC-10 range of machines [KNHH83].
5.4.3.1 The DECSYSTEM-10 Pascal Debug System

The DECSYSTEM-10 Pascal debug system, provides a reasonably good view of the data structure of an executing program. The system has the capability to examine any simple or complex data structure, and display the value symbolically on the screen. The user specifies the data structure by the conventional Pascal syntax. The system can also alter the value of any variable in scope. The new value is specified as a valid Pascal expression. The system also allows display of the stack and the heap in symbolic form, and will also give a display of the dynamic procedure call chain.

The system is a breakpoint type system, with the added capability to interrupt and reenter the execution from debug (using operating system calls). The system executes in two modes: debug, where the user can execute any of the debug instructions, and normal execution, where the execution of the program, and any interaction with the user, is identical to a non-debug execution. The static breakpoints are set and cleared within debug mode. These breakpoints are related to the text listing, as well as the semantic structure of the text, and breakpoints can only be set at points in the text where the start of a statement and the start of a new line coincide. The screen display uses a simple scrolling screen model, where any program output will be lost if debug output is performed.

Summarizing the main points of this system:

Breakpoints  SET, RESET, LISTED by source text line number.

Display    VARIABLE value, STACK contents, HEAP contents, PROCEDURE / FUNCTION call chain and SOURCE TEXT.

Alteration VARIABLE := EXPRESSION to assign new values to variables in the current scope.

The means of breakpoint setting is probably the weakest point of this system from the user point of view. The user must keep a current listing
of the program for reference during debug, or continually request the
system to display text fragments throughout execution.

5.4.3.2 The Pascal-M System

The implementation of Pascal-M was originally implemented on a
VAX system, using VAX VMS Pascal [DIGI81]. The system has been
transported to the DEC-10, with minor modifications by the author.

This system is a modification of the Pascal-P4 compiler/interpreter
system [AmNJ76]. The compiler will produce a P-Code compiled program,
which is then interpreted. The P4 compiler has been modified to produce
two additional files: a text file which provides a cross reference between
the syntactic components of the text (here only statements and conditional
expressions are isolated) and positions within the text, and a symbol file
which provides a symbolic breakdown of the structure of the data space for
the program. The interpreter is altered to load these additional files with
the P-Code file, and also has a high level controller, which transfers control
between the monitor system (the debug component) and the executing
program (the interpreter component).

The display presented to the user is a split-screen type: the text and
debug output are shown on separate areas of the screen. The system allows
symbolic examination of variables, and stack components (but not
assignment of values). The system is also a breakpoint system but here the
breakpoints can be set at the start of any statement or conditional
expression. The breakpoints are set and cleared with reference to the
cursor position of the displayed text on the screen, so there is no need for
additional listings to perform debugging. The system also supports simple
movement of text on the screen, allowing up and down scroll of the
window, and sequential movement to possible breakpoints in either direction,
utilizing the four movement keys on a terminal. Single statement stepwise
execution is also supported by this system. The system also allows the
selection of a number of variables as 'monitored' variables, whose values are
continually displayed on the screen.

The major components of the interactive system are:

**Breakpoints**  SET, RESET by cursor position in text display.

**Display**  VARIABLE value, STACK (top activation record contents only), PROCEDURE / FUNCTION call chain.

**Text Display**  SCROLL text display up and down, MOVE display to next and previous possible breakpoint positions in text.

**Reenter Execution**

BREAKPOINT where next set breakpoint reenters debug.

STATEMENT to execute the next single statement.

The weaknesses of this system are: firstly there is no dynamic assignment to variables of the object program. Thus minor problems with data values cannot be patched during the debug session, but must be accomplished by editing and recompiling the program. Also there is not a consistent method of interruption. The debug system will be reentered either at the next breakpoint, or when 10,000 P-Code instructions have been executed. The latter decision allows a break of execution 'within' a statement, which, when the debug mode is entered, has placed the program in an indeterminate state.

5.4.3.3 Implementation of DECSystem-10 Pascal Debug

The first point to make is a general problem, common to most compiled language debug systems: neither of the systems examined here allows dynamic alteration of source text, or direct alteration of the stack structure. This is a direct result of initially compiling the code, rather than interpreting the source text and it is doubtful whether a full Pascal interpreter would be feasible on the DEC-10 system, and the size and time requirements would be prohibitive in an interactive environment.
From the user's view there is a marked similarity in the capabilities of the two systems. This is not the same with the implementation strategies. The Pascal-M debug system is a totally independent system, and debug and non-debug executions may be different, due to differing semantics of a compiler and this debug compiler/interpreter. The debug system also has limitations in the number of files and system calls that can be performed, which is not present in the DECSYSTEM-10 compiler. This point is very critical, as programs which are compiled to relocatable code and then executed, may not function identically to one which is compiled and executed using the Pascal-M debug system, thus defeating the aim of the dynamic debugging concept entirely. Although this does give the system a certain amount of portability, this may be of little value when the semantics of the debug system and a compiler are markedly different.

The other point to make about the interpreter is the system impact of such a system. The designers have opted for a static stack and heap allocation system, and have also elected to load the text and symbol information into core with the P-Code array. Thus is a significant amount of data space must be reserved by the interpreter. The implementors admittedly had little choice in this decision, as it is forced on them by the decisions to write the interpreter in standard Pascal, and the lack of random I/O procedures in this standard, but it does imply that execution of this system on any non-virtual memory machine has very large overheads in terms of a core demand.

The DECSYSTEM-10 Pascal debug system has opted for efficiency and compatibility with the compiler, at the expense of portability. The compiler here is identical, and the compiled code is identical, so that debugging a program does not imply execution of the program under a different Pascal system. The debug component of system is one which can be added to the compiled code in such a fashion that it has no semantic effects on the object code of the program.

In order to fully describe this system, it is necessary to describe a
number of features of the operating system (Tops-10) which make this debug strategy possible.

The first feature is that all machine error conditions need not cause termination of the program. All such errors are 'trapped' in a consistent fashion. A machine level program may define a routine which will service all such errors. In such a case, when a machine level error occurs (such as arithmetic overflow, or illegal memory references), control is passed to this routine, while storing the location at which the error occurred in a reserved location in the job area (there are a number of such reserved locations, referred to collectively as the Job Data Area). Thus the flow of control can be seen in Figure 5-7.

![Diagram](https://via.placeholder.com/150)

**Figure 5-7: Error trapping on the DECSYSTEM-10**

Thus it is possible to effectively hide all aspects of the machine level from the user, as all machine level error conditions can be trapped, and serviced by a routine which can, at the very least, translate the error diagnostic to a program level diagnostic before halting the execution.

However, as pointed out previously, an effective program language
debug system must provide a higher level of user accessibility into the runtime structure of the program. The next feature of the operating system is the ability to specify a second entry point into a user program: the 'DDT entry point'. This can allow two separate programs to co-exist in the same core area, with control being passed between the two systems in a consistent fashion, so that the user need not be aware of this structure. On one level this debug system can be viewed as a normal component of the runtime system, being entered through normal calls from the compiled code (corresponding to breakpoints in the user program) or entered through calls from the error handling routine. However this gives the user the ability to interrupt execution of the program at any point in execution, and then specify the debug entry point as the continuation point. The most obvious application of this is that infinite loops in the program can be interrupted by the debug system by a simple user command. This feature is not possible with many debug systems, where, if a breakpoint has not been set at a particular location, it is not possible to interrupt execution of the relevant construct. The other advantage of such a system is that the data area of the debug program can be forced to include the complete code and data area of the user program. This structure is as shown in Figure 5-8.

Thus the debug system has the ability to read, and write into the user program area. All that is remaining to complete the debug system is access to the compiler symbol table, which is the mapping between the symbolic description of the user data area, and the core image. This access is provided by the Tops-10 Link-Loader and the Pascal compiler. If the compiler 'dumps' the complete symbol table onto the relocatable binary file that is being generated (including all pointers and structure descriptors), then it is possible to instruct the Link-Loader to load this information into core at the same time as the compiled code and data. Once this table is accessible by the debug system, it is possible to translate any data value into a symbolic value of a particular variable, and given a variable name, it is possible to access (either read or write) the corresponding value.
There is one other interesting aspect of this type of debug support: as the complete set of runtime support routines have also been loaded with the user program, the debug system may be written in the same high level language as the user program. Hence there is no constraint on writing the debug system in a machine level language.

Thus there are three major aspects of the Tops-10 system which can be considered as essential for implementing any complete debug package: the capability to trap and service all machine level error conditions, the ability to declare a debug entry point for the program, so that the user program can be interrupted at any point in execution, and the ability to instruct the Link-Loader to also load the corresponding symbol table for the object program.
The Pascal compiler generates the breakpoint facility by the use of threaded null instructions through the code to mark possible breakpoints, and the symbol table is generated by placing the compiler generated symbol table tree after the end of the executable code. Thus when then code is loaded, the text associations and symbol tables are also loaded in the one operation. The expense of this in terms of execution size is not critical. There is an overhead of 4 instructions per compiled statement and 4 words per symbol in the compiled code. The debug system is a separate program, which uses the complete execution space of the object program as it's data space. The two programs are organised on a single hierarchy, so that when control passes back to the program, the Debug system can only be entered by the execution of a set breakpoint (changing a null instruction to a jump to the debug entry routine) or the occurrence of an exception (user interruption followed by the Debug command or a machine level error).

Certainly when comparing these systems (DECSystem-10 Pascal and Pascal-M), there are only two major differences in the functional modules of the system. Both systems can generate and decode a symbolic breakdown of the execution data space. Both can generate and handle breakpoints in the execution. The first major difference is the runtime association of the original source text with the runtime environment, which is only partially handled by DECSystem-10 Pascal. The second difference is the structuring of the information presented on the screen, to delineate text display and program output in a consistent and simple fashion. The major advantage of DECSystem-10 Pascal is the compatibility within the system, where the Debug sub-system is an additional, separate component of the complete Pascal language system.

5.4.3.1 Implementation of Pascal Debug System

The implementation described here has taken the approach of modifications to the DECSystem-10 Pascal debug system, with the aim of achieving the same user view of the system as that described by Pascal-M, but with the internal consistency of the original DECSystem-10 Pascal debug
sub-system.

The view of an executing program which is generated by this system is very similar to that of the P4 system. The system control switches between two modes: program execution and debug commands. All the necessary information is presented to the user via a split screen arrangement as shown in Figure 5-9.

<table>
<thead>
<tr>
<th>status line</th>
</tr>
</thead>
<tbody>
<tr>
<td>normal I/O area - split from the lower half of the display, this is the normal TTY</td>
</tr>
<tr>
<td>all program I/O is performed within this half-screen</td>
</tr>
</tbody>
</table>

Program Text window This section of the screen is maintained by the debug system to display the source code (and surrounding code) of the statement being executed.

Monitor variable window: the names and values of any monitored variables are displayed in this window

Figure 5-9: Screen Display for Debug System

If the status line displays 'EXECUTION', then the upper screen functions as a normal terminal, and all program input and output to the terminal is performed within the upper screen.

If the status line displays 'DDT COMMAND' the cursor will be positioned within the lower screen, and the actual position will correspond to the point where execution has been suspended. This position is referred to as the 'home' position. The possible commands at this point are:
Cursor movement

- **Up arrow**: scroll the lower window up 1 line
- **Down arrow**: scroll the lower screen down 1 line
- **Left arrow**: move the cursor to the next breakpoint
- **Right arrow**: move to the previous breakpoint
- **HO**: return to the current breakpoint position
- **BN**: move to next breakpoint set position
- **BP**: move to previous breakpoint set position

Breakpoints

- **BS**: set a breakpoint at the cursor position
- **BC**: clear any breakpoints at this position

Display

- **E <expression>**: display the value of the expression on the upper screen
- **MS <variable>**: add this expression to the list of monitored expressions
- **MC <number>**: remove a monitored variable
- **ST**: display the state of the stack
- **HE**: display the state of the heap
- **PR**: display the procedure chain

Assignment

- **A <variable> := <expression>**: perform the assignment

Execution

- **Q**: exit from execution of this program
- **G**: continue until next breakpoint
- **S**: execute 1 statement and resume debug
- **P**: execute 1 statement or 1 procedure call

Status

- **R**: refresh the screen
- **?**: on-line help

Most commands can be preceded by an unsigned number, which acts as a multiplier. Thus **30 <up arrow>** scrolls the screen up 30 lines of text, and **30 S** executes 30 single statements.

Execution of an object program is interrupted after either a single statement has been executed, a set breakpoint is encountered, 10,000 source statements have been executed, or the user interrupts the execution and
then resumes execution via the debug processor.

The aim of the implementation is to use the set of debug calls from the DECSystem-10 Pascal debug implementation, and provide a user interface similar in many ways to Pascal-M. This involves modifications to both the compiler and the runtime support programs. The compiler modifications will be briefly examined first.

The modifications here are aimed to generate statement-based breakpoints in the compiled code rather than source line based, and to generate a text file which can be used by the runtime support to access a selected line or statement.

The first aim is easily achieved by adding a statement counter into the compiler. The debug system is altered to record this statement counter rather than a line counter.

The second aim is not achieved so easily. The structure of this text file must first be defined from the aims of the runtime support system:

- Only the 10 lines of text displayed on the screen are to be held in core.

- Random file access routines are to be used wherever possible to reduce access times.

- Access routines will be of the form: fetch line <line number> or fetch the line containing statement <statement number>.

- The file must be as compact as possible.

The file structure to satisfy these requirements was designed as a three level file, using two levels of tables and the program text. To make the file as compact as possible, these three separate file structures have been interlaced within a single physical file.
- The third, or lowest level contains the text of each line, followed by the positions of all the breakpoints in the line. These line entries are packed into blocks of size equal to the unit of disk I/O (200 words on the DEC-10), with the restriction that no line may cross a block boundary.

- The second level is a table which maps line numbers into entries into the third level. Each line has a packed two word entry which gives the location of the text, number of breakpoints, length, and the statement numbers which start in this line. Blocks of these tables are arranged in a doubly linked list within the file.

- The first level contains the book-keeping entries, such as the first and last line numbers, and a table of the first statement number occurring in each table block, together with the address of that table.

Thus to find the line containing a particular statement number, the first level defines the line table block to examine, and by scanning this block, the address of the line can be retrieved, together with the line number. To retrieve the text of a line number, only the second and third levels of the file need be examined.

As this file cannot be generated using sequential output, the compiler has had random access routines added as extensions to the language. These extensions allow a file to be accessed by a logical component number. Thus the calls GET(f,n) and PUT(f,n) have been added to the compiler to respectively read and write the nth logical component of the file.

The structure of the debug system can be seen by a breakdown into the relevant functional modules as indicated in Figure 5-10.

The screen controller must be a load-time module, specified at compile-time to match the particular terminal on which the debug-system
Figure 5-10: Internal Organization of Screen Debug System

will be used. The base level of terminal I/O uses calls to a virtual
terminal which can support cursor addressing, partial screen scrolling and
selective screen erasure. The screen controller maps such calls into control
sequences for particular terminal types.

For more basic terminals, all input and output to the screen must be
intercepted, and placed on the screen correctly. This involves building an
alternative runtime support module for the procedures GET from, and PUT
to the terminal, which intercepts all terminal traffic and updates the screen
depending on the current mode of operation (Debug or Execution modes).
The overheads of using such terminals are prohibitive on a normal
timesharing system, as the overall approach is based on simulating
multi-window facilities on a terminal which does not provide such facilities
at the terminal hardware (or firmware) level.
6. Further Developments

The major observation that can be made from the material presented in this thesis is that a supportive environment for a programming language implementation entails a far wider set of user tools than a compiler and a runtime support system. This chapter will briefly examine the areas of possible future development of the tools presented in this thesis, to outline the wider goals of language support systems.

6.1 Portability of the Workbench System

The first area of consideration is the portability of this system to other types of computer systems. This can be a major problem area in many systems, where the application programs which build the system assume a particular form of interface to the host operating system, or rely on alterations made to the host operating system to implement certain specialized functions. This system has attempted to use a basic operating system interface in the 'core' areas of the applications programs, so that a basic version of the Workbench system should be readily portable to other systems.

This has been investigated by transporting some of the components of the Workbench to a somewhat different system than the original DEC-10 system. The target host system in this case is a single user micro processor system, using an Intel 8085 8 bit processor, running under Digital Research's CP/M operating system [DIGRS82]. The transportation of the software was done at the Pascal level, transporting the core of the Workbench system, and the screen editor. The implementation effort required for this system centered mainly around building the facility of allowing an executing program to invoke a second program, handing the called program a set of parameters, which is not part of the standard CP/M operating system interface. The screen editor was transported also in a 'core' form, due largely to the size restriction on programs running under CP/M (56 Kbyte
of store). Thus some of the editor code to implement the less common commands was removed to allow a larger data area for the editor. This was considered a better strategy than attempting to produce an editor which used memory overlays, as the overlay controller would introduce inefficiencies, and, as this is not an operating system construct, the implementation would have to include the implementation of such an overlay handler.

Certainly this exercise was successful in demonstrating the portability of the Workbench system, and demonstrates the feasibility of implementing a quite sophisticated user interface using a quite small operating system interface.

A particular user of the Workbench system is automatically assigned this case. Although this has not been attempted, the Workbench would fit quite naturally within a UNIX system. The UNIX interface to the user, the shell program can be seen as a utility program, and this could easily be replaced by the Workbench controller which performs internal calls to the remainder of the Workbench system. Such alternative shell programs are common within UNIX systems, as a means of creating an environment tailored specifically to the user’s application. There are many aspects of the UNIX system which could reduce the complexity of the Workbench system, including the 'FORK' utility as an alternative to the rather elaborate 'background' processing stream of the Workbench. With care, much of the functionality of the shell program could be introduced into the Workbench environment, effectively implementing a command language similar to the shell program itself.

6.2 Extensions to the Workbench

The actual Workbench controller, and the screen editor components of this system make no binding assumptions about the nature of the programming language used, and the system can be easily configured to interface to a number of associated language sub-systems rather than Pascal
alone. Indeed this can be extended to allow the user to invoke processing packages (such as SPSS, Genstat and similar).

These extensions have been implemented on the DEC-10 system to allow novice student users on the system to use the system productively with very little training on how to use the system. The Workbench is the primary interface for a population of 500 student users on the system, and the conclusion from this usage is that the Workbench is a more effective interface than the Tops-10 operating system in the academic environment.

The extensions to the Workbench take the form of an internal table of classes of users, and a profile of the requirements of each of these classes. A particular user of the Workbench system is automatically assigned into one of these classes of users. This allows the Workbench to create a menu-driven environment where the resources and tools at the user’s disposal and the menu entries themselves are determined by the class of user, so that some users can be classified as Pascal programming students, some as statistical package users, some as Pascal and Assembler Language programmers and so on. Thus all students use a very similar, and easy to learn, interface to the system, where the Workbench ‘trims’ the user view of the system to suit the particular needs of each user.

6.3 The Pascal Language Sub-System

The are a number of areas within this Pascal sub-system which indicate further development. Firstly, the Pascal editor - compiler interface can be improved by a tighter binding between the compiler and editor. The current interface involves re-generating the textual representation of the Pascal program within the editor, and then passing this text to the compiler for parsing and code generation. As the editor already maintains the text internally as a syntax-directed structure, the compiler’s parse of the text is redundant, this opens up the possibility of implementing a code-generator which takes the editor’s data structure (the syntax tree) and generates
machine code directly from this, bypassing the unnecessary steps of text
generation and re-parsing.

Secondly, as shown in the Cornell Program Synthesiser [Teir81], it is
possible to embed into the editor’s syntax tree code to allow an interpreter
to execute this structure directly. This would allow the user to execute a
program (or even an incomplete program structure) during editing. Flow of
control presents few problems here, but the area of expression evaluation is
somewhat complex, and would represent the bulk of the implementation
effort (in particular, Pascal constructs that use fields of packed structures
within expressions would prove troublesome, as would expressions involving
the handling of fields of variant records). This inclusion of an interpreter
into the editor environment gives the user the ability to test the execution
of a program without the necessity to generate a textual representation of
the program, compile, link and then execute the program. There is still a
definite role for explicit code generation here, where more efficient
(optimized) code is required, or for production programs which wish to avoid
the overheads inherent in interpretation, but significant attention must be
given to exact adherence to semantic standards within Pascal, as it is an
essential feature that the behaviour of programs be identical when executed
by either the interpreter or by code generation.

The last aspect here is that of the debug system. As presented in
Chapter 5, the debug system uses the compiled code to reproduce a
symbolic environment to allow the interactive interrogation of the state of
execution of a program. Such functionality can be provided more
efficiently, and with considerably less internal complexity, within the
interpreter. This environment is now a symbolic environment, and similar
functionality as described in Chapter 5 can be constructed from the editor’s
syntax tree and from the internal state of the interpreter.

These areas would produce a more integrated view of program
development, and a more efficient interface between the components of the
language system. The internal organization of such a language processor is
indicated in Figure 6-1.

Within the system presented in Figure 6-1, the user can switch contexts between editing, interpretation and debugging of a program with very few overheads, and code generation can be seen as one of the latter steps in the program development process. Such an approach is a logical extension of the system as implemented, utilizing fully the inherent advantages of using a more complex data structure to represent a program.
7. References

AhoU72 'The Theory of Parsing, Translation and Compiling, Volume 1: Parsing'
A. Aho, J. Ullman

AmNJ76 'The Pascal-P Compiler Implementation Notes'
U. Amman, K. Nori, C. Jacobi
Institut fur Informatik, ETH, Zurich, Switz., 1976.

AnBB76 'Semanol (73, A Meta-language for Programming the Semantics of Programming Languages')
E. R. Anderson, F. C. Belz, E. K. Blum

AtMN81 'Context Sensitive Editing as an Approach to Incremental Compilation'
L.V Atkinson, J. Megregor, S.O North

AtkN81 'COPAS - A Conversational Pascal System'
L. Atkinson, S. North
Software - Practice and Experience, 11 8
(August 1981), 819 - 850.

BBN73 TENEX Text Editor and Corrector
BOLT, BERANEK and NEWMAN Inc.,

BROW81 Users Guide to the BROWN CMS Editor
Brown University Computing Centre
Brown University, Providence, R. I., U.S.A., April 1981.

BaAl80 'Pascal-M Pascal System'
C. Barth, W. Agutter, D. Hodson
University of Adelaide, 1980.

BaTW81 'Design of the PEN Video Editor Display Module'
D. Barach, D. Taenzer, R. Wells
SIGPLAN Notices, 16 6 (June 1981), 130 - 136.
What is a Command Language?
D. Beech
published in: 'Command Language Directions'
D. Beech (Ed)

The CRT text editor - NED. Introduction and
Reference Manual
W. Bilofsky

'A Design for a Text Editor'
S. Bourne
Software - Practice and Experience, 11 1
(January 1971), 73 - 81.

Program Debugging
A. Brown, W. Sampson

'Throwaway Compiling'
P. Brown
Software - Practice and Experience, 6 7
(July 1976), 423 - 431.

'Structure-Oriented Editors'
H. Burkhart, J. Nievergelt
Institute für Informatik, ETH, Zurich, Switz, May 1980.

'The Keystroke-Level Model for User Performance
Time With Interactive Systems'
S. Card, T. Moran, A. Newell
CACM, 23 7 (July 1980), 396 - 410.

'Full-Screen Editing in a Hostile Environment'
T. Cargill
Software - Practice and Experience, 11 9
(September 1981), 975 - 982.
CoEA76  'The Design and Implementation of an Interactive Document Editor'
G. Coulouris, et al
Software - Practice and Experience, 6 4
(April 1976), 271 - 279.

CohC77  'A Language for Inquiry about the Run-time Behaviour of Programs'
J. Cohen, N. Carpenter
Software - Practice and Experience, 7 7
(July 1977), 145 - 160.

Cohe82  'Text-Oriented Structure Commands for Structure Editors'
E. Cohen
SIGPLAN Notices, 17 11 (November 1982), 45 - 49.

Coli81  'Notes of the Multics Command Language'
A. Colijn
Software - Practice and Experience, 11 7
(July 1981), 741 - 744.

CouF80  'A Device Driver for Display Terminals'
C. Contant, C. W. Fraser
Software - Practice and Experience, 10 3
(March 1980), 183 - 188.

CunP76  'A Language Independent System to Aid the Development of Structured Programs'
R. Cunningham, C. Pugh
Software - Practice and Experience, 6 10
(October 1976), 487 - 503.

DHKL79  'The MENTOR Program Manipulation System'
V. Donzeau-Gouge, G. Huet, G. Kahn, B. Lang
IRIA-Laboria, France, August 1979.

DIGI78  SOS Reference Manual

DIGI82  TOPS-10 Software Notebooks
DIGH82  
*CP/M 2 User's Guide*  

DOD80  
'Requirements for Ada Programming Support, "STONEMAN"'  

DaDH72  
'Structured Programming'  
O. Dahl, E. Dijkstra, C. Hoare  

DamR71  
'Online text editing: A Survey'  
A. van Dam, D. Rice  
ACM Computing Surveys, 3 (September 1971), 93 - 114.

Davi82  
'String Searching in Text Editors'  
D. Davies  
Software - Practice and Experience, 12  8  
(August 1982), 709 - 718.

Denn81  
'System Design for Usability'  
R.A. Denning  
CACM, 24  8 (August 81), 491 - 501.

Denn75  
'Modularity'  
J. B. Dennis  
published in:  
'Software Engineering - An Advanced Course'  
G. Goos, J. Hartmanis (Eds)  

DeuL67  
'An On-line Editor'  
L. F. Deutsch, B. W. Lampson  
CACM, 12 (October 1967), 793 - 803.

Dijk62  
An Attempt to Unify the Constituent Elements of Serial Program Execution'  

Dijk76  
'A Discipline of Programming'  
E. W. Dijkstra  
DolM80

'Using a Command Language as the Primary Programming Tool'
T. A. Dolotta, J. R. Mashey
published in: 'Command Language Directions'
D. Beech (Ed)

Elli82

'Design of a Simple Screen Editor'
P. Elliot
Software - Practice and Experience, 12 4
(April 1982), 375 - 384.

Ewin83

MINIED User's Guide
R. R. Ewin

FeiM80

'An Incremental Programming Environment'
P.L. Feiler, R. Medina-Mora
Dept. of Comp. Science,

Fins80

'Theory and Practice of Text Editors or a Cookbook for an EMACS'
C. Finseth

Fras79

'A Compact, Portable, CRT-based Text Editor'
C. W. Fraser
Software - Practice and Experience, 9 2
(Febuary 1979), 121 - 126.

Fras80

'A Generalized Text Editor'
C. W. Fraser
CACM, 23 3 (March 1980), 154 - 158.

Fras82

'A Programmable Text Editor'
C. W. Fraser
Software - Practice and Experience, 12 3
(March 1982), 241 - 250.

FrEA74

'A Pascal Compiler Bootstrapped on a DECSYSTEM-10'
G. Friesland et al.
Lecture Notes in Computer Science 7,
FrMS80
'A Standard Operating Command and Response Languages: The ANSI X3H1 Effort'
L.C. Frampton, S. Mellor, C. T. Schlegel
published in: 'Command Language Directions'
D. Beech (Ed)
North-Holland, Amsterdam, Neth., 1980, 83 - 93.

Good81
'Etude and the Folklore of User Interface Design'
M. Good
SIGPLAN Notices, 16 6 (June 1981), 34 - 43.

Gosl81
'A Redisplay Algorithm'
J. Gosling
SIGPLAN Notices, 16 6 (June 1981), 123 - 129.

HaEA81
'The Implementation of Etude, An Integrated and Interactive Document Production System'
M. Hammer et al.
SIGPLAN Notices, 16 6 (June 1981), 137 - 146.

Hall82
'SED: A CRT editor For TOPS-10 and TOPS-20'
A. C. Hall
DIGITAL TOPS-10 On-Line Documentation, April, 1982.

HaSS80
'A Virtual Operating System'
D. Hall, D. Scherrer, J. Sventek
CACM, 23 9 (September 1980), 495 - 502.

Hans71
'Creation of Hierarchic Text with a Computer Display'
W. J. Hansen

Harr80
'The Codasyl COSCL Journal of Development'
T. Harris
published in: 'Command Language Directions'
D. Beech (Ed)

Haze74
'A General Purpose Text Editor for OS/360'
P. Hazell
Software - Practice and Experience, 4 10
(October 1974), 389 - 399.
Haze80  'Development of the ZED Text Editor'
P. Hazell
Software - Practice and Experience, 10 1
(January 1980), 57 - 76.

HenS74  'A Tool for Structured Program Development'
P. Henderson, R. A. Snowdon
published in: 'Information Processing 74'
J. Rosenfeld (Ed)
North Holland, Amsterdam, Neth., 1974, 204 - 207.

HeuL77  'An Automatic Formatting Program for Pascal'
J. Henrich, H. F. Ledgard
SIGPLAN Notices, 12 (July 1977), 82 - 84.

IBM70  System/360 Administrative Terminal System

IBM80  IBM Virtual Machine/system Product: System
Product Editor User's Guide

Inga81  'The Smalltalk Graphics Kernel'
D. H. Ingalls
Byte, 6 8 (August 1981), 168 - 194.

IroD72  'A CRT Editing System'
E. Irons, F. Djorup
CACM, 15 1 (January 1972), 16 - 20.

K. Jensen, N. Wirth

KNNIH83  'Pascal for the DECSystem-10'
E. Kisicki, H. Nagel
IFi-University of Hamburg, Miteilung 37, May 1976.
as revised March 1980 : B. Nebel, IFi-Hamburg
as revised February 1983 : G. Huston, CSC-ANU
on-line documentation, A.N.U.
Faculties Computer Unit DEC-10, March 1983.

KerM81  'The UNIX Programming Environment'
B. W. Kernighan, J. Mashey
Kern78a  
'A Tutorial Introduction to the UNIX Editor'
B. W. Kernighan
Bell Laboratories, Murray Hill, N. J., U.S.A.,
September 1978.

Kern78b  
'Advanced Editing on UNIX'
B. W. Kernighan
Bell Laboratories, Murray Hill, N. J., U.S.A.,
September 1978.

KerP76  
'Software Tools'
B. W. Kernighan, P. L. Plauger

Kerp78  
'The Elements of Programming Style, 2nd Edition'
B. W. Kernighan, P. L. Plauger

KeriP81  
'Software Tools in Pascal'
B. W. Kernighan, P. L. Plauger

Kras81  
'The Smalltalk-80 Virtual Machine'
G. Krasner
Byte, 6 8 (August 1981), 300 - 320.

Lamp78  
'Bravo Manual'
B. P. Lampson

Levi82  
'A Programmable Text-editing System'
M. Levison
Software - Practice and Experience, 12 7
(July 1982), 611 - 622.

LewP80  
'ALBE/P : A Language-Based Editor for Pascal'
J. Lewis, D. Porges
Dept. of Computer Science, Yale Univ., New Haven,
Love82  'The PERQ Workstation and the Distributed Computing Environment'
J. Lovelock

MDFA67  'THOR - A display based Timesharing System'
J. McCarthy, B. Dow, G. Feldman, J. Allen
SJCC, 30, AFIPS Press, Arlington, Va., U.S.A.,

MICR81  WORDSTAR user's guide
MICROPRO International,

Mac77  'Design and Implementation of a Display Oriented Text Editor'
I. Macleod
Software - Practice and Experience, 7 10
(October 1977), 771 - 778.

Mads79  'CCL - A High Level Command Language'
J. Madsen
Software - Practice and Experience, 9 1

MeyD82  'Interactive Editing Systems (Parts 1 and 2')
N. Meyrowitz, A. van Dam
ACM Computing Surveys, 14 3
(September 1982), 321 - 416.

Mike81  'Prettyprinting in an Interactive Programming Environment'
M. Mikelsons
SIGPLAN Notices, 16 6 (June 81), 108 - 116.

Mohi78  'Prettyprinting Pascal Programs'
P. R. Mohiher
SIGPLAN Notices, 13 (July 1978), 34 - 40.

Moli80  'PASDDT - A Debug System for Pascal Programs'
B. P. Molinari
Dept. of Computer Science, A.N.U. (January 1980),
CS1980-02-12.
Moli82  
'Aspects of an Abstract Ada Machine'  
B. P. Molinari  

Moli83  
'Teaching Computer Systems via the Abstract Machine Concept'  
B. P. Molinari  
Dept. of Computer Science, A.N.U., March 1983  
(unpublished).

Mont82  
'Bringing Manual Input into the 20th Century: New Keyboard Concepts'  
E. Montgomery  
Computer, 15 3 (March 1982), 11 - 21.

MorS81  
'The Design of a Language-Directed Editor for Block-Structured Languages'  
J. Morris, M. Schwartz  
SIGPLAN Notices, 16 6 (June 81), 28 - 33.

OIR781  
'An Interactive Manager for Pascal Software'  
R. Oldehoft, W. Ralph, M. Tindall  
Software - Practice and Experience, 11 8  
(August 1981), 867 - 874.

OrFP78  
'Programming Language Structures'  
E. I. Organik, A. J. Forsythe, R. P. Plummer  

Over80  
'UCSD Pascal : A Portable Software Environment for Small Computers'  
M. Overgaard  
Published in: AFIPS Conf.Proc., 1980 NCC  
D. Medley (Ed)  

Palm79  
'VIDED - A Display Oriented Text Editor'  
J. Palme  
Swedish National Defense Research Institute, FOA  

Pete77  
'On the Formatting of Pascal Programs'  
J. L. Peterson  
SIGPLAN Notices, 12 (December 1977), 83 - 86.
Pool71  
'Implementation of an Editing Algorithm Allowing Repeating Corrections'  
M. D. Poole  
Software - Practice and Experience, 1 10  
(October 1974), 373 - 381.

Pool75  
'Debugging and Testing'  
P. Poole  
published in: 'Software Engineering - An Advanced Course'  
G. Goos, J. Hartmanis (Eds)  

Prat81  
'Programming Languages : Design and Implementation'  
T. W. Pratt  

ReLM81  
'db User's Guide'  
S. P. Reiss, M. Lustig, L. Medvène  

RitT78  
'The UNIX Time-sharing System'  
D. Ritchie, K. Thompson  
The Bell System Technical Journal, 56 6, pt2  
(July - August 1978), 1905 - 1930.

SFGT81  
'The Ada Environment: A Perspective'  
V. Steenning, T. Froggat, R. Gilbert, E. Thomas  
Computer, 14 6 (June 1981), 26 - 36.

SIKII82  
'The Star User Interface : an Overview'  
D. Smith, C. Irby, R. Kimball, E. Harslem  
Published in : 'AFIPS Conf. Proc., 1982 NCC'  
H. Morgan (Ed)  

Sand78  
'Programming in an Interactive Environment: The LISP Experience'  
ACM Computing Surveys, 10 1 (March 1978), 33 - 71.

Satt72  
'Debugging Tools for High Level Languages'  
E. Satterthwaite  
Software - Practice and Experience, 2 1  
(July 1972), 197 - 217.
Scow81  'A Survey of some Text Editors'
R. Scowen
Software - Practice and Experience, 11 9
(September 1981), 883 - 906.

SheE81  'PASES - a Programming Environment for Pascal'
E. Shapiro, et al

SiHL77  'A Basis for Executing Pascal Programmers'
A. Singer, J. Heuras, H. F. Ledgard
SIGPLAN Notices, 12 (July 1977), 101 - 105.

SiHL81  'The Annotated Assistant: A Step Towards Human
Engineering'
A. Singer, H. F. Ledgard, J. Huemas
IEEE Trans. of Software Engineering, 7 4
(July 1981), 353 - 370.

Snee78  'User Interface Design for Text Editing : A Case
Study'
J. Sneeringer
Software - Practice and Experience, 8 5
(September - October 1978), 543 - 557.

Snow78  'The Software Tools Project'
C. Snow
Software - Practice and Experience, 8 5
(September - October 1978), 585 - 599.

StaI81  'EMACS : The Extensible, Customizable
Self-Documentating Display Editor'
R. Stallman
SIGPLAN Notices, 16 6 (June 1981), 147 - 156.

StrJ81  'The Implementation and Experiences of a
Structure-Oriented Text Editor'
O. Stromfors, L. Jonesjo
SIGPLAN Notices, 16 6 (June 1981), 22 - 27.

Tass74  'Program Style, Design, Efficiency, Debugging
and Testing'
D. van Tassel
TeRH81 'The Why and Wherefore of the Cornell Program Synthesiser'
T. Teitelbaum, T. Reps, S Horwitz
SIGPLAN Notices, 16 6 (June 1981), 8 - 16.

TeiMS81 'The Interlisp Programming Environment'
W. Teitelman, L. Masinter

TeiR81 'The Cornell Program Synthesiser : A Syntax Directed Programming Environment'
T. Teitelbaum, T. Reps
CACM, 24 9 (September 1981), 563 - 573.

Teit72 'Automatic Programming - The Programmer's Assistant'
W. Teitelman

Tesl81 'The Smalltalk Environment'
L. Tesler
Byte, 6 8 (August 1981), 90 - 147.

ThaT82 'The Practicality of Text Signatures for Accelerating String Searching'
A. Tharp, K. Tai
Software - Practice and Experience, 12 1 (January 1982), 35 - 44.

Toll65 'TVEDIT'
B. Tolliver

VIP69 VIPcom users guide
VIP SYSTEMS

WaiC81 'An Analysis / Synthesis Interface for Pascal Compilers'
W. Waite, L. Carter
Wass80
'Software Tools and the User Software Engineering Project'
A. Wasserman
Published in: 'Software Development Tools'
W. Riddle, R. Fairley (Eds)

Wass81
'Automated Development Environments'
A. Wasserman

WatY74
'Extran - a Top Down Programming System'
T. Watanabe, F. Yamamoto
published in: 'Information Processing 74'
J. Rosenfeld (Ed)
North Holland, Amsterdam, Neth., 1974, 213 - 217.

Wate82
'Program Editors Should Not Abandon Text Oriented Commands'
R.C. Waters
SIGPLAN Notices, 17 7 (July 1982), 39 - 46.

Wegn72
'The Vienna Definition Language'
P. Wegner
Computing Surveys, 4 1 (March 1972), 5 - 63.

WiTD76
'The Design and Implementation of a Table Driven, Interactive, Diagnostic Programming System'
T. Wilcox, A. Davis, M. Tindall
CACM, 19 (November 1976), 609 - 616.

Wirt71a
'Program Development by Stepwise Refinement'
N. Wirth
CACM, 14 (April 1971), 221 - 227.

Wirt71b
'The Design of a Pascal Compiler'
N. Wirth
Software - Practice and Experience, 1 10
(October 1971), 309 - 333.

Wirt76
'Algorithms + Data Structures = Programs'
N. Wirth
The Pascal Abstract Machine

Wood81

"Z - The 95% Program Editor"

S. Wood
SIGPLAN Notices, 16 6 (June 81), 1 - 7.

At the start of this section, reference must be made to the paper 'Aspects of an Abstract Ada Machine' [Mo82], which has been used as a reference for the introductory paragraphs of this Appendix.

The notion of the association of an abstract machine with a programming language is not a recent one, and has been credited to Dijkstra: "A machine defines (by its very structure) a language, namely its input language; conversely, the semantic definition of a language specifies a machine that understands it." [Dijkstra82]

The operational approach to programming language semantics uses this association to define a language via the specification of an abstract machine that is an interpreter for programs of the language. Several approaches to the specification of an abstract machine can be identified. One such approach is exemplified by the Ada reference manual [Wood82], using the informal tool of careful English prose. This described areas of detail of the machine which remain implicitly defined [Mo82]. A second approach involves schemes where the details of the machine are at least made explicit, even though the techniques of description remains informal. This approach includes the "two-pass" structure of many programming language texts [DeV78, Prat81]. A third approach involves the formal specification of the abstract machine in an appropriate meta-language, such as the Vienna Definition Language [Weg82] and Semantics [An84]. This technique used here falls into the second category.

1.1 Basics

The machine described here is a generalization of the von Neumann machine. The structure of the machine is specified by a schematic representation of the components of the machine, and its behavior by a
I. The Pascal Abstract Machine

This appendix is a reference section to this thesis, providing a brief outline of a Pascal abstract machine. At the start of this section, reference must be made to the paper 'Aspects of an Abstract Ada Machine' [Moli82], which has been used as a reference for the introductory paragraphs of this Appendix.

The notion of the association of an abstract machine with a programming language is not a recent one, and has been credited to Dijkstra: "a machine defines (by its very structure) a language, namely its input language; conversely, the semantic definition of a language specifies a machine that understands it" [Dijk62]

The operational approach to programming language semantics uses this association to define a language via the specification of an abstract machine that is an interpreter for programs of the language. Several approaches to the specification of an abstract machine can be identified. One such approach is exemplified by the Ada reference manual [DOD80], using the informal tool of careful English prose. This does lead to areas of detail of the machine which remain implicitly defined [Moli82]. A second approach involves schemes where the details of the machine are at least made explicit, even though the technique of description remains informal. This approach includes the 'run-time' structure of many programming language texts [OrFp78, Prat81]. A third approach involves the formal specification of the abstract machine in an appropriate metalanguage, such as the Vienna Definition Language [Wegn72] and Semanol [AnB87]. The technique used here falls into the second category.

1.1 Basics

The machine described here is a generalization of the von Neumann machine. The structure of the machine is specified by a schematic representation of the components of the machine, and its behaviour by a
textual description. Figure A-1 gives the initial state of the Pascal machine.

![Diagram of the initial state of the Pascal machine]

Figure A-1: The Initial State of the Pascal Machine

The machine structure consists of:

- An (initially empty) heap store subsystem.
- A process subsystem.

The process subsystem consists of:

- A command processor.
- A current activation pointer.
- An activation record (referenced by the current pointer).

- The text of the program (stored in a structure corresponding to the syntactic structure of the text).
The activation record consists of an instantiated activation record, which consists of:

- A data store area.
- A pointer to the structure containing the text of the program (or the sub-structure corresponding to a procedure or function).
- A current command pointer, initially referencing the first statement of the text.
- An environment pointer, referencing an activation record (initially null).
- The calling activation record pointer.

The behaviour of the machine can be specified by the top level algorithm of execution of each single statement of the text component in sequence.

The role of the subsystems, and the manner in which the structure varies dynamically can be elaborated by describing this algorithm in more detail. This involves specifying the effect of executing an arbitrary statement.

### 1.2 Operation of the Pascal Abstract Machine

In operation the machine’s process subsystem will contain a number of activation records, having the structure as shown in Figure A-1. The data store contains a number of data objects, where each object possesses a number of attributes, based on the class of the object. In broad terms, the elaboration of declarations of a block cause the creation of entries into the data store area, and the execution of statements cause the creation and destruction of activation records and the creation and destruction of entries in the heap store area.
The heap store subsystem allows the creation and deletion of dynamic objects via the NEW and DISPOSE calls on pointer variables.

Detailed definition of the actions specified by each of the particular Pascal language constructs is not within the scope of this appendix, but, of interest here is the execution of a procedure or function call.

The execution of a procedure or function call involves:

- Evaluation of the actual parameters, producing values for the value formal parameters, and pointers to the actual parameter for variable formal parameters.

- Creation of a new activation record, with the text pointer set to the sub-structure corresponding to the procedure or function name, as evaluated in the environment of the calling block. The current command pointer of the new activation record is set to the first statement of the this text, the environment pointer set to the parent activation record, and the calling activation record pointer set to the calling activation. The previously evaluated value parameters are placed in the storage area for the corresponding formal parameters, and the variable parameters are initialized to point to the corresponding actual parameters. The current activation record pointer is set to this new activation record.

- Execution of this activation, by executing the statements of the block.

- Termination of the activation (by a non-local GOTO or the more common normal subprogram termination), where the entire activation record is removed and control returns to the calling activation (by resetting the current activation record pointer to the calling activation). Termination of a function also implies the final value of the function is carried back to
the calling environment.

This structure leads to the familiar stack of activation records, as described in many programming language texts ([Prat81], for example).

It must be stressed that this is not a definitive description of a complete Pascal abstract machine, but is intended as a reference for the material presented in Chapters 5 and 7 of this thesis. The descriptions of the actions of this abstract machine given here are only partial, but it is suggested that the descriptions can be refined in a structured fashion to produce a more complete description of this machine.

Most commands are invoked by typing the first letter of the command, followed by the RETURN key. The only exception to this is the pointer movement keys, which are executed directly.

The commands that are accepted by the Workbench are:

**POINTER MOVEMENT**

- **LINE FEED**  Move the pointer to the next file.
- **UP ARROW**   Move the pointer to the previous file.
- **number**      Move the pointer to the numbered file.

**INFORMATION DISPLAY**

- **Help**       Display this text. Entering TON, where N is the first letter of a command, will display info on that command. Entering ‘Helpfilename’ will display the help file called ‘filename.hlp’ in the HELP area. In general this will give useful information about the system or package with the appropriate name. At the end of every screen of help information you have the
II. User Guide for the Pascal Workbench

The Pascal Workbench creates an environment where most aspects of the host operating system are effectively hidden from the user, creating an interactive environment more suitable to the demands of developing Pascal programs, and also takes into account the demands of novice users.

The Workbench displays a two part screen: the upper half contains the names of up to 10 files contained in the user's area, and a pointer to the current file. The lower half contains a list of all commands which can operate on the current file.

Most commands are invoked by typing the first letter of the command, followed by the RETURN key. The only exception to this is the pointer movement keys, which are executed directly.

The commands that are accepted by the Workbench are:

**POINTER MOVEMENT**

**LINE FEED**     Move the pointer to the next file.

**UP ARROW**      Move the pointer to the previous file.

**number**        Move the pointer to the numbered file.

**INFORMATION DISPLAY**

**Help**          Display this text. Entering 'HX', where 'X' is the first letter of a command will display info on that command. Entering 'H:filename' will display the help file called 'filename.hlp' in the HELP area. In general this will give useful information about the system or package with the appropriate name. At the end of every screen of help information you have the
option of returning to the Workbench, continuing with the display or printing the help file you are currently viewing.

Library
List the library directory contents.

ScreenRefresh
Rewrite the screen. This is useful if the screen becomes jumbled for any reason, as this command restores the screen contents.

DISPLAY MODES

Display
This command prompts for a display level number. Level 0 is the full display format, where the screen shows a file window of 10 files, and a five line command menu. Level 1 removes the command menu from the screen, creating a 15 line file window in the top section of the screen. Level 2 does not display a file window, but prompts the user for a command with the prompt:

FILE: <current file name>
*

The system will accept and execute any command that would have been possible in normal mode with the same file. Thus to change the current file, the response to the prompt is to press the <line feed> or <up> key.

EXITING

Logoff
Log the user off the system and save all files.

FILE CONTROL

Delete
Delete the current file.

Undelete
Restores the most recently deleted file.

New file
Create a new file. This command calls the editor to create the file (if the file name specified has the extension '.PAS')
the pascal editor is invoked, otherwise the screen editor is used). On a normal exit from the editor, the new file is entered in the file list. The system will prompt the user for a file name. The command 'N filename' will create the corresponding file without prompts.

Edit

Edit an existing file. Binary files (REL or EXE extensions) cannot be edited. Pascal files will cause the Pascal editor to be invoked by this command.

Print

Print the file. The file can either be printed on the printer or the terminal. Binary files cannot be printed. The command 'P T' will print on the terminal, and 'P M' on the main printer. The command 'P D' will print the job on the printer and delete the file from your area. (Useful for printing off large files which you can't afford to leave in your disk area). The Print command allows multiple files to be printed in the one print request by simply specifying the numbers of the additional files to print. To find the current status of print jobs you have submitted use the P ? command which will list all jobs which you have waiting to be printed.

Run

If the file is a Pascal file it is compiled, loaded and executed (the compiler is only invoked if the editor has been used to alter the file contents since the last compilation of this file).

Format

If the file is a Pascal file, format the file using the Pascal prettyprinter.

Copy

Copy the file into a new file. The user is prompted for a filename for the copy of the file. The command 'C filename' will not prompt the user for the new filename, using the
supplied filename.

Alter name
Change the name of the current file to a supplied name. File extensions cannot be altered by this command. The command 'A filename' will rename the current file to the given name.

Debug
Compile and execute the file with the dynamic debug option turned ON. This command will call the appropriate debug routine for PASCAL source files.

List
The Pascal compiler will not produce a compiler listing file by default. This command will compile the program, and produce both the object (REL) file and a print (LST) file.

<esc>
(The escape key) If the current file is a command file (the '.COM' extension), then this file is interpreted by the Workbench command interpreter. If the current file is an executable file ('.EXE' extension) then the file is executed immediately. Any following text after the escape character is interpreted as textual parameters to the execution of the file.

Background Processing

<Ctrl-B>
if the current file is a PAS file, then the file is compiled and executed in the background processing stream. If the current file is a COM file, then these commands will be processed by the background processing stream. Output generated from this command will be sent to the user specified file. If no file is given in the command line, the user is prompted for a file name.

<Ctrl-B><Ctrl-T>
will display the status of the background processing stream.
<Ctrl-B><Ctrl-C>

will abort any active background processing jobs.

This document is a user guide for the DECSystem-10 implementation of the FRED text editor, developed as a component of the research for this thesis. The user guide provides a description of the user-level environment created by the FRED editor.

### III.4 Invoking FRED

FRED can be invoked by the Pascal Workbench by the EDIT command, or the NEW FILE command. In the first case, FRED uses the current file of the Workbench as the file to be edited, and, if that file was the most recent file to be edited, will restore the edit to the edit parameter settings, and the edit position to that when the editor was previously left.

The second case will attempt to read a file name as a parameter to the NEW FILE command. If no file parameter is found, FRED will prompt for a file name, and then commence editing this initially empty file.

The second method of invoking FRED is from the DECSystem-10 command level. This task has been simplified by the modification of the operating system command interpreter to accept the command "FRED" as a valid command name. The form of command is:

```
FRED <file name>
```

where the file name parameter is optional. If the file name is not specified, then if FRED has been invoked previously, that the most recent file name will be used, otherwise FRED will prompt for a file name before commencing editing. If the file name used for this edit session is the same as that used in the most recent invocation of FRED, then FRED will perform a hot state restore of all edit parameters and edit position at the point when FRED was previously left.
III. User Guide for the FRED Text Editor

This document is a user guide for the DECSYSTEM-10 implementation of the FRED text editor, developed as a component of the research for this thesis. The user guide provides a description of the user level environment created by the FRED editor.

III.1 Invoking FRED

FRED can be invoked by the Pascal Workbench by the EDIT command, or the NEW FILE command. In the first case, FRED uses the current file of the Workbench as the file to be edited, and, if that file was the most recent file to be edited, will restore the edit to the edit parameter settings, and the edit position to that when the editor was previously left. The second case will attempt to read a file name as a parameter to the NEW FILE command. If no file parameter is found, FRED will prompt for a file name, and then commence editing this initially empty file.

The second method of invoking FRED is from the DECSYSTEM-10 command level. This task has been simplified by the modification of the operating system command interpreter to accept the command 'FRED' as a valid command name. The form of command is:

    FRED <file name>

where the file name parameter is optional. If the file name is not specified, then if FRED has been invoked previously, then the most recent file name will be used, otherwise FRED will prompt for a file name before commencing editing. If the file name used for this edit session is the same as that used in the most recent invocation of FRED, then FRED will perform a full state restore of all edit parameters and edit position at the point when FRED was previously left.
III.2 The FRED Display During Editing

On a normal entry, FRED will display the first 21 lines of the file, and position the cursor in the top left of the screen. There will also be displayed a number of status items at the base of the screen (in reverse video if your terminal supports that feature). These are, from left to right, the current edit mode, initially EDIT mode, the current line number of the edit position, and the name of the file being edited.

Edit Line 1 FRED DOC[100.22]

If the file you specified did not exist, FRED will create an empty file, and then select INPUT mode rather than EDIT mode.

While editing, the first 21 lines of the screen are the 'window' on the text. The lower 3 lines of the screen are reserved for status display and for additional prompts and arguments for commands. FRED will not allow you to position the cursor in these lower three lines unless you are entering an argument for a command.

Whenever FRED is waiting for a command the cursor will be positioned within the text on the screen. This position, the cursor position, will be used as the starting place for any commands. Thus a delete character command will delete the character under the cursor, and so on. The line display at the base of the screen will always show the line number within the text at which the cursor is positioned.

III.3 Input and Output File Conditions

There are a number of rules that FRED follows when editing files:

End of Line A line terminator is recognised as the sequence Ctrl-M, Ctrl-J (CRLF). Any other combination of these characters are treated by the editor as stand-alone control characters.

Tab characters If a TAB is encountered in the input file, then the tab will normally be expanded as spaces, to the next 8th column, in
the same fashion as DECSYSTEM-10 tab expansion. This feature can, however, be suppressed with appropriate FRED parameter in the FRED parameter file.

Other Control Characters

If a file contains other control characters, then these characters will be treated as stand-alone control characters. These characters are treated in a similar fashion to normal printable characters by FRED, but are displayed as reverse video characters on the screen. Such characters cannot be edited using the normal command set, as the control characters are FRED commands, not FRED input text. However, whenever the user enters the character pair (Ctrl^char), the input is interpreted by FRED as (Ctrl-char).

Line Widths

FRED has no restrictions on the width of a line. Note that if a line is longer than 80 characters, then, under normal circumstances, the remainder of the line will not be displayed the screen scrolls horizontally in response to cursor movements at the left and right extremes of the display.

III.4 FRED Commands

In all the following commands:

```plaintext
<name> "$<FRED command>
```
<text> stands for a character string (terminated by a RETURN)

<file> stands for a file spec (file name and extension)

<buff> stands for a buffer name (up to 9 characters)

<num> is a decimal number (optionally with a + or - sign)

There are four basic classes of commands to FRED:

1. There are commands which move the cursor and the screen over the text to be edited, i.e. positioning commands.

2. There are commands to manipulate the text in some fashion, by inserting or deleting characters or lines, or text replacement.

3. There are commands to alter the edit mode settings, such as INPUT mode or prompt display.

4. There are commands which invoke other system operations, such as compilation of the text, or performing a directory listing.

Prefixing commands

Commands may be prefixed in either of two ways. The Line feed prefix indicates that the command should be performed over a specified number of lines. The simple numeric prefix indicates that the command is to be repeated a number of times. A command may contain both forms of prefixing, in which case the line feed prefix is specified, followed by the numerical prefix.

e <num> * <FRED command>
This form of command prefix causes the command to be performed over the next <num> lines.

The following commands may be prefixed by a number. This is a decimal number, commonly interpreted as a 'multiplier' for the following command. The number is terminated by a valid command character or by a space character. While entering the number, the value can be edited by the BACKSPACE or DELETE keys. Thus for the commands 0,9,+,-,BACKSPACE, DELETE and SPACE the special case for prefixing these commands is: e <num> <space> <FRED command>.

This form of prefixing applies the command <num2> times over the next <num1> lines.

Cursor Movement

On all terminals the arrow keys are interpreted as a cursor movement command in the appropriate direction.

RETURN moves the cursor to the start of the following line. If the edit mode is INPUT, this will insert a line break at the current cursor position (splitting the current line at this position).

Ctrl-D moves the cursor to the start of the current line.

Ctrl-Y is the cursor home command. The cursor will move to the top left of the screen.

Ctrl-I is the TAB command. The cursor will move right to the next tabstop, or to the end of the line. A positive prefix will move to the nth tab position. A negative prefix will
move left to the corresponding tabstop.

Ctrl-^ will move the cursor to the start of the text on the current line.

Ctrl-_ will move to the end of the text on the current line.

**Character Editing Commands**

Ctrl-A inserts a blank character at the cursor position. All text to the right of the cursor is pushed 1 position right. Prefixing the command with a positive number inserts that number of blank characters.

Ctrl-B deletes the character under the cursor. The remainder of the line is pulled back one space, inserting a blank in the rightmost position. Prefixing the command with a positive number deletes that number of characters to the right of the cursor position to the right. Prefixing with a negative number deletes deletes from the cursor position to the left.

DEL deletes the characters to the left of the cursor. A positive prefix will delete n characters to the left of the cursor.

**Erasing in Line**

Ctrl-E erases the line from the start of the line up to the cursor, replacing all text by blank characters.

Ctrl-V is the erase to end of line command. All text following the cursor will be replaced by blanks.

**String Manipulation**

Ctrl-F <text> is the Find string command. The user is prompted for a text argument. The previous argument given for this command can be selected by entering a <CR> instead of
text. The editor then performs a forward search for this
string until end of file. If the search is successful, the screen
and cursor will be positioned over the next occurrence of
the text. If the command is prefixed by a positive number,
the search will be performed over that many lines. If the
prefix is a negative number, a backward search will be
performed over the given number of lines.

**Ctrl-R** <text1> <text2>

is the Replace command. The editor searches for the first
occurrence of <text1>, and if found, replaces that with
<text2>. The search is only carried out on the current
line. Using a positive line feed prefix, the search is
performed on the next n lines, and a negative prefix, the
preceeding n lines. The cursor will be left positioned at the
original position.

e **Ctrl-R** <text1> <text2>

is the global replace command. All occurrences of <text1>
will be replaced by <text2> from the cursor position to
the end of the file. Using a line feed prefix will perform
global replacement on the next <number> lines, and a
negative prefix will perform the replacement on the
preceeding <number> lines.

e **T** <text>

is the insert text command. The specified string is inserted
at the cursor position.

**Edit mode Selection**

**Ctrl-G**

is the mode select command. With no prefix, this returns
the edit mode to the normal EDIT setting. With a prefix of
1 this enters INPUT mode. In INPUT mode, all characters
will be inserted into the text, rather than overtyping the
original text, the RETURN character will insert a line break at the cursor position, and the TAB key will insert spaces equivalent to a tab to the next tab column.

will toggle prompt mode. Normally prompts are issued whenever arguments are needed for commands, with a negative prefix, this command will turn off all prompts, and a positive prefix will restore prompting.

e Ctrl-I <char>
is the tab set command. The character arguments are:

Z Zeros all tabstops - i.e. no tabstops
C Clears the tabstop at the current position
S Sets a tabstop at the current position
R Resets the tabstops to every 8th column
I Inserts tabstops. The tabstops are cleared, and the user is then prompted for a list of column numbers. Tabstops are inserted at these columns.

Buffer Manipulation

e G <buff> is the get buffer command. The contents of the specified are inserted into the text file. Each line is inserted into the text, following the current cursor position. Prefixing the command with a positive number will get only the first number of lines from the buffer. If a <CR> is given for the buffer name, then the buffer name from the last Get or Write command will be used. If the buffer name 'DELETE' is used, then the text deleted from the most recent delete operation will be retrieved.

e W <buff> is the write buffer command. The remainder of the line is copied into the given file. If a <CR> is given for the buffer name, then the buffer name for the last Get or
Write command will be used. If a positive prefix is given, the next <number> lines will be saved in the buffer.

On-Line Help Commands

e H  is the help command. A brief (2 page) table of commands, organised by functionality, is displayed on the screen. Following this, normal editing is resumed.

e M <char>  will display detailed help on the commands Ctrl-<char>, e Ctrl-<char> and e <char> (i.e. eMM will display help on Ctrl-M (RETURN), e Ctrl-M (ESC RETURN) and e M (ESC M) commands.

Line Commands

Ctrl-K  is the delete line command. The current line is deleted, and the following line is moved to its place. A positive prefix will delete following lines, and a negative prefix will delete the current and preceeding lines.

Ctrl-L  is the create line command. A new line is created following the current line. A positive prefix will create following lines. A negative prefix will create lines immediately preceeding the current line.

Command Control

e R  will repeat the last command, including any number prefix, and any arguments that were given to the command. This command itself may also be prefixed, to repeat the command a number of times.

Screen and Cursor Movement

Ctrl-N  is the goto line command. This command uses the prefix
number as the target line for the move. If no number is given, a 1 is used. Thus the command Ctrl-N will move the cursor (and screen) to the top of the file, while the command e 324 Ctrl-N will move the cursor (and screen) to line 324 of the file.

Ctrl-P

is the Page move command. The screen is scrolled 1 page down the text, displaying the next 20 lines of text. Prefixing with a positive value will move n pages down the text, and a negative value, n pages up the text.

Cut and Paste Command

e P

is the CUT and PASTE command. When issued, all text from the cursor position to the end of the file is placed in a 'CUT' buffer. The remainder of the file is then extended to an 'infinite' length by adding blank lines to the file. In effect the file is cut at the cursor position. This is indicated on the screen by the remainder of the text area of the screen being blanked out, and the word 'CUT' appearing at the base of the screen. When the command is issued again, all text following the cursor is deleted, and the 'CUT' buffer is restored to the cursor position. Thus this command can be used to insert and delete both lines and characters.

Tags

Ctrl-T

is the goto tag position command. Tags are identified by the prefix number to this command. If the tag number has been previously used to establish a position in the text, then the cursor (and screen) will be positioned at this tag position, otherwise no action will be taken.

e Ctrl-T

is the put tag command. The prefix number is used as the
tag identifier, and this tag is placed at the current cursor position in the text.

\( e <\text{num1}> * <\text{num2}> \text{ Ctrl-S <buff}> \)

This is the span command. If \(<\text{num1}>\) and \(<\text{num2}>\)
both define positioned tags, the all text between these tag positions is placed in the named buffer. The order the tags are specified in the command, relative to the positioning of the tags in the text, does not matter here.

**Pascal Compiler Interface**

The following commands are accepted only if the editor is editing file with the '.PAS' extension.

\( e \text{ Ctrl-P} \)

The parse command. The editor state is 'frozen', and the compiler is invoked to parse the current edit text. Once the parse is complete, the editor tags are reset according to the noted positioned of syntax errors in the text.

\( e \text{ Ctrl-L} \)

The locate error command. This command will locate, and open, the next syntax error in the edit text. The diagnostic and error number will appear at the base of the screen, and the screen will be positioned around the next error located. If an unsigned number is used as a prefix to this command, then this is interpreted as an absolute error number. If a signed number is used, then this is a locate relate to the current opened error. If a prefix value of 0 is used then the editor will reposition the display around the currently opened error position.

\( e \text{ Ctrl-R} \)

The tag corresponding to the currently opened error will be removed from the current tag set.

**Screen Restore**
Ctrl-W is the partial screen restore. The screen area following the cursor will restored. This command is useful after receiving messages from the operator, etc.

Ctrl-X is the full screen restore. This will rewrite the complete screen. This command is useful if the screen becomes jumbled.

**Word Manipulation**

Ctrl-\ will delete words from the screen. A word is defined as a sequence of non-blanks, and all following blanks. A positive prefix will delete the next n words, and a negative value will delete n preceding words.

Ctrl-] will move to the start of the next word. A positive prefix value will move to the start of the nth word, and a negative value will move back n words.

**Exit from FRED**

e F <file> is the file exit command. The user is prompted for a file name. The edit text is placed into this file, destroying any previous version of this file (if any). The editor then exits, leaving the input file unchanged, and the results of the edit stored in the given file.

e S is the Save exit command. The input file is saved with the extension ‘.BAK’ and the edit text is saved under the original input file name.

e Q is the unconditional exit command (Quit). No files are saved, and the input file is left unaltered after exit.
IV. The User Subsystem Hardware

This appendix will examine the common input and output devices used to implement, at the hardware level, the user subsystem.

IV.1 The Keyboard

The keyboard is the most common means of interfacing with the system. A 'standard' keyboard is modeled on the typewriter 'QWERTY' keyboard, where all the alphabetic characters, in both upper and lower case, the digits, and punctuation characters are sent to the system by pressing the appropriate key, or keys. This character is encoded into a bit pattern, and sent to the computing system over a communication channel. When referring to character encoding throughout this thesis, the ascii character set conventions has been generally assumed. This does have some relation to keyboard devices, in so far as the user must be able to enter all 128 ascii characters on the keyboard. The 'control' subset of 32 characters is commonly entered by pressing a control key, and the appropriate character.

There are a number of modified keyboards, designed to improve the speed of typing by using a different key layout [MeyD82, MonT82], or use of hand held devices, where a combination from seven buttons are pressed (such as the Microwriter, for example).

Many keyboards have, as well as the 'standard' character keys, auxiliary keys, or function keys, which, when pressed, send a pre-defined sequence of characters to the computer system.

IV.2 Other Input Devices

Input to the computer system can be provided by other means. A 'mouse' device is a small puck with a button, or number of buttons, fitting into the palm of the hand. The mouse will report when a button is pressed, and will also report either the relative movement of the mouse
since the last press on the buttons (on a flat surface, as an X-Y displacement), or the current position of the mouse (on a data tablet). Such devices have become common with personal workstation computing systems, such as the Apple Lisa, the Xerox Star, the PERQ workstation and the SUN micro system. Connection of such devices is either by an A/D converter, and a serial connection, or by use of a data tablet, and a serial connection.

A more advanced input device is the touch sensitive display, which will transmit when the screen is touched with a finger, and will send the location where the display was touched, as X-Y coordinates [Fins80].

IV.3 Display Devices - The Terminal Model

A glass TTY device is modeled on the earlier, and at the time very common, teletypewriter, a printing terminal. A glass TTY uses a video screen instead of a printer, and, in its basic form, adheres in all other respects to the TTY model. However common terminals (keyboard and video screen) can normally perform a number of additional functions, such as cursor positioning on the screen, selective erasure of the display, modifying the display by insertion or deletion of either lines or characters, or variable scrolling windows. These terminals have a display size of typically 80 characters wide and 24 lines deep, and possess a separate processor within the terminal to control both the screen, keyboard and the interface to the computer system. There have been a number of definitions of standards applicable to such terminal devices, defining a standard method of performing particular actions on the screen display. ANSI X3.41-1974 and ANSI X3.64-1979 are two such standards, but general adherence to this is uncommon among terminal manufacturers. Connection to the host system is commonly through a serial line, and the RS232C is a common standard for such connection. Speeds of the connection are typically of 120 characters per second (1200 baud), although connection speeds can vary between 30 characters per second (300 baud) to 960 characters per second.
IV.4 Memory-Mapped Display

This covers a wide range of video display terminals, whose common characteristic is that the screen is mapped from a dedicated portion of the computer memory. A video controller continually scans this section of memory, and constantly updates the screen display as a replica of the bit pattern in video memory. Conceptually this removes some of the common functionality of the computer system's device controller hardware and the terminal's screen controller hardware, and produces a tighter binding of the display to the computer system. The screen display can be created, or modified, at speeds of that of the memory bus, radically altering the model of the display device to that of a high speed, pixel oriented, display. The computer system can then use and manipulate text within a variety of 'soft' fonts, and can also use the display as a graphics device by addressing the video memory at the bit level. This also introduces the capability for the computer to interrogate the current state of the display. This configuration is most common in micro-computer systems, serving a single user. Screen resolution may be quite high (1000 x 800 pixels is common), and colour devices are also supported under such a model. The resolution of the screen is limited by the quality of the display screen and the amount of memory reserved for the video screen display.