A FUNCTION-KEY DRIVEN SYNTAX-DIRECTED EDITOR FOR SOFTWARE SYSTEMS DESIGN

Author: Angelo Paulo Bassanino

Student Number: 81-0321/3

Signed: 

Project supervisor: Dr.A.J.Walker

Submitted January 1986

A Project Report submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

DECLARATION

I declare that this project report is my own, unaided work. It is being submitted for the degree of Master of Science in Engineering in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Signed: _____

13th day of JANUARY 1986

ABSTRACT

Program Description Language (PDL) is a high-level design language used for both hardware and software systems design. Due to the clerical effort involved in creating such a structured program, however, the PDL design is usually bypassed, and coding performed directly. The syntax-directed PDL generator package presented here, written in Pascal for the IBM-PC, is aimed at providing a tool for producing syntactically correct PDL programs with the minimum of effort. Function keys are used extensively for specifying system inputs, and PDL keywords are inserted via construct templates. Syntactical correctness is always enforced while indentation or prettyprinting is automatic. This user-friendly PDL editor thus encourages a top-down iterative design approach while automatically performing syntax and partial semantic error detection. It is believed that this much needed tool will not only promote high-level design principles, but also serve as the basis for automatic code generation for commonly used programming languages.

for my parents
Remo and Maria Bassani.

ACKNOWLEDGEMENTS

I would like to thank the following for their material and moral support throughout the production of this thesis:

-- Dr. A.J.Walker for his dedicated help and advice both conceptually and materially.

-- The Council for Scientific and Industrial Research (Foundation for Research Development) for its financial assistance.

-- Mr K.A.Jackson and Ms D.E.Hosselson for their assistance and moral encouragement in times of need.

CONTENTS OF THIS REPORT

A SYNTAX-DIRECTED PDL GENERATOR FOR SOFTWARE SYSTEMS DESIGN

by A.P.Bassanino

Department of Electrical Engineering,
University of the Witwatersrand,
Johannesburg
December, 1985

Abstract

The need for a machine-independent language for design purposes has been recognized, and Program Description Language (PDL) has *emerged as such a tool.* The clerical effort involved in putting together and editing a PDL program, however, has proved a disincentive to using this powerful design language. This paper describes the features, operation and basic design principles of a PDL syntax-directed editor package. Indentation is automatically performed; syntactical correctness is always enforced; and extensive use of function keys is made to *facilitate program input.* The designer is thus presented with a specialized user-friendly editor for rapidly developing a PDL program in a convenient top-down, iterative manner. It is also explained how expansion of the package will lead to automated program code generation for commonly used programming languages.

INTRODUCTION

The design and documentation of large software systems has always proved a major challenge. Lately, it has become apparent that a high level design language is a useful tool for developing machine independent software. PDL (Program Description Language) is such a high level language. Its structured format and liberal use of comments make PDL not only a designer's tool, but also an effective documentation aid. Due to its descriptive nature, PDL *produces a well-defined model of the design of a project* using conventional programming concepts. PDL has already proved its usefulness in the design of both hardware and software systems. (Caine (1975), Walker (1985)

Program Description Language is an important concept as *it helps* the user to distance himself from language-specific details. PDL therefore becomes a framework for both hardware and software design. A high-level PDL may be used to describe complex system *operation via only a few comments.*

A top-down design would be undertaken in high-level PDL using general descriptive comments initially, and then gradually expanded down into a low-level PDL program. This aids the designer to view the system as a whole initially, and to slowly expand the view to include more detail, until the required implementation level is reached. The design can then be implemented in the most suitable technology. This concept of stepwise refinement iteration is convenient as it agrees to a large extent with the manner in which most designs are created. (Somerville (1984), Vosbury (1984))

PDL in its lowest-level form is similar to any modern programming language in that it makes use of *assignment, decision and looping* constructs to carry out its function. Constants and variables (collectively known as Data Items) are defined rigorously in the Data Description segment which precedes the program or procedure body. The Algorithm segment contains the program body. Indentation is maintained at all times to emphasise the program structure. Thus PDL, when consistently used, produces a well-documented, readable, top-down system design.

PDL does, however, have its problems. The manual entry of a PDL program is tedious due to the rigourous indentation requirements of the ...e. The strict ordering of the Data Description segmen... deletion or insertion of embedded constructs are examp... ...e consuming operations when editing a PDL routine using a ...ntional editor. A tool is thus needed for simplifyi... ...e process of entering a PDL program. Such a tool is known as a syntax-directed editor. (Bassanino (1985a))

THE PURPOSE OF THIS PDL GENERATOR PACKAGE

The syntax-directed editor described here knows the syntax rules of PDL. It combines the text manipulation facilities of a

general-purpose editor with the syntax or error-checking *functions of a compiler.* (Allison (1983))

A syntax-directed function-key driven PDL generator can be used to aid the designer by automating indentation requirements as well as ensuring or enforcing syntactical correctness (and partial semantic correctness in low-level PDL). A user can thus write a PDL program, being interactively warned if any language structure errors are made. This leaves the designer more time to concentrate on the true design of the system instead of being *constantly preoccupied with clerical details.*

This paper presents the features required of such a syntax-directed editor package. Some design methods adopted in its construction are also briefly discussed. This syntax-directed PDL editor is intended to be used for teaching undergraduate engineering students high level system design concepts. The editor will prove useful in teaching programming constructs; emphasizing good programming practices; and also allowing the *user to quickly learn* the rules of PDL. (Teitelbaum (1981), Garlan (1984))

The experienced user will clearly find the editor an invaluable tool for PDL generation. PDL programs can be entered with the minimum of key strokes. Detailed programs produced *by this system* will not only be correct syntactically but will also be in a standard form. This, allied with the fact that indentation is automatic, will aid both the student (in entering the program) and the lecturer (for correcting submitted designs).

The syntax-directed editor package, due to its flexibility, will encourage the development of structured designs using an iterative process. Both high and low level PDL can be interchanged in a program, *with full editing facilities* (ie. copy, move, delete and insert) being available at all times. The user will be able to separate the design thinking from implementation details by iteratively refining this PDL program. As all indentation and syntactical errors are dealt with by the system, the user will find this PDL editor very convenient compared to a conventional editor.

Ultimately, due to the code produced, this PDL tool is intended to be used as a *translator. Thus, it will be possible to target a* PDL program into any one of a few commercially available languages. Pascal, BASIC and FORTRAN are examples of such high-level software languages, while the assembler languages and hardware sequential logic can also be made available as low-level implementation target languages. Compilation and execution of the translated PDL program can then be performed using any standard language-specific compiler.

This PDL tool will therefore provide the missing link between system designing and implementation. A designer will be able to design in PDL and then decide on the most convenient implementation language. An automated walkthrough facility is also envisaged to dynamically enable the user to perform simple but effective execution-error detection and efficiency optimization analysis.

PACKAGE FEATURES

The basic structure

The designed system is aimed at minimizing the effort required to produce a syntactically correct PDL program. (Bassanino (1985b)) It is based on a dynamic set of ten function keys. Each set of function keys defines a new system state. The user will move from state to state depending on what function is required. Physically, the display screen is divided into four logical screens: the Main Screen; the Window Screen; the Prompt Screen; and the Function Key Definition Screen. (See Figure 1) This is done so as to provide the user with a constant format. In this way, as information is always presented in an orderly fashion, the user will never be confused by the information displayed on the screen.

The concept of dividing the physical screen into a number of logical screens greatly helps the user to operate the system with the minimum of fuss. (Good (1981)) A particular type of system response can always be expected in the same physical location on the screen. A prompt, for example, will only ever appear in the Prompt Screen, and the user's attention will be drawn to this line. Similarly, for editing any line, the focus of attention will be on the Window Screen. Figure 1 shows a sample PDL program being edited by the syntax-directed editor package. The logical screen partitions are also distinguished.

Function keys save the user typing time by replacing words, phrases or even constructs. The alphanumeric keys are used only when strictly necessary (ie. when entering text); otherwise , the system is completely function key driven.

The Main Screen is a 20 line screen and is used to display a portion of the formatted contents of the file. This screen contains a cursor (Cursor 1) in the left margin with vertical freedom only. Each line in the Main Screen is numbered sequentially, and system placeholders and system-generated key words are highlighted in various distinguishing video fonts.

The Window Screen is a one-line screen used to obtain responses from the user. All text lines are entered and modified via this screen; a second cursor (Cursor 2) with only horizontal freedom being available for this purpose. The user selects a line for editing in the Main Screen. This line then appears in the Window Screen for modification purposes. After modifying this line, the user can then elect to accept the new line, or revert to the old line. If the new line is accepted, the line in the Window Screen replaces the old line pointed to by Cursor 1 on the Main Screen.

The Prompt Screen represents a one-line screen used for the display of prompt or error messages to inform or warn the user. The system also makes use of a terminal bell function to differentiate between errors and prompts. Extensive error checking is performed by the PDL editor as its syntax-directed nature requires. Prompts and messages are carefully worded so that the novice user can be guided and helped along.

The fourth logical screen is used to display the definition of

the ten *function keys, and as such is known* as the Function Key Definition Screen. A single line is also needed for th's purpose, and function keys which are undefined are not displayed. This screen is updated every time a new state (with a new set of function keys) is entered.

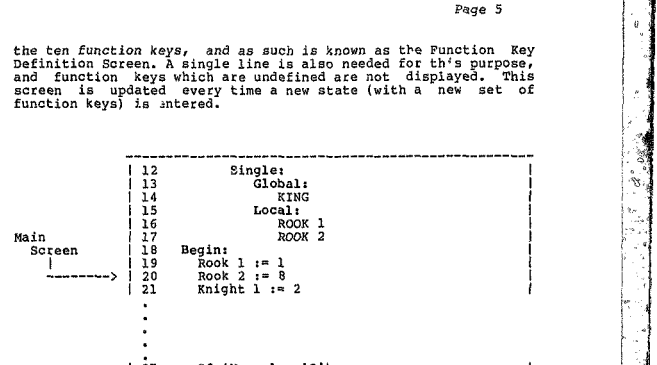


Figure 1: A Sample Editing Situation
The four logical screens are shown in abbreviated form. The two cursors are highlighted and underscored and can be seen at Line 27 (Cursor 1) and at column 10 (Cursor 2). There are ten function keys and their *function abbreviations* are displayed in reverse video font in the Key Definition Screen. Note also the highlighting of system-generated key words in the Main Screen.

At this stage it is important to *explain the various* elements available in a file. The syntax-directed PDL editor generates all standard PDL key words automatically. This is achieved by allowing construct blocks (or templates) to be inserted only as single entities. Where a user-entered text condition or statement is necessary, a placeholder is used. Thus, placeholders (enclosed by < > brackets) must be expanded by the user for the program to be complete.

For a syntax-directed editor to function as such, the template *approach described* above is an attractive one. (Bassanino (1985c)) The user is, however, limited to editing only the user-

editable text lines and placeholders. Key words cannot individually be modified or deleted by the user. Only operations on an entire construct are possible. *This prevents the occurrence* or syntactical errors during an editing session. Key words on the Main Screen are differentiated from user-editable text by highlighting. (See Figure 1) *Placeholders should be* displayed in a third font, as they essentially constitute an error of omission.

Two versions of an edited file are available. There is a coded version of the file which contains data pertaining to indentation levels, key *words, errors,* etc. This version is used by the system alone, and will not be intelligible to the user or any other editor. This is the PDL system's operating file, and it *will always be necessary* to retain it if further editing of that file may be required. This coded form of the PDL program is the file which will also be used for translation purposes in the *future.*

The PDL prettyprinted or formated file is the intelligible *version.* It contains the user-designed program as it is displayed on the Main Screen. This file need not be explicitly stored, as the system does not make use of it. The formatted file may therefore be edited using any conventional editor. This file would usually be stored for printing or display purposes.

The PDL editor, as mentioned before, consists of a number of states. After choosing the file to be edited, the user is initially placed in Base Level. Here, one has the possibility of viewing the file by means of the scroll function key options. More important, however, is the ability to be able to enter *any* of the editing modes (or states) from Base Level. If, for example, insertion of a construct is required, Insert mode must be entered. It is also from Base Level that the *PDL editor can be* exit. To date, only Insert functions have been implemented.

Editor features

As already mentioned, *the editing system operates* using the Main Screen for file viewing, while the Window Screen is used for editing an individual line. If a line is to be modified, Cursor 1 is used in the *Main Screen to choose* the required line. This cursor can be moved using the up and down cursor control keys. Indicating the line to be modified is merely a matter of depressing the *pre-selected function* key from Base Level.

If the chosen line is editable, it will now be duplicated in the *Window Screen. On displaying* the line here, the line number together with any associated indentation is omitted; these attributes only being visible on the Main Screen. The user is now *free to modify* the editable text with the use of the Line Editor features. Key words are not editable. Cursor 2 can be moved under any editable character in the Window Screen by using the left and *right* cursor control keys. When the user is satisfied with the changes made to the line, the ENTER key is depressed. This results in the new line overwriting the old line at Cursor 1 on the Main Screen. If, however, the old line is to be retained without any of the changes made to it in the Window Screen, then the ESC key will be used. (Bassanino (1985b), Bassanino (1985c))

The Line Editor is used extensively for any user-entered text input from the Window Screen. A summary of the functions available together with their meanings is shown below:

->	Cursor 2 moves to the right by one position
<-	Cursor 2 moves to the left by one position
HOME	Cursor 2 moves to the *beginning of* the line
END	Cursor 2 moves to the end of the line
CTRL K	Erases from the cursor position to the end of the line
<==	Destructive backspace deleting function
DEL	Another deleting function
INS	Toggles Insert mode on/off
ENTER	Exits the Line Editor and accepts the new text
ESC	Exits the Line Editor ignoring any *modifications*

Individual line editing is performed using the above method. If, *however, an operation is required on* a line or a block of lines, then Cursor 1 is used in the Main Screen.

For the purposes of viewing any 20-line portion of the PDL file on the Main Screen, extensive scrolling functions are provided in the Base Level. Cursor 1 can be moved up and down by using the up and down cursor control keys. If the cursor is moved beyond the Main Screen limits, *a half-page scroll will occur.* Full page forward and backward scrolling is also available. The top and bottom of a file can be accessed via a single function key depression. The user can also choose a line number where Cursor 1 is required. A summary of the file scroll functions is given below:

↑	-- Cursor 1 up by one line
↓	-- Cursor 1 down by one line
PgDn	-- Page scroll forward
PgUp	-- Page scroll backward
Top of File	-- Cursor 1 to the top of the file
Bot of File	-- Cursor 1 to the bottom of the file
Cur to Line	-- Cursor 1 to a specified line number

Each of the file editing features of the package provides a new *mode from Base Level. The four macro modes* are:

-- Insert Mode
-- Delete Mode
-- Copy Mode
-- Move Mode

Each of these *modes will allow for single* line and block operations. Thus each mode will in turn have its sub-modes which will give the user the necessary functions. These editing features will be described later.

Data Description segment facilities

Before describing the features available for the manipulation of the Data Description segment, it is useful to understand how this *segment is comprised.* In PDL, every data item (known in programming terms as a constant or a variable) is defined according to four characteristics. These characteristics together with their possibilities are listed below:

Function	Type	Structure	Scope
Constant	Boolean	Single	Global
Variable	Integer	Array	Permanent
	Real		External
	Character		Local
	Others		

The various possibilities are almost self-explanatory, but it must be appreciated how data items are described in the Data Description segment. Figure 2 shows an example of such a description.

```
Variable:
    Integer:
        Single:
            Local:
                INPUT A
        Array:
            Local:
                INPUT B (of size 10)
```

Figure 2: Example of a Data Description segment
The data item INPUT A is classified as a local, single integer variable, while INPUT B is a local array consisting of ten integer variables.

It is clear that there is great scope for automation in the insertion and deletion of a data item. The PDL syntax-directed editor makes use of the ten function keys available to allow the user to choose the possibilities specific to a particular data item characteristic. Thus, insertion of a data item is performed as follows: (Bassanino (1985b))

Firstly, the user enters the data item name in the Window Screen. Then, the Function characteristic possibilities are displayed as function key options. When the user has chosen the desired possibility (Constant or Variable), the Type characteristic possibilities (boolean, integer, real, etc.) are displayed. Thus, the user can define the data item characteristic by characteristic until finally the data Scope is defined.

The definition of a data item is flexible in that the user can edit any chosen key word, and even abandon the definition entirely before it is accepted by the system. During Data Item Definition mode, all key words are temporarily displayed in the Window Screen. Only when a data item has been fully defined and accepted will it be positioned in the Data Description segment of the PDL file.

This data item positioning is automatically performed by the system so that only the necessary key words are added to the file. All standard indentation requirements are also automatically satisfied. Thus, the user need never be concerned with the structure of the Data Description segment. Also, there is no possibility of incorrect or incomplete data item definition. This automated facility can be extended to store the

attributes associated with all data items in a data item table. This table can then be used to check for type compatibility and thus semantic errors in the Algorithm segment of the PDL program.

The user is only permitted three operations on the Data Description segment: insertion, modification and deletion. The Insert facility has been described above. The Modify function enables the user to edit the data item name via the Line Editor in the Window Screen. The Delete function restricts the user to manipulating only data items and not their key words.

The Modify function has been described earlier. It is also used for editing any other user-editable text line individually. As lines containing key words are not editable, only the user entered data item name may be altered.

In deleting a data item, the user positions Cursor 1 in the Main Screen on the line containing the data item name which is to be deleted and requests a line delete. Any associated key words are then automatically deleted from the file together with the data item name. Key words may not be tampered with: any attempt to delete them will result in an error message.

A block of *sequentially defined* data items may be deleted using the Block Delete function. This function will allow the deletion of all data items which lie within the chosen block. The block to be deleted must start and end on a data item name for it to be accepted. The data item names included in the block, together with any relevant key words are automatically removed from the file; the remaining Data Description segment being arranged accordingly.

Automation of Data Description segment manipulation can be seen to be highly effective. A large amount of clerical effort is saved due to the function key definition method as well as the automatic placement feature. The user is thus able to define a data item while in the Algorithm segment without having to move to the Data Description segment. *Finally, the basis for semantic* error checking is also provided by the automated system described above.

Algorithm segment facilities

The Algorithm segment contains the statements and constructs which constitute the program body. The key to enforcing syntactical correctness (ie. *no end-of-constructs missing*) is to prevent or disallow syntactical errors. (Teitelbaum (1981)) Thus, a great deal of checking is done by the system to ascertain whether any editing operation requested will still leave the program syntactically correct if performed.

Besides the Modify function, four modes are defined here: Insert, Delete, Copy and Move modes. (Bassanino (1985b)) The Modify function has already been described in the section relating to editor *features. It gives the user the* ability to modify the user-editable section of any line. Thus, placeholders can be expanded and lines modified in the Window Screen, making full use of the Line Editor facilities.

Single line insertion is permitted at almost any location in the

Algorithm segment. This facility is used for inserting PDL statements or comments. Insert Line mode will allow the user to enter a series of lines sequentially. This mode is exit using an appropriate function key. When in this mode, lines on the Main Screen which appear after the line at which insertion is occuring, will not be displayed. Cursor 1 will also not be prosent on the Main Screen.

At this stage it should be pointed out that although this feature is not yet implemented, parsing of each line entered via the Line Editor should be performed. When parsing a line, the PDL grammar can be checked dynamically and any semantic errors flagged. Among the tests that can be performed are: type compatibility; distinguishing between assignment statements and conditions; and checking for illegal (ie. user-entered) key words. As all key words are system generated, they may not be user-typed.

Constructs are inserted as a block or template. This ensures that syntactical correctness is maintained. In all Insert modes, indentation is automatic. An example of an If-then-else construct template can be seen in figure 3. Placeholders represent portions of the construct that must be filled in by the user. With the use of a single construct function key, the user is able to choose any construct template. The templates available are listed below:

```
-- If-then                          -- Case-else
-- If-then-                         -- Cobegin-Coend
-- While-d                          -- Get
-- Repeat-until                     -- Put
-- Case
```

```
If <CONDITION>
  then:
    <STATEMENT>
  else:
    <STATEMENT>
End if:
```

Figure 3: An If-then-else construct template

The template consists of key words (such as "else:") and placeholders (such as <STATEMENT>). Placeholders must be expanded by the user, while key words are not user-editable.

Deletion is also restricted to constructs. Single line deletion is, however, allowed on lines which have been entirely user-entered (ie. have no key words). Construct deletion occurs when an entire construct is removed with the user indicating the construct start line. Block deletion is also possible, but the chosen block must not contain any unterminated constructs.

The user is able to choose a line by one of two methods. A line can be pointed to on the Main Screen with Cursor 1 and a function key used to choose it or a numerical line number in the Window Screen can be specified. Before deletion, the lines chosen are highlighted, and the user is asked to confirm the operation. Placeholders which are outstanding after any Delete operation

will automatically be inserted.

The user has the ability to perform either a single line or a block copy function while in Copy mode. Again, line choice is carried out using one of the two methods described above. The choice of copy block must comply with the persistent law of maintaining the program syntactically correct at all times. In this case, the destination line is also of importance and must be checked for acceptance. Here too indentation is automatic, and any superfluous placeholders which remain after a Copy operation will be removed.

Similarly, a Move function is available for single lines as well as for blocks. With its automatic indentation and friendly user-interface, the user will find the PDL syntax-directed editor an invaluable tool for putting together and editing a PDL design in as short as possible a time period.

It is envisaged that the following functions will eventually be incorporated into the package. (Bassanino (1985b)) A function for obtaining information regarding any particular error chosen on the Main Screen can be provided together with an on-line help facility for the novice. An ellipsis feature for elliding (or temporarily removing) blocks of text so that the outer program levels can be displayed together in the Main Screen, will prove useful. This facility allows the user to effectively view more than 20 lines at a time by removing program details in the form of deeply nested constructs. An "undo" stack is also a useful feature when dealing with peculiarities of syntax-directed template-based editors. (eg. converting a While-do construct into a Repeat-until construct)

PACKAGE DESIGN CONSIDERATIONS

The functions described above have not all been implemented (only Base Level and Insert mode are fully functional), but in building the package, several sound design principles have been adopted which will aid in rapidly expanding the package to its full potential. (Bassanino (1985c))

To make the system as externally programmable (or flexible) as possible, a series of tables have been used. These tables contain specific system operation information such as lists of key words, prompts and next state . It is thus possible, to a large extent, to modify system behavior by modifying the system tables. These tables are file based and are loaded on initialization. Although this is not the objective, these features also make the system adaptable to act as a syntax-directed editor for any other PDL-like language.

The package is designed with a high degree of software design discipline. The system tables are designed as modules, accessible only via certain routine calls. These modules essentially consist of a data structure (known as a resource) surrounded by operators. (Walker (1984)) Separable package features are also designed as modules in such a way that a sound program structure emerges. (Myers (1975), Shankar (1984)) The modules designed can be separately tested via specially written test programs. This allows the designer to test or experiment with any possible

operation on the resource before it is included in the package. This also means that the routines are portable as they were designed as stand-alone modules. Further, a terminal dependant resource with a variety of access operators ensures that the rest of the package is completely terminal independent.

Due to its high structural strength and modularity, the package is easily expanded. Delete, Copy and Move modes can simply be designed separately and integrated into the final package. Extended features are also easily incorporated, while changes in the PDL language are accommodated by modifying the system tables. (Bassanino (1984c))

CONCLUSIONS

Program Description Language (PDL) is a useful, flexible high-level language for describing a design without commitment to any particular implementation technology. A top-down approach is enforced as comment statements can be expanded into algorithm detail. Standard PDL constructs and strict indentation is strongly relied upon to produce a readable PDL program. This very feature of PDL, however, requires the user to spend much design time for manual indentation, and this may detract from the purpose of the language of providing an effective design tool.

A syntax-directed PDL generator is an effective solution to the above problem. The package designed is template based, so that all constructs and key words are system-generated. Syntactical correctness is thus enforced by disallowing incomplete constructs. The system is function key based, thus ensuring maximum efficiency. All editor operations are subject to acceptance testing so as to ensure that no syntactical errors occur. Limited semantic checking can be included for low-level PDL programs. All indentation requirements are automatically dealt with.

This syntax-directed PDL tool is seen as an indispensable tool for the designer. The package can be put to good use in a teaching environment: the student will quickly learn the rules of PDL while the lecturer will be presented with consistent and syntactically correct designs. The clerical effort of typing key words; ensuring correct indentation; and for checking syntactical correctness is eliminated with such a tool. This allows the user more time for designing. The package encourages structured design development using an iterative approach and helps separate design thinking from implementation issues. Step by step top-down design documentation is thus also enforced.

The package has been designed to be as programmable as possible by using a table-based function-key driven approach. Extensions and modifications are easily accommodated due to the highly modular package structure. The system may be modified to act as a syntax-directed editor for a variety of PDL-like languages. The coded program produced by the PDL generator, however, is to be used for translation purposes into the commonly known software languages. The system is also seen as a basis for automated walkthrough facilities. (Chesi (1984), Feiler (1981)) In conclusion, this PDL syntax-directed editor system is destined to become the major tool on the future system designer's work-bench.

REFERENCES

1. Allison,L. (May 1983): "Syntax directed program editing", *Software Practice and Experience (GB)*, Vol.13, No.5, pp.453-465.

2. Bassanino,A.P. (1985a): A Function-key driven Syntax-directed Editor for Software Systems Design, "Literature survey", a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

3. Bassanino,A.P. (1985b): A Function-key driven Syntax-directed Editor for Software Systems Design, "User's Manual", Version 1.0, a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

4. Bassanino,A.P. (1985c): A Function-key driven Syntax-directed Editor for Software Systems Design, "Designer's Reference", Version 1.0, a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

5. Caine,S.H. and Gordon,E.K. (1975): "PDL -- a tool for software design", Proceedings of the National Computer Conference, 1975, pp.271-276.

6. Chesi,M. ,Dameri,E. ,Franceschi,M.P. ,et al (May 1984): "ISDE: An interactive software development environment", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

7. Garlan,D.B. and Miller,P.L. (May 1984): "GNOME: An introductory programming environment based on a family of structure editors", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

8. Feiler,P.H. and Medina-Mora,R. (September 1981): "An Incremental Programming Environment", IEEE Transactions on Software Engineering, Vol.SE-7, No.5, pp.472-481.

9. Good,M. (June 1981): "Etude and the folklore of user interface design", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.34-43.

10. Myers,G.J. (1975): Reliable software through composite design, Pertocelli/charter, New York, 1975.

11. Shankar,K.S. (1984): "Data Types: Types, structures and abstractions", Chapter 12, Handbook of Software Engineering, edited by Vick,C.R. and Ramamoothy,C.V., Van Nosrand Reinold,

REFERENCES

1. Allison,L. (May 1983): "Syntax directed program editing", Software Practice and Experience (GB), Vol.13, No.5, pp.453-465.

2. Bassanino,A.P. (1985a): A Function-key driven Syntax-directed Editor for Software Systems Design, "Literature survey", a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

3. Bassanino,A.P. (1985b): A Function-key driven Syntax-directed Editor for Software Systems Design, "User's Manual", Version 1.0, a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

4. Bassanino,A.P. (1985c): A Function-key driven Syntax-directed Editor for Software Systems Design, "Designer's Reference", Version 1.0, a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

5. Caine,S.H. and Gordon,E.K. (1975): "PDL -- a tool for software design", Proceedings of the National Computer Conference, 1975, pp.271-276.

6. Chesi,M. ,Dameri,E. ,Franceschi,M.P. ,et al (May 1984): "ISDE: An interactive software development environment", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

7. Garlan,D.B. and Miller,P.L. (May 1984): "GNOME: An introductory programming environment based on a family of structure editors", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

8. Feiler,P.H. and Medina-Mora,R. (September 1981): "An Incremental Programming Environment", IEEE Transactions on Software Engineering, Vol.SE-7, No.5, pp.472-481.

9. Good,M. (June 1981): "Etude and the folklore of user interface design", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.34-43.

10. Myers,G.J. (1975): Reliable software through composite design, Pertocelli/charter, New York, 1975.

11. Shankar,K.S. (1984): "Data Types: Types, structures and abstractions", Chapter 12, Handbook of Software Engineering, edited by Vick,C.R. and Ramamoothy,C.V., Van Nosrand Reinold,

1984.

12. Sommerville,I. (1982): Software Engineering, Addison Wesley International Computer Science Series, 1982.

13. Teitelbaum,T. ,Reps,T. and Horwitz,S. (June 1981): "The why and wherefore of the Cornell Program Synthesizer", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.8-16.

14. Vosbury,N.A. (1984): "Process Design", Chapter 25, Handbook of Software Engineering, edited by Vick,C.R. and Ramamoothy,C.V., Van Nosrand Reinold, 1984.

15. Walker,A.J. (1984): Structured information processing system design, Internal publication of the Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1984.

A FUNCTION-KEY DRIVEN SYNTAX-DIRECTED EDITOR FOR SOFTWARE SYSTEMS DESIGN

L I T E R A T U R E S U R V E Y

December 1985

Author: A.P.Bassanino

Signed: 

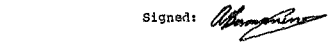
A Project Report submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

A FUNCTION-KEY DRIVEN SYNTAX-DIRECTED EDITOR FOR SOFTWARE *SYSTEMS* DESIGN

L I T E R A T U R E S U R V E Y

December 1985

Author: A.P.Bassanino

Signed: 

A Project Report submitted to the Faculty of *Engineering*, University of the Witwatersrand, Johannesburg *in* partial *fulfillment* of the requirements for the degree of Master of *Science* in Engineering.

CONTENTS

1 THE HISTORY AND DEFINITION OF EDITORS

The editor is one of the most used tools today on an interactive computer system. Line oriented editing is an early form (late 1950's) of text editing usually associated with punch-cards of fixed or variable length. IBM's CMS editor is such an example. Stream editors such as TECO solved the problems of truncation and interline edit experienced by line editors by regarding the entire document as an *infinitely long chain or string of* characters.

The 1960's saw the development of the first basic editors using a TV monitor. Already in 1965, function keys were used in one of the earliest time-sharing CRT based text editors known as TVEDIT. This system, designed at Stanford University, California, offered control functions for inserting and deleting, with facilities for text paging. Due to the high cost of CRT terminals, practical progress at this time was slow.

One of the first classic surveys on text editing in general can be found in Van Dam (1971). Here, on-line editing is established as useful and cost-effective in debugging. An example of an ancient editor can be seen in Irons (1972). This is a line editor which uses the in-built terminal functions and it is interesting to note the curious names given to the most common operations. (The file is compared to a pack of cards, and operations such as *pull, pick and put can be performed.*) *A more recent and very* thorough survey on text editors can be found in Meyrowitz (1982).

In 1974 the first truly useful word processors started to appear. Before this time, editors and word processors had a fuzzy dividing line. A simple text editor is typically used to create or modify a computer program or text file via the use of basic commands such as: delete; move; insert; etc.. Text editors are generally divided into two categories: line editors; and screen *editors. The line editor allows the user to edit only one line at* a time, while the screen editor will permit the editing of a file at any cursor position on the screen. A word processor however can be defined as a sophisticated editor for the production of formatted documents. It includes such attributes as: highlighting; various size lettering; paragraphing; etc.. WordStar (MicroPro 1981) is an example of a popular modern word processor.

Display editors based on the Irons conceptual model (Irons (1972)) essentially constitute the majority of full-screen editors today. Among such editors, we find: PEN (Barach (1981)); % (Wood (1981)); eds (Fraser (1981) ; EMACS (Stallman (1980), Stallman (1981))); and IBM's XEDIT (IBM (1980)). In Irons' model, text is conceived as a quarter-plane with the origin at the top leftmost character and extending infinitely in length and width. The user travels through the file using the cursor keys and changes characters by overtyping. At all times an accurate representation of the displayed file portion is visible. The environment is considered 'modeless': all typing is considered as

text; commands are given via function keys, control characters, escape sequences or by typing in a specified command area in the screen.

Graphics-based interactive editors such as Xerox PARC's Bravo appeared in the mid 1970's. These editors (eg. ETUDE (Hammer (1981))) usually require a high resolution CRT.

Structure editors are character oriented to exploit the natural ordering of a document. (Fraser (1981) and Stromfors (1981)) A Language Based Editor (LBE) makes use of the inherent laws of a language to structure a document and detect any language related errors. The document need not only be a program, but could also represent a binary or graphics file, or a letter or manuscript with subdivisions of chapters, sections, sub-sections and paragraphs. The most common representation used by a language is a hierarchical one.

Syntax-directed editors are used specifically for editing computer programs. They aim to relieve the programmer of the time-consuming task of eliminating syntax errors. Syntax editors are sophisticated structure editors which ensure that on input, syntactic integrity is preserved. Often, these editors will also parse the input into an intermediate tree form that can be used to generate code. Most syntax editors are table driven so that potentially, several languages can be manipulated. Among the first syntax-directed editors are Hansen's EMILY (Hansen (1971)) designed for PL/1; and LISPEDIT written for LISP programs. A comprehensive summary of such edito-s is given in chapter 3.

2 FACTORS INVOLVED IN SYNTAX EDITOR DESIGN

The first section of this chapter gives an idea of the terms used in the literature as well as explaining the pros and cons of using and designing syntax-directed editors. The second lists some ideas and conventions of editors in general with particular attention paid to syntax editors. In the third section the much-spoken-of user interface is considered; many of the important findings in the literature are discussed.

2.1 The general concepts

Syntax-directed editors are editors which know and use the syntax of the language while a program is being edited. A language directed editor combines the text manipulation functions of a general purpose editor with the syntax-checking functions of a compiler. These editors provide an environment which increases the productivity of both beginning and experienced programmers. For the beginner, all the syntax of a language need not be remembered when writing a program. All programmers benefit by the typing time saved and the immediate detection of syntax errors. As indentation and prettyprinting are automatic, programs written using syntax-directed editors are well formatted, readable and syntactically correct.

Many believe in the power of syntax-directed editors. A few such arguments are detailed below. Teitelbaum (June 1981) states that because the user is able to distance himself from the syntactic details of a program, program conception at a high level of abstraction is stimulated, and programming by stepwise refinement is promoted. Meyrowitz (1982) claims that the specification of target data as well connected, well defined units enhances the user's powers of creativity and composition. Syntax-directed editing may change the way that programming is taught and described according to Notkin (1979). Frustrating details such as the placement of statement delimiters (eg. semicolons in Pascal) can be eliminated entirely by the use of templates.

Syntax editors which parse the input into a tree structure are used both as a tool for the programmer and a tool for the compiler. The advantage to using the tree structure is that it is easy to add and delete branches from the tree and when changes are made, the entire tree does not have to be reparsed.

On the other hand, there are those who have not only pointed out the difficulties with this type of editor, but in fact disagree with its basic concept. Wood (1981) for example claims that 95% of editing can be done on a standard editor. He states that syntax-directed editors constrain the user interface complicating normally easy to understand operations. This approach promotes a multitude of editors. He further argues that the representation and editing of a program as a parse tree makes an editor more difficult to implement.

Although many of his points are valid, it must not be forgotten

that syntax editors can also create parse trees, and thus eliminate the need for parsers and compilers. Program trees are however very space intensive requiring on average hundreds of bytes per source line as Fischer (1984) correctly points out. Although parsing consumes processing power and parse trees devour storage space (Morris (1981)), these resources are rapidly becoming more powerful and cheaper today.

The above gave a general introduction to the motivation behind and the issues involved in syntax-directed editors. What follows is a comprehensive list of terms used in the related literature together with their explanations and some associated arguments.

Template --- is the name given for a formatted syntactic skeleton that contains the keywords and punctuation marks of the given statement form. A template includes a placeholder at each position where additional code is required to complete the statement. These act as prompts to the user. Phrases are assignment statements, expressions and variable lists.

Syntax trees --- consist of terminal nodes (leaves) representing variables, constants, static language elements (eg. data type names) and unexpanded program constructs. Figure 1 shows an example of the tree structures adopted by the Cornell Program Synthesizer (CPS) as compared with that used by SED (Allison (1983)). The tree structure of SED contains phrases so that its hierarchy stops at simple statement level. A parser and deparser is used to transform text into a syntax tree and vice versa.

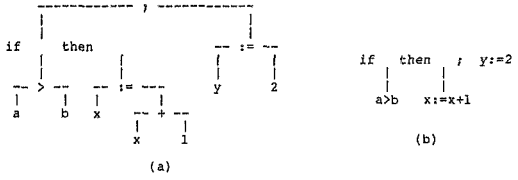


Figure 1: Examples of various Tree Structures
Figure (a) shows the structure used by the CPS, while (b) shows the system used by SED.

Nonterminal nodes --- describe subtrees of a program corresponding to control flow constructs and data definitions in the language (eg. If-then-else construct in Pascal). Information available at each node includes the type of language construct, and references to the parent node and to its offspring. A node can have a fixed or variable number of offsprings. Meta-nodes are best described by 'holes' in the program templates that have not

been expanded.

Short distance syntax --- is defined for error correction ability by Allison (1983) as the syntax of a construct. Long distance syntax involves variables or declarations which can have distant effects on the program. (ie.the program semantics) This is the most difficult error type to check for during editing, as a small change to a declaration can have remote effects on type compatibility. In incremental editing systems, long range errors are left until execution time (eg. Pathcal (Milander (1980))). If checking is done for long range errors during editing as with CPS, validity needs to be relaxed in certain operations.

According to Allison (1983) changes to a program can be implemented in three ways. Structural commands as used in CPS delete, copy, insert and move subtrees. Tree matching and substitution looks like string replacements to the user and this method is adopted in SED. Alternatively, arbitrary textual changes can be allowed to a program as in the CAPS system. This method is known as text editing.

Ellipsis or holophrasting --- is used to abbreviate long sections of code so that they can be displayed on the 24-line VDU screen. Thus a view of the entire program can be obtained and a zoom function can be used to display the details of a particular section. The CPS for example allows the user to label a section of code with a comment before elliding it. (Teitelbaum (June 1981))

Prettyprinter --- is the name given to a software tool used to output text in a structured format on a VDU or printer. Rubin (1983) combines a prettyprinter and a syntax-directed editor to form the idea for a language-independent software development system. For more details on prettyprinting, the following references should be consulted: Nikelsons (1981); Oppen (1980). Teitelbaum (June 1981) distinguishes between monomorphic and polymorphic prettyprinting: the former automatically prettyprints everything which is entered with no attempt made to keep all the text onto the screen (eg. CPS); while the latter (eg. LISPEDIT) prettyprints only in the vicinity of the cursor in an attempt to keep text onto the screen.

Integrated systems --- constitute a set of tools that support program creation, modification, execution and debugging. This means that a user does not have to perform mental context switching between say modifying and debugging a program. An incremental system, however, is a system where immediate execution is interleaved with editing so that, for example, a user can run parts of a program editing any errors which occur; after these errors have been corrected, running of the program can be resumed.

If there is disagreement in the literature on a certain aspect of syntax editors, it must be the generator versus recognizer argument. A summary of both sides of the coin is given below:

The generator approach (sometimes known as programming by selection) usually makes use of templates so that only valid programs can be generated. Programs are created top-down by inserting new templates and phrases within the skeleton of previously entered templates. Syntax error detection is

immediate. Correctness is maintained at all times by preventing the entry of syntactically incorrect programs.

The *generative or error prevention mode* is best suited to development environments and parsers are usually not needed. Typographical errors are possible in user-typed phrases and not in system-supplied templates. Examples of systems based on this method are: the Cornell Program Synthesizer (Teitelbaum(1981)), EMILY (Van Dam (1971)), ISDE (Chesi (1984)), POE (Fischer (1984)), and SUPPORT (Zelkowitz (1984)).

The recognizer approach augments normal editing facilities with lexical, syntactic and semantic analysis to detect any errors. In this way, the user is not constrained to the editor's templates. *Morris* (1981) for example treats everything before the cursor as syntactically correct so that an If-then statement say can be changed to a while construct without much inconvenience. This editing function presents a problem in the generative approach. The recognizer method is consistent and permits arbitrary editing operations on a program while program modification is greatly simplified. Examples of systems which use this method are: *Magpie* (Delisle (1984)), MENTOR (Donzeau-Gouge(1984)), SAGA (Campbell (1984)), Syned (Horgan (1984)), and Z (Wood (1981)).

The generative approach is perhaps more suited to program entry; not program editing. Program by selection may appeal because of its similarity to structured programming's stepwise refinement, but stepwise refinement was developed for creating algorithms; not for entering programs. It is a widely held view that it is bad practice to compose a program at a VDU. Allison therefore concludes that it is doubtful whether programming by selection is a good thing. (*Allison (1983)*)

2.2 Editor functions

Full-screen or display editors operate on the "what you see is what you get" concept with modifications made at the cursor position (cf. chapter 1). These environments are by their nature very comfortable to use. An example of such a system where there is no text mode to enter or leave is given in the MINCE editor (*Moore (1981)*). A *powerful editor should* include many of the following functions:

Cursor movements :- up; down; left; right; to the start or the end of the current line; to the beginning or end of the next line or tab stop. Cursor movement should be possible by character, word, line, sentence, construct, and by screen. Access to the top or bottom of the file should also be readily possible. A horizontally and vertically scrolling screen is ideal, so that all the text can be viewed.

Backspace and delete keys should be provided, and their functions clearly stated. (eg. Thompson (1981) defines the backspace as moving the cursor to the left and deleting the character, while the delete function will delete a character without moving the cursor.) Settable tab or indent and unindent attributes are also desirable.

The undo or restore function is very useful for program

protection. (It can be regarded as a stack: last-done-first-undone). Move, insert, copy, and delete are standard editor functions. Fischer (1983) uses a cursor to pass over the portion of the text that must be copied. Function keys can be assigned to these operations, so that a delete key, say, will be depressed when the line pointed to by the cursor is to be deleted.

String search and string substitution commands prove a time saving feature for the expert user. When a match is found, the cursor can appear at the first match, so that the user can return for another match, or escape. Obviously, function keys could be assigned to most if not all of the above functions, but this does have its disadvantages as will be seen in the next section.

Zelkowitz (1984) states that about 80% of the time spent on an editor is used for editing or maintaining a program as opposed to entering it. This implies that a syntax editor is to be efficient in performing changes to a program after it has been entered. Desirable functions for this type of editor include: syntax error detection or prevention; error correction (this point is debatable); long range error detection by keeping track of data types and checking for incorrect assignments; ellipsis facilities; abbreviations for verbose constructs; prettyprinting; and parsing and deparsing.

2.3 The user interface

There have been a great deal of papers published on the issue of user-friendliness, and this can only be attributed to the low quality of commercial software packages available today. A good idea of the issues involved is presented below but for further details, the following papers and their references should be consulted: Jong (1982); Raduchel (1984) ; Good (1981); and Meyrowitz (1982).

The user interface should present a well defined, consistent conceptual model with the user being familiar and comfortable with the philosophy behind the system. It should be clear and concise, easy to learn and use, and it should provide consistency across different targets (Meyrowitz (1982)). Current technology does a poor job of telling the user what to do as opposed to how to do it. An inefficient editor with a smooth interface is better received and more useful than an efficient editor with a badly designed user interface. Software should be designed and selected not on the basis of what is most machine efficient, but on how well people can use it.

A method for determining user-friendliness is described in Raduchel (1984). A system can be said to be user-friendly if $F > F_t$ in the equation:

$$F = P_o p^n$$

where F_t is the threshold probability value
- P_o is the probability that a user will find a set of steps to solve a problem
- p is the probability that a user can successfully execute each step
- n is the minimum number of steps in the solution

P_o falls as n increases, and p generally increases as each step is made smaller. A user will eventually consider a system with fewer but more complex steps to be the more user friendly. p has an upper limit due to human error (of the order of 0.995). If problems to be solved are not simple, it is unlikely that any general mass-market can be user-friendly.

A system that is easy to learn may not be easy to use. As an example, prompting for a series of steps in a standard operation is easy to learn, but more time-consuming to use. Conversely, a macro system requiring only one command for the entire operation may be easy to use, but not easy to learn. There is thus a trade-off between the power offered by the environment and the ease of learning the system.

The idea of "idiot proofing" found in Jong (1982) is a good one. It is based on anticipating user errors such as: incorrect entries; missing inputs; and inadvertent keystrokes. Deletions larger than a single character should be stored in a stack to be retrieved in case of error. This protects users from themselves. To prevent inadvertent escape or abort routines, a combination of two remote keys should be reserved (eg. CTL-X for quitting in EMACS and MINCE).

For the commands which will have a drastic effect on the file edited or which result in irreversible procedures (eg. block deletes and copies), user confirmation should be requested. A cancel or reset key is a necessity so that long operations can be aborted if so desired. While on this subject, it is important to notice that system speed in interactive environments must be maximized. Good (1981) states that execution time for all operations should be kept below two seconds for acceptability. An absolute maximum of 15 seconds should be imposed for the longest computations.

According to Meyrowitz (1982), an "infinite" undo and redo capability should be provided so that the user can experiment with the system without loss or damage to a document. There are several ways of achieving this. Peck (1981) gives the author access only to a copy of the original file. Alternatively, the most recent keystrokes can be stored. When an undo operation is requested, the latest operation is retrieved from the stack and its inverse operation (found in tabular form perhaps) is performed on the file so as to leave the user with the file before the undesired operation was executed. The ETUDE editor (Good (1981)) displays a list of previous commands for the undo function.

The choice of prompts and messages can greatly influence the degree to which an editor will be accepted. Jong (1982) suggests that messages should be : "polite not imperious; straight not funny; neutral not personal". Although computers have been in use for about thirty years, there are still those who feel threatened by the computer. For this reason, a system should not give the impression of the computer being the dominating person. The user should always feel that he is the master of the computer and not vice versa. For example, a prompt for the next command should rather read "Ready for next command" instead of "Enter next command".

Error messages should tell the user: what went wrong; what has

happened as a consequence; and how to correct the error. When the system is busy and the user is waiting for a long process to be completed, a message to this effect should be presented. Comments should be brief, factual and informative without being abbreviated, humorous or folksy (Good (1981)).

The use of various fonts such as highlighting, underlining and reverse video are very useful to the user in operations such as moving, copying and deleting blocks. The Bank Street Writer editor for example (Lewis (1984)) highlights one string match at a time and asks the user for confirmation of a replace function. The ETUDE editor lights up sections of a file which have been selected for copying or erasing. Reverse video is quoted as the best form of contrasting, while blinking should be used sparingly as it is highly distracting, especially for long messages. Mikelsons (1981) uses different fonts and colors to distinguish keywords from identifiers.

Although the use of programmable function keys is an attractive one, certain operations may be too important for only one keystroke. Both Good (1981) and Jong (1982) agree that major operations or operations that are not always required but can be inadvertently called via a single function key, should not be assigned to function keys. The user should rather be prompted and the whole command typed out.

For both the novice as well as the experienced user, a help facility always proves useful. Documentation, both on-line (in a system supplied help facility) and off-line (in manuals) explaining the conceptual model, user interface and system functions should be provided. The authors of MINCE (Moore (1981)) suggest the code-card approach: a two-sided card with a summary of the functions available and the commands to execute them is given to the user. The experienced user will have a detailed card, while the novice is provided only with a basic set of commands. Bank Street Writer uses an accompanying tutorial to the novice with the functions available on the editor.

The designers of GNOME (Garlan (1984)) have had extensive feedback from students regarding their syntax-directed editor. Difficulty was experienced with the number of hierarchical levels: too many levels lead to confusion as the user gets 'lost' quickly; too few levels results in overcrowding of menus and also proves confusing. The list below accounts for 90% of the errors made by students.

--- undeclared variables
--- variable was declared but not used
--- uninitialized variable
--- type mismatch

Good (1981) suggests that validation of data be done when the data is entered; assuming data or attempting to correct errors usually leads to incorrect results, so that it is best to let the user correct his own mistakes; the editor simply being used for the detection of errors. The designers of EMILY (Van Dam (1971)) report from experience that a light pen is not suitable as a pointing device at engineering workstations, as it is tiring on the arm and obscures vision. A tablet and stylus, mouse or just cursor movements can replace this problem.

The state of a system should always be displayed on the screen (eg. file name, file type, and mode). *Screen subdivision* (as used in EMILY and GNOME) is a useful visual aid to the user, as it refers him to a constant screen location for errors; prompts or *inputs*. The concept of multiple overlapping screens is used successfully in systems such as Magpie (Delisle (1984)), Smalltalk (LRG (1976), Goldberg (1983)) , and PECAN (Reiss (1985)), but these tricks require high resolution or graphics VDUs (eg. the Apollo system).

It must, however, be pointed out that if the screen is subdivided into too many *segments, or if too much* information is presented on a screen, this method looses its effectiveness. Meyrowitz (1982) also suggests that editors be able to offer users multiple contexts on the same display surface. For example, if help is needed, or another file is to be changed while editing a different file, the user should be able to gain access to the relevant routine without having to change mode and return to the original file (ie. transparent access).

3 A REVIEW OF RELATED PROJECTS

In this chapter a detailed summary is given of the available structure editors today. Two case studies are described to clarify the detail of such editors. The Cornell Program Synthesizer is the subject of section 1. This gives the concepts involved in the design and implementation of a template-driven syntax-directed editor in an integrated environment. The Z editor is chosen as the other case study because of its revolutionary ideas in regarding the *language based editor as a simple full*-screen text editor. The last section here gives a few brief words on most of the structure editors and incremental environments found in the literature, presenting a final summary in tabular form.

3.1 The Cornell Program Synthesizer

References: Allison (1983); Meyrowitz (1982); Teitelbaum (June 1981); Teitelbaum (September 1981).

The Cornell Program Synthesizer running on both the Terak personal computer and the VAX family of computers, presents a syntax-directed editor and *programming environment for PL/CS* and more recently, Pascal. Its aims included the provision of a unified programming environment; allowing a high level of abstraction; supporting top-down development; and encouraging good documentation.

The synthesizer is designed for simple terminals which use the cursor keys as the only locator device. A set of possible expansion commands for the current nonterminal is displayed in an *optional window for reminding* the user. In contrast to EMILY, the CPS is a hybrid between the traditional structure editor and the character-string text editor.

The user is presented with three types of high-level entities: templates, placeholders and phrases. *If the placeholder is* a comment or a statement, the user positions the cursor at the appropriate position and types in the relevant phrase. A nonterminal (enclosed in parentheses) requires a template substitution for further expansion. Square brackets indicate that the default value will be used. On depression of the carriage return key, the cursor is moved automatically to the next placeholder.

Variable names are typed in as text, not as structure. These are parsed for syntactical correctness upon pressing carriage return, and are stored and manipulated as text. Semantic checking is also performed as an illegal variable name will be highlighted in reverse video and flagged internally.

The cursor keys enable the user to move through the program structure. Right and down both move the cursor forward through the program, while left and up move the cursor back. Rather than moving character by character, the cursor moves one program

element at a time. (ie. to the beginning of a template, placeholder or phrase) Left and right additionally stop at each character in a phrase. The long down and long up key sequences move the cursor to the next or previous structural element of the same level. Other keys move the cursor to the nearest enclosing structure template and to the beginning of the program.

Insertion and deletion are based on the pick, put and delete buffer concepts. A delete command will delete an entire template with all its associated sub templates. Correcting mistakes can only be done by preserving structural integrity. Thus, the END of a construct for example, cannot be moved forward. Instead, the relevant portion must be moved backwards. This is certainly a more complicated procedure than that for a simple editor. A contributing factor to difficulty in this area is related to the Synthesizer's primitive methods of selection: Meyrowitz (1982) suggests that a pointing device would be more suitable. It must be borne in mind, however, that the time wasted here makes up for the time saved by ensuring that a program is syntactically correct. The major time-wasting operation in simple editors is the tracing and correction of compilation errors but this is not a problem with CPS.

A method for compressing a long program is provided for in CPS. The user can label via a comment statement a section of code, and by using the ellipsis key, the coded statements will be replaced by the comment. This method, besides enabling the user to view the whole program from a high-level viewpoint, promotes good documentation methodologies. Such information hiding still allows single step viewing of a program in which the cursor jumps from one visible high-level unit to the next. Uninitialized variables are flagged, type checking is enforced interactively and duplicate declarations are prohibited, all at edit time, rather than at compile time.

An important contribution of the Synthesizer project is the integration of its syntax-directed editor in a programming environment. The CPS is not used to create text files that will later be passed to a standard compiler, but rather to create a representation of a program suitable for on-line interpretation. The system interacts with an interpreter to allow the programmer to switch between editing and execution in a truly integrated manner.

In CPS, both editing and execution are guided by the syntactic structure of the program. Programs are incrementally compiled. This means that the user can re-edit and experiment with small parts of the program, without having to run the entire program. Whereas templates can only be input in a structurally sound manner, phrases typed textually are allowed to be erroneous. Such an erroneous program can be run at any time. The program will run normally until an error or unfinished program construct is encountered. At this point, an error message is generated interactively, with the offending program component highlighted. When this error is corrected, execution may continue.

During execution, the cursor traces a path through the program. As flow tracing can lead to an added overhead due to the displaying of confusing details on the VDU screen, the ellipsis function proves very useful here. The light intensity gives an indication of the time spent in each program section. Variable

monitoring, pacing and single stepping are also possible with CPS.

The program is stored as a combination of a parse tree for the templates, and as text for the phrases. The prettyprinted code that is displayed is actually an interactively generated view of the internal data structure.

The CPS has recently been implemented as a generator, so that it is now possible to create synthesizers for different languages using attribute grammars to describe the output and semantics for each production of the abstract syntax. Reverse execution is also being implemented.

3.2 The Z editor

References: Allison (1983); Meyrowitz (1982); Wood (1981).

With a text-oriented model of program structure, this editor is both a program editor and a document editor. The designers of Z at the Yale Computer Science Department believe that a text orientation considerably simplifies the design of the editor and presents the user with a simple but powerful model of program structure. This production editor was designed for and is used by undergraduate and postgraduate students and staff at Yale University. Many of the existing features of Z are a result of an iterative procedure involving suggestions from the users.

The quarter plane model for a full-screen editor is used here so that a file is envisaged as an infinite array of infinitely wide lines. There are commands for positioning the display window anywhere in the plane, and for positioning the cursor anywhere within the display window. What the user sees at any moment is precisely what is present in the corresponding section of the file. As opposed to MIT EMACS which is a stream editor, Z allows the user to extend the file or line by simply typing past the last entry. (ie. no end-of-line delimiters exist and the user has complete freedom within the quarter-plane)

Cursor keys are used extensively, but function keys are avoided as the implementers believe that this causes the user to move his hand away from the typewriter keyboard. Control characters and the shift key are used instead for entering predefined commands. Commands are made up of any logical combination of certain key words.

Cursor arguments in Z allow the user to quickly select areas of text to be deleted, moved or manipulated. This selection is done in two ways: a box argument selects a rectangle of text defined by its two opposite corners; and a stream argument selects a stream of text defined from a starting location to an ending location of the cursor.

If the user executes an undesired command by mistake, he can readily undo this command and recover the previous state. If, however, the unwanted command is a lengthy one to execute, the 'cancel' command will cleanly abort the operation. The user is also given the facility to tailor certain editor functions to his particular preference. A line counter which is updated every one

hundred lines, is maintained in one corner of the screen to keep the user informed on the state of a command which takes a long time to execute.

The most recent seven window positions are remembered by the system so that the user can flip back to a previous context without losing the current one. A bookmark facility accepts a number or a name as a label in any file. If a bookmark is in another file, the editor will automatically switch to that file.

It is the program editing features of Z which present an interesting change to many of the conventional syntax-directed environments. For each line of text, the editor only knows about quoted strings, an end of line comment, blank separated words, tab or backtab tokens, and balance tokens. A simple table-driven lexer divides each line of text into the categories. Each editor command is responsible for using this information to impose any additional structure, beyond the text representation, that is necessary to support the program-editing features. For each language supported by Z, there is a modifiable table that categorizes the tokens for the language. The editor currently supports the following languages: LISP, BLISS, Pascal, RATFOR, and APL.

The Z editor does not rigidly adhere to a set prettyprinting format, but rather 'suggests' an indentation amount whenever the newline command is used to enter a line. The indentation is relative to the first non-blank character of the current line. For block structured languages, the cursor position on the next line is determined as follows:

-- Each language type is associated with a table of tab/backtab tokens. When a newline command is invoked, the editor examines the last token on the current line and using the table performs the associated tab/backtab indentation. (This deals with tokens that open and close blocks.)

-- If this token is not in the table, the last token of the previous line is checked to see whether it is a tab token that implicitly opens a block. If this is the case, a backtab command is performed. (This deals with the case of a loop with only one statement within it.)

-- If none of the above are successful, the cursor is placed in the same column as the first non-blank character of the current line. (This deals with lists of statements, field names, etc..)

The advantage of such a system is that it is extremely simple to implement and gives correct results most of the time. By disabling the automatic indentation feature, the user can effect his own indentation style by hand or correct any errors made in automatic mode.

A balanced expression is one that can be regarded as a block (eg. An expression within parentheses or a Begin-end block) The reason for the need for balanced expressions is twofold:

-- The editor must be able to close off the most recently opened block, and indicate the location of the matching tokens.

-- The editor should allow cursor movements by blocks if required.

The Z editor provides both these functions via the use of its balance facility.

Provided the programmer is consistent, the indentation of a program provides all the information necessary for defining block levels. In Z, the indentation is interpreted in one of the following ways:

-- The display level of the line is the number of tab stops from the beginning of the line to the first non-blank character of the line.
-- The display level must be an integral number of tab stops and must differ from the display level of the previous line by plus or minus one tab stop. If this condition is not met, then the display level is that of the preceding line.

The zoom command specifies the maximum level to display. An *infinite zoom parameter* displays all lines, while a zero zoom parameter displays only the top level declarations and procedure definitions. Groups of lines that are not displayed are represented collectively by a single dotted line. Selecting this line with the cursor implies the selection of all the hidden lines represented by that line.

Thus without the need for the user to understand a program in terms of the complex semantics of a parse tree, the Z editor provides extensive structured program editing facilities with the use of indentation and the balancing function.

The designers of Z feel that the programmer is the best person to decide whether his program is correct and ready for compilation. Thus, no further syntactic or semantic error detection or correction is performed. The designers tried to improve the communication interface between the editor and the compiler so that the user can display the error message after compilation by moving the cursor to the location of the error in the file.

The provision of a link to Multiple User Forks (a program which maintains multiple user contexts in parallel) allows the user to exit from Z and enter any of the other forks (perhaps to read another document or check the execution of a program) with transparent return to the Z editor.

3.3 *Other systems*

ALOE (Feiler (1981)) is a general structure editor where semantic correctness is not enforced. Program entry is by selection and a tree structure is formed, but there is no parsing of text. Only certain structures are allowed to be modified (eg.moved) as a single block.

CARS (Allison (1983); Wilcox (1976)) is a teaching system with integrated interpreter and debugger. It is aimed at the novice and makes use of a full-screen editor. Changes are seen by the user as textual and errors are rejected on entry for immediate

correction.

Cedar (Teitelman (1985)) is a single programming environment providing: a sophisticated editor, a document preparation facility and a variety of tools for the programmer to use in construction and debugging of programs. The Cedar programming language is a strongly typed compiler-oriented Pascal-like language. This system makes use of high quality graphics terminals.

COPE (Archer (1981)) is a COoperative Programming Environment developed at Cornell University. Similar to POE, it is a text editor with integrated execution facilities. It uses an intelligent parser and supports undo and redo commands.

EMACS (Jong (1982); Meyrowitz (1982); Stallman (1981)) is a large well-established extensible, customizable and self-documenting text editor. It can be used for the structured editing of any text file. It is a display editor supporting many windows. It has a primitive undo function: the entire history of commands can be run and the user can stop this process where he sees fit.

Emily (Allison (1983); Meyrowitz (1982); Van Dam (1971)) developed at Argonne National Laboratory was one of the first useful syntax-directed editors. The screen area is divided into three logical screens : text, menu and message. The text area contains the text under construction, with nonterminals highlighted by underlining; the current nonterminal being enclosed in a rectangle. The menu screen contains possible replacements for the current nonterminal and the message screen is used for entering identifiers and displaying status and error messages. Prettyprinting is settable and all nonterminals must be replaced before the program is completed. It also features a limited undo function and ellipsis facilities.

ETUDE (Good (1981); Meyrowitz (1982)) is a document production system. Although it uses prefix notation, it provides many useful functions: an infinitely deep undo command; a cancel command to abort a lengthy operation; and an again command to repeat the command last executed. ETUDE's help facility is interesting in that it displays the history of commands already executed. Its adaptable interface supports inputs using any of the following methods: menu selection; cursor movements; pointing device; command recognition; or function keys.

Gandalf (Habermann (1982)) was one of the pioneering projects at Carnegie-Mellon University for the design of an integrated environment involving a syntax-directed editor. The use of templates is encouraged in its generative approach. A series of generators have emerged from this project, and ALOE is an example of a syntax editor which stemmed directly from this system.

GNOME (Garlan (1984)) is the Gandalf NOvice Programming Environment developed and in use at Carnegie-Mellon University. The system consists of a family of four structure editors among which are the FORTRAN and Pascal syntax-directed editors. A program is entered by selection and no parsing of text is performed. Changing a program is done explicitly by modifying the parse tree directly. A form of ellipsis is provided via the use of multiple views.

Interlisp (Teitelman (1981)) is a fully integrated system with a single command for editing, debugging and programming.

ISDE (Chesi (1984)) is a language independent Interactive *Software Development* Environment. It can be used for editing of general text files. It supports the generator approach but does not have an ellipsis function. It makes use of function keys and multiple editing is possible because of the multi-window screen. A type checker does static semantic error detection.

Magpie (Delisle (1984)) uses the same technique of overlapping screens in an integrated environment. It uses the recognizer approach and limits syntax error occurrence by partitioning segments of code. It provides information regarding any highlighted errors, and execution cannot proceed before all *static errors have been removed.*

Mentor (Allison (1983); Donzeau-Gouge (1980) and (1984)) is an extensible editor used as a structured document manipulation system. It uses a generative approach with a tree representation and provides ellipsis.

MINCE (Jong (1982); Moore (1981)) is a self-acknowledged spin-off of EMACS: Mince Is Not Complete Emacs. It too makes use of a full-screen editor and has facilities for viewing two files simultaneously. It also provides an undo facility and conditional and unconditional *string replacement.*

Pathcal (Allison (1983); Wilander (1980)) was developed at Cornell University with the same principles of CPS. It is an integrated environment and semantic error correction is done at execution time.

PECAN (Reiss (1984) and (1985)) is a program development system generator for algebraic programming languages. The program development systems it produces support multiple views of the user's program, its semantics and its execution. The program *views include a syntax-directed editor, a declaration editor, and* a structured flow-graph editor. The semantic views include expression trees, data type diagrams, flow graphs and the symbol table. Execution views show the program in action and the stack contents as the program executes. This system is currently implemented on APOLLO workstations, and requires high-resolution graphics for its re-targeting. The system is modelled on Cedar. Parsing is done on request and it uses a keyboard and pointing device as input resources. It gives the user the option of using either templates or typing the constructs manually and provides an extensive redo/undo capability. A program is modified by *explicit tree manipulation.*

PEN (Barach (1981)) represents a Portable Editing Nucleus. It is terminal independent but does not support the idea of a "free" full-screen editor as defined by the designers of Z. Instead, it uses special characters on the end of each line to show if more text is to follow.

POE (Fischer (1984)) is a full-screen Pascal Oriented Editor. It was inspired by the Synthesizer, but is more similar to COPE. It does not use templates, but has a set of required and optional prompts that must and need not be *expanded respectively.* Although it is not part of an integrated system, semantic checking is done

and errors are highlighted. Information on the errors can be obtained on request by positioning the cursor at the relevant error. Contrary to some views, this editor provides error correction so that incorrect automatic corrections must be undone and corrected manually by the user.

SAGA (Campbell (1984)) is an integrated Software Automation Generation and Administration system. Although it makes use of full-screen facilities, this editor involves many modes. Changing of a file is by text manipulation, but a tree structure is used internally. Even though the recognizer approach is taken, on request, the user will be supplied for a list of tokens which can be inserted at any cursor position.

sds (Fraser (1981); Meyrowitz (1982)) is a general structure editor for graphics files, documents or programs. The user is presented with a tree view of the file, and operations are done explicitly on this tree. An ellipsis function is provided.

SED (Allison (1983) is a Syntax EDitor developed at the University of Western Australia. It uses a higher level tree structure than used by the like of CPS (See Figure 1). It takes a recognizer approach and makes extensive use of error correcting capabilities. One of its principal aims was to be able to use files from any other editor and correct partially created programs. Prettyprinting is not enforced, and an ellipsis function is provided.

Smalltalk (Goldberg (1983); Goldberg (February 1983); LRG (1976)) is an integrated software development environment. It makes use of extensive graphics facilities and uses a mouse as the input device. The system is menu-driven and modeless. It makes extensive use of overlapping screens to emphasize the idea of papers on a work-desk.

SUPPORT (Zelkowitz (1984)) is a Still Unnamed Production Programming Oriented Research Tool environment developed at the University of Maryland. Editing is allowed for only one line at a time. Function keys are used to expand nonterminals, and a high level Program Design Language (PDL) is being added to the system so that a designer can produce fully documented programs in a structured top-down way by interspersing PDL with the Pascal code. The user can work on the data definition segment and in the program itself simultaneously. A powerful string matching routine enables the user to find all the uses of a certain variable. As the system is integrated, execution is completely interactive. An ellipsis function is also provided.

Syned (Gansner (1983); Horgan (1984)) is a language-based editor which uses multiple entry parsing to give it its general text processing abilities. It makes use of the concept of transactions. It takes a recognizer approach and provides an undo function, but makes no use of menus.

Many of the results above are summarized in the following table.

System Name	Language + Used	Incr. System?	Semantic checks?	Generator/ Recognizer	Ellipsis Function?	Errors Corrected?
CAPS	Various	Yes	Yes	Generator	---	No
CPS	PL/CS	Yes	Yes	Generator	Yes	No
Emily	PL/1	No	---	Generator	Yes	No
GNOME	Various	Yes	No	Generator	Yes	No
ISDE	Various	Yes	Yes	Generator	No	No
Magpie	Pascal	Yes	Yes	Recognizer	No	No
Mentor	Various	No	No	Recognizer	Yes	No
PBCAN	Various	Yes	Yes	Generator	No	Yes
POE	Pascal	No	Yes	Generator	Yes	Yes
SAGA	Various	Yes	Yes	Recognizer	No	No
sds	Various	No	No	Generator	Yes	No
SED	Pascal	No	No	Recognizer	Yes	Yes
SUPPORT	Pascal	Yes	Yes	Generator	Yes	No
Syned	Various	No	No	Recognizer	---	No

4 REFERENCES AND BIBLIOGRAPHY

1. Allison,L. (May 1983): "Syntax directed program editing", Software Practice and Experience (GB), Vol.13, No.5, pp.453-465.

2. Archer,J. and Conway,R. (June 1981): "COPE: A cooperative programming environment", Technical Report, Cornell University, TR 81-459.

3. Barach,D.R. ,Taenzer,D.H. ,Wells,R.E. et al. (June 1981): "The design of the PEN video editor display module", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.130-136.

4. Campbell,R.H. and Kirslis,P.A. (May 1984): "The SAGA project: A system for software development", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

5. Chesi,M. ,Dameri,E. ,Franceschi,M.P. ,et al (May 1984): "ISDE: An interactive software development environment", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM *SIGSOFT/SIGPLAN Software Engineering* Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

6. Clowes,T. (July 1982): "Move and Copy commands for text processing systems", IBM Technical Disclosure Bulletin (USA), Vol.25, No.2, p.869.

7. Delisle,N.M. ,Menicosy,D.E. and Schwartz,M.D. (May 1984): "Viewing a programming environment as a single tool", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM *SIGSOFT/SIGPLAN Software Engineering* Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

8. Donzeau-Gouge,V. ,Huet,G. ,Kahn,G. ,et al (May 1980): "Programming environments based on structured editors: The Mentor experience", Workshop on Programming Environments, Ridgefield, CT, presented June 1980.

9. Donzeau-Gouge,V. ,Kahn,G. Lang,B. ,et al (May 1984): "Documents structure and modularity in Mentor", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

10. Feiler,P.H. and Medina-Mora,R. (September 1981): "An Incremental Programming Environment", *IEEE Transactions on Software Engineering*, Vol.SE-7, No.5, pp.472-481.

11. Fischer,C.N. ,Johnson,G. ,Pal,A. et al. (1983): "An introduction to editor Allan POE", SOFTFAIR A Conference on software development tools: technologies and alternatives, Proceedings, Arlington, VA, USA, 25-28 July 1983 (Silver Spring, MD, USA: IEEE Comput. Soc. Pres 1983) pp.245-250.

12. Fischer,C.N. ,Pal,A. ,Stock,D.L. ,et al (May 1984): "The POE language-based editor project", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

13. Fountain,A.M. and Hyder,A.F. (February 1981): "Extended function programmable keys for display systems", IBM Technical Disclosure Bulletin (USA), Vol.23, No.9, p.4327.

14. Fraser,C.W. (June 1981): "Syntax-directed editing of general data structures", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.17-21.

15. Gansner,E.R. ,Horgan,J.R. ,Moore,D.J. et al (1983): "SYNED -- A language-based editor for an interactive programming environment", Spring COMPCON 83, Intellectual Leverage for the Information Society, San Francisco, California, USA, 28 February - 3 March 1983, IEEE, New York, USA, pp.406-410.

16. Garlan,D.B. and Miller,P.L. (May 1984): "GNOME: An introductory programming environment based on a family of structure editors", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

17. Goldberg,A. (February 1983): "The influence of an object-oriented language on the programming environment", Proceedings of the ACM Computer Science Conference.

18. Goldberg,A. and Robson,D. (1983): Smalltalk-80: The language and its implementation, Addison-Wesley, Reading, Mass..

19. Good,M. (June 1981): "Etude and the folklore of user interface design", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.34-43.

20. Grappel,R.D. and Hemenway,J. (June 1980): "The CREDIT goes to Intel", Mini-Micro Systems (USA), Vol.13, No.6, pp.119-122.

21. Habermann,A.N. and Notkin,D. (January 1982): "The Gandalf software development environment", Technical Report, Carnegie-Mellon University, Computer Science Department.

22. Hammer,M. ,Ilson,R. ,Anderson,T. et al. (June 1981): "The implementation of Etude, an integrated and interactive document preparation system", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.137-146.

23. Hansen,W.J. (July 1971): "Creation of hierarchic text with a computer display", Argonne National Laboratory, Rep. ANL7818, Argonne, Illinois.

24. Horgan,J.R. and Moore,D.J. (May 1984): "Techniques for improving language-based editors", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

25. Irons,E.T. and Djorup,F.M. (January 1972): "A CRT Editing System", Communications of the ACM, Vol.15, No.1, pp.16-20.

26. Jong,S. (April 1982): "Designing a text editor? The user comes first" , Byte (USA), Vol.7, No.4, pp.50-53.

27. Learning Research Group (March 1976): "Personal dynamic media", Xerox Palo Alto Research Centre, Tech. Rep. SSL-76-1, Palo Alto, California.

28. Lewis,T.G. (April 1984): "Word processing for the masses: A review of bank street writer", IEEE Software, Vol.1, No.2, pp.89-92.

29. Meyrowitz,N. and Van Dam,A. (September 1982): "Interactive editing systems: Parts I and II", ACM Computing Surveys, Vol.14, No.3, pp.321-415.

30. MicroPro (1981): "WordStar user's guide", MicroPro International Corporation, San Rafael, California.

31. Mikelsons,M. (June 1981): "Prettyprinting in an interactive programming environment", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.108-116.

32. Moore,J. (Spring-summer 1981): "Mince -- a product review", SIGPC Notes (USA), Vol.4, No.1-2, pp.47-50.

33. Morris,J.M. and Schwartz,M.D. (June 1981): "The design of a language-directed editor for block-structured languages", Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.28-31.

34. Oppen,D.C. (October 1980): "Prettyprinting", ACM Transactions On Programming Languages And Systems, Vol.2, pp.465-483.

35. Peck,J.E.L. and Maclean,M.A. (May 1981): "The construction of a portable editor", Software Practice and Experience (GB), Vol.11, No.5, pp.479-489.

36. Raduchel,W.J. (May 1984): "A professional's perspective on User-Friendliness", Byte (USA), Vol.9, No.5, pp.101-106.

37. Reiss,S.P. (May 1984): "Graphical program development with PECAN program development system", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh,

Pennsylvania , April 23-25, 1984, pp.100-200.

38. Reiss,S.P. (March 1985): "PECAN: Program development systems that support multiple views", *IEEE Transactions on Software Engineering*, Vol.SE-11, No.3, pp.276-285.

39. Rubin,L.F. (March 1983): "Syntax-directed pretty printing -- a first step towards a syntax-directed editor", IEEE Transactions on Software Engineering, Vol.SE-9, No.2, pp.119-127.

40. Stallman,R.M. (August 1980): "EMACS manual for TWENEX users", Artificial Intelligence Laboratory, AI Memo.556, Massachusetts Institute of Technology, Cambridge, Mass..

41. Stallman,R.M. (June 1981): "EMACS, the extensible, customizable self-documenting display editor", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.147-156.

42. Stromfors,O. and Jonesjo,L (June 1981): "The implementations and experiences of a structure-oriented text editor", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.22-27.

43. Teitelbaum,T. ,Reps,T. and Horwitz,S. (June 1981): "The why and wherefore of the Cornell Program Synthesizer", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.8-16.

44. Teitelbaum,T. and Reps,T. (September 1981): "The Cornell Program Synthesizer: A syntax directed programming environment", Communications of the ACM, Vol.24, No.9, pp.563-573.

45. Teitelman,W. (March 1985): "A tour through Cedar", IEEE Transactions on Software Engineering, Vol.SE-11, No.3, p.285.

46. Thompson,H.B. (March 1981): "Text editing with compuview's VEDIT", Byte (USA), Vol.7, No.3, p.262, 266, 268-270.

47. Van Dam,A. and Rice,D.E. (September 1971): "On-Line Text Editing: A Survey", ACM Computing Surveys, Vol.3, No.3, pp.93-114.

48. Wilander,J. (1980): "An interactive programming system for Pascal", BIT, Vol.20, pp.163-174.

49. Wilcox,A.M. ,Davis,A.M. and Tindall,M.H. (1976): "The design and implementation of a table driven interactive diagnostic programming system", Communications of the ACM, Vol.19, No.11, pp.609-616.

50. Wood,S.R. (June 1981): "Z -- the 95% program editor", ACM Sigplan Notices, Vol.16, No.6, SIGPLAN/SIGOA Symposium on Text Manipulation, Portland, Oregon, June 8-10 , 1981, pp.1-7.

51. Zelkowitz,M.V. (May 1984): "A small contribution to editing

with a syntax directed editor", ACM Sigplan Notices, Vol.19, No.5, (ACM Software Engineering Notes, Vol.9, No.3), ACM SIGSOFT/SIGPLAN Software Engineering Symposium Practical Software Development Environments, Pittsburgh, Pennsylvania , April 23-25, 1984, pp.100-200.

A FUNCTION-KEY DRIVEN SYNTAX-DIRECTED EDITOR FOR SOFTWARE SYSTEMS DESIGN

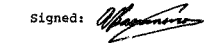
U S E R ' S M A N U A L

(Version 1.0)

December 1985

Author: A.P.Bassanino

Signed:



A Project Report submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

CONTENTS

* These system levels have not yet been implemented.

LIST OF FIGURES

1 INTRODUCTION

In this chapter some of the important terms and concepts involved in the package are introduced. The first section is introduced mainly to give the user a feel for the structure of the Program Description Language (PDL) for which the package was designed. Secondly, a broad descriptive account is given of the system, paying particular attention to the definition of new terms. In this second section, the advantages of using the package are also emphasized.

1.1 A Brief Summary of the Features of PDL

PDL (Program Description Language) is a machine-independent language written in a structured english format which is used to express a design in a logical, high-level notation. Due to its descriptive nature, a well-defined model of the design of a project can be obtained using conventional programming concepts. The use of comments in PDL is encouraged so that a design can be developed in program form via a series of stepwise refinement iterations.

This top-down design approach aids the designer to view the system as a whole initially, and to slowly expand the view to include more detail until the required implementation level is reached. A design that is fully analyzed using a PDL approach will result in a structured Pascal-like high-level program. This can then be implemented using any suitable technology. Although such an approach is useful in both hardware and software designs, the full power of PDL can only be appreciated in the latter.

A very high-level description of a project can be written in a few lines of PDL using a couple of descriptive comments. This is known as high-level PDL (See Fig.1.1 (a)). PDL in its lowest level form is similar to any modern programming language in structural strength and integrity. As can be seen from Fig.1.1 (b), a low-level PDL description is in many ways similar to a computer program.

A PDL program is usually divided into a single program module and an associated set of procedures or subroutines. The procedures will list all their relevant input and output variables as well as any external procedures accessed by them. Extensive use of procedures can be made for the design of large systems, so that program complexity can be controlled. This type of fragmentation also aids in error detection and assists in system building by providing reusable modules.

Strong typing of the constants and variables is enforced in the Data Description segment before the start of the program or procedure body. The five characteristics used for this purpose are known as: Function, Type, Structure, Scope and Name. A detailed description of these terms can be found in Appendix A

which gives the formal specification for PDL. Rigid indentation disciplines in the area of data description ensure ease of readability as is evident from Fig.1.1. .User-defined types may also be specified; the familiar and useful "record" concept in Pascal is also available in PDL.

(a)

```
Program ROOTS
Begin:
  *Get the variable co-efficient B*
  If (Determinant is positive)
    then:
      *Determine the two roots*
      *Output the two roots*
    else:
      *Output an error message*
  End if:
End:
End Program:
```

(b)

```
Program ROOTS
Constants:
   Real:
      Single:
         Local:
            A = 5
            C = 6
Variables:
   Real:
      Single:
         Local:
            B
            X1
            X2
Begin:
  Get:
    Keyboard: B
  End Get:
  Determinant := B**2 - 4*A*C
  If (Determinant >= 0)
    then:
      X1 := (-B + SQR (Determinant)) / (2*A)
      X2 := (-B - SQR (Determinant)) / (2*A)
      Put:
        Console: 'The two roots are', X1, 'and', X2
      End Put:
    else:
      Put:
        Console: 'The roots are imaginary.'
      End Put:
  End if:
End:
End Program:
```

Fig.1.1: An example of low and high-level PDL descriptions
A high-level PDL description of a program for determining the roots of a quadratic equation is shown in figure (a). The corresponding low-level or detailed PDL for this same program can be seen in figure (b).

The Algorithm segment which comprises the program body makes use of assignment, selection, and iteration constructs. The control constructs which will be familiar to the programmer include the following:

--- If - then - else

--- Case

--- Repeat - until

--- While - do

Get and Put routines are used for input and output purposes respectively. Sections of code which run concurrently (or "in parallel"), as opposed to the conventional sequential order, can be distinguished via the Co-begin and Co-end keywords. Comments, enclosed by asterisks, are freely allowed at any point in the program.

Indentation is vital to the understanding of any program, thus PDL should always be appropriately indented Fig.1.1 (b)). Entering such a program manually thus involves large amount of effort. This is especially true in the Data Description segment or in the case of inserting or deleting an imbedded construct. The clerical effort involved in producing a readable PDL program may overshadow the use of PDL as an effective design aid. It is mainly for this reason that this PDL generator package has been produced.

1.2 A General Description of the Package

The package presented here is aimed at producing a tool for simplifying the input of a program written in Program Description Language (PDL). A syntax-directed editor package such as this, knows the syntax rules for the language for which it will be used. Such a system combines the text manipulation facilities of a general-purpose editor with the syntax or error-checking functions of a compiler. A user can thus write a program, being interactively warned of any language structure errors that would be made.

The aim of this editor package is to enable the user to put together and edit a PDL design with the minimum of effort. The user should not be constantly preoccupied with petty issues such as indentation correction and program structure, but rather be left with the actual task of designing the required system. This package will thus ensure that programs written using this system will emerge well formatted, readable, and syntactically correct. The syntax-directed editor was built with many modern editor improvement concepts in mind, so that the resulting system should prove easy to learn and operate.

The entire system is function-key driven, saving the user unnecessarily wasted time for typing (eg. The insertion of a construct with associated indentation can be achieved by the depression of a single function key.) The keyboard is used only

when necessary for entering text lines or for specifying filenames or other required responses. Otherwise, the entire system is driven with the use of a dynamic set of ten function keys.

Indentation (or prettyprinting) is done automatically by the editor, leaving the user more time to concentrate on the true design rather than the program format. Also, a PDL program produced using this editor will have a consistent format. In this way, the documentation layout can be standardized.

The editor can also perform syntax and limited semantic error detection. This means that compilation errors such as "missing end of construct" or "undefined variable name" would be flagged *as errors at edit time. Thus, a completed program, if it could be* run, would be free of the most common compilation errors.

Such an editor will prove to be useful as a teaching aid as it emphasizes good programming practices and allows the user to learn the rules of PDL quickly. Because of its time-saving nature, such an editor is of great use as a design tool. Programs produced by this system will not only be correct syntactically and semantically, but will also be in a standard prettyprinted format and thus readable form.

2 THE SYNTAX-EDITOR STRUCTURE

This second chapter is aimed at introducing the user to the system by elaborating on the available features of the package. This chapter should give the user a global view of the package; its interface to the outside world; and the basic mechanisms involved behind it. The first section explains the existing screen partitions and visual cues (or fonts), with a brief description of the purpose of each window. The editor operation is also clarified. The second section deals with a broad description of system operation, while the last section gives the reader a good idea of the hierarchical levels involved.

2.1 The Screen Divisions

To provide a standard interface which is simple to operate, the physical VDU screen, which has space for 25 lines of text, is divided into a number of logical screen partitions. A logical screen partition is an area on the physical screen dedicated for a specific function. Screen divisions, when used consistently, are of great use to the user, as he will know where to expect, say, prompts or errors, and thus never be confused with the physical display of information.

Four logical screens are defined in this package: the Main Screen; the Window Screen; the Prompt Screen; and the Function Screen. This arrangement is depicted in Figure 2.1. Figure 2.2 gives an example of what the screen may look like during a typical editing session.

The Main Screen (Screen 1) is the 20 line file display screen. (An even number of lines can be selected to suit the physical VDU screen length.) Here, a 20 line section of the formatted program is presented for editing with its corresponding line numbers. A cursor with vertical freedom only (Cursor 1) is available in this screen in the form of a reverse videoed line number. Cursor movement is effected via the use of the up and down, page forward and page backward cursor control keys. A half screen scroll is used when the cursor is moved beyond the screen limits. The fonts which are used in this screen are as follows:

--- Reverse Video line number to indicate the position of Cursor 1 along the left margin.

--- Errors (semantic and unexpanded placeholders) will also be highlighted in reverse video font. These errors are editable.

--- Highlighted text representing standard or system-defined key words. These words are not directly editable.

--- Non-highlighted, non-underlined text representing user entered, editable syntactically and semantically correct statements.

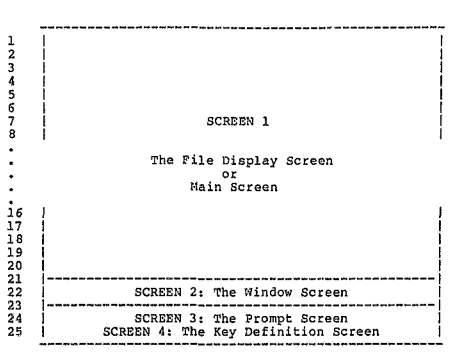


Fig.2.1: The Logical Screen Partitions
Physical screen line numbers are shown in the leftmost column. The logical screen locations and names are also given.

The Window Screen (Screen 2) is a one-line screen used for the editing of text lines and for obtaining other responses from the user. When editing is required, the true VDU cursor will at all times be resident on this screen. This cursor (Cursor 2) is given horizontal freedom only and cursor movement is controlled by the left and right cursor function keys. The Window Screen supplies the user with powerful line editing functions as all entry and editing is performed here. This concentrates the user's attention on Screen 2 for editing purposes, and on Screen 1 for viewing *purposes.*

Available fonts on Screen 2 are similar to those of Screen 1 with the exception of reverse video which is reserved for more emphatic highlighting. If a line is to be edited, it is chosen on the Main Screen using Cursor 1. A Line Edit is then requested on this line using a function key. If the line is individually editable, a copy of the line is made in Screen 2. The line number as well as any associated indentation is not shown in the Window Screen so that the user is presented with the entire line extent. Depending on the terminal type, the length of the screen will determine the maximum length of any line (usually 80 characters).

The usual fonts will be used to distinguish key words from editable text; the user then being able to edit the editable text string. This Window Screen text is now editable using the Line Editor facilities. (See Section 3.1) After the line has been entered, the Window Screen is cleared, and the old line referenced by Cursor 1 is replaced by the new edited version. In

this way, parsing can be done at entry on a line by line basis.

```
| 12          single:                                    |
| 13             Global:                                 |
| 14                KING                                 |
| 15             Local:                                  |
| 16                ROOK 1                               |
| 17                ROOK 2                               |
| 18    Begin:                                           |
| 19      Rook 1 := 1                                    |
| 20      Rook 2 := 8                                    |
| 21      Knight 1 := 2                                  |
  .
  .
  .
  .
  .
| 27      If (Move 1 = 'O')                              |
| 28        then:                                        |
| 29           *Castling*                                |
| 30        else:                                        |
| 31           *Check for other possibilities*           |
|-------------------------------------------------------|
|If (Move 1 = 'O')                                      |
|-------------------------------------------------------|
| ** Editing Line 27 **                                 |
|1.PaB  2.PaF  3.ToF  4.BoF  5.ToL  ...                 |
---------------------------------------------------------
```

Fig.2.2: A Sample Editing Situation

The four logical screens are shown in abbreviated form. The two cursors are highlighted and underscored and can be seen at Line 27 (Cursor 1) and at column 10 (Cursor 2). There are ten function keys and their function abbreviations are displayed in reverse video font in Screen 4.

The Prompt Screen (Screen 3) is a one line screen used to warn or *inform the user via error messages or prompts.* Information on any semantic error (which is highlighted in the Main Screen) can be obtained on this screen. The user can consistently expect to find any form of system comment on this screen only. Highlighted and blink fonts are the only fonts needed for this logical screen.

The Function Screen (Screen 4) is another single line screen used for the sole purpose of displaying the function key definitions. As the entire system is function key driven, each of the 10 function keys is dynamically defined here. A number, 1 to 10, (corresponding to the function key number) together with a six character abbreviation of the associated function is used in Screen 4 to define each function key. This screen uses only highlighted and reverse video fonts to present a display in *consistent format.*

2.2 Basic System Operation

From the user's point of view, as explained previously, Cursor 1 is used for moving through the file line by line, while Cursor 2 brings the focus of attention on a single character in the line chosen by Cursor 1 on the Main Screen. A line on Screen 1 is chosen for editing on Screen 2 by depressing a predefined Line Edit function key. The system will allow only lines which contain text that has been entered by the user, or construct placeholders to be edited via the Window Screen. This line is released from the editor after it is entered. It is important to note that a line is shown in the Window Screen without indentation so that line length restrictions can be accurately portrayed. Text presented on the Main Screen is done so in indented or prettyprinted form.

Definition of data items in the data description segment is largely automated. Besides deleting or modifying a specific data item, the user is not permitted to tamper with this PDL segment. A data item definition can only be deleted by deleting its corresponding data item name line. The system will essentially take care of removing any associated definition key words. Any error in system usage will be displayed as a warning in the Prompt Screen.

The system is template-based. This means that a construct will be inserted as a block, with placeholders (denoted by <> brackets) to be expanded by the user. (See Fig.2.3) This method ensures that syntactical correctness is maintained by preventing the occurrence of any such error. (This eliminates the "missing end of construct" compilation error.) Copy, delete and move functions are thus restricted in that they can only be performed on certain blocks of text defined by the constructs. This choosing of a block during editing, is aided by the system's feature of highlighting the line numbers of the defined block. This helps the user visually to see the physical block which he is about to modify. Block restrictions on the standard move, copy, and delete functions have to be imposed if syntactical correctness is to be maintained. This method may seem slightly inconvenient to the novice, but the instantaneous compilation facilities coupled with the automatic indentation features of this syntax-directed editor will soon begin to show their effectiveness.

```
Case <CONDITION> of:
    <STATEMENT>
  else:
    <STATEMENT>
End Case:
```

Fig.2.3: An example of a predefined block construct
The Case-else construct can be inserted by the depression of a single function key. The placeholders are identified by the <> brackets, and highlighted in reverse video font. Key words are highlighted and are thus not directly editable.

The PDL program can be stored in formatted or coded form. A program stored in coded form will be unintelligible to the user when displayed, while a formatted program will represent the prettyprinted program as generated by the syntax-directed system on the Main Screen.

The coded version of a program is used by the system and will be unintelligible to any other editor. This version is labelled with a ".COD" extension, and it is always necessary to retain this file if the program is to be re-edited using this system. This coded form of the program will also be used for translation purposes in the future.

The formatted version of a program can be stored in a file if so desired by the user. This can be done for documentation or printing purposes. This file will be a physical copy of what the user has generated in Screen 1. Although this file is fully comprehensible and can be modified with the use of any conventional text editor, it is of no use to the system unless a coded version exists.

2.3 The System Levels

The PDL syntax-directed editor package is function-key driven, and as such, operates from a hierarchical structure. At any time, the user will be in a certain mode defined by the display of funtion keys in Screen 4. Each mode or level can be distinguished from another by the significance assigned to the ten function keys. These levels have purposefully been kept to a minimum so as to simplify usage of the editor.

The system consists of a principal Base Level, from which the user would usually operate. This level is responsible for providing:

--- File scrolling functions (such as Top of file, Page forward, etc.)

--- The single line modification facility as referred to in the previous section

--- A gateway into other file manipulation levels (eg. copy, delete, etc.)

The base level is thus mainly used for "browsing" through the file. It should be noted that if the terminal keyboard has explicit cursor control keys as well as page scroll keys, any subsequent mode or level of operation in the system will allow the following file scroll movements without the need for returning to base level:

↑	Cursor up one line
↓	Cursor down one line
Pg Up	Forward page scroll
Pg Dn	Backward page scroll

Base mode is, however, the only level from which an editable line

can be modified. This level is described fully in section 3.3.

The so called Front-end of the package provides an interface into the outside world by allowing the user to specify input and output filenames and formats. This routine leads directly into the system's Base Level. If the editor is exit, the Front-end routine is again called on for external interfacing purposes. Further detail of this level can be found in section 3.2.

The other four levels or modes which exist are: Insert, Delete, Copy, and Move modes. (See sections 3.4 - 3.7 for details) In these modes, the user has limited file scrolling abilities as listed above, but can perform any of the defined functions associated with that mode. Each of these modes may again have their sub-levels. It can be said, in general, that a RETURN function key will move the user to a previous level, while the CONTINUE function key will move him into the next level.

A sketch of the system's levels in hierarchical form is shown in Figure 2.4 below. It should be noted that although the basic package structure has been formulated, only the Base Level and *Insert Mode have been physically implemented as yet.* (See the highlighted blocks of Fig.2.4.) Addition of the other system blocks is merely an expansion using principles which have already been developed.

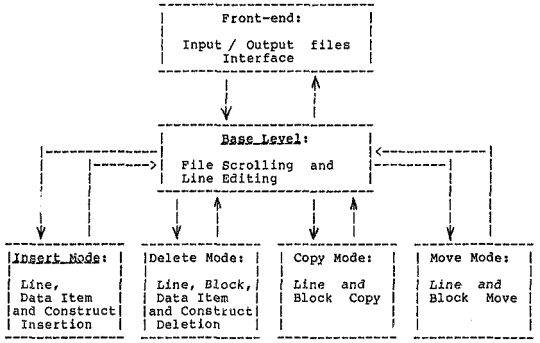


Fig.2.4: System Level Hierarchy
A simplistic representation is given of the main system levels together with their associated functions. The interconnections indicate the possible movements between levels open to the user. The highlighted blocks and underlined headings indicate existing system features.

The levels and sub-levels are described in detail in the next chapter. For this, extensive use is made of "tree structure diagrams" (eg. F'g.3.3). These diagrams are a representation of the modes which exist, and the transitions which are possible between them. Each mode corresponds to a unique set of 10 function keys, and each of the keys can potentially lead to a new mode. In the figures, all the defined keys of a mode are shown on the same line, with arrows indicating transitions. For a detailed walkthrough of one of these trees, section 3.3 should be studied. For reasons of clarity, the tree structure diagrams are presented in fragmented form. Appendix B, however, presents all these diagrams in a detailed summary of the system's available levels.

3 THE EDITING FACILITIES

This chapter is a detailed account of all the editor's functions. With the aid of suitable examples, the reader is lead through the various modes and methods of the syntax-directed PDL generator. After a look at the line editing facilities, the reader is introduced to the various system levels in top-down order. It will be useful to refer back to Fig.2.4 every now and then, so as not to loose track of the overall picture.

3.1 The Line Editor

The Line Editor plays an important part in the flexibility of the system. It is responsible for the management of text entered in the Window Screen. It is easy to note when the Line Editor is in use: the cursor is positioned in the Window Screen. In this case, a keyboard response is usually expected.

The user is presented with a line in the Window Screen which is to be edited. Key words (ie. "reserved words" generated by the package) are highlighted, and the user is not permitted to edit these words in the Line Editor. All indentation is removed, and the user is given a limited line length the size of the screen width.

There are *essentially two modes of operation: the Text-enter mode* and the Insert mode. The text-enter mode is the familiar mode where the user can overtype existing characters or add extra characters after the end of the line. Besides the character-by-character horizontal cursor movements, the cursor can also be moved to the beginning or end of the line by the depression of a single predefined key. A function is also available to erase the end of the line from the current cursor position.

The Delete function is available for deleting a single character at a time. If the user is positioned say in the middle of a text line, depression of the backspace key will cause the character to the left of the cursor to be deleted as the cursor moves back by one position to the left together with the entire string on its right. Thus the name "destructive backspace" assigned to this key.

This type of deleting is used by some editors and amounts to the following: the text to the right of the cursor including the character on which the cursor is positioned is left intact but moves to the left thus deleting the character immediately to the left of the cursor. Although this method is difficult to become acquainted with initially, it supplies the experienced user with just as much flexibility and power as do other methods (eg. that used by the IBM machines).

As a second delete key is found on IBM compatable keyboards, (the DEL key) a second type of deleting could also be included. Here,

the cursor remains stationary, while the character that was at the cursor position is deleted. At the same time, the string to the right of the cursor moves one character to the left. The operation of these two types of character deleting funtions is shown graphically in Fig.3.1.

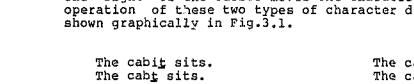


Fig.3.1: The Line Editor's character delete funtions
The two types of delete functions are shown to correct the phrase "The cabit sits." to "The cat sits.". Method (a) uses the destructive backspace, while method (b) uses the DEL key. The highlighted, underscored character indicates the cursor position.

An insert mode is also necessary and this mode can be toggled using the INS key. On entering Insert mode, the text after the cursor (including the character under the cursor) will move to the right by one position; a blank character being inserted at the cursor position. The cursor therefore now lies under a blank token and any text entered will be inserted here. The cursor will move with the entered text, thus always remaining under the blank token.

This insert token is used as a visual cue to remind the user that he is in Insert mode. Even though this cue may not always be evident, the cursor movement keys will give a definite indication of the mode in use: moving the cursor while in Insert mode will *result in the next character exchanging places with* the blank insert token. The Erase-end-of-line and Delete functions operate in the same manner as in Text-entry mode. On exiting Insert mode using the INS key again, the text after the cursor is moved back by one position and the blank insert token vanishes. Figure 3.2 gives an example of Insert mode operation.

Operation	Visual Representation
The original line	The cat sits.
Enter Insert mode	The ca_t sits.
Move cursor left	The c_at sits.
Use destructive backspace key	The _at sits.
Enter "g"	The g_at sits.
Enter "o"	The go_at sits.
Exit insert mode	The goat sits.

Fig.3.2: The Line Editor's Insert mode operation
Here, cursor movement, deleting and typing are demonstrated in an exercise to change "The cat sits." to "The goat sits.".

The user is allowed to edit the entire unhighlighted text string and if an attempt is made to type beyond the line limits, a warning will appear in the Prompt Screen. The last character on

the line will always be overtyped, with an audible warning to the user that the end of the line has been reached. When the line has been satisfactorily edited, the Enter key can be depressed for the system to accept the new line. If the old version of the line (still visible in Screen 1 at Cursor 1) is required, the ESC key will exit the line editor, ignoring any changes made to that line.

A summary of the function keys which are operational in the Line Editor, together with their associated meanings is presented below:

->	Cursor moves right by one position
<-	Cursor moves left by one position
HOME	Cursor moves to the beginning of the line
END	Cursor moves to the end of the line
CTRL K	Erases from cursor position to the end of the line
<==	Destructive backspace deleting function
DEL	Another deleting function
INS	Toggles Insert mode on/off
ENTER	Exits the Line Editor and accepts new text
ESC	Exits the Line Editor ignoring any modifications

3.2 The Front End of the Package *Note: this level is as yet unimplemented*

As explained in Chapter 2, the Front-end of the PDL generator package is responsible for dealing with input and output to the external world. With reference to Fig.3.3, the operation of this interface will be clarified.

On entering the editor package, the system tables are loaded. The user will initially be placed on the level containing the function key options EDIT, INFO and EXIT. This can be regarded as the system's top state. The user may then choose a function key to proceed further.

The EXIT key will exit the editor and return to the computer's operating system. The INFO key can be used to obtain further information on how to operate the system. On having obtained this information, the user is returned to the package's top level.

The EDIT function key will lead the system into asking the user to enter a filename which is to be edited. The cursor will be positioned in the Window Screen, and using the Line Editor facilities, a filename can be specified. (Note that as a ".COD" extension is assumed, no extension must be specified here.) Once the ENTER key is depressed, the package will load the required file (if it exists), and move on to Base Level.

The Base Level is characterized by the function keys: PageB, PageF, TopF, BotF, ToLin, EditLn, End and More. It is thus a simple procedure to enter the PDL generator package. From here, the file can be edited until the user is satisfied. When the user wishes to exit the editor, or edit another file, Base Level must be returned to.

Here, the END key will cause the system to ask the user: "Do you want to SAVE or ABANdon this file?". Simultaneously, the choices

SAVE, ABAN and RETURN will appear as function key options. The RETURN option will (as conventionally expected) return the user to Base Level again, ignoring the last question.

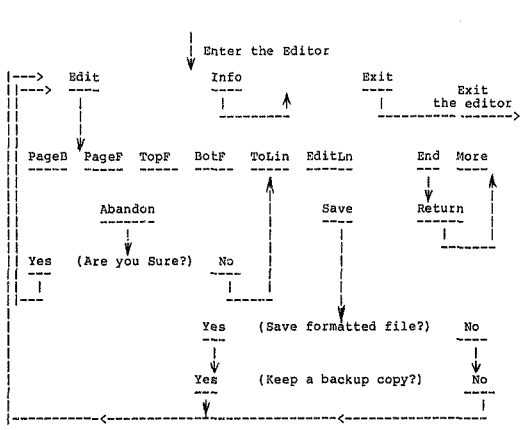


Fig.3.3: The front-end tree structure
Bracketed questions show the display which appears in the Prompt Screen, and to which the user must respond.

If the ABAN option is chosen, the system will ask: "Are you sure you want to ABANDON all the edits of this file?". To this, the user answers (still using the function keys) either YES or NO. If abandoning is not really required, the user will be returned to Base Level. If the user, however, confirms his desire to abandon or quit the file, he is returned to the package's top level, and any edits or changes he may have made to the file during the latest session, will have been lost or ignored. This abandon or quit function is useful where unwanted or unintended edits have been made to a file.

If, on the other hand, the SAVE function is chosen, the system will enquire: "Do you want to save a copy of the formatted file?". The file alluded to here is the prettyprinted version of the file which is actually presented to the user on Screen 1. If a copy of this is required in a file for printing or documentation purposes, then the user would choose the YES option; otherwise, the NO function key needs to be depressed.

Either decision will lead to the next question: "Do you want to keep a BACKUP copy of the old file?". For safety, it is suggested that the novice user always keep backup copies of the old file versions. This can prove useful if two slightly different versions of a PDL design are to be kept. Whether a backup copy is required or not, the user is then taken back to the system's top level.

The package will write the newly edited file into a file with the name given to it initially. The formatted file will have extension ".PDL", while the coded version will be labelled ".COD". Backup files will end in the letters BU, and will be labelled with their appropriate extensions.

The user is now again in the package's top level, and from here, he can either choose to edit another file (or indeed the same file), or exit the PDL generator package. If the user chooses to edit another file, the system tables need not be re-read from disk; only the new file to be edited needs to be loaded. This saves much time and user frustration in the long-run.

3.3 The System's Base Level

Base Level has effectively a set of 20 function keys due to its many functions. This is achieved by reserving one function key (the MORE key) for the sole purpose of displaying the remaining 10 functions. It is important to understand that, while the function key definitions are changed on depression of the MORE key, the Base level is still effective. Fig.3.4 below defines the system's Base Level.

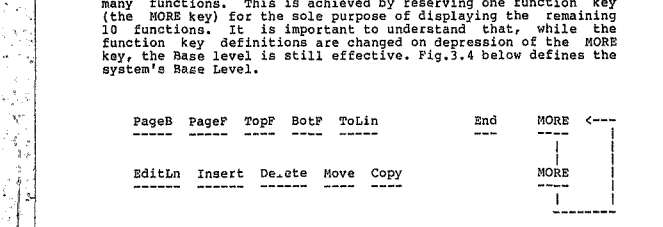


Fig.3.4: The Base Level tree structure

3.3.1 The Scrolling functions

Approximately half of the Base Level is essentially devoted to the scrolling functions. The user will make use of these facilities for viewing the PDL file in Screen 1 (20 lines at a time). It is in order to mention again that the cursor up, cursor down, page forward and page backward keys, being separate from the function keys, are operational in all major levels.

The cursor up and down keys will move Cursor 1 on Screen 1 up and down respectively. If the cursor is moved beyond the limits of the logical screen, Screen 1, a half page scroll (either forward

or backward) is performed, thus positioning the cursor centrally on Screen 1. If it is attempted to move the cursor beyond the first or last line of the file, an appropriate message will be displayed in the Prompt Screen.

The PageB and PageF function keys have the same associated functions as the explicit PgUp and PgDn keys available on the keyboard. They will, where possible, scroll the file backward or forward respectively by a page. Under normal circumstances (ie. if the file is long enough) the page will be advanced by 19 lines (ie. in scrolling forward, the bottom line on Screen 1 before the scroll will become the top line after the scroll); the cursor remaining at the previous screen position. If the file is too short, a page scroll is attempted, with the cursor being positioned in the appropriate position.

The TopF and BotF keys will move Cursor 1 to the top or bottom lines of the file respectively. The ToLin function key will allow the user to move Cursor 1 to any valid line number in the file. The line number required must be entered in the Window Screen. Only the line numbers between the current top and bottom line numbers of the file will be accepted. Although the user is limited to a maximum of a 4 digit line number (a file longer than 9999 lines is quite unlikely not to mention counterproductive), the full line editing facilities are available when in the Window Screen. A valid line number which requires a screen scroll will always (if possible) position Cursor 1 at the centre of Screen 1.

As shown in Fig.3.5, after the execution of any of the scrolling functions, the user is returned to Base Level again. The END key is the only key in Base Level which could move the user to another permanent level (as explained in section 3.2).

Fig.3.5: The scrolling functions' tree structure

3.3.2 The Modify or Edit Line function

This is one of the most important function keys for editing a file on a line by line basis. Any line displayed on Screen 1 in unhighlighted or reverse video fonts (correctly user entered text, placeholders and errors) can be edited using this function. *This is because highlighted, system-generated key words are not* editable. A line on Screen 1 which contains a key word only, is thus not editable. Any other line will be at least partially editable.

If a line is editable, it will be loaded into the Window Screen, from which the Line Editor can be used to modify its contents. When the user is satisfied with the edited line, it is ENTERed, and the new line will replace the old at Cursor 1 on the Main *Screen. If the edits performed* on the line in the Window Screen,

however, are not required, the ESC key will return the user to Base Level without making any changes to the chosen line on Screen 1.

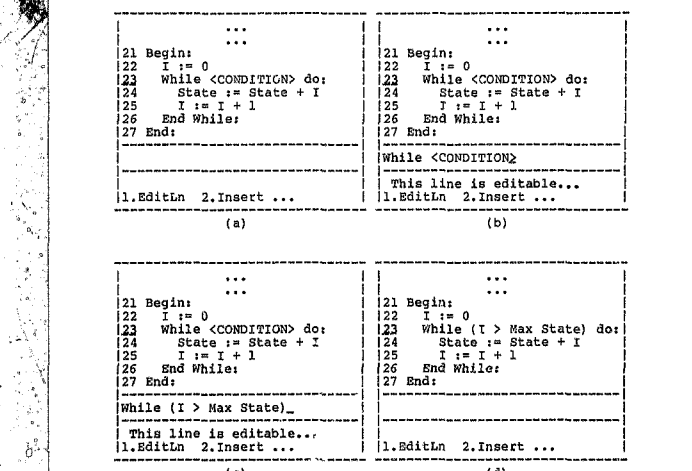


Fig.3.6: Using the EditLn function

The highlighted, underscored character in the figures above represent Cursor 2 on the Window Screen. In the Main Screen, Cursor 1 is shown as a highlighted, underscored line number, while highlighted words represent system-generated key words. In figure (a), the required line to be modified is chosen with Cursor 1. Figure (b) shows the result after the EditLn key has been depressed. It can be seen that the Window Screen is loaded and the user is prompted. Figure (c) shows the Window Screen after the user has edited it using the Line Editor, and figure (d) is the result after the new line has been ENTERed.

This function is typically used when expanding a <CONDITION> placeholder or when editing an error. In the Data Description segment, only the user-entered data items can be modified. On exiting the Line Editor, parsing of the new line is done by the

system, and any errors introduced will be displayed on Screen 1 in reverse video font. A sample editing procedure is shown in Fig.3.6.

3.3.3 The Insert facility

From Base Level, Insert mode can be entered using the appropriate function key. The Insert key is basically a gateway into the insert functions provided by the system. (This also applies to the Delete, Copy and Move keys.) As shown in Fig.3.7, returning from Insert mode is simply achieved by depressing the RETURN function key.

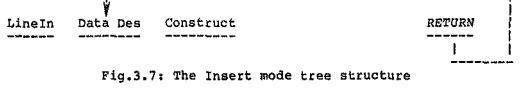


Fig.3.7: The Insert mode tree structure

The necessary functions in Insert mode include:

--- the Line Insert facility for entering text or program lines sequentially

--- the Data Item Insertion facility for automatic definition and placement of data items used in the PDL program

--- the Construct Insertion facilities used for inserting construct blocks (with associated placeholders) in the PDL program's algorithm segment

These functions will be sufficient for the writing of any PDL design. Placeholder deletion is automatic when required. The construct facility will ensure that syntactical correctness is maintained. If each defined data item is coded, and all lines of text are parsed, full semantic error checking facilities will become available. Section 3.4 gives the details of the functions available in Insert mode.

3.3.4 The Delete facility

Fig.3.8 shows the tree structure relevant to the Delete mode. Again, the Delete key in Base Level can be regarded as a gateway to the delete functions available in the package. The Delete functions include:

--- Single Line Deletion

--- Construct Deletion
--- Block Deletion

All the delete functions require some type of user input to indicate the lines which should be deleted, and extensive error checking is performed to protect the user from himself. Placeholder insertion is automatically done when necessary. Data items can be deleted either by using the Single Line Deletion facility directly on the appropriate line, or by using the Block Deleting facility.

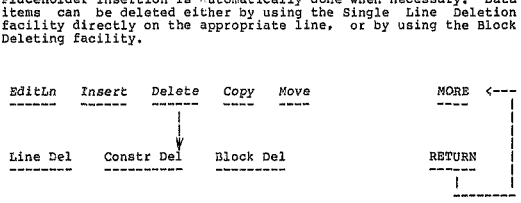


Fig.3.8: The Delete mode tree structure

Construct Deletion is useful when in the Algorithm segment, while Block Deletion is more general. Here, the user is allowed to enter a delete range; the system checking whether such a range will affect the syntax of the remaining PDL program. If not, the entire specified block will be removed. In this mode, extensive visual cues are used to facilitate block identification. Section 3.5 of this manual will elaborate on the details of Delete mode functions.

3.3.5 The Copy facility

Another gateway facility is provided here for copying lines of a *PDL program elsewhere in the program.* Fig.3.9 depicts the function keys involved. Basically, only two types of copy functions exist:

--- the Line Copy facility
--- the Block Copy facility

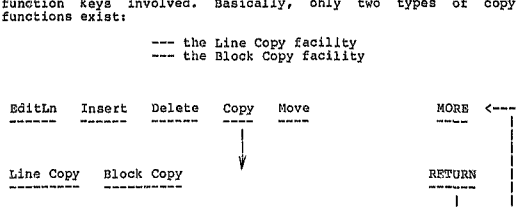


Fig.3.9: The Copy mode tree structure

No copy (or move) operations are permitted in the Data Description segment. The major criterion between distinguishing between the two types of copy functions is convenience. A Block Copy requires the user to define a legitimate block of PDL (which is highlighted), and, if this is accepted by the system, a destination can then be specified; this too being system-checked. Section 3.6 gives a more detailed description of the Copy functions.

3.3.6 The Move facility

The Move gateway is used from Base Level to access functions *which will enable the user to move lines of PDL around in the* program without disturbing the syntactical correctness of the PDL structure. Fig.3.10 shows the tree structure adopted.

Again, similarly to the Copy functions, two functions have been provided for user convenience. During a Move operation, the block to be moved must be accepted (and highlighted) before a destination (this also being checked) can be specified. Extensive system checking is required to maintain the PDL program syntactically correct. Further details on these functions can be obtained in section 3.7.

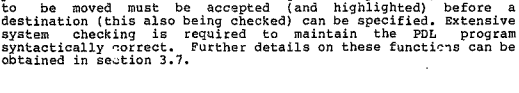


Fig.3.10: The Move mode tree structure

3.4 Insert Mode

3.4.1 Single Line Insertion

This function is indispensable when using any editor. It enables the user to enter a number of text lines (not constructs) sequentially. Fig.3.11 shows the tree structure of this construct.

Fig.3.11: The tree structure for Line Insert mode

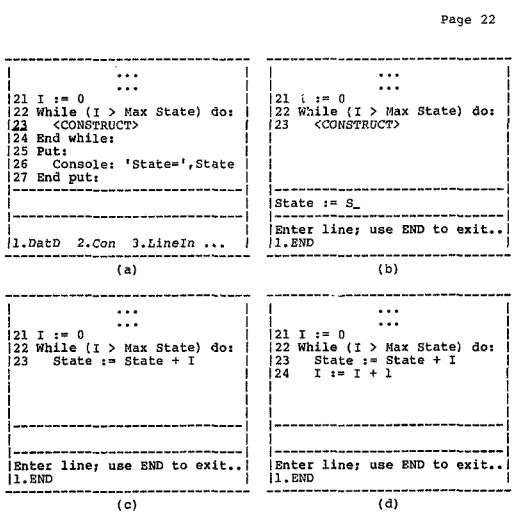


Fig.3.12: An example using the Insert Line facility
Two lines of code are to be inserted at line 23 of figure (a). Figure (b) shows the result after depressing the LineIn key and typing part of the required line. Figure (c) shows the layout after the new line has been ENTERed. Figure (d) shows another user-entered line of text. The last figure (figure (e)) portrays the final product after the END key has been depressed.

Assume, as an example, that a placeholder is to be expanded to a few lines of PDL code. Cursor 1 is placed on the line after which insertion is required, and the LineIn key is depressed. This results in Screen 1 being cleared of all lines after the line indicated by Cursor 1. The cursor on this screen is also turned off. (Thus, if no cursor is present on Screen 1, then the user can safely deduce to be in Line Insert mode.)

Cursor 2 appears in the Window Screen, and full line editing facilities are available for entering the text line. When a line is entered, it will be inserted (with correct indentation) after the last line currently displayed on Screen 1. Each depression of the ENTER key will cause whatever text is displayed in Screen 2 to be added to the last line of Screen 1. Once the ENTER key has *been depressed, there is no way of editing the newly entered line* without leaving the present mode (Insert mode).

If instead of ENTER, the ESC key is used, the line present in the Window Screen will be discarded, and no line will be added to Screen 1. The END function key is used for exiting this mode. It should be remembered that all lines which are to be inserted, must be ENTERed before the END key is used. The END key will return the user to Insert mode; restore Screen 1 to contain the complete 20 lines; and return Cursor 1 to the last line which was inserted.

While inserting, this mode is similar to the Construct Insert mode in that indentation is automatic, and unnecessary placeholders are removed. The pictorial example in Fig.3.12 should clear up any uncertainties in the above verbal description.

3.4.2 Data Description definition

The Data Description segment is perhaps one of the areas where automation can be of major help in alleviating the designer's clerical effort. This segment is usually present at the beginning of a program, and it is here that all constants and variables are defined. Due to the strict indentation and ordering laws which apply to this segment, manual entry in this area is usually very *frustrating and time consuming. For this reason* it can be regarded as a prime candidate for automation.

When the DataD key is depressed, Data Description Inset mode is entered, and here, the format of the Window Screen changes somewhat (see Fig.3.14). A data item definition field is defined in the Window Screen. The user is asked to enter or edit the data item name here. A tree structure is presented layer by layer via the Function Screen. The user makes use of the function keys to choose the data function, data type, data structure and data scope successively. This tree structure is seen in the diagram of Fig.3.13.

The Window Screen is also used to display these various classifications of the data item so that only once the user is through with defining the data item can it be accepted for automatic indentation and positioning on the Main Screen. A diagram of the Window Screen in Data Definition mode is shown in Fig.3.14.

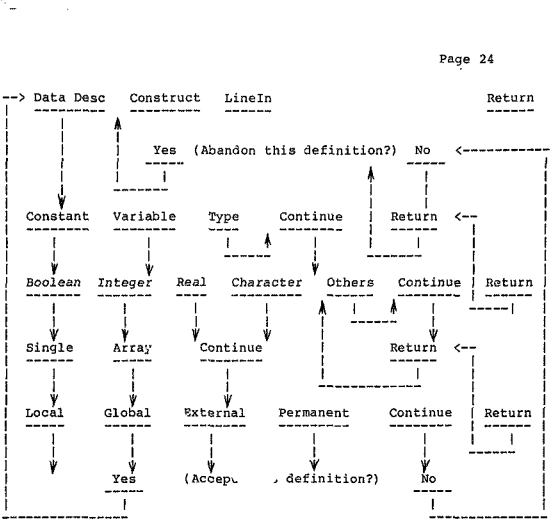


Fig.3.13: The tree structure of the Data Description Insert mode
Although all prompts and conditional branches have not been labelled for the sake of clarity, bracketed questions indicate prompts requiring user response.

In the Data Type level, choosing of the Others key option will *result in a new set* (if *any* exists) of user-defined data types. Types are defined by entering a text line; except where records are defined, in which case a more structured approach is taken. The functions associated with these two related keys (Type and Others) have not yet been implemented.

The Data Definition tree can be traversed in both the forward and backward directions for key word editing purposes using the Continue and Return keys respectively. Visual representation of the user's position in the tree is given by highlighting one of the fields of Fig.3.14 in reverse video font. Each of these fields is individually editable. Simultaneously, the user is able to edit the data item in field 5 with the full power of the Line Editor.

After the user has defined the data item, he is prompted for its acceptance or rejection. A newly defined data item will be positioned correctly in the Data Description segment of the program without further user intervention. If the defined data

item is to be rejected, no action occurs.

If Cursor 1 is currently in the Data Description segment of the program on Screen 1, then the cursor is automatically positioned at the newly inserted data item. If the cursor is in the Algorithm segment of the program, then the cursor remains where it is, with only the different line numbers showing that an insertion has occured in the Data Description segment. Thus, a user can define any new variable while in the program body, without having to return to the Data Description segment.

```
|Variable: Boolean: Array: Local: This array of size (1..100)  |
|----------|--------|------|------|-----------------------------|
| Field 1  | Field 2 | Field 3| Field 4|          Field 5           |

Field 1 -- The Data Function field
Field 2 -- The Data Type field                  (10 characters wide)
Field 3 -- The Data Structure field
Field 4 -- The Data Scope field
Field 5 -- The editable Data Name field  (40 characters wide)
```

Fig.3.14: The Window Screen in Data Description mode
The field written in boldface and underscored represents the field highlighted in reverse video font. This field will correspond to the level (See Fig.3.13) that the user is on. The highlighted, underscored character in Field 5 indicates the position of Cursor 2.

An entered data item can only be deleted by deleting the line containing the data item, or by redefining it. (The user is asked whether he wishes to replace the old item definition with the new.) This makes the Data Description segment fully automated as the user never deals explicitly with its formatting.

3.4.3 Construct Insertion

The system is a template-driven syntax-directed editor so that the Construct Insertion facility consists of the various constructs available in PDL. When a construct template is chosen, the skeleton structure is displayed on Screen 1, with non-terminals or placeholders highlighted in reverse video font. The user can then choose to fill in the placeholders using the editor's normal functions, or return to them at a later stage. A list of the templates available is given below:

--- If - Then
--- If - Then - Else
--- While - Do
--- Repeat - Until
--- Case
--- Case - Else
--- Co-begin-Co-end
--- Get
--- Put

Insertion of a construct will occur after the line on which Cursor 1 is positioned. The user is warned if Block Insertion is not allowed in the chosen position. Indentation of the construct block is automatic, and any unnecessary placeholders are removed. Fig.3.15 below shows the steps involved in inserting a nested If-then-else construct.

```
                                  ...
                                  ...
21 I := 0                                  21 I := 0
22 While (I > Max State) do:               22 While (I > Max State) do:
23    <CONSTRUCT>                          23    If <CONDITION>
24 End while:                              24       then:
                                  ...      25          <CONSTRUCT>
                                  ...      26       else:
                                  ...      27          <CONSTRUCT>
                                  ...      28    End if:
                                  ...      29 End while:
-------------------------------------      -------------------------------------
Use f-keys & choose construct              1.DatDes  2.Con  3.LineIn...
1.If-t  2.If-t-e  3.While...
-------------------------------------
            (a)                                        (b)
```

Fig.3.15: Inserting a block construct

Positioning Cursor 1 on line 22 or 23 will have the same effect, as the unwanted placeholeder is removed. Figure (a) is the screen layout before Construct Insertion, while figure (b) shows the result after depressing the If-t-e function key.

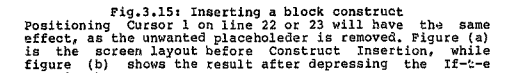


Fig.3.16: The tree structure for Construct Insert mode

It can be seen in Fig.3.15, that the indentation is automatically maintained, while the existing <CONSTRUCT> placeholder is removed; the If-then-else template (with its own placeholders) *being inserted. It can also be noted* that the cursor is positioned on the first line of the template which contains an unexpanded placeholder. When the construct has been inserted, the user is returned to Insert mode so that the Insert Line function can then be used to expand the placeholders. This can also be

seen from the tree structure of Fig.3.16.

3.5 Delete Mode *Note: This level is as yet unimplemented*

3.5.1 Single Line Deletion

This function forms one of the primitives of any conventional file editor. In this PDL generator package, this function will enable the user to delete the line on the Main Screen on which Cursor 1 is positioned. Due to the nature of the syntax-directed editor, however, not any line may be deleted individually. If a line may not be deleted, an error message will appear in Screen 3 and the user will be returned to Delete mode. Fig.3.17 shows the relevant tree structure for this sub-mode.

Fig.3.17: The tree structure for Single Line Delete mode

If a line is deleted, reformating of the Main Screen and line renumbering is automatic. The scope of this function can be divided into two categories: the lines in the Data Description segment and those in the Algorithm segment.

In the Algorithm segment, after deleting a line, a placeholder will be inserted if it is required. Also, Cursor 1 remains physically in the same position in the Main Screen (ie. same line number) while indicating the following line. (See Fig. 3.18) A line containing any key word (ie. a highlighted word in the Main Screen) may not be deleted via the Line Delete function. This restriction is necessary so as to maintain the syntactical correctness of the PDL program at all times. Key words in the Algorithm segment can only be deleted as they were inserted: via special block manipulation functions. The Construct and Block Delete modes serve this purpose. (See sections 3.5.2 and 3.5.3)

In the Data Description segment, two types of deletion are possible: single and multiple data item deletion. The Line Delete function can be used to delete a single data item at a time (together with any relevant system-generated definition key words). In the Data Description segment, the user has no direct control over the data item definition structure: when an item is to be deleted, all the relevant key words will be arranged accordingly without further user intervention.

Similarly to the Algorithm segment, a system generated key word cannot be directly deleted in the Data Description segment of a PDL program. Thus, Cursor 1 must be on the line containing the data item name when that data item is to be deleted via the Line Delete function. After a data item is deleted, Cursor 1 is

positioned at the line following the deleted data item name. When deletion of more than one adjacent data item is required, the Block Delete function can be used (See section 3.5.3).

```
|                 ...                |
|                 ...                |
|                 ...                |
|21 I := 0                           |
|22 While (I > Max State) do:        |
|23   State := State + I             |
|24   I := I + 1                     |
|25 End While:                       |
|26 Put:                             |
|27   Console: 'State=',State        |
|------------------------------------|
|                                    |
|------------------------------------|
|                                    |
|1.LineDel  2.ConstrDel ...          |
--------------------------------------
                  (a)

|                 ...                |
|                 ...                |
|                 ...                |
|21 I := 0                           |
|22 While (I > Max State) do:        |
|23   I := I + 1                     |
|24 End While:                       |
|25 Put:                             |
|26   Console: 'State=',State        |
|27 End Put:                         |
|------------------------------------|
|                                    |
|------------------------------------|
|                                    |
|1.LineDel  2.ConstrDel ...          |
--------------------------------------
                  (b)

|                 ...                |
|                 ...                |
|                 ...                |
|21 I := 0                           |
|22 While (I > Max State) do:        |
|23   <CONSTRUCT>                    |
|24 End While:                       |
|25 Put:                             |
|26   Console: 'State=',State        |
|27 End Put:                         |
|------------------------------------|
|                                    |
|------------------------------------|
|                                    |
|1.LineDel  2.ConstrDel ...          |
--------------------------------------
                  (c)
```

Fig.3.18: Using the Line Delete function

Lines 23 and 24 in figure (a) must be deleted. Delete mode is thus entered. Depressing the LineDel function key, figure (b) results. After performing another line delete, figure (c) is obtained. It will be noted that the <CONSTRUCT> placeholder was inserted for the purposes of maintaining syntactical correctness. Further line deletes on line 23 will produce an error.

3.5.2 Construct Deletion

This function is supplied purely for the sake of convenience. It allows the user to remove an entire construct from a PDL program with the use of only two function key depressions. In so doing,

the remaining program will maintain its syntactical correctness. The tree structure for this function is shown in Fig.3.19.

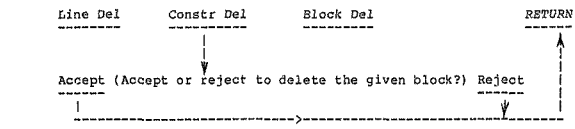


Fig.3.19: The tree structure for Construct Delete mode

This deletion mode can only be available in the Algorithm segment. The user will position Cursor 1 in the Main Screen at a particular line. After depressing the ConstrDel function key, the line numbers of the group of lines, above and below the chosen line, with an indentation equaling or exceeding the indentation of the chosen line, will be highlighted in reverse video font. At this point Cursor 1 vanishes.

The user is prompted for acceptance to delete the block which is highlighted. If he rejects, Cursor 1 re-appears, while the block line numbers are reverted to normal font. If the user accepts to delete the block, the highlighted line numbers will be removed from the file, and the cursor positioned at the line after the deleted block.

This function thus enables one to delete blocks of text rapidly. At all times, syntactical correctness is maintained and any placeholders which are required will be automatically inserted.

3.5.3 Block Deletion

This is the more general delete function which is available in most simple editors. The function allows the user to delete a specifiable block of text. In this PDL syntax-directed editor much checking of the chosen block must be performed so that only a deletion which will not affect the syntactical correctness of the PDL program will be accepted. The tree structure for this function is shown in Fig.3.20.

If a block of text is to be deleted, after the Block Delete function key has been depressed, the user is prompted to use the Begin Block key when Cursor 1 on Screen 1 indicates the beginning of the block to be deleted.

The user has the choice to use either the cursor movements coupled with the Begin Block function key to indicate the block start line, or use the Line Editor to specify a numeric line number. Thus, the Window Screen will contain the message: "Start line number =".... A line number is ENTERed for acceptance. Thus, if the Begin Block key is used to enter the numerical line number

in the Window Screen, an error will occur. This means that the Begin Block function key chooses the line indicated by Cursor 1 on Screen 1 only when there is no line number specified in the Window Screen.

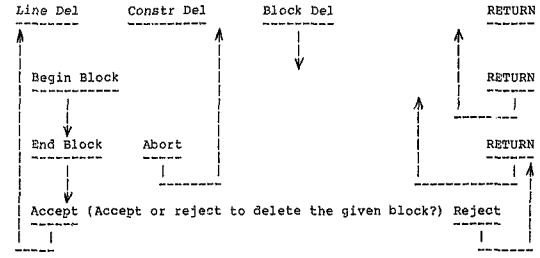


Fig.3.20: The tree structure for Block Delete mode

Before a start line can be accepted, it must be checked. If any error occurs (eg. trying to delete the first line in the program) the user is warned and the level of operation remains unchanged. If no error occurs, the next state is entered to allow the user to specify the end of the block which is to be deleted. It is worthy to note here that the RETURN function key will position the user in the previous system state, thus implying an exit from Block Delete mode.

In choosing a line to end the delete block, again the user can make use of the cursor movements together with the End Block function key, or the *Line Edit features for specifying* a numerical line number. The Window Screen prompt will now be:

"Start line number = XX; End line number ="...
where XX is the numerical value of the previously chosen line

A useful feature here is the dynamic highlighting of any lines chosen for the delete block as *Cursor 1 is moved.*

At this level, the RETURN function key will move the user back to the stage where an initial start line is still to be specified. The Abort function key is useful for cleanly aborting any unwanted operation at this stage.

Once an end line has been chosen, *extensive error checking occurs* to determine whether the chosen block can be deleted without disrupting the PDL program's syntax. As an example, the If-then part of an If-then-else construct cannot be deleted, as it leaves an incomplete (or syntactically incorrect) construct in the

program.

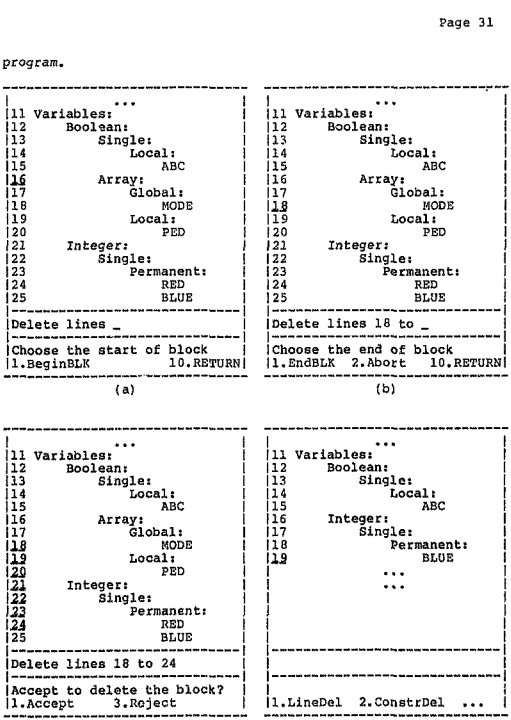


Fig.3.21: Using the Block Delete function

An error-free example of Block Deletion in the Data Description segment is shown above. Figure (a) shows the screen layout after the BlockDel function key has been depressed in Delete mode. The Line Editor features are used to ENTER the line number "18". As shown in figure (b), Cursor 1 moves to line 18 and the user is prompted for the end line of the block. This time, the cursor movement keys

are used. As the cursor moves, the lines from line 18 to the current cursor position are highlighted dynamically, and when the EndBLK function key is depressed, the display screen will be as in figure (c). Note that in the Window Screen, the end line number was automatically filled in. Figure (d) shows the final result after the block from line 18 to line 24 has been deleted. Note that block limits start and end at a data item name. Note also that, although the block specifies lines 18 to 24, only the data items included in that block have been deleted. (Any necessary key words included in the block have been retained, while unnecessary key words outside the block limits have been deleted.) Key word ordering has also been dealt with automatically.

It is important to note that a block with a start line in the Data Description segment and an end line in the Algorithm segment is not acceptable. Only a pure Data Description segment delete block, or a pure Algorithm segment delete block will be accepted.

In the Data Description segment, the start and end lines of the delete block may only contain a data item name. A block starting or ending on a data definition key word will not be accepted. All the deletion of intermediate key words will automatically be taken care of by the system. Fig.3.21 will help to illustrate the operation and power of the Block Delete function as applied to the Data Description segment.

When an acceptable block has been defined, it will be highlighted, and the user will be asked if he wishes to delete the chosen block. Rejection will return the user to redefine the end line. (From here, this mode can also be aborted.) Acceptance will automatically delete the indicated block, with any necessary placeholders inserted. In the Data Description segment, all unnecessary key words will automatically be removed. The Main Screen is reformatted; Cursor 1 is positioned on the line after the deleted block; and the user is returned to Delete mode.

3.6 Copy Mode *Note: This level is as yet unimplemented*

3.6.1 Single Line Copy

The Line Copy function enables the user to copy any line of text to any destination, if in so doing the PDL program remains syntactically correct. As shown in the tree structure of Fig.3.22, first the line to be copied, and then the destination need to be specified by the user.

Again, as in Delete mode, the user is given the ability to enter the required inputs either pictorially (by using the cursor movements together with the This Line or After This Line function keys) or numerically (using the Line Editor facilities in the Window Screen).

After depressing the Line Copy function key, the Window Screen prompt will be: "Copy line "..... The user then either moves Cursor 1 to the required line and uses the This Line function key

to choose it or ENTERs a line number in the Window Screen. The chosen line number (corresponding to the line number which will be copied) is displayed in reverse video font, and the user is prompted for the destination. If instead of choosing a copy line, the RETURN key is used, Line Copy mode will be abandoned.

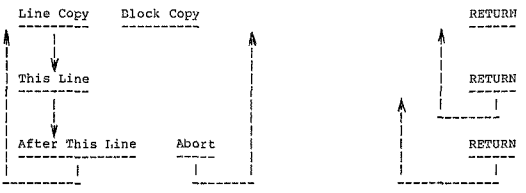


Fig.3.22: The tree structure for Line Copy mode

Note that no acknowledgement from the user is required. The reason for this is twofold: the Line Copy function involves the addition of only one extra line; and a copy process is in any case easily reversed by deleting any extra lines created.

When a copy line has been chosen, the Window Screen will display:

"Copy line XX after line "...
where XX is the line number of the copy line.

The line which is to be chosen is the line after which the copy line will be copied. Use of the Abort function key at this stage will cleanly abandon this Line Copy mode, whereas the RETURN key will request that the user re-enter the copy line (ie. previous state).

Both the copy line and the destination lines are checked for possible syntactical errors. For example, no line containing a key word or a placeholder may be copied. Also, neither the copy nor destination lines may occur in the Data Description segment. Essentially, only a user-entered algorithm statement may be copied. It is important to note that when copying is performed, indentation of the copied line is calculated, and the new line automatically formatted. After a copy line operation Cursor 1 is positioned at the new copied line.

3.6.2 Block Copy

The Block Copy function is the more useful function where a block of text can be copied to another section of the file. The user will specify a block of text and its destination. If the required copy operation will not affect the PDL program's syntactical

correctness, then it will be performed. The tree structure for this function is as shown in Fig.3.23.

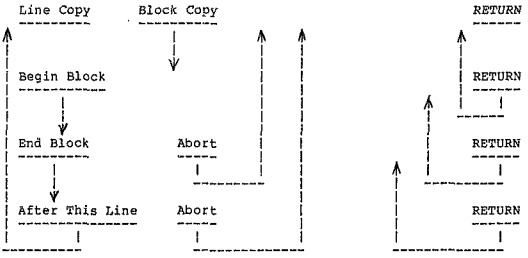


Fig.3.23: The tree structure for Block Copy mode

Again no acknowledgement from the user is required before the operation can be executed; the reason being a compromise between system user speed and user friendliness.

The copy block is chosen in the usual way, with the Window Screen presenting a numerical input alternative. The format used for this purpose is:

"Copy lines XX to YY after line ZZ"

where XX = start of the copy block line
YY = end of the copy block line
ZZ = destination line

The block is highlighted as it is chosen (ie. dynamically). At all times the RETURN function key will essentially undo the previous choice, thus taking the user back one level. The Abort key is available for cleanly exiting this Block Copy mode ignoring any choices made.

When the copy block has been chosen and highlighted on the Main Screen, an alternative Cursor 1 font is required so that the user can choose the destination. A blinking line number in the margin of the Main Screen is thus used for Cursor 1. As destination, the user must indicate the line after which the copy block is to be placed. Once a destination line has been chosen (either by using the After This Line function key, or the Line Editor facilities) the copy operation will immediately (if it is legitimate) be performed.

Clearly, to maintain the program's syntactical order, checking is done at every stage of user input. No copy function can be performed (wholly or partially) on the Data Description segment. Only entire constructs may be copied; part of a construct cannot

be copied unless it contains no system generated key words.

On successfully copying a block of text, the new block will be appropriately indented according to its location in the PDL program. Any extra placeholder in the destination's vicinity will be removed. Cursor 1 will be positioned at the beginning of the newly inserted text block afer the copy operation is complete.

3.7 Move Mode *Note: This level is as yet unimplemented*

This mode basically involves the use of a Copy function followed by a *Delete function.* (Firstly the original lines are copied to the destination; then the original lines are deleted.) Thus the Move mode is not regarded as a primitive editor function, but a derived function. Although this is the case, most modern editors include a Move facility as it greatly alleviates the user's editing burden.

3.7.1 Single Line Move

The user makes use of this function to move a single line to another file position. In so doing, the PDL program's syntax must remain intact. From the tree structure of Fig.3.24 it can be seen that no confirmation of the action is requested from the user. Again this is done so as to increase usage speed. The Move function is also easily reversed (particularly for a single line).

Fig.3.24: The tree structure for Line Move mode

The user is presented with an almost identical procedure as that of Line Copy mode. Input lines are entered via the function keys This Line and After This Line or using the Line Editor in the Window Screen to enter the line numbers directly. The Window Screen prompt will be as follows:

"Move line XX after line YY"

where XX = the line number to be moved
YY = the line number after which line XX must be moved

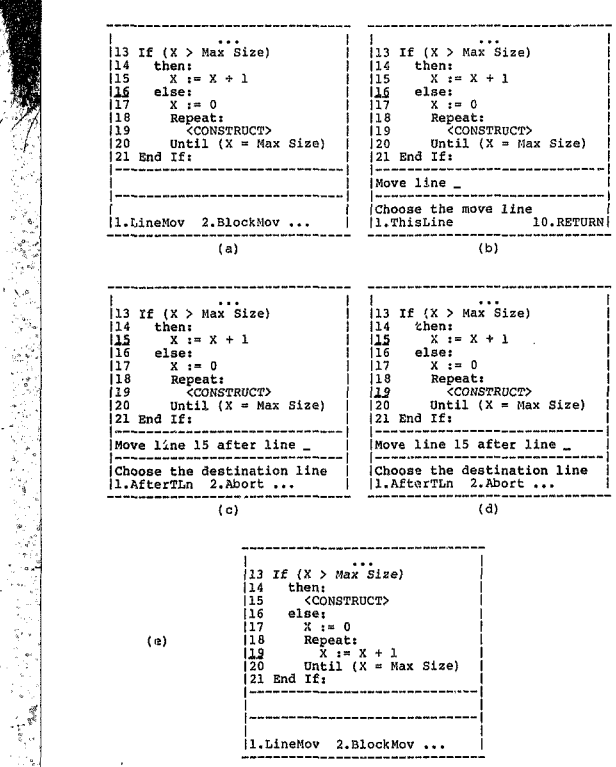


Fig.3.25: Using the Line Move function
The screen layout in Move mode before the LineMov function key is depressed is shown in figure (a) while figure (b)

shows the layout after the Line Move function has been selected. At this stage, the Main Screen display is unchanged. The user then moves Cursor 1 (now at line 16) to line 15 and depresses the ThisLine function key. This results in figure (c). A destination line is now chosen by moving Cursor 1 (now in blinking font) to line 19 as in figure (d). After the AfterTLn function key has been depressed, figure (e) results. Note that the destination line number in the Window Screen is never displayed if the user enters it pictorially via Cursor 1 movements. Also, it can be seen that a placeholder has been inserted at line 15 while one has been removed at line 19 to maintain the program syntactically correct. The newly moved line (line 19) can be seen to have a new indentation to blend with its *new position.*

RETURN and Abort function keys have their usual meanings and provide the user with full flexibility when in this mode. The same restrictions for copying a line apply to the Move facility. Thus the user may only move completely user-entered text lines in the Algorithm segment.

When a line is moved, any necessary placeholders will be inserted or deleted as required (see Fig.3.25). The line which is moved is automatically indented to match its new surroundings. After a move operation, Cursor 1 will be positioned at the newly moved line. Figure 3.25 shows an example of using the Move function correctly with its associated features.

3.7.2 Block Move

This function is used to move a specified PDL block to a specified location in the file. As in Block Copy mode, much checking for legitimate inputs is needed to maintain syntactical correctness. The tree structure used in Block Move mode can be seen in Fig.3.26.

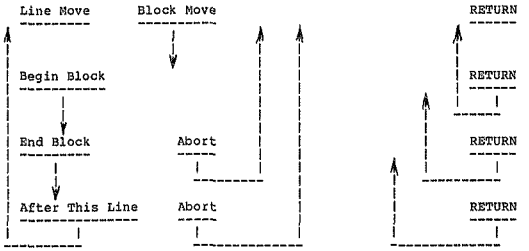


Fig.3.26: The tree structure for Block Move mode

Almost identical to the Block Copy mode operation, the user enters the move block via the two alternative methods. The Window Screen display is as follows:

"Move lines XX to YY after line ZZ"

where XX = the start line for the move block
YY = the end line for the move block
ZZ = the line after which the move block will be placed

The usual Abort and RETURN functions are available to enhance flexibility. After the move block has been chosen and highlighted on Screen 1, Cursor 1 reverts to a blinking cursor thus enabling the user to enter the destination line. The same restrictions of *Block Copy mode apply here for maintaining syntactical order.*

A successfully moved text block will be automatically indented to suit its new location. Additional placeholders will be removed, while necessary placeholders are inserted. After the operation, Cursor 1 will be placed at the beginning of the newly moved block.

4 UNIMPLEMENTED AND EXTENDED FEATURES

4.1 Package Completion

Although the full package has been described in the chapters above, many functions still have to be implemented. The present version of the system has no Front-end. It makes use of the ESC key from Base Level to exit the editor. A primitive facility for specifying the input and output files is provided in the main program, but a more flexible interface is needed.

At this stage, errors and placeholders are not displayed on Screen 1 in reverse video font. This is partially because there is as yet no form of parsing of an entered line or semantic detection such as checking for undefined variable names. The Delete function has been investigated, but not included in the package due to unforseen problems. The Move and Copy facilities also have not yet been implemented. Although the above shortcomings may seem restrictive, the skeleton structure and principal concepts have been finalized and are functional.

4.2 Semantic Checking Abilities

As PDL can be written in a very verbal manner, it is difficult to try to incorporate a compiler facility to check for semantic correctness into such a package. However, if the user is given a choice of whether parsing is to be performed or not, then designing down to any desired level is possible. A bottom level design can thus be fully compiled on entry. This would involve a parser to check each line for correctness after it has been entered. In a similar manner, constants or variables which have not been declared in the Data Description segment will be flagged as errors.

An interesting feature of the system is that information can be obtained on any error in the program (highlighted in reverse video font on Screen 1) by depressing a predefined key. A counter value can also be kept on the number of errors so that the user will know when the program is completely error-free.

4.3 Ellipsis Facility

An ellipsis function can easily be provided from the indentation values of each line. This function will ellide (or temporarily remove from view) blocks of the program on Screen 1 which are indented beyond a certain ellipsis level. This will permit the user to view the whole program on Screen 1, and then descend into the lower levels of interest by selecting the appropriate

ellipsis level.

4.4 The Undo Stack

An UNDO facility will prove useful to both the novice and the experienced user. The user will be able to recover from any accidental editing operation that has been performed on the file.

Related to this function, a set of keys can also be made available to store a block of program on the stack and bring it back when required. This function would be especially useful when performing tricky operations with the *syntax-directed template-based* editor.

Fig.4.1 depicts an example where this function could be used. It involves the changing of a Repeat-until construct into a While-do construct. This procedure may seem more complex than the standard text editor method, but it is the price which must be paid to preserve syntactical correctness.

```
|                    |
|        ...         |
|        ...         |
|13 I := 0           |
|14 Repeat:          |
|15   I := I + 1     |
|16   Line := Line + 1|
|17   Call FORMATTER (I, Line)|
|18 Until (I > Max Line)|
|19 Call POINTER MOVE (Line)|
```

(a)

```
|                    |
|        ...         |
|        ...         |
|13 I := 0           |
|14 Call POINTER MOVE (Line)|
|        ...         |
|        ...         |
|        ...         |
```

(b)

```
|                    |
|        ...         |
|        ...         |
|13 I := 0           |
|14 While <CONDITION> do:|
|15   <CONSTRUCT>    |
|16 End While:       |
|17 Call POINTER MOVE (Line)|
|        ...         |
|        ...         |
```

(c)

```
|                    |
|        ...         |
|        ...         |
|13 I := 0           |
|14 While (I > Max Line) do:|
|15   I := I + 1     |
|16   Line := Line + 1|
|17   Call FORMATTER (I, Line)|
|18 End While:       |
|19 Call POINTER MOVE (Line)|
```

(d)

Fig.4.1: Using the "undo" stack

In figure (a), lines 15 to 17 have been saved on the stack. As can be seen, the display is not affected. Lines 14 through 18 are now deleted using a Block Delete function. This is shown in figure (b). With Cursor 1 positioned at line 13, a While-do block is inserted as shown in figure (c). The condition is now filled in using the Line Modify function, after which, with Cursor 1 on line 14 or 15, the stack contents is retrieved to produce the result of figure (d).

4.5 Standard Text Editor Compatibility

As is, the PDL syntax editor uses a coded version of the file (extension ".COD") for its internal functions. It is not currently possible to obtain a coded version of a text file. The translating of a text file into a system-usable coded form is regarded as an extra, as the design effort in the construction of such a parsing program is considerable. The problem that arises here is one of compatibility, because if a text file is presented to the system in a faulty state, a great deal of computation is needed for each line to determine all the program faults.

If such a translator can be constructed, a program written with a standard text editor can be coded and thus prettyprinted by the system. In this way, such an externally created program can then be accessed with this PDL syntax-directed editor where all its convenient features can be used for any subsequent editing.

4.6 Language Translator Possibilities

A major spin-off from this system will come in the form of a series of translators which will convert low-level PDL into any one of a commonly used language such as BASIC, FORTRAN, Pascal or Ada. For this, the system would make use of the coded text output file. The final user will thus be able to design in PDL, and then decide on the best target language for the design. This is seen as one of the ultimate uses of this package.

APPENDIX A: A Program Description Language

CONTENTS

1 Introduction

The primary purpose of a Program Description Language (PDL) is that of a means of communication between people concerning an information processing task to be undertaken. It must not be seen as equivalent to any particular high level language whose primary function is to serve as a means of communication between people and a computing machine. Low and high level software technologies (i.e. languages) have strict syntax and are terse in construction. These requirements serve the needs of program compilers but not so effectively the needs of the program reader. Certain high level languages serve the needs of the program writer and the reader reasonably well (i.e. Pascal, Algol, Ada). Others serve the needs of the program writer but leave the reader with a hard time trying to understand the processing task (i.e. *PL1, Fortran, Apl, Forth,* Assembler). It must not be thought that it is not possible to write programs in say, Fortran, that communicate effectively to the reader, as well as to the compiler, but this requires a high order of self-discipline by the writer to restrict himself to using features of the language which emphasise design and lead to maintainable code.

The design and development of information processing tasks requires a high level of self-discipline, simply because in the nature of things it is a human characteristic to be woolly, unstructured and illogical in our thinking.

A challenge which becomes readily apparent in any processing task of note, is controlling the growth of program complexity. It is remarkably simple to design programs which are conceptually complex, time consuming to implement, and almost impossible to modify and extend. No program of usefulness to a party other than the designer should be considered as a static entity. Rather, it should be viewed as a dynamic object, having an initial simple existence, and then growing in complexity and size. The point is reached where further evolution and growth is halted because the degree of complexity becomes unmanageable and the structure too degenerate to redeem. There are effective methods for managing complexity, and these will be examined in due course.

The purpose of this document is not to introduce a formal notation. In a sense this would detract from its primary purpose. Nevertheless, there is a well defined framework of concepts which will need to be adhered to. A disciplined approach to the design and implementation of program tasks is introduced, as also are the essential tools which are required to achieve this.

The subjection of oneself to a discipline is sometimes construed as an attack on creativity. Experience, however,

has shown (not only in this field of endeavour) that the voluntary subjection to a discipline in fact leads to greater freedom in being able to concentrate on the task on hand, and not having to concern oneself afresh with major infrastructural decisions each time a program is developed.

2 The Program Body

A Program is the name given to an information processing task which executes for a limited time duration. The name Task is used to refer to an information processing task, which after initialisation, may execute indefinitely. It is a particularly useful concept in the design of information processing systems which may comprise many tasks executing *simultaneously*.

The body of a program comprises well defined sections, as set out below -

```
Program    The Name of this Program

    * data description segment *

External Procedures:
    * the procedures are listed *

Begin:
    * the algorithm segment *
End:
End   Program:
```

The major segments of the program include the data description, a listing of procedures (or subroutines) external to this program *module*, and the *body of executable* statements introduced by Begin and terminated by End.

3 The Body of a Procedure

In most respects this is similar to that listed for the body of a program or task. It differs in that parameters which are passed from the program (or another procedure) are listed.

```
Procedure    The Name of this Procedure

Inputs:   Identifier A, Identifier B etc
Outputs:  Identifier C, Identifier D etc

    * data description segment *

External Procedures:

    * external procedures listed *

Begin:
    * the algorithm segment *
End:
End   Procedure:
```

4 *Data Description Segment*

The primary purpose of this segment is to unambiguously communicate to the reader the characteristics of the data structures which have been employed. The five characteristics are -

- function
- type
- structure
- scope
- name

and are individually considered.

4.1 Data Function

The two data functions are -

a) Constants A data value which is set once and does not change is termed a Constant.

b) Variables An item of data which may be changed at will in the course of program execution.

4.2 Data Type

The type of an item of data describes its essential nature and may be one of several possibilities, including a Character, Boolean, Integer or Real. The nature of these types are further considered.

a) Character The data word is said to be able to represent all printable (and quite a few non-printable) characters i.e. from {A - Z}, {a - z}, {0 - 9}, to {!@ ...%) }. The ASCII (American Standard Code for Information Interchange) code is one of the codes widely used to represent these characters in a data word.

b) Boolean A data word with the type of Boolean (logical) can only take on the values of TRUE or FALSE.

c) Integer A data word with Integer representation implies that the range of whole numbers which may be represented is the range {- 2**n to + 2**n -1} where n represents the number of bits per memory word.

d) Real A data item with Real representation implies that a number is stored in scientific or floating point form.

Beyond these basic data types, a new type may be defined in terms of these basic types or in terms of an enumerated set of items. Aggregates of data (i.e. records and arrays) may also be defined as basic data types. A list data structure will, for example, require an array of records, where the record may be defined as a basic type as

```
List Entry = record:
                Forward Pointer : integer
                Symbol          : char
             End record:
```

4.3 Data Structure

This refers to the number of data items forming the data item. The two possibilities with which we will concern ourselves are those of -

a) Single implies that a single data item is considered.

b) Array implies an aggregate of data items of the same type is considered.

4.4 Data Scope

The scope of data refers to its degree of accessibility. In some instances it is desirable that a data item be accessible only to the immediate environment of that program or procedure. On other occasions we require that a data item be accessible to a number of (or all) procedures. Four degrees of accessibility are defined -

a) Local implies that the data item is accessible within the present procedure only. When that procedure is not executing then that data item does not exist.

b) External When a data item is declared as external then the implication is that it has been defined in a higher level procedure. In this instance the data item must appear in the parameter list (input and/or output) of the called procedure.

c) Global As its name implies, a data item which is declared as global in scope is available to all procedures for manipulation. This breadth of accessiblity is necessary in many instances, but can result in intractable problems e.g. in trying to establish which procedure may be responsible for corrupting a data value. To limit the breadth of this accessibility it is useful to place such data items in partitions.

d) Permanent The concept of a permanent identifier is particularly useful in information processing systems which comprise multiple tasks. It is used in the sense that once created it exists for the duration of the system. A permanent identifier may be local or global in scope.

4.5 Data Name

The name of a data item is most frequently referred to as an identifier. The identifier may be as long as necessary to clearly convey its purpose. Meaningless abbreviations do not serve the reader and hinder comprehension.

5 Data Segment Construction

The five characteristics referred to in the Data Description are linked together in the following order of precedence -

{function} {type} {structure} {scope} {identifier(s)}

in a structured fashion beginning with the function. For the sake of clarity each of the characteristics is placed on separate lines with successive indentation. The resulting graphical presentation of the data description enhances the reader's comprehension. The following example will help to clarify these concepts.

```
Types:

    List Entry = Record:
                     Forward Pointer : integer
                     Symbol : Char
                 End Record:


Constants:
    Char:
        Array:
            Local:
                Doublespace := '  '
    Integer:
        Single:
            External:
                Maximum Items
                Maximum record length
Variables:
    Char:
        Array:
            External:
                Text record = of size(Maximum
                                      record length)
    Boolean:
        Single:
            Local:
                Condition
    Integer:
        Single:
            External:
                This record length
    List Entry:
        Array:
            Local:
                List = of size (Maximum Items)
```

6 Algorithm Segment

The constructs required to implement any processing task include those of sequence, selection, and iteration. While it is true that these are sufficient, used in isolation the problem of complexity will rapidly become evident. The major tool for controlling and reducing complexity is the procedure facility.

In addition to those *already mentioned, facilities* are required for the computing machine to communicate with its enviroment.

6.1 Assignment

This construct will always feature the process of assignment i.e.

identifier := function of other identifiers

The expression -

A := B + C

is a simple example of assignment

6.2 Selection

The simplest example of selection is the If-Then-Else construct with its two possibilities of choice. More generally useful is the Case construct, which allows a choice from amongst any number of constructs.

6.2.1 If-Then-Else

A test is made on a condition which must be a boolean variable.

```
If (Condition is true)
    Then:
        Sequence A
    Else:
        Sequence B
End if:
```

For the true condition Sequence A is executed, after which execution continues after the End if: termination of the construct. For the not true condition Sequence B is executed, whereafter execution continues after the end of the construct.

In many instances the else condition is not required. The following simpler form should then be used.

```
If (Condition is true)
    Then:
        Sequence A
End if:
```

6.2.2 Case

In contrast to a condition which was tested, the Case construct allows the test item to be any one of a discrete nature i.e. character, boolean or integer. The form of this construct is -

```
Case Option of:
     Option 1:  Sequence A
     Option 2:  Sequence B
     . . . .
     Option M:  Sequence M
     Else:      Sequence N
End case:
```

If the variable Option does not match one of the stated options (Options 1 to M) then control passes to the Else: option and Sequence N is executed. In each instance, once the appropriate Sequence has been executed, control passes to the next statement following the End Case: statement.

If the Else condition is not required, then it should not be included in the construct.

6.3 Iteration

Two forms of iteration are considered. The Repeat-Until and the While - Do constructs.

6.3.1 Repeat - Until

The primary feature of this construct is that the executable steps between Repeat and Until are executed at least once.

```
Condition := false
Repeat:
    Step A
    Step B
    . . . .
    Step to modify test condition
Until: (Condition is true)
```

6.3.2 While - Do

In contrast to the Repeat - Until, the executable steps enclosed by this construct might not be executed at all, depending on the initial value of the test condition. In the example shown the statements will be executed at least once, since the initial value of Condition is false.

```
Condition := false
While (condition is false)
    do:
        Step A
        Step B
        . . . .
        Step to change the state of Condition
End while:
```

6.4 Procedure

A procedure is invoked by the statement -

Call (procedurename)(input parameters) output parameters

The input parameters list is enclosed between ( ) to unambiguously indicate the direction of the passed parameter. If there are no input parameters then the ( ) simply appear after the procedure name. A parameter that must be passed in and out of a procedure will appear in both the input and output lists.

Perhaps the most prevalent view of the use of the procedure construct is that of simply replacing multiple instances of a code sequence with a single statement which results in that replaced code sequence being executed. While this concept is certainly valid, it takes no account of the power of the procedure concept from a coherent system building viewpoint.

The most powerful conceptual use of the procedure which has emerged from recent research in software engineering is that of 'information hiding'. The concept here is that of surrounding a data structure with all the necessary functions (i.e. procedures) to access and modify that structure. Take, for example, the data structure of a stack. The relevant operators to provide all required functions on the stack are:

- Initialise stack ( ) stack status
- Push onto stack (data value) stack status
- Pop off the stack ( ) data value, stack status
- Top of stack ( ) data value, stack status

It is now quite irrelevant to the user of these operators exactly how the stack is implemented. We have 'hidden' the stack behind four operators, and at the same time provided for the benefit of the user four 'abstractions' which encompass all meaningful operations on the stack.

This concept of 'information hiding' is considered valuable for the following reasons -

- Managing system complexity

  A set of operators which embody all required functions to be performed on a data structure enables the designer to build large and complex systems which are understandable. Unless determined steps are taken to control complexity, the designer will lose appreciation of how the parts affect the whole.

- Assists system evolution

  By surrounding all data structures with a relevant set of operators, the system may be allowed to evolve in a systematic fashion.

- Aids in the location of errors

  By restricting access of each data structure to a limited set of operators, the location of a source of data corruption is immediately traceable to that set of operators responsible for manipulating that data aggregate.

- It provides reusable modules

It has been suggested that the greatest enhancement in software productivity has come from the concept of 'modularisation' of software. Well-known examples of this are the libraries of mathematical subroutines provided in common high level languages. The concept here is that of accumulating, with time, modules comprising data structures and their associated operators.

As in most instances where a course of action has substantial benefits, there are attendant costs. The principal cost in this case is the additional effort that must be devoted to developing the required set of operators. A further penalty is the possible loss in program execution efficiency and an increase in memory requirements.

6.5 Machine - Environment Interface

Experience shows that there is more disparity in the approach to handling data input and output than in any other area of language construction. In this PDL data input and output are handled through the constructs of GET and PUT, respectively. Unless the origin and destination of the data are specified, it will be assumed that the Keyboard and the Console are the input/output devices, respectively.

6.5.1 Data Input

The general form of this construct is -

```
Get:
   Source device: identifier(s)
End get:
```

For example,

```
Get:
   Diskfile: Text record
End get:
```

6.5.2 Data Output

The general form of this construct is -

```
Put:
    Output device: * identifier(s) and/or
                                text strings *
End Put:
```

For example -

```
Put:
    "Text messages between quotes"
    Newline
    "On this line";
    "The semi-colon implies stay on the same line"
    Identifier 1
    "And another message"
    Newline
End put:
```

The above example shows the various elements of a Put statement in a language independent fashion. Printable text is enclosed between quotes. The Newline implies skip to the next line. The semicolon is used to prevent skipping to the next line on completion of the last task, whether it be printing a message or the value of an indentifier.

6.5.3 Abbreviated Forms of Get and Put

Since the process of moving an item of data to or from a device is conceptually equivalent to the process of assignment, the following form of device/identifier statement is useful.

```
Device:     := identifier    or
identifier := device:
```

A device is distinguished from an identifier by the colon following the device name.

7 Array References

A simple array will require a single pointer to reference a given location.

```
Array Name:Pointer := data item
```

The colon indicates that Array Name is referenced by Pointer. A field of a record in an array of records is referenced as follows.

```
Array Name.Field Name:Pointer := data item
```

The name following the dot after the array name is the referenced field.

8 Arithmetic and Logic Operators

The following symbols are used for common arithmetic and logical operations

Operation	Symbol(s)
plus	+
minus	-
multiply	*
divide	/
Raise to power n	^ n
Modulo n operation	MOD N
And	AND
Or	OR
Exclusive Or	XOR
Complement	NOT

9 Comments

Comments may be freely interspersed in the text, and are indicated as such by placing the comment text between

asterisks. For example -

```
* this is an example of a comment *
```

10 Sequential and Parallel Construct Execution

The need frequently arises to distinguish between constructs intended for execution in serial or parallel fashion. In some instances the need arises to allow for both forms of construct execution in the context of a single task.

The PDL discipline already introduced assumes that an information processing task can be represented by a series of constructs, as follows.

```
Begin:
    Construct A
    Construct B
    Construct C
    - - - - - -
    Construct N
End:
```

The implicit assumption here is that executions begins with the construct following the keyword 'Begin:'. The execution of Construct A then proceeds to completion before continuing to execute Construct B. Execution will ultimately be terminated at the keyword 'End:'. Serial execution of the constructs is implied.

When it is desired that the constructs execute in parallel, the description may be viewed as

```
Begin:
    Construct A; Construct B; .... ; Construct N
End:
```

where the semi-colon implies simultaneous construct execution.

A significant textual problem arises in adopting such a documentation convention in view of the obvious constraint on page width. A useful expedient here is to retreat to the alternative serial listing of constructs, with the condition that alternative keywords be used to reflect the parallel construct execution, as follows.

```
Cobegin:
    Construct A
    Construct B
    Construct C
    - - - - - -
    Construct N
Coend:
```

All constructs between Cobegin: and Coend: are considered to execute in parallel or concurrently.

When the occasion arises to distinguish between those constructs executing in parallel and those executing in

serial fashion, the successive application of the relevant keywords surrounding the constructs of interest will serve to indicate the correct sense.

```
Begin:
    Construct A
    Construct B
    Construct C
    Cobegin:
        Construct D
        Construct E
        Construct F
    Coend:
    Construct G
    Construct H
End:
```

Constructs A to C first execute serially. Constructs D to F then execute in parallel. Only when execution of this block is completed is serial execution of the remaining statements undertaken.

11 Data Flow Nomenclature

Data Flow analysis partitions a system into processes and resources. It is helpful to define the specific format of program modules that use this form of system construction.

11.1 Process Description

A process as defined as an area of activity endlessly consuming and processing instructions or data. This behaviour is described as

```
Process    {the name of this process}

{local data description}

Begin:
  Repeat: forever
    Begin:
       * fetch an instruction (or data item) *
       * process the instruction (or data item) *
    End:
End:
End Process:
```

11.2 Resource Description

A resource comprises a data aggregate and a series of operations to be performed on that aggregate. The description of the data aggregate is undertaken separately from a description of the operations.

11.2.1 Resource Data Aggregate Description

The form of this description is as follows.

```
Resource      {name of this resource}

Permanent data Description

{all aspects of the common data aggregate are described }

End Resource:
```

11.2.2 Resource Operations Description

The form of description used for each operation on the common data elements of the resource is described as

```
Operation     {Resource Name}{Name of this operation}

Inputs:       {list as needed}
Outputs:      {list as needed}

Begin:
     * the algorithmic description for this
       operation *
End:
End Operation:
```

12 An Example of a PDL Application

An example is given below of how a program using PDL notation is presented. A vitally important aspect is the indentation of the constructs as a graphical aid towards assisting reader comprehension.

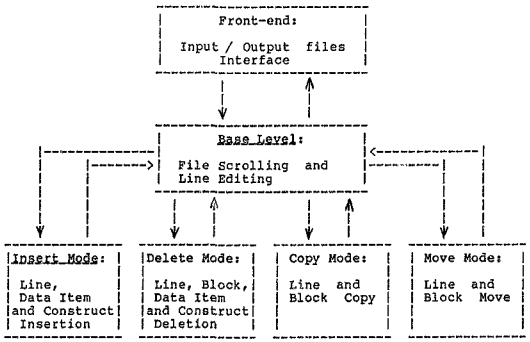
```
Procedure      Move Cursor

 ***********************************************
 *   A Routine to move the cursor by an        *
 *   externally defined number of moves and    *
 *   in a given direction.                     *
 ***********************************************

Inputs:          Number of moves, Cursorfunction
Outputs:         * None *

Constants:
     Char:
          Array:
               Permanent:
                    Vducodes
     Integer:
          Single:
               Permanent:
                    Console
                    Keyboard
                    Diskfile
```

```
Variables:
    Char:
        Single:
            Local:
                Cursorcode
    Integer:
        Single:
            External:
                Cursorfunction
                Numberofmoves
            Local:
                Movecounter
                Consolestatus

External Procedures:

                Putchar

Begin:
    If (Numberofmoves  exceeds zero)
        Then:
            Movecounter := Numberofmoves
            Cursorcode  := Vducodes:Cursorfunction

            Repeat:
                Put:
                    Console: Cursorcode
                End Put:
                Movecounter := Movecounter - 1
            Until: (Required number of moves are made)
    End If:
End:
End Procedure:
```

- - - O O O - - -

APPENDIX B: SUMMARY OF SYSTEM LEVELS

From the System Level Hierarchy figure on the next page, the system layout is presented in block format. The remainder of this appendix details the tree structure diagams of the syntax-directed editor package. The Front-end is regarded as a separate system block, but Base Level is considered as the gateway into the subsequent editing modes. It is with this in mind that the following figures have been organized. A list of the contents of this appendix is given below:



System Level Hierarchy

A simplistic representation is given of the main system levels together with their associated functions. The interconnections indicate the possible movements between levels open to the user. The highlighted, underscored blocks indicate existing system features.

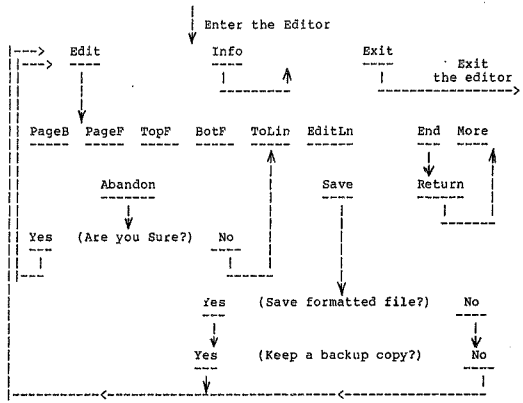


Fig.1: The Front-end tree structure

Bracketed questions show the display which appears in the Prompt Screen, and to which the user must respond.

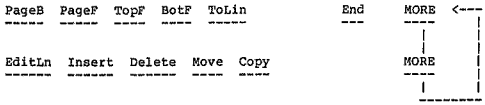


Fig.2: The Base Level tree structure

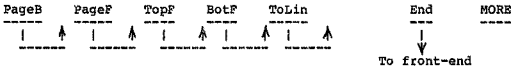


Fig.2.1: The scrolling functions' tree structure

Fig.2.2: The tree structure for Edit Line mode

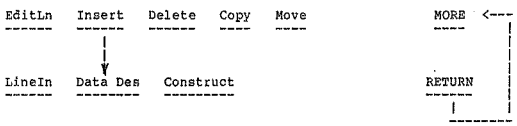


Fig.2.3: The Insert mode tree structure

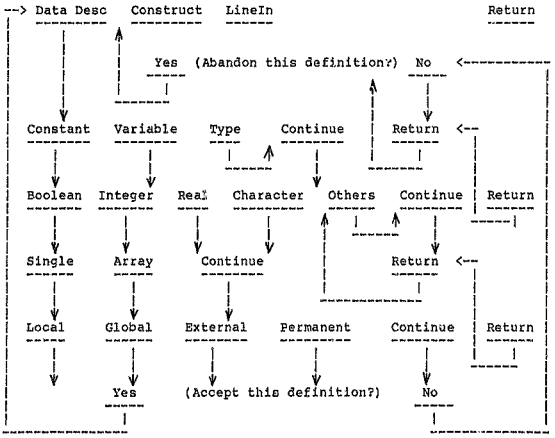


Fig.2.3.1: The tree structure of the Data Description Insert mode

Although all prompts and conditional branches have not been labelled for the sake of clarity, bracketed questions indicate prompts requiring user response.

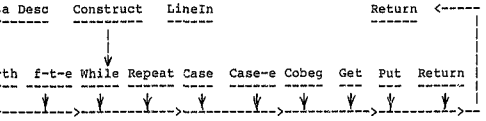


Fig.2.3.2: The tree structure for Construct Insert mode

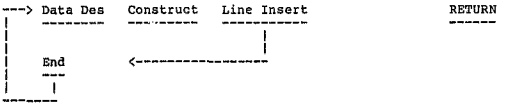


Fig.2.3.3: The tree structure for Line Insert mode

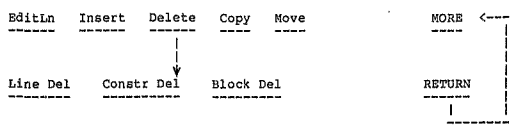


Fig.2.4: The Delete mode tree structure

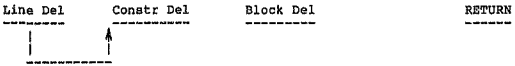


Fig.2.4.1: The tree structure for Single Line Delete mode

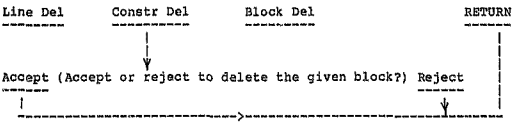


Fig.2.4.2: The tree structure for Construct Delete mode

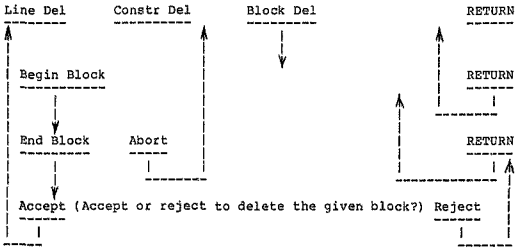


Fig.2.4.3: The tree structure for Block Delete mode

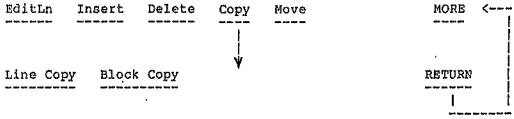


Fig.2.5: The Copy mode tree structure

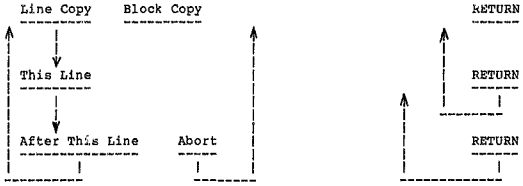


Fig.2.5.1: The tree structure for Line Copy mode

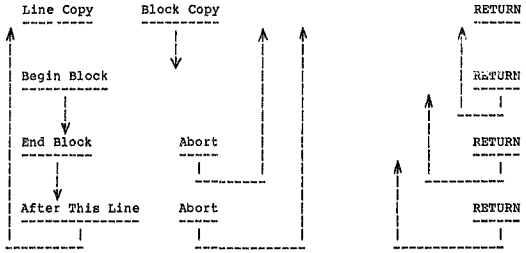


Fig.2.5.2: The tree structure for Block Copy mode

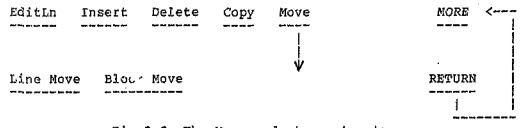


Fig.2.6: The Move mode tree structure

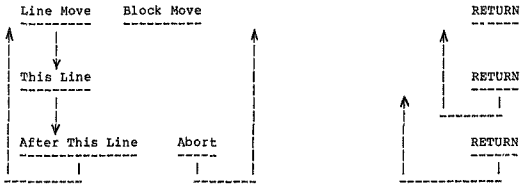


Fig.2.6.1: The tree structure for Line Move mode

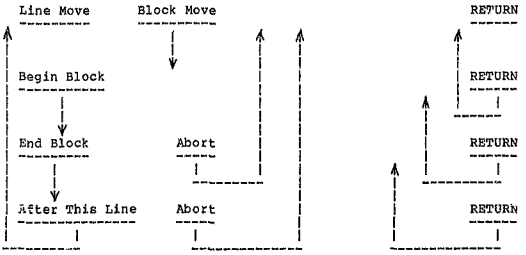


Fig.2.6.2: The tree structure for Block Move mode

APPENDIX C: GLOSSARY OF TERMS USED

construct -- a program block specific to the programming language adopted (usually used to indicate decision or looping).

data flow concepts -- the design principles developed by Dr.A.J.Walker using processes and resources as the key elements.

ellipsis -- (also known as holophrasting) is the replacement of a program block by an ellipsis symbol so as to be able to see the major top levels of a program with minimal scrolling effort.

font -- a terminal-specific video attribute (eg. blinking, highlighting, etc.).

function key -- a physical key on the keyboard which is programmed to execute a certain function.

key word -- a system-generated, language dependent reserved word.

mode -- a system level in which the user can operate.

operator -- a routine used by a process to access a resource. Each operator has a specific function on the resource.

parser -- a process which analyzes an input line to determine if any semantic errors are present.

PDL -- Program Description Language: a terminal-independant high-level structured design language used to describe both hardware and software related designs.

placeholder -- a non-terminal in a program construct template which must be expanded (usually into a condition or statement) by the user.

prettyprinting -- the formatting of a program so as to emphasize its logical structure; this is usually performed by indenting each program line appropriately.

process -- an element used in design description to indicate an activity. Processes are used for the management of resources.

reserved words -- language specific words which cannot be used in a program other than for their predefined purpose.

resource -- a passive element used in design description to represent a data structure. Operators represent the

user's only access to a resource.

semantic errors -- long distance, compiler detectable errors created by the user disobeying language rules (eg. undefined data item; type incompatibility; etc.).

state -- a stopping point in the program where some action must be taken for a transition to another state to occur.

syntax errors -- short-distance errors created due to incomplete program constructs.

template -- the skeleton structure of an entire construct block containing key words and placeholders.

tree structure -- a method used in this manual to illustrate system levels and their transitions (a system level is recognized by a specific set of ten function keys).

VDU -- Visual Display Unit: (also known as a display terminal or Cathode Ray Tube) is a screen used for visual output.

A FUNCTION-KEY DRIVEN SYNTAX-DIRECTED EDITOR FOR SOFTWARE SYSTEMS DESIGN

D E S I G N E R ' S R E F E R E N C E

(Version 1.0)

December 1985

Author: A.P.Bassanino

Signed: 

A Project Report submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg in partial fulfillment of the requirements for the degree of Master of Science in Engineering.

CONTENTS

LIST OF FIGURES

1 INTRODUCTION

Throughout this guide, constant reference will be made to such terms as PDL, Processes, Resources, and States. It is the intention of this introductory chapter to familiarize the user with the above terms in the context of complex software systems design. The concepts expressed here have been devised by Dr. A.J.Walker of the Department of Electrical Engineering at this university, and for further information on any of the topics mentioned here, the relevant departmental documents should be consulted. (Walker (1984), Bassanino (1985b))

As the package described here is a syntax-directed PDL editor, the reader should be familiar with the concept of a Program Description Language (PDL). (Caine (1975)) This high-level program-like language is used throughout this manual to describe any pertinent algorithm features of the system. The reader is referred to Appendix A of the User's Manual for the formal PDL specification. (Bassanino (1985b))

In information processing system analysis, the data flow concept is a useful one. For this, two elements, processes and resources, are necessary. A process is essentially an active element which is responsible for the management of resources. A resource, on the other hand, is a passive entity responsible for the management of data.

Resources are characterized by their detail management of data. These resources are accessed (by processes) via operations performed on them. The operations will vary depending on the required logical behavior of the resource. This can be compared to an abstract data type, (Shankar (1984)) where the user knows only of the operations and not of the internal implementation details. Examples of physical resources are printers, VDUs, and disk drives, while software or memory-based resources include stacks, queues, and linked lists.

The resource concept exploits the principle of "information hiding", in which the user is unaware of the data structure behind the operations presented to him by the resource. This concept forms the basis for a software component library. Thus, resources can be compiled and tested separately to the system for which they will be used. (Miller (1984)) All that is required is to test that the operations performed on the resource will function correctly under all circumstances. This thorough testing procedure will ensure that the resources are 100% functional before they are included for operation with the process. This system of design lends itself to modular programming techniques.

Processes are machines used to transfer data between resources. These are the entities which access the resources via their operations in order to produce a working system. An example of a process-resource diagram is given in Fig.1.1. Here, the Printer process accesses the Buffer Queue resource, and transfers data to the Printer resource at an acceptable rate.

Fig.1.1: A Process-resource Example
The process is represented by the rectangular box, while the resources are shown as the oval shapes. The process accesses the resources via operators which are written on the connecting lines (eg. Read). The arrows indicate the direction of data flow which is always from left to right.

Processes can be designed using a state approach. The activities of a process can be categorized into a number of states. Each state is a "stopping point" in an activity. From this state, depending on the outcome of some particular test condition, a transition will occur into another state. An example of a graphical representation of a state diagram is shown in Fig.1.2. The idea of a state concept to represent process behavior is a key issue in this design.

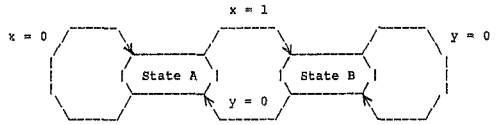


Fig.1.2: Example of a State Diagram
States are written within the state bubble. The transitions together with their associated conditions are drawn as connecting lines.

The above methods, coupled ... process decomposition, have proved an effective approac' ...e system design. Among the merits of the process-resc ...dology, we can recognize the ease of comprehension o1 ...hod because of its logical nature; its simple graphica... iption of system behavior; its power in terms of defining reusable "components"; and its independence of technology for its implementation.

In the first level design, a single process is defined which represents the entire system's capabilities. The resources used by this only process, are those that represent inputs or outputs to the environment external to the process.

The second level of design would see the single system process decomposed into its major constituing activities. Associated with these new, smaller processes, the necessary internally accessed resources will emerge. New resources may be required due to the internal coupling necessary between sub-processes. Similarly, the process decomposition described above can continue to a stage where further fragmentation is either impossible, illogical, or disadvantageous.

The method described above is iterative, in that subsequent levels of design examine each of the constituent activities of the first level process in terms of their behavior and demand for local resources. This clearly reduces the problem of complexity, as design boundaries are defined for each level; only the specific operations required of a resource being investigated for the building of that resource.

This iterative methodology also leads to effective documentation. At each level of design, each process's behavior may be described in terms of its usage of resources. Each resource, in turn, may be described in terms of its required behavior and the operations to be performed on it. It is this approach which has been used in the subsequent chapters of this manual.

2 THE SYSTEM DESIGN

2.1 Top Level Design

From a superficial level, the system can be seen as in Fig.2.1. The user is aware only of the keyboard and diskfile as input devices or resources, while the outputs are routed either to the terminal display for visual inspection, or to the diskfile for backup documentation purposes.

Fig.2.1: Top Level Design using processes and resources

The resources can clearly be subdivided. The keyboard will contain the usual alphanumeric keys as well as the IO function keys which are dynamically defined by the system. Diskfile is used both as an input and as an output resource. On loading the editor, diskfile is accessed for the system tables and the coded PDL file. After an editing session, the new file is rewritten into diskfile for future reference. The VDU display is divided into four logical screens which will aid the user in operating the system effectively.

```
Process  Syntax-directed PDL Editor
Begin:
  *Read the necessary system files to load the editor*
  Repeat:
    *Read the required coded file which corresponds to the file
     which is to be edited*
    Repeat:
      *Allow the user to modify the file using the keyboard as
       the input device, and the VDU screen as the output device*
    Until (The user wishes to terminate the editing of this file)
    *Write the edited coded file back onto disk*
  Until (The user wishes to exit the editor)
End:
End Process:
```

Fig.2.2: A PDL description of the Top Level Design

Fig.2.2 gives a very general description of the PDL syntax-directed editor package. The system initially loads the required files from disk. The user is then allowed to edit a PDL file according to some complex algorithm; *all logical representations* of the file being given on the VDU screen. When the user is through with editing this file, it will be stored back onto disk.

2.2 System Principles

The second and subsequent levels of design, as explained in Chapter 1, involve the decomposition of the single system process into its constituent parts. *Before delving into this problem,* it will be constructive to take a look at the principles involved behind the system construction. In this section some past work in this area is briefly mentioned; the resources and basic system structure for efficient system operation also being discussed.

Work was first started on the syntax-directed PDL editor package in 1984 (Bassanino (1984), Master (1984)). The groundwork was laid for future development and expansion, even though by the end of that year only a demonstration system was available. In 1985 this project was continued to provide facilities for program *storage. Many of the resources* and concepts of the previous year were incorporated into this design, and similarly, many new features and design changes were also necessary.

The system was designed as a template-based editor. The choice of a generator approach as opposed to a recognizer approach was mainly because of user convenience in program entry. *Using this* method, syntactical correctness is maintained at all times by preventing any syntactically incorrect operations. This programming by selection allows the user to choose a construct for insertion. A construct is inserted with a placeholder indicating any construct section which is to be user-entered. (eg. the <CONDITION> in an If-then-else construct)

The generator approach has been used successfully as the basis for syntax-directed editors such as the Cornell Program Synthesizer. The literature survey of Bassanino (1985a) gives a detailed account of such projects. The *recognizer approach* is very similar to a conventional editor, and syntax errors are allowed. It is believed that this approach, in giving the user almost unbounded freedom, does not satisfactorily teach or aid the user when writing or editing a syntactically correct PDL program.

Programming by selection is suitable for program entry and gives the designer the power to enter a PDL program in a series of stepwise refinement iterations. This is one of the main objectives of the package designed here, and with some adjustment, the user will soon become accustomed to the different *method of operation of a syntax-directed* editor as opposed to a conventional text editor.

There must of necessity be a difference between the operation of these two types of editor because of the increased power of the former. Limitations on illegal operations are automatically performed so that editing (as opposed to entry of) a PDL program is also a simple task. These limitations, due to the necessity to

maintain syntactical correctness, is a price which must be paid if the program is to emerge correct. The age-old problem of converting an If-then-else construct into a While-do construct is easily resolved by providing a temporary storage stack. (See section 7.7)

From reading the User's Manual (Bassanino (1985b)), it will be clear that a variety of visual cues are used on the four logical screens to differentiate between errors, system-generated key words, placeholders and function key definitions. The management of the logical screens (Main Screen, Window Screen, Prompt Screen and Key Definition Screen) requires a considerable amount of effort if only primitive functions are to be used.

It is advantageous to group these screen-based functions into a single module for convenience. This module, the Terminal Resource, will have a set of high-level operations which will make the system, as seen from outside the module, terminal independent. Thus, the Terminal Resource will be responsible for terminal or VDU based operations. All terminal dependence is dealt with from within this module. The designer will have access to this resource only via the specialized operations available for each logical screen. This resource is thus used by processes which require any form of screen management. The Terminal Resource is explained further in section 3.1.

The PDL generator system has a set of standard PDL "reserved words" (such as the "If" or "End if:" of an If-then-else construct). Each of these key words can be associated with an indentation code. This code is used by the system for positioning the key words in a formatted form on the Main Screen. The Key Code Table is a resource containing any such key words together with all the available placeholders. Each of these words is associated with a key code for easy identification, and an indentation code. This table allows the system to combine the key words in almost any desired order, and thus contributes to the editor's flexibility. Further details of this resource can be found in section 3.3.

The Prompt Table is an obvious resource which emerges with the need for a prompt screen. The messages displayed on this screen are used to guide the user. As a full 80 character line of text must be accounted for, any repeated prompts which are hard programmed into the system will result in a large wastage of memory space. The Prompt Table prevents this wastage by assigning a unique integer code to each prompt line. Besides the obvious memory space saving, all prompts will now be conveniently situated in one location and can now become totally file based. This too aids in making the system flexible and adaptive to any possible changes in prompt message wording. Details regarding the Prompt Table can be found in section 3.4.

Again, to make the system's operation as flexible as possible, instead of hard coding the logic into the system routines, a table driven approach was opted for. Using the concept of states, the Definition Table was devised. This table corresponds closely in operation to the tree structure diagrams adopted in the User's Manual to describe system operation. Each state is associated with a specific function key definition line. The depression of any valid key will possibly lead to a new state, with a prompt being displayed in the Prompt Screen. The Definition Table

accounts for this by providing fields for a key code, next state and prompt code for each key in the key definition line for that state. This means that the calling program will only specify a starting state; all subsequent states being read from the *Definition Table*. Thus, a great deal of system behavior can be modified by changing the Definition Table data. This resource is described fully in section 3.2.

The system uses a line-by-line editing concept as described below. A line is chosen for editing from the file, and is edited separately in the Window Screen. All line editing will be performed using the Line Editor. When the line has been edited, it can be inserted back into the file to replace the old line at Cursor 1. Line operations, such as line delete or copy, are also available. Thus, editing of the file is done using two editors: one for horizontal, character editing; and the other for vertical, line editing. All character editing is done on a line-by-line basis in the Window Screen using Cursor 2, while file editing (regarding a line of text as a single element) is performed using Cursor 1 on the Main Screen.

The obvious choice of resource when dealing with extensive manipulation of textual data is the linked list. This resource allows the user insertion, deletion, and pointer moving facilities which form the basis of any editor. Due to the specialized nature of the PDL generator system's editor, two linked lists are needed. The Line Linked List is used in conjunction with the horizontal Line Editor, and is responsible for character (or microscopic) manipulation within a text string. The File Linked List instead, is used for the implementation of the vertical file editor which deals with the macroscopic manipulation of lines of text. A more detailed description of *these two resources is given in sections* 3.5 *and* 3.6.

The coded form of a file produced by this syntax-directed PDL editor is labelled with an extension of ".COD". This file contains the edited PDL description in coded form, and is structured as follows: each line of PDL program corresponds to two lines in the coded file. The file is coded line by line as it is generated, and stored in the File Linked List. Each line can be coded by means of three integer codes and one text field. The text field would contain any segment of user-entered code which is editable. Placeholders would thus also fall into this category.

The three integer fields contain the indentation code and two key codes of the line in question. The indentation code is an integer which corresponds to the level of indentation of that particular line. This indentation code, multiplied by a certain constant (usually 2) will give the absolute indentation of the line from the left hand margin.

There are two key codes for the mere reason that there exist cases where more than one key word can be present on one line (eg. While <CONDITION> do:). If two key words are present on the same line, then it is assumed that the text line is placed between the two key words. Similarly, if there is only a single key word and a text line associated with a PDL line of program, then it is assumed that the line is in the form of the key code first, followed by the text line. A line containing a key word *only will have a null text line associated with it. These*

conditions hold true for PDL. In Appendix D, the coded and formatted files are compared and explained.

It is with the above knowledge of the basic system operation that the process decomposition can be performed meaningfully.

2.3 Process Decomposition

A second level design would essentially recognize the distinct processes involved in the system. These can be seen to be: the File Editor; the Line Editor; and the Formatter. At this stage only these three main processes will be considered. It will become evident that they can be further decomposed.

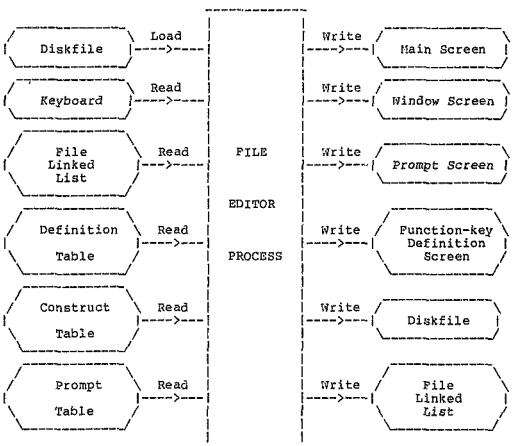


Fig.2.3: Process-resource diagram for the File Editor Process

The File Editor process will essentially be responsible for the management of the Definition Table and thus a large portion of the total system operation. This process includes all the functions of Base Level. Thus, Main Screen scrolling; Insert mode; and Delete mode are all controlled by this process. It will ensure that the user is moved from state to state. It is for this reason that the File Editor process is the only process to access the Function Key Definition Screen. Fig.2.3 shows the process-resource diagram of the File Editor process.

Due to the variety of functions performed by the File Editor process, it is clear that most of the resources will be employed by it. With reference to Fig.2.3, the input and output resources are explained as follows. On initialization, the coded file is read from diskfile and written into the File Linked List. The system will accept user-entered input in the form of function key depressions or a requested input value; thus the Keyboard Resource.

For system value input purposes, the Window Screen is sometimes used to display a permanent prompt. In Data Description Insert mode, the Window Screen is used extensively for data item editing purposes. The Main Screen is used to display the PDL program, and as such will form a resource accessed by the File Editor process whenever a screen scroll or Cursor 1 movement is required.

The Definition and Prompt Tables are a necessity in this process so as to allow a user to be prompted as a state is changed. The corresponding output resources are the Prompt Screen for displaying error or prompt messages, and the Function Key Definition Screen used to display the current system level in terms of the ten function keys. The Construct Table is used for Insert mode, and contains all the templates available. The File Linked List is accessed both for reading and writing purposes when the file needs to be modified after a block edit operation. Diskfile is written to when the user is through with editing a PDL file.

The Line Editor is an important part of the system design as it stands alone as an autonomous process and is used whenever a user-entered alphanumeric input is required. The process deals with the manipulation of characters in a text line. A few important functions such as delete, insert, etc. are provided. Fig.2.5 shows a PDL high-level description of the editor's behavior, while Fig.2.4 is a pictorial representation of the process with its associated resources.

Inputs are taken from the keyboard. If a PDL text line is to be modified, it is loaded into the Line Linked List from the File Linked List. This line can then be modified using the various Line Edit functions. For modification purposes, constant access is made to the Line Linked List. The Window Screen is used as the visual display screen for the Line Editor. Any errors or warnings that occur regarding line editing will be displayed in the Prompt Screen. On exiting the Line Editor process, the contents of the Line Linked List is rewritten in logical order in the File Linked List via a Dump operation.

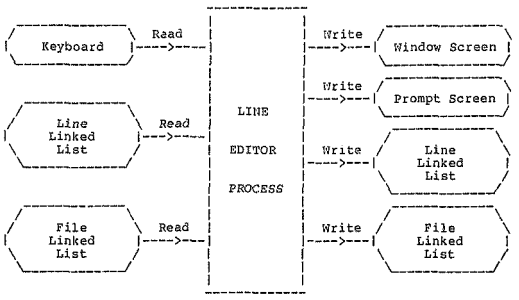


Fig.2.4: Process-resource diagram for the Line Editor Process

```
Process Line Editor
Begin:
  If (Text line is to be modified)
    then:
      *Load the Line Linked List with the line in the File Linked
       List*
    else:
      *Initialize the Line Linked List*
  End If:
  Repeat:
    *Get an input character from the keyboard*
    *Process  the input according to whether it is a  command  or
     data*
    If (Input implies an error)
      then:
        *Write out the error in the Prompt Screen*
      else:
        *Perform the required operation in the Window Screen*
    End If:
  Until (Line Editor must be exit)
  If (Text line is to be written in file)
    then:
      *Dump the line in the Line Linked List into the File Linked
       List*
  End If:
End:
End Process:
```

Fig.2.5: A PDL description of the Line Editor Process

The Formatter is a vital system component as it is responsible for the formatting or prettyprinting of the coded file. Here, a line is read from the File Linked List. This line is in coded form. The Key Code Table is used to convert the key codes into key words. Indentation is calculated, and the prettyprinted line is written out to either diskfile for documentation purposes, or (more frequently) to the Main Screen to give the user a visual representation of the true formatted file. Fig.2.6 shows the process-resource diagram for the Formatter process, while Fig.2.7 gives a broad outline of the process behavior.

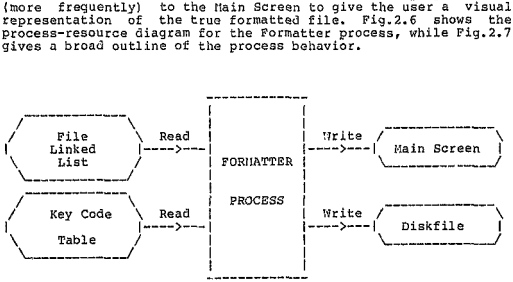


Fig.2.6: Process-resource diagram for the Formatter Process

```
Process Formatter
Begin:
  *Read the coded line from the File Linked List*
  *Look up the key codes in the Key Code Table*
  *Calculate the line's indentation*
  If (output is directed to diskfile)
    then:
      *Write the formatted line to diskfile*
    else:
      *Write the formatted line to the Main Screen*
  End If:
End:
End Process:
```

Fig.2.7: A PDL description of the Formatter process

Clearly, the File Editor process is easily decomposed into sub-processes (ie. third level design). Each of the Base Level; Insert mode; Delete mode; etc. are individual processes, and each of these in turn will have its own set of sub-processes. For purposes of compactness and convenience, the third level process-resource diagrams are not included in this document. As lower levels of design are reached, more detail is involved. This type of detailed decomposition for the File Editor process can be found in sections 4.3 and 4.4.

The Line Editor process too can be further subdivided into its constituent functions. This detailed level can also be seen in section 4.1. The Formatter process, however, is almost at minimal level as depicted above. Section 4.2 elaborates more explicitly the functions which are needed for prettyprinting via this process.

The Terminal Resource described earlier in this section is used whenever any form of output is required to the display terminal. Thus, all three major processes will require access to this resource. In the process-resource diagrams, however, it has been omitted from the left hand side of the processes for the sake of clarity.

3 THE RESOURCES

In this chapter, the software resources described briefly in *sections 2.2 and 2.3 are discussed in detail.* Each resource is explained in terms of its function, operators and structure. The Resource Function clarifies the use of the resource in the PDL generator program, also listing its benefits. The Resource Operators section in this chapter is a list of each routine together with its relevant parameters which can be called to operate on the resource. The routine calls are given in *Pascal,* with all the parameters explained. The Resource Structure section details the physical arrays necessary to maintain the resource. The file-based resources have their tables detailed in Appendix C.

3.1 Terminal Resource

3.1.1 Resource function

This resource is designed to deal with every kind of output which is presented to the user by the system on the VDU terminal. Thus, to write any output to the VDU, the system will use the operators provided by the Terminal Resource. Typical examples of the functions provided include the choosing of any of the terminal's video fonts (such as reverse video, underscore, blink or highlight) and the positioning of the cursor at any particular point on the screen.

Screen management is also dealt with by the Terminal Resource. *The syntax-directed PDL generator makes extensive use of this* feature for its four special-purpose logical screens. To make the Main Screen, Window Screen, Prompt Screen and Key Definition Screen completely terminal independent, the necessary operations have been devised in this resource.

Thus, without the need for the user to get involved with terminal dependent ASCII or octal codes, which are specific only to a particular system, the writing of a message to the Prompt Screen, for example, can be simply achieved by using the operator PS_WRITE ('Message') from the Terminal Resource. The operator will essentially deal with any cursor positioning, font selection or line clearing functions required. The user need only make use of the high-level routine calls available from the Terminal Resource as operators.

The concept of a terminal resource is also attractive from a software portability viewpoint. All *terminal specific functions* are available in one resource. Thus, if the package is to be made available under a new operating system which adopts a different screen management approach, the designer need only modify the Terminal Resource, resting assured that the rest of the package is completely terminal independent.

3.1.2 The operators

The operators are divided into groups to differentiate between their function. The routine names, together with their input and output parameters are listed below in their Pascal format. A description of the general behavior of each operator is also given. (Walker (1985))

Keyboard accessing operations

KBD_GET (VAR SYMBOL: CHAR)

Function: To obtain a character symbol from the keyboard without using the ENTER key and without echo to the screen.
Inputs: *none*
Outputs: SYMBOL -- The character which has been read.

Screen management primitives

BELL

Function: Sounds the terminal's bell once.
Inputs: *none*
Outputs: *none*

BLINK_ON

Function: Turns the blink attribute on. Any character subsequently written to the screen will appear in blinking font.
Inputs: *none*
Outputs: *none*

BOLD_ON

Function: Turns the bold attribute on. Any character subsequently written to the screen will appear in highlighted font.
Inputs: *none*
Outputs: *none*

RVID_ON

Function: Turns the reverse video attribute on. Any character subsequently written to the screen will appear in reverse video font.
Inputs: *none*
Outputs: *none*

UDSC_ON

Function: Turns the underscore attribute on. Any character subsequently written to the screen will appear in

nderscored font.
Inputs: *none*
Outputs: *none*

RESTORE

Function: Turns off any selected font or combination thereof. It thus reverts to normal font. Any character subsequently written to the screen will appear in unblinking, unhighlighted, non reverse videoed, non underscored font. (ie. normal font)
Inputs: *none*
Outputs: *none*

Cursor control primitives

SET_CP (ROW: INTEGER;
COL: INTEGER)

Function: Will set the cursor position corresponding to the desired row and column coordinates. Row and column values beyond the ranges set below will cause the cursor to wrap around the screen.
Inputs: ROW -- An integer row number (1 to 25) where the cursor is to be positioned.
COL -- An integer column number (1 to 80) where the cursor is to be positioned.
Outputs: *none*

READ_CP (VAR ROW: INTEGER;
VAR COL: INTEGER)

Function: Reads the cursor position returning its row and column coordinates.
Inputs: *none*
Outputs: ROW -- The integer row number where the cursor is positioned. (1 to 25)
COL -- The integer column number where the cursor is positioned. (1 to 80)

HOME

Function: Positions the cursor at the HOME position (ie. topmost left hand corner) on the screen.
Inputs: *none*
Outputs: *none*

CLR_SCR

Function: The entire screen is cleared and the cursor placed at the HOME position.
Inputs: *none*
Outputs: *none*

CLR_LINE

Function: The current line is cleared from the current cursor position onward.
Inputs: *none*
Outputs: *none*

UP_SCR (INC: INTEGER)

Function: Moves the cursor up the screen by the number of rows specified by the increment, while maintaining the current column position.
Inputs: INC -- An integer value which defines the number of rows to be moved up the screen. If the increment is too large to be accommodated, the cursor is placed in the first row.
Outputs: *none*

DN_SCR (INC: INTEGER)

Function: Moves the cursor down the screen by the number of rows specified by the increment, while maintaining the current column position.
Inputs: INC -- An integer value which defines the number of rows to be moved down the screen. If the increment is too large to be accommodated, the cursor is placed in the last row.
Outputs: *none*

CUR_RIGHT

Function: Moves the cursor to the right by one position while maintaining the current row. If this function is performed when the cursor is in column 80, then no action is taken.
Inputs: *none*
Outputs: *none*

CUR_LEFT

Function: Moves the cursor to the left by one position while maintaining the current row. If this function is performed when the cursor is in column 1, then no action is taken.
Inputs: *none*
Outputs: *none*

Logical screen formatting

SCR_FORMAT

Function: Displays on the physical VDU screen the four logical

screen partitions.
Inputs: *none*
Outputs: *none*

Operators for the Main Screen

MS_CLEAR

Function: Clears the logical Main Screen (ie. Screen 1)
Inputs: *none*
Outputs: *none*

MS_CUR_ON (MS_CUR_POS: INTEGER;
MS_TOP_LINE: INTEGER)

Function: Turns on the cursor (Cursor 1) in the Main Screen by reverse videoing the line number.
Inputs: MS_CUR_POS -- The Main Screen CURsor POSition is an integer value from 0 to 19 which specifies (from the top of the screen) the row of the Main Screen on which Cursor 1 is to lie.
MS_TOP_LINE -- The Main Screen TOP LINE is an integer value specifying the actual line number of the line displayed on the top line of the Main Screen. Using the two above inputs, the actual line number of Cursor 1 can easily be calculated as (*MS_TOP_LINE* + *MS_CUR_POS*).
Outputs: *none*

MS_CUR_OFF (MS_CUR_POS: INTEGER;
MS_TOP_LINE: INTEGER)

Function: Turns off the cursor (Cursor 1) in the Main Screen by rewriting the actual line number in normal font.
Inputs: *as above*
Outputs: *none*

MS_WRITE (LINE_NUMBER: INTEGER;
MS_LINE_POS: INTEGER;
VAR STRING_ONE: STRING (81) OF CHAR;
VAR STRING_TWO: STRING (81) OF CHAR;
VAR STRING_THREE: STRING (81) OF CHAR)

Function: Will write out a line of text (represented by the three strings) at a specified row in Screen 1. A long text line will be truncated to fit on a single physical screen line, so that no overwrapping is allowed.
Inputs: LINE_NUMBER -- An integer variable giving the actual line number to be displayed.
MS_LINE_POS -- An integer variable (0 to 19) specifying the row number where the line is to be displayed in Screen 1.
STRING_ONE -- A *text string which represents* the

first part of the line to be written. This string is displayed in highlighted font. (It is normally reserved for the first key word.)

STRING_TWO -- A text string which represents the second part of the line to be written. This string is displayed in normal font. (It is normally reserved for the user-entered text.)

STRING_THREE -- A text string which represents the third part of the line to be written. This string is displayed in highlighted font. (It is normally reserved for the second key word.)

Outputs: *none*

Operators for the Window Screen

WS_CLEAR

Function: Clears the logical Window Screen (ie. Screen 2).
Inputs: *none*
Outputs: *none*

WS_ASET_CP (COL: INTEGER)

Function: Sets the cursor position to the absolute column specified in the Window Screen.
Inputs: COL -- An integer variable (1 to 80) used to specify the column which the cursor is to be moved to. If the number specified is beyond the given range, the cursor will wrap around the screen.
Outputs: *none*

WS_RV_WRITE (START_COL: INTEGER;
VAR TEXT_STRING: STRING (80) OF CHAR)

Function: Writes the text string in reverse video font in the Window Screen starting at the specified starting column. This function is used mainly for highlighting fields in the Data Definition Insert mode.
Inputs: START_COL -- An integer parameter used to specify the position in the Window Screen from which the text must be written.
TEXT_STRING -- The text string which must be written out.
Outputs: *none*

WS_HI_WRITE (START_COL: INTEGER;
VAR TEXT_STRING: STRING (80) OF CHAR)

Function: Writes the text string in highlighted font in the Window Screen starting at the specified starting column. This function is used mainly for highlighting fields in the Data Definition Insert mode.

first part of the line to be written. This string is displayed in highlighted font. (It is normally reserved for the first key word.)

STRING_TWO -- A *text string which represents* the second part of the line to be written. This string is displayed in normal font. (It is normally reserved for the user-entered text.)

STRING_THREE -- A text string which represents the third part of the line to be written. This string is displayed in highlighted font. (It is normally reserved for the second key word.)

Outputs: *none*

Operators for the Window Screen

WS_CLEAR

Function: Clears the logical Window Screen (ie. Screen 2).
Inputs: *none*
Outputs: *none*

WS_ASET_CP (COL: INTEGER)

Function: Sets the cursor position to the absolute column specified in the Window Screen.
Inputs: *COL -- An integer variable (1 to 80) used to specify the column which the cursor is to be moved to. If the number specified is beyond the given range, the cursor will wrap around the screen.*
Outputs: *none*

```
WS_RV_WRITE (    START_COL: INTEGER;
              VAR TEXT_STRING: STRING (80) OF CHAR)
```

Function: Writes the text string in reverse video font in the Window Screen starting at the specified starting column. This function is used mainly for highlighting fields in the Data Definition Insert mode.
Inputs: START_COL -- An integer parameter used to specify the position in the Window Screen from which the text must be written.
TEXT_STRING -- The text string which must be written out.
Outputs: *none*

```
WS_HI_WRITE (    START_COL: INTEGER;
              VAR TEXT_STRING: STRING (80) OF CHAR)
```

Function: Writes the text string in highlighted font in the Window Screen starting at the specified starting column. This function is used mainly for highlighting *fields in the Data Definition Insert mode.*

Inputs: *same as above*
Outputs: *none*

WS_LO_WRITE (START_COL: INTEGER;
VAR TEXT_STRING: STRING (80) OF CHAR)

Function: Writes the text string in normal font in the Window Screen starting at the specified starting column. This function is used mainly for distinguishing fields in the Data Definition Insert mode.
Inputs: *same as above*
Outputs: *none*

Operators for the Prompt Screen

PS_CLEAR

Function: Clears the logical Prompt Screen (ie. Screen 3).
Inputs: *none*
Outputs: *none*

PS_WRITE (MESSAGE: STRING (81) OF CHAR)

Function: Writes a message to the Prompt Screen in highlighted font.
Inputs: MESSAGE -- The message which is to be written in the Prompt Screen.
Outputs: *none*

Operators for the Function Screen

FS_CLEAR

Function: Clears the logical Function or Key Definition Screen (Screen 4).
Inputs: *none*
Outputs: *none*

FS_WRITE (KEY_STRING: STRING (81) OF CHAR;
FLAG_ARRAY: ARRAY [1..10] OF BOOLEAN)

Function: Writes out the 10 function key options in the Function Screen. Only the function keys which correspond to a TRUE flag value (ie. a valid function key) will be displayed as dictated by the key string. Valid function key options will be written in reverse video font at the field position in Screen 4 corresponding to their number.
Inputs: KEY_STRING -- This is the text line containing the 10 function key definitions which are to be selectively displayed in Screen 4.
FLAG_ARRAY -- The boolean array indicating the valid

function keys which will be highlighted in reverse video in Screen 4. There are 10 flags available; one for each function key. A true flag will indicate a valid function key.

Outputs: *none*

3.1.3 Resource structure

The Terminal Resource is not a memory-based resource as no common data structure is needed. Each operator is built as a procedure, and is essentially independent of external data structures. For IBM-PC implementation, a common terminal display function (DOSXQQ) is made generally available for use by the screen management primitive operators. Global constants include the following:

```
ROWS_PER_PAGE  = 25 -- The number of lines in the physical screen
MS_SIZE        = 20 -- The number of lines in the Main Screen
WINDOW_ROW     = 22 -- The Window Screen row number
PROMPT_ROW     = 24 -- The Prompt Screen row number
FUNCTION_ROW   = 25 -- The Function Screen row number
MESSAGE_COLUMN = 1  -- The message column number
```

Besides the constants above, an input and an output text file is also specified. These two files (INP and OUP) are used exclusively for all types of inputs and outputs which are not directed to an external file.

It can be noted, when looking at the available operators, that a few distinct categories exist. They are so divided for easy reference to the designer. The screen management and cursor control primitives are the backbone routines on which the other operators depend. It is thus true to say that only these few routines will need to be changed if the package is transported to another VDU type system.

3.2 Definition Table

3.2.1 Resource function

This resource was constructed for the main purpose of making system operation as flexible as possible. Instead of hard-coding system features into the cyntax-directed PDL generator, a means of making the system as programmable as possible was sought. Working with the idea that each new function key definition screen represents a new system state, a definition or state table solution emerged.

The Definition Table consists of system data for each function key of each state present in the package. The data held for each function key includes two Prompt Codes, two Next States and a Key Code. The Prompt Code corresponds to a prompt which will be displayed after the depression of that key. The Next State determines the state to which control will be passed after that

key is depressed. The Key Code is a code given to that particular function key as defined in the present state. This code may be used by the program for intelligent checking.

The two Prompt Codes and Next States are used, one, for normal operation, and the other in case of the system trapping a user-entered error. Thus, two possible branches are provided for any function key in any state. Error detection is done external to this resource, and the appropriate branch is then chosen. The system will therefore access this resource to find its next state. This makes the entire package programmable from the Definition Table.

The Definition Table is one of the system tables which is loaded from file on entering the syntax-directed editor. The file used for this purpose is "DT.SYS". This facility for loading the system files from disk makes the package easy to modify without the need for recompilation or linking. Speed is greatly enhanced by loading the table into memory. The detailed Definition Table can be found in Appendix C.

3.2.2 The operators

There are three operators for this resource. The initialization operator is used only when the table is to be loaded from diskfile. As there exist no operators for modifying the contents of the Definition Table, the loading operation is performed only once: on entering the PDL editor.

There are two read operations: A and B Reads. The A Read operator *is used initially to obtain the function definition line.* The user is then presented with the appropriate state. When a decision is made using a valid function key, then the B Read operator is used to obtain the subsequent Prompt Code and Next State.

```
DT_INIT (VAR NO_OF_STATES: INTEGER)

Function: Initializes the Definition Table by loading the data
          from diskfile. This operation is used on editor entry
          to load the data into dynamic memory.
Inputs:   *none*
Outputs:  NO_OF_STATES -- An integer value indicating the number
                          of states available in the table.


   DT_A_READ (    THIS_STATE: INTEGER;
                  NO_OF_STATES: INTEGER;
              VAR KEY_DEF_STRING: STRING (80) OF CHAR;
              VAR KEY_FLAG_ARRAY: ARRAY [1..10] OF BOOLEAN;
              VAR STATUS: INTEGER)

Function: Reads the key definition line and its associated key
          flag array from the Definition Table using a given
          state. The status flag is set unsuccessful when the
          input state does not exist.
Inputs:   NO_OF_STATES -- This number is usually taken directly
                          from the output of the DT_INIT routine
```

and represents the number of states available in the Definition Table.

THIS_STATE -- The integer input state which is used to look up the key definition line. The *range is between 1 and NO_OF_STATES.*

Outputs: KEY_DEF_STRING -- A line of text containing the definition of the 10 function keys for the particular state chosen.

KEY_FLAG_ARRAY -- A boolean array of 10 (one for each function key) to determine which keys are valid. A TRUE flag will indicate a valid function key.

STATUS -- An integer variable which is set unsuccessful if the input state is not available in the table and successful otherwise.

```
DT_B_READ (    THIS_STATE: INTEGER;
               THIS_KEY: INTEGER;
           VAR NEXT_S1: INTEGER;
           VAR NEXT_S2: INTEGER;
           VAR KEY_CODE: INTEGER;
           VAR PROM_C1: INTEGER;
           VAR PROM_C2: INTEGER)
```

Function: This operator is used to obtain the two Next States, the two Prompt Codes and a Key Code associated with any valid function key, when it is depressed.

Inputs: THIS_STATE -- The state in which the user was when the function key was depressed.

THIS_KEY -- The number of the function key (1 to 10) which was depressed

Outputs: NEXT_S1 -- The Next State number if no error has occurred.

NEXT_S2 -- The Next State number if an error has occurred.

KEY_CODE -- A Key Code integer number assigned to each function key for the purpose of program segment identification. This key code may be used in the calling program for further calculations.

PROM_C1 -- The code of the prompt to be displayed after the function key has been depressed, and if no error has occurred.

PROM_C2 -- The code of the prompt to be displayed after the function key has been depressed, and if an error has occurred.

3.2.3 Resource structure

There are two global constants which are worthy of note, and these are:

MAX_STATES = 20 -- The maximum number of states for which space has been allocated in the Terminal Resource.

MAX_KEYS = 10 -- The number of function keys per state.

The memory based data structure is given below in PDL.

```
Types:
------
        FX_KEY_RECORD = Record:
                          KEY_FLAG    : Boolean
                          KEY_CODE    : Integer
                          PROMPT_C1   : Integer
                          PROMPT_C2   : Integer
                          NEXT_S1     : Integer
                          NEXT_S2     : Integer
                        End Record:
        FX_KEY_ARRAY = Array [1..MAX_KEYS] of FX_KEY_RECORD

        STATE_RECORD = Record:
                         FX_KEY      : FX_KEY_ARRAY
                         KEY_DEF_LN  : String (80) of Character
                       End Record:

        KEY_DEFN_TABLE = Array [1..MAX_STATES] of STATE_RECORD

Variables:
----------
    KEY_DEFN_TABLE:
        Single:
            Permanent:
                DEFN_ARRAY
```

Thus, the key definition line of, say State 5, can be pointed to in Pascal as shown below:

```
DEFN ARRAY [5]. KEY_DEF_LN
```

while the key code of the third function key in State 5 is referred to as:

```
DEFN_ARRAY [5]. FX_KEY [3]. KEY_CODE
```

The data structure is available to all the operators of the resource. It can be seen that this table requires a large section of memory space. The Definition Table is loaded from the file "DT.SYS", and this file must be in the format shown below:

```
                                ...
                                ...
  DD Seg   Constr   Ins Ln                                    RETURN
50    1      1      3    3
60    8      8      8    8
70    10     10     10   10
0     0      0      0    0
0     0      0      0    0
0     0      0      0    0
0     0      0      0    0
4000  0      0      9    9
0     0      0      0    0
0     0      0      0    0
                                ...
                                ...
```

One state is shown above. The first line represents the key

definition line: each of the 10 fields is 8 characters wide; the first two of which are reserved for the function key number (inserted by the PS_WRITE routine) thus leaving six characters to define the function key. The next 10 lines define the data associated with the 10 function keys. The data defined is as follows: Key Code; Prompt Code 1; Prompt Code 2; Next State 1; Next State 2. A key code of zero implies an invalid function key.

The Definition Table, like all resources, is built as a module which is separately compilable. A test program is thus available to test the three operators. In this test program, extensive testing for erroneous inputs is performed, so that the designer can experiment with all input combinations to determine the *resource's behavior before it is included in the program.* An extra function is also available in the test program which displays the logical structure of the Definition Table and its related data in a convenient form. The test program format used by the designer can be found in Appendix A.

3.3 Key Code Table

3.3.1 Resource function

This resource represents the system's list of "reserved words". *The Key Code Table consists of a list of key words or phrases* which can be identified by a unique key code. Each key word also has an indentation and edit flag associated with it. The indentation number, where applicable, will be an indication of the relative indentation which must be added to the present indentation to obtain the final prettyprinted key word. The boolean edit flag determines whether the line on which that particular key word appears will be editable or not.

All words which will appear in Screen 1 in highlighted font (data description and construct key words) can be found in this table. All placeholders will also be found here because these words are also system-generated. Null text key words are other types of elements needed in this table. These key words consist of no key *words at all, but only an indentation value. They represent* a relative indentation which must be assigned to a user-entered, editable text line. This means that even a line of text which has been entirely user-entered contains a key word which will determine its extra indentation which is to be added to its associated indentation level value.

The key codes are chosen carefully in all cases to ensure that enough room is allowed for the purposes of system expansion. The integer value of the key code is used in the program for intelligent decision taking. The key code determines whether the associated key word is placed in the pre-Data Description segment (a negative key code); the Data Description segment (0 to 40); or in the Algorithm segment (> 40).

The existence of a Key Code Table makes the syntax-directed PDL generator package flexible, as new key words and placeholders can easily be added, and old key words erased or modified if required. Full relative indentation control is also offered via this resource. This table too is loaded on initialization from

diskfile so that system key words and indentation can be modified without the necessity for re-compilation. Details of the KCT.SYS file are given in Appendix C.

3.3.2 The operators

This resource does not have an operator facility for modifying table contents, as it is assumed that any text editor can be used to access and modify the data file KCT.SYS. For this reason, the initialization operator should only be used once, and this on entering the PDL editor. Only a single operator, besides the initialization operator, is required to read the contents of the Key Code Table given a key code as input.

```
KCT_INIT (VAR KCT_SIZE: INTEGER)

Function: Used to load the file KCT.SYS into the memory-based Key
          Code Table.
Inputs:   *none*
Outputs:  KCT_SIZE -- The number of key code entries loaded into
                      the Key Code Table.


KCT_READ (    THIS_KEY_CODE: INTEGER;
          VAR THIS_KEY_WORD: STRING (30) OF CHAR;
          VAR THIS_INDEN: INTEGER;
          VAR THIS_EDIT_FLAG: BOOLEAN;
              KCT_SIZE: INTEGER;
          VAR STATUS: INTEGER)

Function: Reads the key word, relative indentation, and edit flag
          from the Key Code Table given an input key code. The
          status flag is returned unsuccessful only if the input
          key cannot be found.
Inputs:   THIS_KEY_CODE -- The input key code which will be
                           searched for in the table. When a
                           match is found, its associated
                           characteristics will be output.
          KCT_SIZE      -- This integer variable is usually taken
                           directly from the output of the
                           initialization operator.
Outputs:  THIS_KEY_WORD -- The key word corresponding to the key
                           code. Its maximun length is 30
                           characters.
          THIS_INDEN    -- The relative indentation correspon-
                           ding to the key code.
          THIS_EDIT_FLAG -- The boolean flag used to determine
                           whether the line consisting of the
                           given key code is editable or not. A
                           TRUE flag indicates editability. It
                           should be noted that only the first
                           key code of a line will determine its
                           editability. The second key code's
                           edit flag is not used.
          STATUS        -- This is an integer error flag which
                           will return unsuccessful if the input
                           key code is not found in the table,
                           and successful otherwise.
```

3.3.3 Resource structure

Global constants used in this procedure include:

```
KCT_CAP = 60  -- The capacity of the Key Code Table.  This is the
                 maximum number of key codes for which  provision
                 has been made.
KW_CAP  = 30  -- The maximum allowable length for any key word in
                 the table.
```

A PDL description of the data structure used in dynamic memory for the table is as follows:

```
Types:
------
        KCT_FL = Record:
                   KEY_CODE     : Integer
                   KEY_WORD     : String (KW_CAP) of Character
                   INDEN        : Integer
                   EDIT_FLAG    : Boolean
                 End Record:

        KCT_ARRAY = Array [1..KCT_CAP] of KCT_FL

Variables:
----------
    KCT_ARRAY:
        Single:
            Permanent:
                KCT_FILE
```

The file from which the Key Code Table is initialized (KCT.SYS) must be in the format shown below:

```
                        ..
                        ..
                N 102    2 else:*
                        ..
                        ..
```

The first field specifies whether the key code implies an editable line or not (Y=Yes; N=No). This is followed by a single blank character. The key code then follows. The third field is the relative indentation, and is followed by a single blank character. The key word then appears, with a "*" delimiter to demarcate the end of the key word. Thus, the key word will be identified as the last characters up to but excluding the last "*" symbol.

A menu-driven test program is available to investigate the function of the operators of this table. Full input variable testing is performed. An extra routine is used to display the logical contents of the Key Code Table. Appendix A shows how the designer can investigate the operation of this resource via the test program.

3.4 Prompt Table

3.4.1 Resource function

This is the simplest resource but yet it is of importance. It is a store of all the system prompts or error messages which are available. A prompt code obtained from the Definition Table is used to access the Prompt Table. The associated message is then passed back to be displayed in the Prompt Screen.

The purpose of this table is twofold: firstly, error messages can easily be modified; and secondly, space is saved when dealing with duplicate messages. Prompts are not "hard programmed" within the package and can thus easily be edited by changing the data file contents. This saves the designer time as no re-compilation need be performed. Flexibility is added to the package in that it is a simple matter to add extra prompts. Also, if the same prompt is used twice in an algorithm, only the integer prompt code rather than the entire eighty-character prompt line needs to be duplicated. This accounts for a large saving in memory space at the cost of a slightly slower response time.

The Prompt Table is loaded initially from diskfile (PT.SYS) and thenceforth may not be modified. Appendix C contains the Prompt Table for the PDL syntax-directed editor package.

3.4.2 The operators

As in the Key Code Table, initialization is only performed once for loading the system file into memory. Hereafter, only the read *operation may be requested.* As the Prompt Table does not have operators to modify its contents dynamically, operations are restricted to the two mentioned above.

```
PT_INIT (VAR PT_SIZE: INTEGER)

Function: Initializes the Prompt Table by loading the system file
          PT.SYS into memory.
Inputs:   *none*
Outputs:  PT_SIZE -- The number of prompts which have been loaded
                     into the Prompt Table.


PT_READ (     THIS_CODE: INTEGER;
          VAR THIS_PROMPT: STRING (80) OF CHAR;
              PT_SIZE: INTEGER;
          VAR STATUS: INTEGER)

Function: Reads the prompt associated with the input prompt code.
Inputs:   THIS_CODE -- The input prompt code which corresponds in
                       line number to the required prompt.
          PT_SIZE   -- The size of the Prompt Table which is
                       usually taken directly from the output of
                       the PT_INIT operation.
Outputs:  THIS_PROMPT -- The prompt or message which corresponds
                         to the input prompt code. It has a
                         maximum length of 80 characters.
          STATUS      -- An integer variable which will return
```

successful if the prompt code exists, and unsuccessful otherwise. As the prompt code corresponds to the entry line number of the prompt, if the prompt code is greater than the PT_SIZE, then the prompt code is beyond the allowable range.

3.4.3 Resource structure

The global constants available in this resource are as follows:

```
PROM_TABLE_CAP = 20 -- The space allowed for prompts in the
                       Prompt Table. (ie. a maximum of 20 prompts
                       are allowed)
PROMPT_CAP     = 80 -- The capacity of the prompt message to be
                       displayed in the Prompt Screen in terms of
                       characters.
```

The data structure adopted is as follows:

```
Types:
------
       PROMPT_LINE = String (PROMPT_CAP) of Character

       PROMPT_LN_ARRAY = Array [1..PROM_TABLE_CAP] of PROMPT_LINE

Variables:
----------
    PROMPT_LN_ARRAY:
        Single:
            Permanent:
                PROMPT_ARRAY
```

The file PT.SYS which is to be loaded into memory initially consists simply of a number of lines; each containing a prompt. The prompt code associated with each prompt will correspond to its line number. (eg. The third prompt in the file will be on line 3 and will thus have a prompt code of 3.) A prompt must be no longer than 80 characters and must be entered in the table as it is to be displayed on Screen 3 on a single line.

A test program for the Prompt Table resource is also available for the user to experiment with its functions. A logical display routine is needed for the purpose of displaying the Prompt Table on the screen. Appendix A shows the layout of this test program.

3.5 Line Linked List

3.5.1 Resource function

The Line Linked List is used solely by the Line Editor process. It enables a user to manipulate characters within a text line. The linked list offers the basic editing primitives. With the use of a list, it is possible to perform an endless amount of edits

on a line with a fixed allocated memory space. The linked list ensures that a memory element which is deleted will remain on a space list to be used whenever another memory location is needed. The designer will access the linked list only by means of its operators.

Operators are provided for: initializing the list to zero (ie. emptying the Line Linked List) getting and returning a record for the purposes of writing or deleting a character; reading and writing a record; and moving the list pointer. Operators are also available for returning the list pointer value and for returning the edited line in order. With these primitive operators, any text editing function can be constructed.

In the following paragraphs it will be explained how the basic line editing functions can be constructed by combining one or more of the above operators. Linked list operation is explained in section 3.5.3, but for further clarification, the notes of Dr. A.J.Walker (1984) should be consulted. By following the application examples below, however, a good idea of this resource's behavior should be obtained.

To start editing a new line, the list is initialized. This sets the record pointers in consecutive order and effecitvely clears all data records. Writing a character into the list firstly requires the retrieval of a record. Hereafter, the list pointer will indicate the new record, and thus a character can be writen into it. It is imperative to note that if a new character is to be added, a new record must be fetched before a Write operation is performed.

The list pointer is initially at the zero position, and a Write operation here will be unsuccessful. If, however, the pointer is at an existing record in which a character is already written, a Write operation here will cause overtyping (ie. replacing of the old character by the new). After a Write operation, the pointer remains at the newly edited character (ie. all pointer movements must be performed explicitly).

An insert operation is also a Get-record operation followed by a Write-record operation. It should, however, be noted that a record is always inserted after the current pointer position, and the pointer is then positioned at the newly inserted record. Thus, the first record is inserted at character position 1 by performing a Get-record operation when the list pointer is on the zero position. If a series of characters is to be inserted sequentially, the convention is favourable, as new records are always inserted after the previously entered character.

The delete function of the Line Editor is implemented by making use of the Return-record operator in the Line Linked List. With the list pointer on the character which is to be deleted, a Return-record operation is performed. That record will then effectively be deleted, and the pointer moved back by one position. Thus, if a series of characters is deleted sequentially, it will become obvious that this function is associated with the destructive backspace key.

By moving the list pointer, the line editor cursor can effectively be moved under any character. Pointer movement is, however, incremental and not absolute, so that a function to

determine the list pointer location is useful. The Read-record operator will return the character at the current pointer position. After a series of edits on a line, it is useful to have an operator which will return the entire new line in logical order. This is the Log-string operator.

The functions described above will be used in the Line Editor process of section 4.1. This section should be consulted for detailed PDL descriptions of the above line editing routines. The linked list is thus a powerful resource for any form of editor. The Line Editor, as well as the File Editor of this PDL syntax-directed editor package are based on the Line Linked List and File Linked List respectively.

3.5.2 The operators

LIST_INITIALISE

Function: Initializes the Line Linked List, setting all records to the null character.
Inputs: *none*
Outputs: *none*

LIST_GET_RECORD (VAR STATUS: INTEGER)

Function: Fetches a record from the space list and inserts it into the linked list.
Inputs: *none*
Outputs: STATUS -- An integer error flag having one of the following possible outcomes:
Successful -- record was successfully fetched
Empty_space_list -- the list is full and no more records can be inserted.

LIST_RETURN_RECORD (VAR STATUS: INTEGER)

Function: Returns a record to the space list, effectively deleting it from the linked list. The record returned is the one pointed to by the list pointer. After the operation, the pointer is moved back by one position.
Inputs: *none*
Outputs: STATUS -- The integer error flag. It has one of the following outcomes for this operation:
Successful -- record was sucessfully returned
Empty_link_list -- the list is empty and thus no record can be returned
LLP_outside_list -- with the Logical List Pointer in the zero position, no record can be returned.

LIST_READ_RECORD (VAR DATA_ITEM: CHAR;
VAR STATUS: INTEGER)

Function: Reads and returns the charac (or value of the da item) present in the record pointed to by the li pointer.

Inputs: *none*
Outputs: DATA_ITEM -- The character which is read from the linked list
STATUS -- The integer error flag having one of the following possible outcomes:
Successful -- the operation was performed successfully
Empty_link_list -- no read operation can be performed on an empty list
LLP_outside_list -- no read operation can be performed with the list pointer at the zero position.

LIST_WRITE_RECORD (DATA_ITEM: CHAR;
VAR STATUS: INTEGER)

Function: Writes the input character in the record which is pointed to by the list pointer.
Inputs: DATA_ITEM -- The input character which is to be written at the current list pointer position.
Outputs: STATUS -- The integer error flag having the following possible outcomes:
Successful -- the operation was performed successfully
Empty_link_list -- no records are available for writing into
LLP_outside_list -- no writing can be performed with the list pointer in the zero position.

LIST_MOVE_POINTER (INCREMENT: INTEGER;
VAR STATUS: INTEGER)

Function: Moves the list pointer forward or backward by a positive or negative increment.
Inputs: INCREMENT -- A positive, negative, or zero integer value which will move the list pointer forward, backward, or not at all by the specified amount from the current pointer position. If too large an increment is specified, the pointer is moved as far as possible.
Outputs: STATUS -- The integer error flag having the following possible outcomes:
Successful -- the operation was performed successfully
Empty_link_list -- the pointer cannot be moved if the list is empty
LLP_outside_list -- the increment is too large or too negative, so that the list pointer would have to be moved beyond the end of the list or before the beginning of the list. In this case, the pointer is still moved to the relevant list limit, but the STATUS flag tells of the overshoot problem.

```
LIST_LOG_INFO (VAR LIST_POINTER: INTEGER)

Function: Returns the value of the logical link list pointer.
Inputs:   *none*
Outputs:  LIST_POINTER -- The list pointer value.


      LIST_LOG_STRING (VAR T_STRING: STRING (80) OF CHAR;
                       VAR RETURN_LOOP_COUNT: INTEGER;
                           LIST_POINTER: INTEGER)

Function: Returns the linked list elements in logical order in a
          single string variable from the specified input logical
          list pointer value.
Inputs:   LIST_POINTER -- The logical list pointer position from
                          which the rest of the linked list is to
                          be returned.
Outputs:  T_STRING     -- The linked list elements in logical
                          order listed from the specified list
                          pointer position.
          RETURN_COUNT -- The number of characters contained in
                          the returned text string.
```

3.5.3 Resource structure

The Line Linked List data structure is as follows:

```
Constants:
----------
    Integer:
        Single:
            Local:
                MR  = 81  *Maximum Records -- the capacity of the
                           resource*
                SUCCESSFUL       = 0
                EMPTY_SPACE_LIST = 1  *possible outcomes of the
                EMPTY_LINK_LIST  = 2   status flag*
                LLP_OUTSIDE_LIST = 3

Types:
------
        LIST_RECORD = Record:
                        FP: Integer    *Forward Pointer*
                        BP: Integer    *Backward Pointer*
                        DI: Character  *Data Item*
                      End Record:

        MY_LIST = Array [1..MR] of LIST_RECORD

Variables:
----------
    Integer:
        Single:
            Local:
                L_LLR    *Line Link List Rock*
                L_SLR    *Line Space List Rock*
                L_PLP    *Line Physical List Pointer*
                L_LLP    *Line Logical List Pointer*
                L_LLS    *Line Link List Size*
```

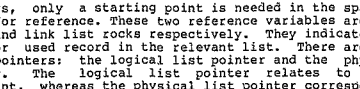
```
MV_LIST:
    Single:
        Permanent:
            L_LIST
```

A brief explanation of linked list operation is given below:

The linked list is used for manipulating large blocks of text efficiently without wasting memory space. To do so, each element of text is regarded as a record or data item. (For the Line Linked List, a character is the data item, while for the File Linked List a text line is regarded as a data item.) The linked list will use the forward and backward pointers to link all the data items in such a way as to form the text block. Data items which are not used are stored in the space list, while records in use are stored in the link list.

As all records are linked via their forward and backward pointers, only a starting point is needed in the space and link lists for reference. These two reference variables are called the space and link list rocks respectively. They indicate the first free or used record in the relevant list. There are two major list pointers: the logical list pointer and the physical list pointer. The logical list pointer relates to the user's viewpoint, whereas the physical list pointer corresponds to the actual location of the data item in the linked list.

The following example should help to clarify the linked list structure and operation. Assume that a linked list of 10 characters exists, with the word "MESSGE" written sequentially in it as shown below:



Looking at the forward pointer (FP) for the letter "M", a value of 2 indicates that the next data item (DI) linked after "M" is found in location 2. In location 2, the first "E" of "MESSGE" is found, and its forward pointer points to the letter "S. This is continued until all the letters of the word are linked. Note that the last letter of the listed word has a forward pointer which indicates the end of the list (Eol).

In this way it can be seen that the word "MESSGE" is linked character by character both in the forward direction (by the forward pointer) and in the backward direction (by the backward pointer (BP)). These characters which exist in the list comprise the link list, while the remaining records constitute the space list. The space list elements are also linked via forward and backward pointers.

Let us now, for the sake of clarity, consider the list to have a forward pointer only. The following can be said:

LLR = 1 -- Link list rock indicates the location of the first record of the list
SLR = 7 -- Space list rock indicates the location of the first free record
LLS = 6 -- Link list size is of six characters: M,E,S,S,G,E.

Now, the word "MESSGE" is to be corrected to read "MESSAGE". This requires an "A" to be inserted after the second "S". Thus, the logical pointer is moved to position 4. (Note that this also corresponds to the physical pointer position.) The Get-record operation is now performed. This results in a blank record being inserted after the second "S" of "MESSGE". Physically, this involves the increasing of the link list size by one, and a corresponding decrease in size of the space list. After the "A" has been written into the new record, the schematic representation of the word in the list is as follows:

Location:	0	1	2	3	4	5	6	7	8	9	10
DI	Eol	M	E	S	S	G	E	A			
FP		2	3	4	7	6	0	5	9	10	0

It can be seen that at position 4, the forward pointer indicates the character "A" at position 7. The letter "A" in turn indicates position 5 so that the word "MESSAGE" results. The list parameters are now as follows:

LLR = 1 -- The start of the link list is still unchanged.
SLR = 8 -- The first free record has been used so that the space list size has decreased by one.
LLS = 7 -- The size of the link list has increased by one.

The pointer is now located at the letter "A" (as seen from the underscoring in the previous figure). This position corresponds to :

PLP = 7 -- Physical list pointer is the actual pointer location in the data structure.
LLP = 5 -- Logical list pointer is the logical location of the pointer in the word "MESSAGE" (ie. M=1; E=2; S=3; S=4; A=5; G=6; E=7).

The above example illustrates both the power and the complexity of the linked list. An interactive menu-driven test program with full input condition testing is available for the designer to become acquainted with Line Linked List operation. Appendix A explains further the facilities of this resource using both a logical and a physical model.

3.6 File Linked List

3.6.1 Resource function

The File Linked List is similar in operation to the Line Linked List described above, however, it deals with the manipulation of text lines. This resource treats an entire PDL text line as a record. This enables the designer to use this resource in the File Editor process for line manipulation purposes. Section 4.4 details how insertion is performed with the help of this resource. It should also be noted that the coded version of the PDL file is stored in this resource, and not the formatted version. This means that a record in the File Linked List will contain the key codes, indentation and text line fields as well as the necessary forward and backward pointers.

3.6.2 The operators

The operations which can be performed on the File Linked List are identical to those for the Line Linked List, with one exception. The LIST_LOG_STRING operator in the Line Linked List is not used for this resource. The PDL file stored in the File Linked List need only be arranged when an editing session is ended. Separate routines for outputing formatted and unformatted files are provided in the front-end level of the package. (See FILE_UF_DUMP in Chapter 5)

FLL_INITIALISE

Function: Initializes the File Linked List by resetting its records to the null data item.
Inputs: *none*
Outputs: *none*

PLL_GET_RECORD (VAR STATUS: INTEGER)

Function: A record corresponding to a PDL line is obtained from the space list and inserted in the link list after the File Linked List pointer.
Inputs: *none*
Outputs: STATUS -- An integer error flag having one of the following possible outcomes:
Successful -- record was successfully fetched
Empty_space_list -- the list is full and no more records can be inserted.

FLL_RETURN_RECORD (VAR STATUS: INTEGER)

Function: A text line (or record) is returned to the space list, thus deleting it from the link list. The record pointed to by the pointer is returned; after the operation the pointer is moved to the previous record.
Inputs: *none*
Outputs: STATUS -- The integer error flag. It has one of the following outcomes for this operation:

Successful -- record was sucessfully returned
Empty_link_list -- the list is empty and thus no record can be returned
LLP_outside_list -- with the Logical List Pointer in the zero position, no record can be returned.

```
FLL_READ_RECORD (VAR INDEN_CODE: INTEGER;
                 VAR KEY_CODE_1: INTEGER;
                 VAR KEY_CODE_2: INTEGER;
                 VAR TEXT_LINE: STRING (81) OF CHAR;
                 VAR STATUS: INTEGER)
```

Function: Reads the record fields associated with the record pointed to by the list pointer (ie. the attributes associated with that text line).
Inputs: *none*
Outputs: INDEN_CODE -- The indentation code of the PDL line which determines its absolute placement from the left-hand margin when pretty-printed.
KEY_CODE_1 -- The first key code associated with a system-generated key word which precedes any user-entered text.
KEY_CODE_2 -- The second key code associated with a system-generated key word which follows any user-entered text.
TEXT_LINE -- The variable string containing the user-entered text.
STATUS -- The integer error flag having one of the following possible outcomes:
Successful -- the operation was performed successfully
Empty_link_list -- no Read operation can be performed on an empty list
LLP_outside_list -- no Read operation can be performed with the list pointer at the zero position.

```
FLL_WRITE_RECORD (    INDEN_CODE: INTEGER;
                      KEY_CODE_1: INTEGER;
                      KEY_CODE_2: INTEGER;
                      TEXT_LINE: STRING (81) OF CHAR;
                  VAR STATUS: INTEGER)
```

Function: Writes the information associated with the File Linked List record into the record pointed to by the list pointer.
Inputs: * same as the field outputs for the FLL_READ_RECORD operator above *
Outputs: STATUS -- The integer error flag having the following possible outcomes:
Successful -- the operation was performed successfully
Empty_link_list -- no records are available for writing into
LLP_outside_list -- no writing can be performed with the list pointer in the zero position.

```
FLL_MOVE_POINTER (    INCREMENT: INTEGER;
                      VAR STATUS: INTEGER)
```

Function: Moves the list pointer forward or backward by a positive or negative increment.
Inputs: INCREMENT -- A positive, negative, or zero integer value which will move the list pointer forward, backward, or not at all by the specified amount from the current pointer position. If too large an increment is specified, the pointer is moved as far as possible.
Outputs: STATUS -- The integer error flag having the following possible outcomes:
Successful -- the operation was performed successfully
Empty_link_list -- the pointer cannot be moved if the list is empty
LLP_outside_list -- the increment is too large or too negative, so that the list pointer would have to be moved beyond the end of the list or before the beginning of the list. In this case, the pointer is still moved to the relevant list limit, but the STATUS flag tells of the overshoot problem.

```
FLL_LOG_INFO (VAR LIST_POINTER: INTEGER)
```

Function: Returns the value of the logical link list pointer.
Inputs: *none*
Outputs: LIST_POINTER -- The list pointer value.

3.6.3 Resource structure

The data structure for the File Linked List is as follows:

```
Constants:
----------
    Integer:
        Single:
            Local:
                MR  = 81  *Maximum Records -- the capacity of the
                          resource*
                SUCCESSFUL       = 0
                EMPTY_SPACE_LIST = 1  *possible outcomes of the
                EMPTY_LINK_LIST  = 2   status flag*
                LLP_OUTSIDE_LIST = 3

Types:
------
        LIST_RECORD = Record:
                        FP : Integer          *Forward Pointer*
                        BP : Integer          *Backward Pointer*
                        DI1: Integer          *Indentation code*
                        DI2: Integer          *Key code 1*
                        DI3: Integer          *Key code 2*
```

```
                    FT : String (81) of Character      *user-
                                                entered File Text line*
                End Record:

    MY_LIST = Array [1..MR] of LIST_RECORD

Variables:
---------
    Integer:
        Single:
            Local:
                LLR     *File Link List Rock*
                SLR     *File Space List Rock*
                PLP     *File Physical List Pointer*
                LLP     *File Logical List Pointer*
                LLS     *File Link List Size*
    MY_LIST:
        Single:
            Permanent:
                MY_LIST
```

The structure of the linked list is as described in section 3.5.3. The only point of interest here is the multiple fields involved in describing a record. To define any PDL line in the system, the codes and text line mentioned above are required. With the edited file being list-based, extensive line manipulation is thus possible via this resource.

A test program for this resource is provided (see Appendix A). The designer can thus become familiar with this resource due to the extensive input error checking facilities available. A physical and logical display of the state of the resource is also available via this test routine.

4 THE PROCESSES

4.1 The Line Editor

4.1.1 Process function

The Line Editor is designed to be a self-contained procedure for use as a package when any user-entered response is expected in the Window Screen. The Line Editor functions are needed when the user is entering or modifying any line or entering a filename as requested by the system. This process contains the usual editing functions required of a line editor. The user can overtype, insert, delete and move the cursor (Cursor 2) under any character in the editable string.

The Line Editor is exit by one of three methods:

1. Using the ENTER key --- The new edited text string is accepted and the editor abandoned.
2. Using a valid Function Key --- The valid set of function keys is determined by the boolean array. On depression of one of these keys, the new line is accepted, while a change of states also occurs.
3. Using the ESCAPE key --- This key will exit the editor without accepting the new edited line.

Complete error checking is performed within the Line Editor package to trap errors such as line overflow and other illegal operations. All errors display messages in the Prompt Screen, simultaneously sounding the terminal bell.

The editor functions are explained in section 4.1.3 with the use of the subroutines which have been used to create the Line Editor. Each subroutine corresponds to a certain Line Editor function. Section 4.1.2 gives the Line Editor program's structure which is used in this package.

4.1.2 Process structure

The routine call which invokes the Line Editor is shown below, together with its external input and output parameters.

```
LINE_EDITOR (     WS_WIDTH: INTEGER;
                  START_COL : INTEGER;
              VAR KEY_TEXT: STRING (80) OF CHAR;
              VAR IN_STRING: STRING (80) OF CHAR;
              VAR WSP: INTEGER;
                  KEY_FLAG_ARRAY: ARRAY [1..10] OF BOOLEAN;
              VAR OUT_STRING: STRING (80) OF CHAR;
              VAR OUT_KEY: INTEGER)
```

where the inputs are:

WS_WIDTH -- (Window Screen WIDTH) Defines the upper bound of the editable line in the Window Screen.

START_COL -- (START COLumn) defines the lower bound of the editable line in the Window Screen. The Line Editor will only be concerned with text between the two above limits.

KEY_TEXT -- The first key word in the Window Screen line. Note that this key word will be displayed in highlighted font and is not user-editable.

IN_STRING -- The user-editable input text string. This string is written in normal font immediately after the key word in the Window Screen and is completely user-editable.

WSP -- (Window Screen Position) The required starting position of Cursor 2 in the Line Editor. If this variable is input out of range, a best attempt is made at placing the cursor as close as possible to the required position. WSP can only be set to a character in the IN_STRING (ie. the key word is not editable and thus the cursor cannot be placed here).

KEY_FLAG_ARRAY - An array of flags for determining the function keys which will exit the Line Editor. A TRUE flag indicates that the corresponding function key will successfully exit the Line Editor. This array is usually obtained directly from the A Read operation on the Definition Table.

and the outputs are:

OUT_STRING - The user-editable text string which is output when the Line Editor is exit.

OUT_KEY -- An integer code which is passed back to the calling program to indicate what key combination terminated the line editing session. The possible outcomes of this variable are:

- 1 - 10 -- depending on which function key was used (any changes made in the Line Editor will be accepted)
- -1 -- if the ENTER key was used for exiting (all changes made in the Line Editor are accepted)
- -100 -- if the ESC key was used (any changes made while in the Line Editor are ignored and OUT_STRING is set to the IN_STRING value).

This Line Editor package can thus be used as a module where required, with the designer only being concerned with its interfacing to the calling program. This makes such a process usable in many other applications where a line editor is needed.

An example of initializing the Line Editor is given in Fig.4.1.

```
Inputs: WS_WIDTH  = 20
        START_COL = 2
        KEY_TEXT  = "Until "
        IN_STRING = "<CONDITION>"
        WSP       = 1
        KEY_FLAG_ARRAY = *all elements set to FALSE*
```

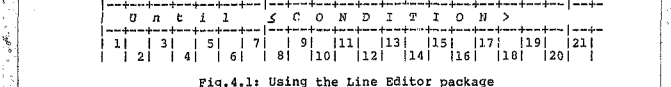


Fig.4.1: Using the Line Editor package

The figure shows a line in the Window Screen for a certain set of Line Editor input parameters. The numbers represent the column position from the left-hand margin of the Window Screen. The editable portion of the Window Screen is restricted to between characters 8 and 20 due to the WS_WIDTH, START_COL and KEY_TEXT input values. The cursor is not allowed outside these limits. The KEY_TEXT input is highlighted and written starting at the specified START_COL position. In this example, the <CONDITION> placeholder is to be expanded via the Line Editor. WSP was input as 1 which is clearly out of range. (Acceptable WSP inputs lie from 8 to 20.) The cursor (shown as an underscored character) is thus positioned under the first editable *character* (the "<"). For exiting, only the ENTER and ESC keys will be effective.

The resources used in the Line Editor are:

Line Linked List -- for character by character manipulation
Terminal Resource -- for output purposes in the Window Screen

Section 3.5.3 has already elaborated on how the Line Linked List can be used to form the basic features of a Line Editor. The Terminal Resource is used for the positioning of Cursor 2 in the Window Screen; the highlighting of the key word; and the displaying of the user-editable text line.

The Line Editor's routines are listed below:

```
LOAD        -- initializes the Line Editor
MOVE_CUR_F  -- moves the cursor (Cursor 2) forward by one
               position
MOVE_CUR_B  -- moves Cursor 2 backward by one position
WS_HOME     -- positions the cursor at the start of the editable
               text
END_OF_LINE -- positions the cursor at the end of the existing
               text line
ENTER_TEXT  -- allows the user to enter text one character at a
               time both in Type and Insert modes
INSERT      -- toggles the Character Insert mode
DELETE      -- deletes a single character immediately to the left
```

```
                of the cursor
ERASE_EOL    -- deletes all characters from the current cursor
                position to the end of the line
DUMP         -- dumps the resultant text line into the OUT_STRING
                variable.
```

A general description of the Line Editor process is given in Fig.4.2. The principal global variables used for this process are listed below:

```
INSERT_STATUS -- detects Insert mode (can be either ON or OFF)
LINE_LENGTH   -- keeps track of the current length of the user-
                 editable text line
MAX_LENGTH    -- determines the maximum length of the user-
                 editable text line
LLP           -- (Logical List Pointer) is the current pointer
                 position in the user-editable text in the Line
                 Linked List
WSP           -- (Window Screen Position) is the logical cursor
                 position (measured in columns from the left-hand
                 margin) in Screen 2.
```

The STATUS variable indicates the status of many of the operations performed on the Line Linked List, but is used sparingly.

```
Process  Line Editor
Begin:
  Repeat:
    Call LOAD
    If (Status <> Successful)
      then:
        Status := Successful
    End If:
    Get:
      Keyboard: Key
    End Get:
    Case (Key) Of:
        Cur Forward : Call MOVE_CUR_F
        Cur Backward: Call MOVE_CUR_B
        Home        : Call WS_HOME
        End         : Call END_OF_LINE
        Ins         : Call INSERT
        Del         : Call DELETE
        Cntrl K     : Call ERASE_EOL
      else:
        Call ENTER_TEXT
    End Case:
  Until (Key implies termination)
  Call DUMP
  *Set OUT_KEY to appropriate value*
End:
End Process:
```

Fig.4.2: A PDL description of the Line Editor

With reference to Fig.4.2, the Line Editor operation is described as follows. Initially, the LOAD routine is used to initialize the editor to its initial state as set by the inputs (see Fig.4.1). A *Repeat-until block is entered until the key obtained from* the keyboard (a single character) implies termination. The Line Editor can be exit by using a valid function key; the ENTER; or the ESC key. An unsuccessful status is set to successful before a key is obtained from the keyboard.

A somewhat more complex Case construct than that shown in Fig.4.2 is used to perform key capture and identification. A simplified version is given here, but the next section will elaborate on the subtleties involved when describing a similar procedure in Base Level. When the editor is exit, the DUMP routine is called and the OUT_KEY output variable is set. The DUMP routine is responsible for producing an output text string corresponding to the edited line.

The Line Editor is built as a stand-alone module so that it can be used in any line editing situation. A test program has been constructed allowing the designer to experiment with input parameters and editor operation. If any modifications are made to the *Line Editor, the test program can be used to check* if it performs as expected before it is integrated into the PDL generator package. The format of the test program can be seen in Appendix A.

4.1.3 Process routines

Load

The LOAD routine and its parameters is as shown below, while the PDL description of its behavior can be found in Fig.4.3.

```
LOAD (     START_COL: INTEGER;
      VAR MAX_LENGTH: INTEGER;
      VAR LINE_LENGTH: INTEGER;
      VAR WSP: INTEGER;
          KEY_TEXT: STRING (80) OF CHAR;
          IN_STRING: STRING (80) OF CHAR)
```

```
Procedure  Load
Begin:
  *Write KEY_TEXT in highlighed font in Screen 2*
  *Write IN_STRING in normal font in Screen 2*
  *Set the MAX_LENGTH of the line*
  *Set the LINE_LENGTH to the length of the IN_STRING*
  *Ini .ialize the Line Linked List*
  *Write the IN_STRING in the Line Linked List*
  *Check for any errors in the WSP input*
  If (There are any errors in WSP)
    then:
      *Correct WSP to beginning or end of line*
  End If:
  *Set Cursor 2 to the WSP value*
  *Move list pointer to the position corresponding to WSP*
End:
End Procedure:
```

Fig.4.3: The PDL for the Line Editor Load routine

This routine is used initially, each time the Line Editor is called. It is responsible for setting up the Line Editor as specified by the input parameters. The MAX_LENGTH, LINE_LENGTH and WSP variables are set in this routine. The Line Linked List will also be initialized and loaded with the IN_STRING; the list pointer being positioned by the WSP input value.

Cursor Forward

This routine moves Cursor 2 forward by one position on depression of the *Cursor Right* key. It is possible to move the cursor beyond the last character of the text line as long as the Window Screen bounds are not exceeded. If it is attempted to move the cursor beyond the right Window Screen bound, an error will occur. The calling routine is given below, while Fig.4.4 elaborates on the procedure construction.

```
MOVE_CUR_F (     INSERT_STATUS: BOOLEAN;
            VAR LINE_LENGTH: INTEGER;
                MAX_LENGTH: INTEGER;
                LLP: INTEGER;
            VAR WSP: INTEGER;
            VAR STATUS: INTEGER)
```

```
Procedure Move Cursor Forward
Begin:
  If (Cursor is moved beyond right bound in Window Screen)
    then:
      *Put out an error -- cursor at end of line*
    else:
      *Check if last character is a blank*
      If (Cursor is moved beyond end of text line)
        then:
          *Get a record*
          *Write a blank character in the new record*
          Call CUR_RIGHT
          WSP := WSP + 1
          LINE_LENGTH := LINE_LENGTH + 1
        else:
          If (Insert Status is ON)
            then:
              *Move pointer forward by one position*
              *Read the character at the pointer position*
              *Write the character followed by a blank*
              Call CUR_LEFT
              WSP := WSP + 1
            else:
              *Move pointer forward by one position*
              Call CUR_RIGHT
              WSP := WSP + 1
          End If:
      End If:
  End If:
End:
End Procedure:
```

Fig.4.4: The PDL for the Move Cursor Forward routine

Cursor Backward

This routine is used to move the cursor in the Window Screen to the left by one character. The Cursor Left key is assigned to this function. If the cursor is moved beyond the beginning of the editable text line, an error will occur and the terminal bell will sound. As in the Cursor Forward routine, when in Insert mode, the cursor will always be under a blank token. Moving the cursor either to the left or to the right will result in the previous or following character being interchanged with the blank token, again leaving the cursor under the blank insert token. The procedure is called as shown below; Fig.4.5 giving its internal structure.

```
MOVE_CUR_B (     INSERT_STATUS: BOOLEAN;
                 LLP: INTEGER;
           VAR WSP: INTEGER;
           VAR STATUS: INTEGER)
```

```
Procedure  Move Cursor Backward
Begin:
  If (Insert Status is ON)
    then:
      If (List pointer is at the zero position)
        then:
          *Put  out an error message -- cursor is at beginning of
           line*
        else:
          *Read the character pointed to by the list pointer*
          Call CUR_LEFT
          *Write a blank to the screen*
          *Write out the character obtained above*
          Call CUR_LEFT
          Call CUR_LEFT
          WSP := WSP - 1
          *Move the pointer back by one position*
      End If:
    else:
      **(ie. Insert Mode is OFF)**
      If (List pointer is at the start of the line)
        then:
          *Put out an error message -- cursor is at beginning  of
           line*
        else:
          Call CUR_LEFT
          WSP := WSP - 1
          *Move list pointer back by one position*
      End If:
  End If:
End:
End Procedure:
```

Fig.4.5: The PDL for the Move Cursor Backward routine

Cursor Home

This routine will move Cursor 2 to the beginning of the user-editable text line in the Window Screen. If in Insert mode, the cursor is placed at the first editable character position, but

the entire text string is shifted to the right by one character to make space for the blank insert token under which the cursor will lie. Fig.4.6 describes this simple procedure in PDL while the routine call is shown below:

```
WS_HOME (     INSERT_STATUS: BOOLEAN;
              MAX_LENGTH: INTEGER;
              LLP: INTEGER;
          VAR WSP: INTEGER;
          VAR STATUS: INTEGER)
```

```
Procedure  Home
Begin:
  *Set WSP to first editable character*
  *Move list pointer accordingly*
  *Set Cursor 2 to the WSP value*
  If (Insert Status is ON)
    then:
      *Move list pointer back by one position*
      Call LIST_LOG_STRING
      If (LINE_LENGTH = MAX_LENGTH)
        then:
          *Trim the above output string to its length minus one*
      End If:
      *Write out above output string*
  End If:
End:
End Procedure:
```

Fig.4.6: The PDL for the Line Editor's Home routine

Cursor to End of Line

```
Procedure  End of File
Begin:
  If (LINE_LENGTH = MAX_LENGTH) and (Insert Status is ON)
    then:
      *Put out an error*
    else:
      *Move list pointer to end of text string*
      If (LINE_LENGTH = 0)
        then:
          *Set WSP to first editable character*
        else:
          *Set WSP to end of text string*
      End If:
      If (Insert Status is ON)
        then:
          Call LIST_LOG_STRING
          *Write out the characters obtained above*
          WSP := WSP + 1
          Call CLR_LINE
        else:
          *Set Cursor 2 to WSP value*
      End If:
  End If:
End:
End Procedure:
```

Fig.4.7: The PDL for the End of Line routine

This routine (described in Fig.4.7) will move the cursor to the last character of the user-editable text string in the Window Screen. For the purposes of logical convenience, this operation is not performed when the line is full and Insert mode is ON. The routine call and parameters are given below.

```
END_OF_LINE (    INSERT_STATUS: BOOLEAN;
                 LINE_LENGTH: INTEGER;
                 LLP: INTEGER;
             VAR WSP: INTEGER)
```

Enter_Text

As this routine will be used most frequently in both typing and Character Insert mode, it involves many tests as can be seen from Fig.4.8. This function will concatenate any continuously user-entered characters into the text string. If overtyping is performed, the old character is replaced by the new, while typing beyond the last character of the line will add characters to the text string. In Insert mode, characters entered will be inserted sequentially at the blank insert token.

An error will result in normal Typing mode if it is attempted to type beyond the right bound of the Window Screen. In this case, the last character of the line is overtyped, and the user warned. *When in Character Insert mode, however, an error will occur once* the line length exceeds the MAX_LENGTH value. In checking for line length, if the last character is a blank, it will be discarded. The calling routine is presented below, while Fig.4.8 gives a more detailed PDL description of the Enter Text function.

```
ENTER_TEXT (     KEY: CHAR;
                 INSERT_STATUS: BOOLEAN;
             VAR LINE_LENGTH: INTEGER;
                 MAX_LENGTH: INTEGER;
                 LLP: INTEGER;
             VAR WSP: INTEGER;
             VAR STATUS: INTEGER)
```

```
Procedure  Enter Text
Variables:
     Boolean:
        Single:
           Local:
              POSSIBLE
Begin:
  If (Insert Status is ON)
    then:
      POSSIBLE := TRUE
      If (LINE_LENGTH = MAX_LENGTH)
        then:
          *Check if last character is a blank*
          If (Last character is not a blank)
            then:
              *Put out an error -- line is full*
              POSSIBLE := FALSE
            else:
```

```
            *Return the last blank character*
            LINE_LENGTH := LINE_LENGTH - 1
          End If:
        End If:
        If (POSSIBLE = TRUE)
          then:
            *Write the input key to the screen*
            WSP := WSP + 1
            *Get a record from the space list*
            *Write the input key in the new record*
            LINE_LENGTH := LINE_LENGTH + 1
            If (Line length is a maximum and pointer is at end)
              then:
                Call CUR_LEFT
                WSP := WSP - 1
              else:
                Call LIST_LOG_STRING
                *Put out a blank character to the screen*
                If (LINE_LENGTH = MAX_LENGTH)
                  then:
                    *Trim output string to its length minus one*
                End If:
                *Put out the output string*
                *Set Cursor 2 to the WSP value*
            End If:
        End If:
      else:
        **(ie. Insert Mode is OFF)**
        *Write the input key to the screen*
        WSP := WSP + 1
        If (LINE_LENGTH = 0)
          then:
            *Get a record from the space list*
            LINE_LENGTH := LINE_LENGTH + 1
        End If:
        *Write the input key in the list*
        *Move list pointer forward by one position*
        If (Status implies a Pointer-Outside-List error)
          then:
            If (LINE_LENGTH >= MAX_LENGTH)
              then:
                *Put out an error -- overtyping last character*
              else:
                *Get a record from the space list*
                *Write a blank character in the list*
                LINE_LENGTH := LINE_LENGTH + 1
            End If:
        End If:
    End If:
End:
End Procedure:
```

Fig.4.8: The PDL for the Line Editor's Enter Text routine

Insert

This routine is used to toggle Character Insert mode ON or OFF. Character Insert mode is used for inserting a character before the character pointed to by Cursor 2. Under normal circumstances,

this insert mode will produce a blank insert token under the cursor as a visual reminder. This mode cannot, however, be entered if the line is full. The routine call is described below, while Fig.4.9 shows its PDL description.

```
INSERT (VAR INSERT_STATUS: BOOLEAN;
        VAR LINE_LENGTH: INTEGER;
            MAX_LENGTH: INTEGER;
            LLP: INTEGER;
        VAR WSP: INTEGER)
```

```
Procedure  Insert
Variables:
     Boolean:
        Single:
           Local:
              POSSIBLE
Begin:
  If (Insert Status is ON)
    then:
      Call LIST_LOG_STRING
      *Write output string from above operation*
      If (LINE_LENGTH <> MAX_LENGTH)
        then:
          Call CLR_LINE
      End If:
      If (WSP is at the end of the file)
        then:
          WSP := WSP - 1
      End If:
      *Set Cursor 2 to the WSP value*
      *Move list pointer forward by one position*
      *Set Insert Status OFF*
    else:
      POSSIBLE := TRUE
      If (LINE_LENGTH = MAX_LENGTH)
        then:
          *Check if last character is a blank*
          If (Last character is a blank)
            then:
              *Put out an error -- line is full*
              POSSIBLE := FALSE
            else:
              *Return the last character to the space list*
              LINE_LENGTH := LINE_LENGTH - 1
          End If:
      End If:
      If (POSSIBLE = TRUE)
        then:
          Call LIST_LOG_STRING
          *Put out output string from above operation*
          *Move list pointer back by one position*
          *Set Insert Status ON*
      End If:
  End If:
End:
End Procedure:
```

Fig.4.9: The PDL for the Line Editor's Insert routine

Delete

This routine will delete the character immediately to the left of the cursor whether in Insert mode or otherwise; the remaining text string to the right of the cursor moving to the left by a single character position. Deleting beyond the beginning of the user-editable text line is not permitted. Below, the routine call is shown, while the PDL for the Delete routine is described in Fig.4.10.

```
DELETE (     INSERT_STATUS: BOOLEAN;
         VAR LINE_LENGTH: INTEGER;
             LLP: INTEGER;
         VAR WSP: INTEGER;
         VAR STATUS: INTEGER)

Procedure  Delete
Begin:
  If (Insert Status is ON)
    then:
      If (List Pointer is at zero position)
        then:
          *Put  out an error -- deleting beyond start of line not
           allowed*
        else:
          Call LIST_LOG_STRING
          Call CUR_LEFT
          *Write a blank character to the screen*
          *Write out the string obtained above*
          If (LINE_LENGTH <> MAX_LENGTH)
            then:
              Call CLR_LINE
          End If:
          WSP := WSP + 1
          *Set Cursor 2 to the WSP value*
          *Return the record to the space list*
          *LINE_LENGTH := LINE_LENGTH - 1
      End If:
    else:
      **(ie. Insert Status is OFF)**
      If (List pointer is at the start of the line)
        then:
          *Put out an error -- deleting beyond start of line  not
           allowed*
        else:
          Call LIST_LOG_STRING
          Call CUR_LEFT
          *Write out the string obtained above*
          Call CLR_LINE
          WSP := WSP - 1
          *Set Cursor 2 to WSP value*
          *Move list pointer backward by one position*
          *Return the record to the space list*
          *Move list pointer forward by one position*
          LINE_LENGTH := LINE_LENGTH - 1
      End If:
  End If:
End:
End Procedure;
```

Fig.4.10: The PDL for Line Editor's Delete routine

Erase End of Line

This routine is responsible for erasing all characters from the current cursor position to the end of the line. Fig.4.11 gives the PDL for this routine while the calling procedure is shown below.

```
ERASE_EOL (     INSERT_STATUS: BOOLEAN;
           VAR LINE_LENGTH: INTEGER;
               LLP: INTEGER)
```

```
Procedure  Erase End of Line
Begin:
  While (List pointer is less than LINE_LENGTH) Do:
    *Move list pointer forward by one position*
    *Return the character at the list pointer to the space list*
    LINE_LENGTH := LINE_LENGTH - 1
  End While:
  If (Insert Status is OFF)
    then:
      *Write a blank character at the end of the line*
  End If:
  Call CLR_LINE
End:
End Procedure:
```

Fig.4.11: The PDL for the Erase End of Line routine

Dump

This routine deals with the outputting of the OUT_STRING variable. This string is obtained by reading the Line Linked List in order of ascending forward pointer values. The LIST_LOG_STRING routine is useful for this purpose (see section 3.5 2). Any blank characters at the end of the text line are ignored. The routine call is shown below, while Fig.4.12 gives the PDL description of the Dump routine.

```
DUMP (     MAX_LENGTH: INTEGER;
      VAR OUT_STRING: STRING (80) OF CHAR)
```

```
Procedure  Dump
Begin:
  Call LIST_LOG_STRING
  *Trim  the output string of any blank characters at the end  of
   the line*
End:
End Procedure:
```

Fig.4.12: The PDL for the Line Editor's Dump routine

4.2 Formatter

4.2.1 Process function

The Formatter process is a short routine which is used to format a PDL program, line by line. This routine provides a method for converting the coded version of a PDL file into its logical equivalent. By accessing a line in the coded file (which is resident in the File Linked List resource), the Formatter process is able to convert the various codes into a text line comprising three distinct text strings. These three strings form the output of the Formatter routine. The output strings, when strung together sequentially, will form the required text line.

The three output strings thus represent the first, second and last parts of the prettyprinted line. The output was chosen in this form to make the formatter a more generally usable routine. On the Main Screen for example, the three strings represent the first highlighted system-generated key word; the unhighlighted user-entered text; and the second system-generated highlighted key word.

The output routine which writes the file to disk in formatted form (see section 5.2) also uses the Formatter process, but does not make any distinction between the three strings (highlighted or not). Here instead, the strings are combined into a single line without distinguishing between key words. It will be apparent that not all lines will contain all three of these output strings, but three strings are supplied for completeness.

4.2.2 Process structure

A high-level PDL description of the Formatter process operation is given in Fig.4.13. Its operation is described below.

```
Process Formatter
Begin:
  *Read the File Linked List at the current pointer position*
  *Assign the user-editable text string to the second output
   string*
  *Read the Key Code Table for the first key code*
  *Calculate the line's absolute indentation*
  *String the absolute indentation space and the key word
   together and assign to the first output string*
  *Read the Key Code Table for the second key code*
  *Assign the second key word to the third output string*
End:
End Process:
```

Fig.4.13: The Formatter routine

Firstly, the File Linked List is read corresponding to the current pointer position. Thus, the line which is to be formatted will be the line referenced by the File Linked List pointer on

entering the Formatter routine. From the File Linked List, the indentation level; the first and second key codes; and the user-editable text string are obtained.

Next, the Key Code Table is read, using the first key code to determine its associated key word and relative indentation. The indentation level, together with the relative indentation obtained from the first key code, is then used to calculate the line's absolute indentation measured from the left hand margin. It should be noted here, that only the first key code is used to determine indentation. The relative indentation provided by the second key code is meaningless, and thus not used.

The absolute indentation space is strung before the first key word to form the first output string. The second output string merely consists of the user-editable text string read from the File Linked List. The Key Code Table is then again accessed to determine the second key word from the second key code. This key word forms the third output string.

The Formatter routine has been structured in such a way that it is easily usable. This routine is used wherever a formatted version of the coded file is required. This includes prettyprinting for the Main Screen, as well as for the formatted output file. The three strings are used so as to be able to display the formatted line on Screen 1 in its appropriate mixed font style.

4.3 System Base Level

4.3.1 Process function

The Base Level is the system's foundation level from where all its functions are accessible. In this level, the user is given the ability to move around in the file via the scrolling functions such as cursor up and down; page forward and backward; top and bottom of file; etc.. It should be noted, however, that the cursor movements and page scroll functions are also available in most other modes. Also, the system is designed to scroll correctly for a Main Screen with an even number of lines only.

Another important function of the Base Level is that of providing a gateway into the system's sub-levels. Insert, Delete, Copy and Move modes will all be accessible from Base Level. The construction of this process is such that any number of functions can be added by making use of the available function keys. A gateway into a sub-level is easily achieved by performing a call to the new level's procedure whenever the relevant function key is depressed.

As this process is somewhat specialized to suit the PDL generator package, further details of the program structure can be found in section 4.3.2, while section 4.3.3 describes the scrolling routines and the gateway into Insert Level in more detail.

4.3.2 Process structure

The Base Level process is built in modular form and can be addressed using the following convention:

```
FILE_SCROLL (    TOP_STATE: INTEGER;
                 NO_OF_STATES: INTEGER;
                 KCT_SIZE: INTEGER;
                 PT_SIZE: INTEGER;
            VAR MS_CUR_POS: INTEGER;
            VAR MS_TOP_LINE: INTEGER;
            VAR FILE_BOT_LINE: INTEGER)
```

Where the inputs are:

```
TOP_STATE     -- The state from which Base Level operates.
NO_OF_STATES -- The number of states in the Definition Table.
KCT_SIZE      -- The number of key words in the Key Code Table.
PT_SIZE       -- The number of prompt lines in the Prompt Table.
```

and the outputs:

```
MS_CUR_POS     -- (Main Screen CURsor POSition) the line number of
                  the Main Screen (0 = top line, 19 = 20th line)
                  on which the cursor is positioned.
MS_TOP_LINE    -- (Main Screen TOP LINE) the actual number of the
                  line displayed at the top of the Main Screen.
FILE_BOT_LINE -- (FILE BOTtom LINE) the actual number of the last
                  line in the file.
```

These last three variables can be used exclusively to keep track of any cursor movements or screen manipulation. The position of Cursor 1 is at all times determined by MS_CUR_POS, while the line number which it is on can be found by (MS_TOP_LINE + MS_CUR_POS). The FILE_BOT_LINE is used to monitor the length of the file.

The primary scrolling functions in Base Level will require the use of the following resources:

```
Terminal Resource -- for screen management reasons
File Linked List  -- for reading and pointer manipulation
Key Code Table    -- for displaying the key codes on Screen 1
Formatter Process -- for formatting prior to displaying on Screen 1
```

Due to the presence of the Line Edit facility as well as the various gateways in Base Level, it is necessary that this level involve the use of every resource and process available.

Besides the resources, the following procedures are employed in the Base Level:

```
CUR_UP        -- moves Cursor 1 up the screen
CUR_DOWN      -- moves Cursor 1 down the screen
PAGE_BWD      -- scrolls backward by a page
PAGE_FWD      -- scrolls forward by a page
TOP_OF_FILE   -- displays the top of the file
BOT_OF_FILE   -- displays the bottom of the file
CUR_TO_LINE   -- displays the requested file line
EDIT_LINE     -- used for modifying a single line
INSERT        -- the Insert gateway
```

These procedures are described in greater detail in section 4.3.3. It will be noted that the gateway into the Insert facility is provided by the INSERT procedure. This procedure will then in turn act as an entry point into the Insert sub-routines.

The PDL structure of the Base Level's main program is shown in Fig.4.14. Referring to this figure, it can be seen that an A Read operation is initially performed on the Definition Table, with the TOP_STATE as input. This is done so that the Base Level's function key set can be displayed in the Key Definition Screen. A repeat-until loop is used, so that only if the user depresses the ESC key will he exit the Base Level (and indeed the PDL editor).

```
Procedure  Base Level
Begin:
  Call DT_A_READ
  Call FS_WRITE
  Repeat:
    Call KBD_GET
    Case (ASCII key code) Of:
        Special keys: KBD_GET
                      If (ASCII key implies a valid function key)
                        then:
                          Case (Function key) Of:
                             1: Call PAGE_BWD
                             2: Call PAGE_FWD
                             3: Call TOP_OF_FILE
                             4: Call BOT_OF_FILE
                             5: Call CUR_TO_LINE
                             6: Call EDIT_LINE
                             7: Call DT_B_READ
                                Call INSERT
                          End Case:
                        else:
                          If (ASCII key implies any used key)
                            then:
                              Case (Used key) Of:
                                Cursor up  : Call CUR_UP
                                Cursor down: Call CUR_DOWN
                                Page up    : Call PAGE_BWD
                                Page down  : Call PAGE_FWD
                              End Case:
                          End If:
                      End If:
    End Case:
  Until (ASCII code implies ESCAPE)
End:
End Procedure:
```

Fig.4.14: The PDL structure for Base Level

The Case construct which is embedded within the loop forms a standard structure for multiple function-key driven operations. This method, although not completely externally programmable, does have large scope for expandability. Extra functions can merely be inserted as a separate Case option in the form of a procedure call. Thus, the program will remain simple in basic structure.

The gateway facility can easily be identified by the multiple statements used for the function key 7 option. Here, the Definition Table is again accessed using the B Read operator so that the next state can be obtained. It is this next state which is used as the top state for the INSERT procedure. Details of this procedure will be given in section 4.4.

The structure of Fig.4.14 also brings out the possibility of using special keys on the keyboard (eg. the cursor control keys or the Pg Dn and Pg Up keys) to execute an associated routine. Thus, special-purpose function keys can easily be added. A constant is assigned to the ASCII codes generated when a key is depressed so as to make the system as portable as possible. The system designer need only change the constant values at the beginning of the program to suit the new keyboard, resting assured that the program will run as expected without further intervention.

A more general structure for being able to trap any key or key combination of the keyboard is given in Fig.4.15. It is assumed that function keys and special keys (including Cntrl keys) put out two ASCII codes when depressed.

```
Call KBD_GET
Case (ASCII code) of:
  Groupi special keys: Call KBD_GET
                       Case (ASCII code) of:
                         Groupl fx keys: Case (fx key) of:
                                           1: ...
                                           2: ...
                                           else:
                                             *do nothing*
                                             **ie. fx key
                                               invalid**
                                         End Case:
                                  else:
                                         Case (ASCII key) of:
                                           Cursor up  : ...
                                           Cursor down: ...
                                                ...
                                                  else:
                                                    *do nothing*
                                             **ie.no other keys
                                               are valid**
                                         End Case:
                     End Case:
  Group2 special keys:
                ...
                ...
                ...
  Groupn special keys:
                           ...
                           ...
                else:
                     *text is entered*
End Case:
```

Fig.4.15: The PDL structure for using any key or key combination

4.3.3 Process routines

Cursor_Up

This routine is responsible for moving the cursor in Screen up by one line. This function is assigned to the Cursor Up key. The procedure call can be seen below.

```
CUR_UP (VAR MS_CUR_POS: INTEGER;
        VAR MS_TOP_LINE: INTEGER;
            KCT_SIZE: INTEGER)
```

The variables MS_CUR_POS and MS_TOP_LINE are used entirely for the management of the Main Screen. As explained previously, given the top line of the Main Screen and an absolute cursor position, all other unknowns are easily calculated. The tasks which need to be performed in this routine include the moving (or effective switching off and on) of the cursor (Cursor 1) on the Main Screen; the checking for scrolling conditions; and the checking for an error.

```
Procedure Cursor Up
Begin:
  If (Cursor 1 is not at the top of Screen 1)
    then:
      *Move File Linked List Pointer Back by one line*
      *Turn the old cursor OFF*
      MS_CUR_POS := MS_CUR_POS - 1
      *Turn the new cursor ON*
    else:
      **(ie. Cursor 1 is at the top of the screen)**
      If (MS_TOP_LINE = 1)
        then:
          *Put out an error message -- top of file*
        else:
          If (An exact half screen scroll is possible)
            then:
              *Decrease MS_TOP_LINE by a half screen*
              *Set MS_CUR_POS to the centre of Screen 1*
            else:
              **(ie. An exact half screen scroll is not possible)**
              *Set MS_CUR_POS to its appropriate value*
              *Set MS_TOP_LINE to 1*
          End If:
          *Rewrite the entire Main Screen*
          *Turn the new cursor ON*
      End If:
  End If:
End:
End Procedure:
```

Fig.4.16: The PDL for the Cursor Up routine

An error will occur if the user tries to move the cursor above the file's top line. Page scrolling is needed when the cursor reaches the top of the logical Main Screen, but not the top of the file. A half page backward scroll is performed when scrolling

is required so that the cursor is positioned in the centre line of Screen 1. This centre line is taken as the (MS_SIZE / 2)th line. A skeleton PDL structure of this routine is given in Fig.4.16. The various condition checks are clearly depicted.

The comment "*Rewrite the entire Main Screen*" involves moving the pointer in the File Linked List to access the required 20 lines. Each line is formatted and written to the Main Screen individually. This routine is used often when either part of or the entire Main Screen is to be updated. Thus, Fig.4.17 is included to demonstrate the steps involved.

```
*Move File Linked List pointer to the first line to be displayed*
*Initialize the loop counter*
Repeat:
  Call FORMATTER
  Call MS_WRITE
  *Move the File Linked List pointer forward by 1*
  *Increment the loop counter*
Until (The correct number of lines have been updated on Screen 1)
*Move File Linked List pointer back to the new cursor position*
```

Fig.4.17: PDL for updating the Main Screen

```
Cursor Down
Procedure Cursor Down
Begin:
  If ((The cursor is currently not on the last line) and
      (Scrolling is not required))
    then:
      *Move File Linked List pointer forward by 1*
      *Turn the old cursor OFF*
      MS_CUR_POS := MS_CUR_POS + 1
      *Turn the new cursor ON*
    else:
      If (The cursor is on the last line)
        then:
          *Put out an error message -- bottom of file*
        else:
          **(ie. Scrolling must be performed)**
          If (An exact half screen scroll is possible)
            then:
              *Increase MS_TOP_LINE by a half screen*
              *Set MS_CUR_POS to the centre of Screen 1*
            else:
              *Set MS_CUR_POS to its appropriate value*
              *Set MS_TOP_LINE to one screen less than the
               FILE_BOT_LINE*
          End If:
          *Rewrite the entire Main Screen*
          *Turn the new cursor ON*
      End If:
  End If:
End:
End Procedure:
```

Fig.4.18: The PDL for the Cursor Down routine

This routine moves Cursor 1 on the Main Screen down by one line. This function is assigned to the Cursor Down key. The procedure call can be described in Pascal as follows:

```
CUR_DOWN (VAR MS_CUR_POS: INTEGER;
          VAR MS_TOP_LINE: INTEGER;
              FILE_BOT_LINE: INTEGER;
              KCT_SIZE: INTEGER)
```

The usual variables are used, however, the FILE_BOT_LINE input is the variable (globally accessible) used to denote the last line of the file.

Again, checking must be performed for scrolling and error conditions. An error is detected when the user attempts to move the cursor beyond the file's last line. A half page forward scroll is performed when the cursor is moved beyond the last line of the logical Main Screen. A scroll will position the cursor (from the top of the physical screen) at the ((MS_SIZE / 2) + 1)th line. The skeleton structure for this routine is shown in Fig.4.18.

Page Backward

```
Procedure  Page Backward
Begin:
  If (MS_TOP_LINE = 1)
    then:
      If (MS_CUR_POS = 0)
        then:
          *Put out an error -- Top of file*
        else:
          *Move file pointer to top of file*
          *Turn old cursor OFF*
          MS_CUR_POS = 0
          *Turn new cursor ON*
      End If:
    else:
      **(ie. scrolling is required)**
      If (exact page scroll is possible)
        then:
          *Move file pointer back by one page minus one*
          *Decrement MS_TOP_LINE by above amount*
        else:
          If (Cursor 1 can be moved without overshooting)
            then:
              *Set MS_CUR_POS to its appropriate value*
            else:
              MS_CUR_POS := 1
          End If:
      End If:
      *Rewrite entire Main Screen*
      *Turn new cursor ON*
  End If:
End:
End Procedure:
```

Fig.4.19: The PDL for the Page Backward routine

This routine will make every attempt to perform a full page scroll backward. If possible, MS_TOP_LINE is decreased by a full screen size minus one while MS_CUR_POS remains unchanged. This means that the first line of the page displayed on Screen 1 before scrolling will become the last line of the page displayed after scrolling. If the screen's top line cannot be decremented by the complete ideal amount, then the cursor is moved the equivalent of one screen backward. If no decrement of MS_TOP_LINE is possible, then the cursor is moved to the top line of the file and screen.

This procedure, which is also assigned to the Pg Up key, can be invoked using the following:

```
PAGE_BWD (VAR MS_CUR_POS: INTEGER;
          VAR MS_TOP_LINE: INTEGER;
              FILE_BOT_LINE: INTEGER;
              SCT_SIZE: INTEGER)
```

The usual global parameters are passed for the sake of being explicit. Besides the tests needed as mentioned above, an error will occur if the user attempts a backward scroll when Cursor 1 is positioned at the top of the file. The skeleton PDL structure is given in Fig.4.19.

```
Page Forward
Procedure  Page Forward
Begin:
  If (No scrolling is required)
    then:
      If (Cursor 1 is already at the bottom of the file)
        then:
          *Put out an error -- Bottom of file*
        else:
          *Turn old cursor OFF*
          *Set MS_CUR_POS to appropriate bottom line*
          *Turn new cursor ON*
          *Move File pointer to end of file*
      End If:
    else:
      If (Exact page scroll is possible)
        then:
          *Move file pointer forward by a page minus one*
          *Increment MS_TOP_LINE by the same amount*
        else:
          If (Cursor 1 can be moved without overshoot)
            then:
              *Set MS_CUR_POS to its correct value*
            else:
              *Set  MS_TOP_LINE such that file bottom line is  at
                end of Main Screen*
          End If:
      End If:
      *Rewrite entire Main Screen*
      *Turn new cursor ON*
  End If:
End:
End Procedure:
```

Fig.4.20: The PDL for the Page Forward routine

This routine will make every attempt to perform a full page scroll backward. If possible, MS_TOP_LINE is decreased by a full screen size minus one while MS_CUR_POS remains unchanged. This means that the first line of the page displayed on Screen 1 before scrolling will become the last line of the page displayed after scrolling. If the screen's top line cannot be decremented by the complete ideal amount, then the cursor is moved the equivalent of one screen backward. If no decrement of MS_TOP_LINE is possible, then the cursor is moved to the top line of the file and screen.

This procedure, which is also assigned to the Pg Up key, can be invoked using the following:

```
PAGE_BWD (VAR MS_CUR_POS: INTEGER;
          VAR MS_TOP_LINE: INTEGER;
              FILE_BOT_LINE: INTEGER;
              KCT_SIZE: INTEGER)
```

The usual global parameters are passed for the sake of being explicit. Besides the tests needed as mentioned above, an error will occur if the user attempts a backward scroll when Cursor 1 is positioned at the top of the file. The skeleton PDL structure is given in Fig.4.19.

```
Page Forward
Procedure  Page Forward
Begin:
  If (No scrolling is required)
    then:
      If (Cursor 1 is already at the bottom of the file)
        then:
          *Put out an error -- Bottom of file*
        else:
          *Turn old cursor OFF*
          *Set MS_CUR_POS to appropriate bottom line*
          *Turn new cursor ON*
          *Move File pointer to end of file*
      End If:
    else:
      If (Exact page scroll is possible)
        then:
          *Move file pointer forward by a page minus one*
          *Increment MS_TOP_LINE by the same amount*
        else:
          If (Cursor 1 can be moved without overshoot)
            then:
              *Set MS_CUR_POS to its correct value*
            else:
              *Set  MS                such that file bottom line is  at
               end of                 n*
          End If:
      End If:
      *Rewrite entire Main screen*
      *Turn new cursor ON*
  End If:
End:
End Procedure:
```

Fig.4.20: The PDL for the Page Forward routine

Similarly to the Page Backward routine, this routine attempts to scroll the file on Screen 1 forward by a page minus one. This means that the bottom line before scrolling becomes the top line after scrolling. All the possibilities described in the PAGE_BWD routine above are accounted for, with an error occuring if a forward scroll is attempted with Cursor 1 on the last line.

This procedure is assigned to a function key in Base Level, and also to the permanently available Pg Dn key on the keyboard. This routine has the following parameters:

```
PAGE_FWD (VAR MS_CUR_POS: INTEGER;
          VAR MS_TOP_LINE: INTEGER
              FILE_BOT_LINE: INTEGER;
              KCT_SIZE: INTEGER)
```

The parameters are again the usual global parameters and Fig.4.20 shows the structure of the routine in PDL form.

Top of File

This routine moves the cursor to the top line of the file. Cnecking is performed to determine if scrolling is needed and an error will be detected if this function is attempted with the cursor already on the top line. A better idea of the nature of the procedure can be obtained by studying the PDL description of Fig.4.21. The routine name and parameters are as follows.

```
TOP_OF_FILE (VAR MS_CUR_POS: INTEGER;
             VAR MS_TOP_LINE: INTEGER;
                 FILE_BOT_LINE: INTEGER;
                 KCT_SIZE: INTEGER)
```

```
Procedure  Top of File
Begin:
  If (MS_TOP_LINE = 1)
    then:
      If (MS_CUR_POS = 1)
        then:
          *Put out an error -- Top of file*
        else:
          *Move File Linked List pointer to top of file*
          *Turn old cursor OFF*
          MS_CUR_POS := 0
          *Turn new cursor ON*
      End If:
    else:
      **(ie. scrolling is required)**
      *Rewrite the entire Main Screen*
      MS_CUR_POS := 0
      MS_TOP_LINE := 1
      *Turn new cursor ON*
  End If:
End:
End Procedure:
```

Fig.4.21: The PDL for the Top of File routine

Bottom of File

Similar to the Top of File routine, this procedure will move the cursor to the last line of the file. Again, checking is done to determine whether scrolling is required or not. An error will occur if this function is attempted when Cursor 1 is on the last line of the file. The routine's parameters are shown below; the PDL structure of this procedure is given in Fig.4.22.

```
BOT_OF_FILE (VAR MS_CUR_POS: INTEGER;
             VAR MS_TOP_LINE: INTEGER;
                 FILE_BOT_LINE: INTEGER;
                 KCT_SIZE: INTEGER)
```

```
Procedure  Bottom of File
Begin:
  If (Scrolling is not required)
    then:
      If (Cursor 1 is already at the bottom of file)
        then:
          *Put out error message -- Bottom of file*
        else:
          *Turn old cursor OFF*
          *Set MS_CUR_POS to new value*
          *Turn new cursor ON*
          *Move File Linked List pointer to end of list*
      End If:
    else:
      **(ie. scrolling is required)**
      *Rewrite the entire Main Screen*
      *Turn new cursor ON*
  End If:
End:
End Procedure:
```

Fig.4.22: The PDL for the Bottom of File routine

Cursor to Line

This function allows the user to choose a line number to which Cursor 1 is to be moved. The line number is entered in the Window Screen via the Line Editor. An illegal line number is not accepted and an error message is displayed in the Prompt Screen. Only an integer number of four digits or less which is between 1 and FILE_BOT_LINE will be accepted.

If the line number requested appears in the present Screen 1 display, the cursor is merely moved to this line. If the line number is not on Screen 1, scrolling occurs, with all attempts being made to place the cursor with the required line in the centre of the screen. In this routine, the centre of the screen is taken as the (MS_SIZE / 2)th line. The procedure is defined as shown below, with Fig.4.23 giving the structure of the Cursor-to-line routine in PDL.

```
CUR_TO_LINE (VAR MS_CUR_POS: INTEGER;
             VAR MS_TOP_LINE: INTEGER;
                 FILE_BOT_LINE: INTEGER;
                 KCT_SIZE: INTEGER)
```

```
Procedure  Cursor to Line
Begin:
  *Prompt user for input line*
  *Set inputs for Line Edito. *
  *Set WSP to start of line*
  *Set all function key flags to FALSE*
  *Set length of editor to allow only a 4 digit number*
  Call LINE_EDITOR
  Call WS_CLEAR
  Call PS_CLEAR
  If (Input line number is not valid)
    then:
      *Put out an error*
    else:
      **(ie. line number is valid)**
      If (Scrolling is not required)
        then:
          *Move pointer to correct value on Main Screen*
          *Turn old cursor OFF*
          *Set MS_CUR_POS to new value*
        else:
          **(ie. scrolling is required)**
          If (Requested line too near top of file to be placed in
              centre of screen)
            then:
              *Move file pointer to requested line number*
              MS_TOP_LINE := 1
              *Set MS_CUR_POS to value indicated by line number*
            else:
              If (Requested  line too near bottom of file  to  be
                  placed in centre of screen*
                then:
                  *Move pointer to line number*
                  *Set MS_TOP_LINE so that file bottom line is at
                   end of Main Screen*
                  *Set  MS_CUR_POS  to  value indicated  by  line
                   number*
                else:
                  **(ie.  Requested line can be located at centre
                     of Main Screen)**
                  *Move file pointer to line number*
                  *Set  MS_TOP_LINE such that line number  is  in
                   centre of Main Screen*
                  *Set  MS_CUR_POS  to  half of the  Main  Screen
                   size*
              End If:
          End If:
          *Rewrite entire Main Screen*
      End If:
      *Turn new cursor ON*
  End If:
End:
End Procedure:
```

Fig.4.23: The PDL for the Cursor To Line routine

It will be noted that the Line Editor package is used initially. The length of the line is set so as to restrict the user to a maximum of four digits. The highlighted prompt which is displayed

in the Window Screen ("Line Number =") is not editable.

Also worth noting is the task involved in displaying the error message "Line Number .... is out of range. Acceptable range = 1 to FILE_BOT_LINE." with the FILE_BOT_LINE and the "...." being replaced by their appropriate values. Clearly, integer to string conversions and string concatenation facilities are needed to assemble this type of message in the form of a single string.

Edit_Line

This function allows the user to pick any line from the file with Cursor 1 and edit it, if possible, with the Line Editor in the Window Screen. Any line containing a key word only will not be editable and an error message will be output in the Prompt Screen. The user is able to edit the user-entered text or a placeholder in the Window Screen, without affecting the system-generated highlighted keywords. The accepted modified line is ENTERed and will replace the old line in Screen 1. If the ESC key is used, the line will remain as before and the Line Editor exit.

As no modification to the Screen 1 cursor position is performed in this routine, the editor routine is invoked by the statement:-

EDIT_LINE

```
Procedure Edit Line
Begin:
  *Read File Linked List at current pointer position*
  *Read the Key Code Table using the first key code obtained from
    above*
  If (Editing is allowed)
    then:
      *Initialize settings for the Line Editor*
      Call LINE_EDITOR
      If (The new line differs from the old line)
        then:
          *Write the new text line in place of the old text line
           in the File Linked List*
          Call FORMATTER
          Call MS_WRITE
          *Turn the cursor back ON*
      End If:
    else:
      *Put out an error message -- line not editable*
  End If:
End:
End Procedure:
```

Fig.4.24: The PDL for the Edit Line routine

Fig.4.24 shows a PDL structure of the procedure. This routine is virtually self explanatory, but the Line Editor inputs can perhaps be elaborated on. The user is given 80 columns of line to edit (Inclusive of the key word). Only the first key word will be

displayed. The key word is obtained by reading the Key Code Table, while the editable text line is taken directly from the File Line List. All function key flags are set to FALSE so that only the ENTER and ESC keys will exit the Line Editor.

Insert_Gateway

See the main program in the next section (Section 4.4.2).

4.4 Insert Mode

4.4.1 Process function

Insert mode is used for the insertion of one or more lines of PDL into the PDL file. This process deals with data insertion; construct or block insertion and line insertion. Data items are inserted by entering a specialized Data Description mode where, via the use of function keys, a data item can be quickly defined, and automatically positioned in the program's Data Description segment. Using this method the system provides a friendly user interface, while constantly defining each entered data item for future semantic error checking. The automatic placement of the data item in the program with its required key words and indentation is also of great help to the user.

The Construct or Block Insert function makes use of templates. A template with associated placeholders will be inserted in the PDL program below the current cursor position, with all indentation considerations accounted for. This method of program generation ensures syntactical correctness as all constructs are terminated in the correct manner due to the templates used.

The Line Insert routine will allow multiple lines to be inserted at certain permissible points in the PDL program. Again, indentation is automatic, and insertion occurs after Cursor 1. This function can be used to expand a placeholder or add statement or comment lines. The Data Description segment of the PDL program may not be accessed via this function and the user will be warned if line insertion is not permitted.

The structure of Insert mode is such that a number of insert routines (besides the ones listed above) can be added modularly to the package by simply expanding the number of valid function keys in this mode. Cursor 1 can be moved up and down in Insert mode with the use of the cursor control keys. The Pg Up and Pg Dn keys for viewing the file by pages, are also operational.

4.4.2 Process Structure

The Insert process can be called using:

```
INSERT (TOP_STATE: INTEGER)
```

The TOP_STATE variable represents the state or mode from which

any type of insertion can be chosen. This input is passed from the calling program as the NEXT_STATE which is obtained from the Definition Table when the Insert function key is depressed in Base Level. A PDL description of the basic Insert program is given in Fig.4.25. The familiar Case structure (also found in Base Level) is again apparent here. This gives the designer greater flexibility in the adding to or modifying system operation.

```
Procedure  Insert
Begin:
  Repeat:
    Case (Input Key) Of:
      special keys: Call KBD_GET
                    If (Key implies a function key)
                      then:
                        If (Function key is valid)
                          then:
                            Call DT_B_READ
                            Call PT_READ
                            *Write prompt to the Prompt Screen*
                            Case (Function key) Of:
                               1: Call DATA_DESCRIPTION
                               2: Call ALGORITHM
                              10: Call DT_A_READ
                                  Call FS_WRITE
                            End Case:
                        End If:
                      else:
                        Case (Key) Of:
                          Cur Up: Call CUR_UP
                          Cur Dn: Call CUR_DOWN
                          Pg Up : Call PAGE_BWD
                          Pg Dn : Call PAGE_FWD
                            else:
                                *Do nothing*
                        End Case:
                    End If:
              else:
                    *Do nothing*
    End Case:
  Until (Key Code implies RETURN)
End:
End Procedure:
```

Fig.4.25: The PDL for the Insert mode routine

The procedure is similar to Base Level: an A Read operation is performed on the Definition Table with the TOP_STATE as reference so as to obtain the function key definitions. Once a valid function key has been depressed, a B Read operation is performed on the Definition Table. From the codes obtained, a prompt is displayed and the next state is obtained for use by the subsequent routines.

The Case option will then call the routine corresponding to the function key number (1 to 10) which is depressed. It can be seen

that only one routine call is needed each for data item insertion, construct insertion and line insertion. The variables passed to these routines include the globals: NO_OF_STATES, KCT_SIZE and PT_SIZE; the Main Screen management variables: MS_CUR_POS, MS_TOP_LINE and FILE_BOT_LINE; and the NEXT_STATE obtained above.

Only the RETURN function key (which is always assigned to function key 10) does not make use of a separate routine. Instead, an A Read operation is performed on the NEXT_STATE, thus returning to the previous state and displaying a set of function keys in Screen 4. This is also the only key (having a key code of 4000) which will terminate the program's Repeat-until loop.

The next section (section 4.4.3) describes more clearly the implementation details involved in the design of the individual insert routines.

4.4.3 Process routines

Data Description segment routine

The defining routine name, together with its parameters is shown below:

```
DATA_DESCRIPTION (    TOP_STATE: INTEGER;
                      N_STATE: INTEGER;
                      NO_OF_STATES: INTEGER;
                      KCT_SIZE: INTEGER;
                      PT_SIZE: INTEGER;
                  VAR MS_CUR_POS: INTEGER;
                  VAR MS_TOP_LINE: INTEGER;
                  VAR FILE_BOT_LINE: INTEGER)
```

For editing of the data item definition which is being entered, a temporary data structure is necessary. The data structure adopted is as follows:

```
Type:
-----
      DD_RECORD = Record:
                    KC: Integer                    *Key Code*
                    KW: String (10) of Character   *Key Word*
                  End Record;

      D_ARRAY = Array [1..4] of DD_RECORD

Variables:
----------
    D_ARRAY:
        Single:
            Local:
                DD_ARRAY
    Character:
        Array:
            Local:
                DI_STRING (string of 80)
```

The DI_STRING contains the user-entered data item. The DD_ARRAY represents the key codes (KC) and key words (KW) of each of the four data description fields (function, type, structure and scope) used to define the data item. Both key codes and key words are stored because of the convenience of not having to access the key code table continuously to translate the key codes.

The resources used by this process involve:

Terminal Resource	-- for the specialized formatting of the Window Screen as well as for displaying inserted data items on the Main Screen
File Linked List	-- for placement and positioning of the defined data item in the PDL file
Definition Table	-- for determining the next state, key code and prompt codes associated with any function key depression
Key Code Table	-- used to find the key word and indentation corresponding to the defining key code for visual display in Screens 1 and 2.
Prompt Table	-- used to translate the prompt code into a prompt which can be written to the Prompt Screen when required
Formatter	-- used for the visual formatting of the newly inserted data structure in Screen 1
Line Editor	-- for editing of the user-entered data item name in the Window Screen

Fig.4.26 gives a high-level PDL description of the Data Description Insert algorithm. This routine makes extensive use of the Definition table for determining the key codes, next states, and prompt codes associated with any function key. In Fig.4.26 a little detail is shown for the mechanism which moves the user through all the Data Description levels. The function of some of the variables used is given below:

TOP_STATE -- the Insert mode state
THIS_STATE -- the present state
NEXT_STATE -- the next state (obtained from the Definition Table)
PREV_LEVEL -- the previous level of data item definition (ie. function=1, type=2, structure=3, scope=4)

The outermost While-do loop will enable the user to remain in this routine until the NEXT_STATE equals the TOP_STATE. Within the loop, if the NEXT_STATE is identical to the present state, then no system action is required. Otherwise, if a new field is defined, the previous field is written in highlighted font, while the next field is highlighted in reverse video font. Also, if the next and present states differ, then a new function key definition line must be written to Screen 4.

The "*Test for an error*" procedure almost exclusively refers to the error which occurs when the user tries to CONTINUE without having defined the present field. In this error case, the second next state and prompt code values obtained in the Definition Table are used instead of the first.

After the top state has been returned to, the key code is tested. If abandonment of the defined data item is implied, no system

action occurs (the user is returned to Insert mode as if Data Definition Insert mode had never been entered). If, on the other hand, the defined data item is to be accepted, then the Placement routine is called. This routine will insert the user-entered data item together with any relevant key words into the PDL file automatically.

```
Procedure Data Description Insertion
Begin:
  *Initialize data structure*
  THIS_STATE := TOP_STATE
  PREV_LEVEL := 1
  *Initialize the key code to a non-existant value*
  While (NEXT_STATE <> TOP_STATE) Do:
    If (NEXT_STATE <> THIS_STATE)
      then:
        Call KCT_READ
        If (Key code was found)
          then:
            *Ensure that key word is exactly 10 characters long*
            DD_ARRAY [THIS_STATE-2]. KC := Key Code
            DD_ARRAY [THIS_STATE-2]. KW := Key Word
        End If:
        WS_HI_WRITE ((PREV_LEVEL-1)*10, DD_ARRAY [PREV_LEVEL]. KW)
        If (Next State implies one of the 4 definition modes)
          then:
            PREV_LEVEL := *the definition mode in question*
            WS_RV_WRITE((PREV_LEVEL-1)*10,DD_ARRAY[PREV_LEVEL].KW)
        End If:
        THIS_STATE := NEXT_STATE
        Call DT_A_READ
        Call PS_WRITE
    End If:
    Repeat:
      Call LINE_EDITOR
    Until (There is no error in the user-entered Data Item)
    Call DT_B_READ
    *Test for an error*
    Call PT_READ
    Call PS_WRITE
  End While:
  If (Key code implies abandonment of the present data item
      definition)
    then:
      Call PLACEMENT
  End If:
  Call DT_A_READ
  Call PS_WRITE
End:
End Procedure:
```

Fig.4.26: The Insert mode structure in PDL

A PDL skeleton of the complex PLACEMENT routine is described in Fig.4.27. The key codes are used for determining not only where the user is situated in the PDL program, but also where the newly defined data item is to be inserted. An integer search as opposed

to a string search makes key codes a very good solution to automating the placement process.

Only the relevant key words (in the set of four) are inserted where necessary, together with the data item name. If the user is in the Algorithm segment, inserting a data item will be transparent. The user is left at the same point in the Algorithm segment while only the line numbers in the Main Screen tell of a data item insertion elsewhere. If in the Data Description segment instead, Cursor 1 will be positioned at the new data item name; any necessary scrolling being dealt with by this placement routine.

```
Procedure  Placement
Begin:
  *Search  the Data Description segment by key codes to determine
   where new data item should be placed*
  *Determine  which  key words (if any) are to be  added  to  the
   file*
  *Insert  the  data item and any necessary key word in the  Data
   Description segment*
  If (User is in the Algorithm segment)
    then:
      *Change only the line numbers on the Main Screen*
    else:
      *Attempt to place Cursor 1 and the data item in the  centre
       of the Main Screen*
  End If:
End:
End Procedure:
```

Fig.4.27: The PDL structure for the Placement routine

Algorithm segment routine

This routine is responsible for Construct Insert mode. Here the user can insert an entire construct template in the Algorithm segment by depressing a single function key. This template-based system ensures that no syntactical errors occur. The routine is called as shown below:

```
ALGORITHM (     TOP_STATE: INTEGER;
                NO_OF_STATES: INTEGER;
                KCT_SIZE: INTEGER;
                PT_SIZE: INTEGER;
           VAR MS_CUR_POS: INTEGER;
           VAR MS_TOP_LINE: INTEGER;
           VAR FILE_BOT_LINE: INTEGER)
```

The Algorithm routine is built as a module and its structure can be seen in Fig.4.28. Construct insertion is not permitted in the Data Description segment; between an "If" and a "then:" of an If-then or If-then-else construct; or after the end of the file. The chosen construct is inserted after the line indicated by Cursor 1.

```
Procedure  Construct Insert
Variables:
     Boolean:
       Single:
         Local:
           EXIT_FLAG
Begin:
  *Read the File Linked List at the current Cursor 1 position*
  If (Key code obtained above implies error)
    then:
      *Put out an error -- insertion is not permitted here*
    else:
      *Perform an A READ operation on the Definition Table*
      *Write the key definition obtained above in Screen 4*
      EXIT_FLAG := FALSE
      Repeat:
        Call KBD_GET
        Case (Key) Of:
          special keys: Call KBD_GET
                        If (Key implies a valid function key)
                          then:
                            EXIT_FLAG := TRUE
                            Call DT_B_READ
                            If (Key code does not imply RETURN)
                              then:
                                Call GET_CONSTRUCT
                            End If:
                          else:
                            Case (Key) Of:
                              Cur Up: Call CUR_UP
                              Cur Dn: Call CUR_DOWN
                              Pg Up : Call PAGE_BWD
                              Pg Dn : Call PAGE_FWD
                                else:
                                  *Do nothing*
                            End Case:
                        End If:
                    else:
                      *Do nothing*
        End Case:
      Until (EXIT_FLAG = TRUE)
      Call DT_A_READ
      Call FS_WRITE
  End If:
End:
End Procedure:
```

Fig.4.28: The PDL structure for the Algorithm routine

The *resources* and routines used by this procedure are as follows:

Terminal Resource -- for displaying in Screens 1, 3 and 4

File Linked List -- for insertion of the block construct into the PDL file

Definition Table -- for determining the next states, prompt codes and key codes corresponding to the chosen template

Key Code Table -- for determining the relative indentation associated with each key word in the chosen template

Prompt Table -- for finding the prompt to display in the Prompt Screen from the prompt code obtained from the Definition Table

Construct Table -- for determining the structure of the template chosen by the user

Formatter -- for reformatting the display of the Main Screen whenever any part of it is to be updated

The *GET_CONSTRUCT routine* is called upon when a construct is to be inserted. Fig.4.29 shows the PDL structure for this routine. The key code obtained from the Definition Table on depression of a *function key* corresponds to a specific predefined template. The GET_CONSTRUCT routine will search the system table CONSTR.SYS to find the required template. The single line of data thus obtained (see Appendix C) is decoded and inserted in the File Linked List as a number of lines constituting the required template. The Main Screen is arranged so as to attempt to place the newly inserted construct at the top of the page, with Cursor 1 being positioned at the first line of the construct which contains a placeholder.

```
Procedure  Get Construct
Begin:
  *Search CONSTR.SYS table for the requested template*
  *Decode the data obtained above*
  *Write the template into the File Linked List*
  *Arrange  the  Main Screen display so that Cursor 1  is  on  the
   first template placeholder*
End;
End Procedure;
```

Fig.4.29: The PDL for the Get Construct routine

Indentation is automatically calculated, and any superfluous placeholders removed by the INDENT routine of Fig.4.30. This procedure compares the indentation of the line before and the line after which insertion is to occur. The line with the greatest indentation determines the indentation of the inserted construct.

To determine the indentation of the two lines mentioned above, the absolute indentation stored in the File Linked List needs to be added to the relative indentation obtained from the Key Code Table (see also Appendix D). A placeholder which is superfluous is detected when the line whose indentation was followed consists entirely of a "<CONSTRUCT>" placeholder. In this case, this line is removed. This important routine is also used in the Line Insert mode.

```
Procedure  Indent
Begin:
  *Read the File Linked List at the current pointer position*
  *Read the File Linked List at the next pointer position*
  *Read the Key Code Table for the key code of the first line*
  *Read the Key Code Table for the key code of the second line*
  *Calculate  the  total absolute indentations for the  lines  by
   using the indentation level and absolute indentations obtained
   above*
  If (Indentation of first line > Indentation of second line)
     then:
        Indentation := Indentation of first line
        If (The first line is a <CONSTRUCT> placeholder)
          then:
            *Return this line to the space list*
            FILE_BOT_LINE := FILE_BOT_LINE - 1
            MS_CUR_POS := MS_CUR_POS - 1
        End If:
     else:
        Indentation := Indentation of second line*
        If (The second line is a <CONSTRUCT> placeholder)
          then:
            *Return this line to the space list*
            FILE_BOT_LINE := FILE_BOT_LINE - 1
        End If:
  End If:
End:
End Procedure:
```

Fig.4.30: The PDL for the Indent routine

Line Inserting routine

This routine is responsible for dealing with Line Insert mode. Once entered, this mode allows the user to insert lines sequentially using the Line Editor. The routine call and the resources used are listed below:

```
INSERT_LINE (     TOP_STATE: INTEGER;
                  NO_OF_STATES: INTEGER;
                  KCT_SIZE: INTEGER;
                  PT_SIZE: INTEGER;
             VAR MS_CUR_POS: INTEGER;
             VAR MS_TOP_LINE: INTEGER;
             VAR FILE_BOT_LINE: INTEGER)
```

```
Terminal Resource -- for displaying purposes
File Linked List  -- for insertion into the PDL file
Definition Table  -- for determining the next state
Key Code Table    -- for use in the INDENT routine
Prompt Table      -- for displaying prompts in the Prompt Screen
```

Again insertion is not permitted in the Data Description segment; between an "If" and "then:" key word line; or after the end of the program. On entering the Line Insert mode, all the lines on the Main Screen from the current cursor position are cleared, and Cursor 1 vanishes. The user is then placed in the Window Screen

and uses the Line Editor facilities to ENTER a line of text. As the line is inserted, the INDENT routine mentioned previously is called, so that indentation and superfluous placeholder elimination are automatically dealt with. When the user exits this mode, the Main Screen is "closed up" and Cursor 1 reappears. Fig.4.31 shows the PDL structure adopted.

```
Procedure  Insert Line
Begin:
  *Read the File Linked List at the current cursor position*
  If (The key code obtained above implies an error)
    then:
      *Put out an error -- insertion not permitted here*
    else:
      *Clear the Main Screen of lines after Cursor 1*
      *Turn Cursor 1 OFF*
      Repeat:
        Call LINE_EDITOR
        *Insert the new line into the File Linked List*
        *Put out the new line to the Main Screen*
      Until (Insert Line mode is exit)
      *Close-up the Main Screen*
      *Turn Cursor 1 back ON*
  End If:
End:
End Procedure:
```

Fig.4.31: The PDL for the Insert Line routine

5 THE MAIN PROGRAM

5.1 Operation

The Main program deals with the loading of the system tables; the inputting and outputting of editor files; and the calling of the Base Level. On entering the *PDL-editor*, the system tables are loaded into dynamic memory. The user is then prompted for the input file. This is the file which is to be edited. This file is loaded into the editor and displayed on the Main Screen. At this stage, the user is in Base Level and all the editing functions described earlier can be accessed.

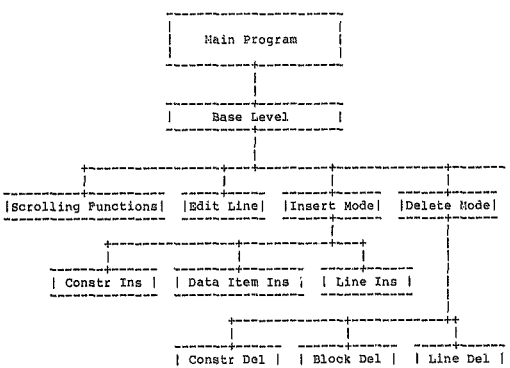


Fig.5.1: The structure of the PDL generator package Delete mode is shown, but has not yet been implemented. Similarly, Copy and Move modes are easily added from Base Level.

When the file has been satisfactorily edited, the ESC key will move the user back to the Main program. Here, the output filename

is prompted for. This is the file in which both the formatted and unformatted versions of the PDL program will be stored. (Extensions of ".PDL" and ".COD" will distinguish between the formatted and unformatted files.) The user is then able to edit another file via the same procedure or exit the package entirely.

5.2 Structure

It should be clear now that a modular hierarchical approach has been used for designing this PDL syntax-directed editor package. It is for this reason that the Main program will call upon the Base Level routine only, which in turn will call on the various modes available on request. The structure of the system is shown in Fig.5.1.

The Main program uses the following variables as global parameters:

```
-- MS_CUR_POS    : INTEGER      -- PT_SIZE      : INTEGER
-- TOP_STATE     : INTEGER      -- NO_OF_STATES: INTEGER
-- MS_TOP_LINE   : INTEGER      -- STATUS       : INTEGER
-- FILE_BOT_LINE: INTEGER       -- FILENAME     : STRING (8) OF CHAR
-- KCT_SIZE      : INTEGER
```

All these variables have been described previously, with the exception of FILENAME. This variable is provided to enable the user to specify input and output filenames from or to which data is to be obtained or stored. Fig.5.2 shows the PDL structure of the main program.

```
Program Main
Begin:
  *Initialize the TOP_STATE variable*
  Call SCR_FORMAT
  Call KCT_INIT
  Call DT_INIT
  Call PT_INIT
  Repeat:
    *Get the input filename from the user*
    Call FILE_LOAD
    Call FILE_SCROLL
    *Get the output filemame from the user*
    Call FILE_F_DUMP
    Call FILE_UF_DUMP
    *Determine whether the user wishes to exit the package*
  Until (User wishes to exit the package)
  Call CLR_SCR
End:
End Program:
```

Fig.5.2: The PDL for the Main program

It can be seen from Fig.5.2 that the screen is formatted into the four logical screens and all the system tables are loaded before the Repeat-until loop is entered. This looping construct enables the user to edit a number of files without having to re-load the system tables continuously. Input and output filenames can be user-specified, thus providing flexibility in this area.

The FILE_SCROLL routine constitutes Base Level. The user will remain in this routine until the ESC key is depressed from Base Level. On exiting the package, the physical display screen is cleared.

The FILE_LOAD, FILE_F_DUMP and FILE_UF_DUMP are routines used for initialization and termination of any editing session. Figures 5.3, 5.4 and 5.5 give virtually self-explanatory PDL descriptions of these routines.

```
Procedure  File Load
Begin:
  HS_TOP_LINE := 1
  HS_CUR_POS := 0
  Call FLL_INITIALISE
  *Read the input file and transfer it to the File Linked List*
  *Set the FILE_BOT_LINE variable*
  *Display the first 20 lines of the input file in formatted form
   on the Main Screen*
  *Turn Cursor 1 in the Main Screen ON*
End:
End Procedure:
```

Fig.5.3: The PDL for the File Load procedure

```
Procedure  Formatted File Dump
Begin:
  *Open an output file with the name specified and with a  ".PDL"
   extension*
  *Move the file list pointer to the beginning of the list*
  Repeat:
    Call FORMATTER
    *Write  the  strings obtained above on a single line  of  the
     text file*
    *Move the list pointer forward by one position*
  Until (All  items in the File Linked List have been  stored  on
         file)
End:
End Procedure:
```

Fig.5.4: The PDL for the Formatted File Dump routine

```
Procedure  Unformatted File Dump
Begin:
  *Open  an output file with the name specified and with a  ".COD"
   extension*
  *Move the list pointer to the beginning of the list*
  Repeat:
    *Read a record*
    *Write the data into two lines in the text file:  one for the
     key codes; the other for the text line*
    *Move the list pointer forward by one position*
  Until (All  items in the File Linked List have been written  in
         the file)
End:
End Procedure:
```

Fig.5.5: The PDL for the Unformatted File Dump routine

6 IMPLEMENTATION AND PORTABILITY CONSIDERATIONS

Due to limited facilities at the time, the package was initially implemented on the Eclipse S140 multiprocessing computer system in MP/Pascal (Version 2.3) (DGC (1979a, 1979b, 1980)). The Dasher Data General display terminals were used (DGC (1979c)) where blinking, reverse video, underscoring and highlighting are all possible.

The package had, however, severe limitations when implemented in this environment. Cursor positioning on the screen is not consistent, and this causes major problems when accurate cursor positioning is constantly required. Being a multitasking system supporting five terminals, compilation and linking is very slow. The major disadvantage , however, was the limited memory space available for executable program (64 KBytes). With such a large package, these limits were soon exceeded.

Due to new facilities, the PDL generator package was moved to an IBM Personal Computer. Here, all the problems mentioned above vanished: cursor positioning is accurately and consistently defined; compilation and linking are fast while 256 KBytes of dynamic memory are available for program storage.

The Pascal routines were easily transported; the only variations occuring in the areas of string manipulation and input and output. The screen management differences, however, proved to be vast. The compiler used is IBM Pascal Version 1.0 (also compatible with Microsoft Pascal version 3.2) under MS-DOS version 2.0 (IBM (1981, 1983)).

The Terminal Resource is the resource which deals with screen management, thus leaving the rest of the package terminal independent. On the IBM machines, the MS-DOS operating system (IBM (1983)) is used for this purpose so that it is necessary for the CONFIG.SYS file to contain the statement DEVICE = ANSI.SYS when the computer system is started up. Although it may be initially difficult to adapt the Terminal Resource to a particular system, once the primitives have been defined, the rest of the package will be functional in this respect. (Walker (1985))

As mentioned above, string manipulation and input and output facilities are the only features of the package affected by transporting it.

As the IBM-PC has set a new standard among personal computers, the package implemented hereon is likely to be fully developed on the IBM before being transported to another system. Microsoft Pascal also makes use of modules to differentiate between system blocks. *A modular design approach is encouraged in many texts* (Myers (1975), Parnas (1972)) as it makes a program separately compilable and thus portable. All these features are supported by Microsoft Pascal.

The resources are built as single data structures (eg. KCT_RES; PT_RES; etc. (see Appendix B)) surrounded by the operators. These operators are written as routines, and represent the only method whereby the user may access any resource. The resource is *compiled as a separate module*. Linking of this module to a process is performed by including the OPS file (eg. KCT_OPS; PT_OPS) in the process. This file contains all the operators available for the specific resource together as external procedures with any relevant input and output parameters. This enables the system designer to use the operators from any resource by including the OPS file.

Portability is further aided by the test programs which have been written for all the resources as well as the Line Editor. Using the test programs, the designer can ensure that resource operation is as expected. The Pascal used is also kept as standard as possible (Jensen (1974)), and this is another advantage if the package is to be transported.

7 EXTENSIONS, MODIFICATIONS AND RANDOM THOUGHTS

7.1 The Front-end

At this stage, the front-end of the package is regarded as the Main program. Here, on first entering the package, the system tables are loaded. This loading from diskfile is only done once, and this on entering the PDL editor package. Once the tables have been stored in dynamic memory, the user is free to edit a number of files.

From the Main program, before the FILE_SCROLL routine is called, the user is prompted for an input filename. This means that any coded file can be called on for further editing at this stage. It is assumed that the filename here has an extension of ".COD", as these are the only file types which can be understood by the package. If a new file is to be created, a set of function key options should be provided giving the user a choice of the type of program templates available. This choice should include:

-- Program
-- Procedure
-- Operation
-- Process
-- Resource

The user should thus either type the name of an existing file (with the use of the Line Editor in the Window Screen) which is to be re-edited, or use a function key to define a new file. Any new file must always be started with a template so as to give the system a reference point from which to work. Clearly, using a function-key template option for choosing a new file when a filename has been specified in the Window Screen constitutes an error.

Once an input filename has been chosen the user is placed in Base Level via the FILE_SCROLL routine. The User's Manual (Bassanino (1985b)) describes a level above Base Level (ie. a pre-Base level) where general package functions could be made available (eg. Information on package operation; or facilities for setting system parameters such as indentation settings or line length; etc.). This idea is used successfully in many editors (eg. IBM-Wordstar and IBM-Professional Editor) and greatly enhances the flexibility and power of the package.

When a file has been satisfactorily edited, a "Save" or "Abandon" function key is used from Base Level. This should lead the system through a series of questions so as to interactively determine the exact needs of the user. Confirmation should always be requested before an Abandon operation is performed. If, however, a file is to be saved, the user should be asked for an output filename; if a formatted copy of the file is to be stored; and if a backup copy of the file is to be created.

An alternative solution to output file naming is that of requesting a filename initially. This filename is then used as the new filename, without having to specify an output filename. Thus, if the file entered in the Window Screen exists, it will be loaded in the editor, while if it does not exist, a new file of this name is created.

This method has the disadvantage that if a file is to be used as a skeleton for the production of a few routines, the first file produced will take the name of the skeleton file. This means that the remaining programs which were to be based on the skeleton file cannot use it again. This problem can be resolved by maintaining a backup copy of the old file (the skeleton file) after having edited the first new file. This backup file can then be used again for the production of the outstanding routines.

It can be seen that the first method of naming input as well as output files is not popular because of the added effort involved in typing the various filenames; while the second method of specifying a single filename suffers from flexibility limitations. A good compromise seems to be that of adopting the first method, with automatic typing of the output and backup filenames. Thus, under normal circumstances, the user is only required to ENTER the filename displayed in the Window Screen. If so desired, however, a new filename can be specified by overtyping the existing filename in the Window Screen using the Line Editor facilities.

The user is now returned to the pre-Base system level where another file can be edited, or the package exited entirely. This pre-Base level is thus useful as a point where system settings can be adjusted before the next file is edited. In this level, the Main Screen may even acquire a new format. The present package version includes only limited facilities for file naming and storage, but this front-end interface needs to be perfected for the system to maintain its power and user-friendliness.

7.2 Delete Mode

In the User's Manual, three types of delete functions are described: Line Delete; Construct Delete; and Block Delete.

The Line Delete function has been implemented already (its PDL structure is shown in Fig.7.1), but in so doing has raised a new problem. In Line Delete mode, the line pointed to by Cursor 1 is deleted, while Cursor 1 remains at the same physical screen position. This means that the bottom of the file will effectively move up as deletion occurs. Thus, continuous deletion will delete consecutive lines with Cursor 1 remaining in the same screen position.

A screen scroll is thus never required in Line Delete mode. This does, however, imply that the bottom line of the file will not necessarily always lie in the last line of the Main Screen when a file is longer than MS_SIZE and consecutive lines are deleted near the end of the file. As the package scrolling routines were designed to operate maintaining the Main Screen full at all times, the delete function causes a problem. It is thus necessary

to devise a system whereby the file bottom line can appear at any position on the Main Screen before this delete function can be implemented successfully.

Returning to Fig.7.1, it can be seen that, initially, tests are performed to ensure that the line in question is not a line containing a keyword or a placeholder. If this test is passed, then the line may be deleted. Deletion is simply performed by returning the record pointed to by the File Linked List pointer, and decreasing the file length by one. Logically, only the portion of the Main Screen below Cursor 1 is rewritten; the cursor remaining in the same position.

```
Procedure  Delete Line
Begin:
  If (Line is a placeholder) or (Line contains a keyword)
    then:
      *Put out an error -- deleting is not allowed*
    else:
      *Return the record pointed to by the list pointer*
      FILE_BOT_LINE := FILE_BOT_LINE - 1
      *Rewrite the relevant portion of the Main Screen*
      *Turn new cursor ON at old position*
  End If:
End:
End Procedure:
```

Fig.7.1: The PDL for Line Delete mode

The Construct Delete function operates only in the Algorithm segment and is simply implemented by identifying the construct immediately surrounding the chosen line. The construct block is identified by its indentation level. Any consecutive lines above or below the chosen line which have an indentation level greater than or equal to that of the chosen line, will be included in the delete block. This block construct is to be highlighted so as to give the user a chance to confirm the deletion. This thus involves the accessing of the Definition Table for changing the function key definitions. The only challenge here (besides the usual screen scrolling necessities) lies in the insertion of any outstanding placeholder.

Determining whether a placeholder is to be inserted is a function of the template surrounding the delete block. If the lines before and after the delete block are both part of a system template, then the placeholder which appears between these two lines in the template is to be inserted. This requires that the CONST.SYS table be searched for a key code match. This method may seem time consuming, but due to the search being for an integer value and not a string, the Construct Table search is fast. The method is powerful in its generality and is also used in the Block Delete routine for insertion of outstanding placeholders.

The Block Delete function is divided into two parts: block deletion in the Data Description segment and block deletion in the Algorithm segment. Before discussing the implementation of

any of these, it is necessary to understand that this function requires the user to choose the block limits, and in so doing offers unbounded freedom. This freedom must be checked so as to protect the user from disrupting the syntactical correctness of the PDL program.

An upper and lower line limit is to be chosen by the user, and at every stage checking is performed. The upper line limit cannot be chosen as a data definition keyword. If a line in the Data Description segment is chosen as one block limit, the other limit cannot be placed in the Algorithm segment. Block deletion can thus not be performed across the Data Description - Algorithm segment dividing line.

A delete block is chosen in one of two ways: using the function keys or using line numbers. As explained in the User's Manual (Bassanino (1985b)), this dual method of choosing a block makes the system flexible and convenient to the user. The function keys are available to choose the line on which Cursor 1 is positioned. Simultaneously, the user is allowed to ENTER a line number in the Window Screen via the Line Editor facilities.

This dual mode of entry is possible due to the Line Editor's ability to be exit using a function key. The chosen block is dynamically highlighted. This means that if only one limit has been chosen, moving Cursor 1 will cause the line numbers from the chosen limit to the cursor to be highlighted. Once the delete block has been chosen, it is highlighted and the user prompted for its acceptance. This highlighting is easily accomplished using the MS_CUR_ON and MS_CUR_OFF routines of the Terminal Resource.

The deletion of a block in the Algorithm segment is a simple matter as every line between the two limits must be deleted. The only problem is to determine if the block may be deleted or not. The only law necessary here is the following: a delete block cannot contain any incomplete constructs. This requirement implies that any construct keyword present in the delete block is to have every other keyword associated with that construct present within the block.

Thus, the Construct table is again to be referred to so as to determine the keywords associated in a template with any other construct keyword found in the delete block. Again this seems a lengthy operation, but if a keyword search is performed from the top line of the delete block, an error is easily detected, while searching is greatly simplified. The indentation level of the constructs coupled with the top down order of checking for correctness ensures that this method functions correctly. It is, however, possible that a stack will need to be kept if deeply nested constructs exist.

In the Data Description segment, deletion is a more complex affair. A block is specified here by defining a begin and an end data item. All the data items lying within these two limits (including the two at the end points) will be deleted. The difficulty lies in deleting all the corresponding keywords. An algorithm has been devised for this purpose, but not yet implemented. It seems to cater for all cases using a simple procedure. The PDL algorithm is given in Fig.7.2. It is responsible for deciding which keyword lines should remain after

deleting the required block.

```
Procedure  Block Delete
Begin:
  Function Field := 0
  Type Field := 0
  Structure Field := 0
  Scope Field := 0
  *With the pointer at t.e start of the delete block ...*
  *Move pointer back one position*
  Repeat:
    If (Line contains a key code implying a keyword)
      then:
        *Set the delete block's top limit to this line*
        *Move pointer back one position*
        Status := Continue
      else:
        Status := Terminate
    End If:
  Until (Status = Terminate)
  Repeat:
    *Move pointer forward by one position*
    Case (key code) implying:
      Function : Function Field := Line Number
      Type     : Type Field := Line Number
      Structure: Structure Field := Line Number
      Scope    : Scope Field := Line Number
      Name     : *Do Nothing*
    End Case:
  Until (Last line of delete block has been reached)
  *Move pointer forward by one position*
  Case (key code) implying:
    Function : Function Field := 0
               Type Field := 0
               Structure Field := 0
               Scope Field := 0
    Type     : Type Field := 0
               Structure Field := 0
               Scope Field := 0
    Structure: Structure Field := 0
               Scope Field := 0
    Scope    : Scope Field := 0
    Name     : *Do Nothing*
  End Case:
  *Delete  all the lines in the delete block excepting  the  line
    numbers stored in the four data definition field variables*
End:
End Procedure:
```

Fig.7.2: A PDL algorithm for Data Description Block Deletion

The algorithm of Fig.7.2 uses the four data definition fields as running variables. It fills the variables with the line numbers of any possible key codes which could be retained. Working top-down, any new key code lines found will be used as new line numbers, replacing the old in the relevant *field variable*. The top and bottom limits must also be taken into account if the line

before or the line after the delete block is not a data item. On termination of the routine, the four variables will contain either a zero or a line number. These line numbers will represent the lines which are to remain after the delete block is deleted.

Clearly, Main Screen formatting and Cursor 1 positioning are also issues, but the intelligent mechanism behind this type of Block delete mode has been described above.

7.3 Copy Mode

In Copy mode, Block and Line Copy functions should be provided. Again the user should be presented with the dual method for choosing a block as well as destination line. Here too, the copy block is highlighted. After the copy block has been chosen, however, Cursor 1 must be displayed in a different font so as to allow the user to choose a destination line. A blinking cursor in reverse video should be conspicuous enough for this purpose. This requires an additional function to the Terminal Resource, but then this cursor display font can also be used for other purposes such as Move mode.

The physical copy operation is a relatively simple task using the linked list, but again, extensive checking of the syntactical correctness of the operation is necessary. No copying is permitted in the Data Description segment or this section of the PDL program is strictly controlled by the PDL editor. Indentation of the new copied line is performed automatically by the system, and this is simply accomplished using the INDENT routine adopted in Insert mode. This routine will also determine whether to delete any superfluous placeholder in the vicinity of the copied line.

When in Line Copy mode, only entirely user-entered lines may be copied. Thus, when attempting to copy a single line, no line containing a system-generated keyword will be accepted. The destination line is subject only to the limitations of Line Insert mode. Thus, a line cannot be copied after the "If <CONDITION>" line, or after the end of the file, or in the Data Description segment. These same limitations on destination line apply to the Block Copy mode.

In Block Copy mode, the copy block may not contain any incomplete construct. This is the same requirement used for Block Delete mode. Thus, only entire construct blocks may be copied; part of a construct may not be copied unless it contains no system-generated keywords. Automatic indentation is accomplished by using the INDENT routine on the top line of the copied block. The indentation level thus obtained becomes the new base indentation which is to be added to all the other lines in the copied block.

It can thus be seen that Copy mode merely uses routines which have already been developed above for a different purpose. Besides the screen management, this mode is thus easily implemented.

7.4 Move Mode

In Move mode too, there are two possible functions: the single line move operation; and the block move operation. The move function is a combination of a Copy operation followed by a Delete operation. Again the user is given the dual mode of stipulating lines, with a blinking cursor in reverse video for choosing the destination line where the move block is to be placed.

As a Move operation is a conbination of two of the functions developed above, only some kind of integration is necessary. Clearly, a line or block which may be copied, may also be deleted. Thus, checking of the chosen copy and destination line/(s) is performed using the Copy mode algorithm described above. The move block is then copied to the destination position.

The usual requirements of deleting any superfluous placeholders and automatic indentation are automatically dealt with by the Copy operation. Now, without refreshing the Main Screen display or prompting for the delete block, a Delete operation is performed on the original move block. This delete function will also deal with the insertion of any necessary placeholder as discussed in section 7.2. After updating the Main Screen, an effective Move operation will have been performed.

It can thus be seen that Move mode is a derivation of the other modes discussed above. An initial interface almost identical to the Copy mode must be used to enable the user to choose the relevant move and destination lines. The Copy mode checking requirements are then applied to the inputs to determine their validity. A valid Move operation is performed by firstly copying the move block after the destination line, and then deleting the original move block. The screen scrolling algorithm used is that of the Copy mode, so that Cursor 1 is positioned at the beginning of the moved block after the operation has been completed.

7.5 Semantic Error Detection

As the package is built, syntactical errors are prevented due to the generative approach adopted. The template based system and all the functions described above, ensure syntactical correctness at all times. Semantic errors, however, are more complex to determine due to their long-range or far-reaching nature.

It must be realized when dealing with a PDL generator that it is difficult to draw the line between a semantically correct and an incorrect high-level program. Due to its high-level nature, a PDL program can be written in terms of comments only and still be correct. It is for this reason that the user should be able to choose a level of error checking before entering a program. A low level of error checking would be chosen if a high-level PDL program is being generated, while a high degree of error checking is desirable for a low-level PDL design. It is probably wise to allow only two levels of error checking: syntactical error checking only for high-level PDL; and semantic and syntactical error checking for low-level PDL programs.

7.4 Move Mode

In Move mode too, there are two possible functions: the single line move operation; and the block move operation. The move function is a combination of a Copy operation followed by a Delete operation. Again the user is given the dual mode of stipulating lines, with a blinking cursor in reverse video for choosing the destination line where the move block is to be placed.

As a Move operation is a combination of two of the functions developed above, only some kind of integration is necessary. Clearly, a line or block which may be copied, may also be deleted. Thus, checking of the chosen copy and destination line/(s) is performed using the Copy mode algorithm described above. The move block is then copied to the destination position.

The usual requirements of deleting any superfluous placeholders and automatic indentation are automatically dealt with by the Copy operation. Now, without refreshing the Main Screen display or prompting for the delete block, a Delete operation is performed on the original move block. This delete function will also deal with the insertion of any necessary placeholder as discussed in section 7.2. After updating the Main Screen, an effective Move operation will have been performed.

It can thus be seen that Move mode is a derivation of the other modes discussed above. An initial interface almost identical to the Copy mode must be used to enable the user to choose the relevant move and destination lines. The Copy mode checking requirements are then applied to the inputs to determine their validity. A valid Move operation is performed by firstly copying the move block after the destination line, and then deleting the original move block. The screen scrolling algorithm used is that of the Copy mode, so that Cursor 1 is positioned at the beginning of the moved block after the operation has been completed.

7.5 Semantic Error Detection

As the package is built, syntactical errors are prevented due to the generative approach adopted. The template based system and all the functions described above, ensure syntactical correctness at all times. Semantic errors, however, are more complex to determine due to their long-range or far-reaching nature.

It must be realized when dealing with a PDL generator that it is difficult to draw the line between a semantically correct and an incorrect high-level program. Due to its high-level nature, a PDL program can be written in terms of comments only and still be correct. It is for this reason that the user should be able to choose a level of error checking before entering a program. A low level of error checking would be chosen if a high-level PDL program is being generated, while a high degree of error checking is desirable for a low-level PDL design. It is probably wise to allow only two levels of error checking: syntactical error checking only for high-level PDL; and semantic and syntactical error checking for low-level PDL programs.

Semantic error checking involves much computation as many possible errors exist. It is interesting to note, however, that the designers of GNOME (see Bassanino (1985a)) list the following four errors as comprising 90% of all semantic errors made by students.

-- undeclared variable
-- unused but declared variable
-- uninitialized variable
-- type mismatch

It is clear that all the above errors deal with the relationship between the Data Description and Algorithm segments. Thus, a record must be kept of each variable which is either defined in the Data Description segment or used in the Algorithm segment. The concept of a Variable Table therefore emerges. Each data item appearing in the PDL program is stored in this table by name and by code number. The code number is a unique number used to identify the type of the data item for easy reference purposes. The Variable Table also needs to maintain a record of the position of every occurrence of the data item in the PDL file.

The best way for checking errors is at input time. When any line is edited (either by using the Line Edit function or Line Insert mode) *parsing of that line should be performed* after it has been ENTERed. Thus, a line would have any errors highlighted on the Main Screen in reverse video. These errors can then be re-edited (the line being reparsed) for correction. Unexpanded placeholders should also be regarded as errors (errors of omission) as a program cannot be compiled if any placeholder is left unexpanded.

An Error file is necessary to keep track of the position and nature of all the errors present in a PDL file. This Error File would also be used for reference when highlighting the errors on the Main Screen. A field in this table can be set aside for determining the type of error (eg. mismatched parentheses; undeclared variable; etc.). According to the error type, the corresponding error message would be displayed (perhaps via use of the Prompt Table) on request.

In this way, a user can at any time request information regarding a particular error as the editor maintains a record of all current errors. This system will also enable the user to obtain a breakdown of all the errors and their types which have been identified in the PDL file. When the system lists no errors, then the user can with confidence deduce that the PDL file generated is virtually 100% error-free.

In the Data Description Insert mode, a newly defined data item should also be checked for acceptance against all existent data items. The user will therefore be warned if the defined data item already exists, and will be asked if replacement is desired. The Data Description segment also causes other problems, because if a data item name is changed here, *the rest of the file must be* checked (or reparsed) for any occurrence of this data item. This will be by far the most time consuming operation available (especially if the file is a long one), and thus time will be well spent in the designing of an efficient batch parser.

7.6 Ellipsis Facilities

The elliding or temporary removal of a program section is a useful feature if the entire PDL program is to be viewed on the Main Screen without its internal details. This feature also helps to identify constructs. The ellipsis feature will replace a section of PDL code with an ellipsis token (usually "...."). The section of code to be ellided can be chosen in one of two ways.

An ellision level can be specified so that a zero ellision level will display the program in full detail, while an infinite ellision level will show only the outermost program level. Another method for choosing ellision is to point to a line on the level below and including which ellision must occur. By depressing an Ellide key, this program block will temporarily be removed from view on the Main Screen.

Besides the user-interface described above for choosing an ellision block, the physical implementation of the function is elementary. The "level" used for ellision purposes corresponds exactly to the indentation level adopted for coding each PDL line. Thus, identification of an ellision block is simply performed by searching for all lines in the file which have an indentation level greater than or equal to the ellision level.

7.7 The UNDO Stack

A stack needs to be maintained if multiple "undo" operations are to be allowed. The undo operation is a very convenient facility for both the experienced and the novice user. With the help of this function, a user can essentially return to a stage where the last edits to the file have not been performed. This helps recover from accidental errors of deletion, also giving the user a chance to experiment with system features. The user can thus edit a PDL file with the knowledge that any operation which is performed can be undone at any stage. This gives a system almost unlimited power making it truly foolproof.

The *Undo* function can be implemented by maintaining a stack of previous commands. When an Undo operation is requested, the last command is popped from the stack and an opposite command performed so as to reverse the previous operation. This will effectively produce the file as it was before the last command was performed. The depth of the stack will determine the depth of the Undo function. A table is also necessary here so that the inverse of any command can be found.

The Undo stack can also be used for inconvenient block editing operations. Converting a While-do into a Repeat-until construct, for example, is possible by merely pushing the relevant block onto the stack; inserting the new construct; and popping back the contents of the stack. In this way, the flexibility removed from a syntax-directed template-driven editor is regained. The conversion of, say a Program, into a Procedure, poses a problem, however, as the function keys must again offer the user the choices given when a new PDL file is created.

7.8 Possible Design Improvements

The Data Description segment Insert mode is as yet incomplete, due to the missing feature of *allowing a user to define his own* data types. User-defined data types are a feature of many modern programming languages, including PDL. When wanting to define a data type, the user should be given a "Record" function key option. This function key will enable a record type to be defined. Associated with this function key there will, therefore, also be a possible further set of function keys which will aid the user in defining a record in an orderly manner.

The record fields must be stored by the system if extensive semantic error checking is to be performed. A parser is indispensable if data types are to be user-defined. The user has the possibility of defining a type incorrectly due to the flexible manner in which it may be defined. Thus, a parsing algorithm would be responsible for checking the grammar of the type definition.

An alternative solution to the problem of inaccuracy associated with defining a new type, is the use of templates for type *definition. The possible type definition frameworks* must first be decided upon so that the Data Description segment will be completely standardized. Templates introduce the added problem that functions such as Delete, Copy and Move allowed in the Algorithm segment now need to be allowed in the Data Description segment. Also, unexpanded placeholders in the Data Description segment will become a reality.

Thus, the problem of data type definition is to be considered very carefully not only in terms of an elegant solution, but also in terms of a solution which will not require major system design changes. When the "type" field of a data item is to be chosen, the "Others" function key should lead into a state where all the user-defined types are displayed as function key options. This facility is clearly challenging to implement.

In the Data Description Insert mode, scrolling facilities may also prove useful (especially when wanting to know the definition of a new type while defining a data item). A fast abort function key for exiting this mode is also a necessity for speeding-up system operation.

As the package was designed, there are still some prompts which are hard-programmed into the system routines. It should be attempted to make the Prompt Table the sole dispenser of all prompts, so that system flexibility is enhanced. The designer is then able to change any prompt to suit the particular client.

Presently there is no facility for entering multiple lines for, say, a *CONDITION placeholder. This necessary reature* is limited by the 80 character line length of the Line Editor. If the line length is not to be made more flexible, then a method has to be devised whereby a line can be extended on the following line; indentation automatically being taken care of.

The unusual indentation required when listing multiple external procedure names, inputs or outputs, requires that the items are listed one under the other, relative to the first item. Fig.7.3

shows the indentation required when defining input and output parameters of a procedure. If a method is devised for extending a line onto the next line while maintaining the same indentation of the user-entered text portion of the line above, then this problem, together with the problem mentioned above is solved.

The Case construct also presents a problem, due to the single placeholder present under the Case line. Indentation of user-entered lines must be alligned under the desired option's colon and not, as the package dictates, directly under the option. This problem may be solved by parsing the input lines so as to determine if a line entered is an option or merely an algorithm statement. The indentation can then be determined accordingly for the lines following a Case option. (See Fig.7.3)

```
Procedure Test
  ...
Inputs: Inden Level
        Cursor Position
Outputs: First String
         Second String
         Third String
  ...
 Begin:
    ...
      Case (Cursor Position) of:
        Home: First String := 'X'
              Second String := 'Z'
        End of Line: First String := 'A'
                     Second String := 'B'
                     Third String := 'C'
      End Case:
    ...
 End:
 End Procedure:
```

Fig.7.3: Indentation for Inputs, Outputs and Case construct

Another problem associated with constructs involves the Get and Put templates. In PDL, these two constructs are used exclusively for input and output purposes only. Thus, it is not logical to insert another construct within a Get or Put block. This problem is solved by the system checking if the cursor lies within one of the above constructs before any system-defined template is inserted.

A parser is also required to check for allowable user-entered text lines within any construct. The text within a Get construct, for example, will differ from an algorithm line, and this in turn will be different from a condition. The parser would thus distinguish if a line is correct or not according to the key code of the placeholder or surrounding text line. This parser routine is thus seen to be a very critical item in the construction of a successful synax-directed editor package.

As already mentioned in section 7.2 in this chapter, the requirement that the Main Screen always be full when the bottom

of a more than 20 line long file is reached is a severe restriction. Both Line Delete and Line Insert modes suffer the consequences of this law. The bottom of the file should be demarcated by an end of file line, and scrolling beyond this line should not be permitted. This line should be allowed to be positioned at any point in the Main Screen, as dictated by any needy operation. This will solve a few problems as well as make the system more general.

In Base Level, the Edit Line facility retains the function key definitions. These definitions should either be removed, or allowed as exit options while in the Line Editor. It is doubtful whether the scrolling functions should be allowed while editing a line, as this will only serve to confuse the user. It is thus probably wiser to erase the Function Key Definition Screen when dealing with the Edit Line function.

To avoid repetition of function key screens in the Definition Table, a pointer could be used to the Prompt Table. This would mean that the Prompt Table could become a more flexible Text Line Table having fields for a line code and a text line of 80 characters long. The Definition Table takes up much space and this method could serve as a system to decrease this large memory requirement. Function keys in the Definition Table which are not used could also be eliminated so as to maintain only the absolute minimum information required for system operation.

The conversion of a Program into a Procedure using the Undo stack requires that the function key screen displayed on initializing a new file be re-displayed for the user to choose the required procedure type. This operation is by no means trivial and more detailed thought must be give to this problem.

A neater method for re-displaying the whole of the Main Screen is also necessary, so that the list pointer will never be pushed beyond its limits. As the File Linked List resource used has a zero position, the list pointer should be started from a higher position and then incremented to the bottom of the Main Screen. Thus, after displaying an entire screen, the list pointer will always be on the last line of the Main Screen, whether the bottom of the file has been reached or not. Fig.7.4 shows this algorithm which can be compared to the old system used in Fig.4.17.

```
*Move list pointer to one line before the first line to be
 displayed*
i := 0
While (i < MS_SIZE)
  *Move pointer forward by one position*
  i := i + 1
  Call FORMATTER
  MS_WRITE (MS_TOP_LINE + i - 1, i - 1, ....)
End While:
```

Fig.7.4: A neater algorithm for updating the Main Screen

It may also be an idea to create routines for the scrolling of the Main Screen to give a particular line number at a particular

screen position, so that the Insert, Delete, Copy and Move functions will not be involved with explicit screen management details.

7.9 Future Package Expansion and Integration

Most of the expansion features have been discussed above in this chapter, however, there are greater plans for the syntax-directed PDL generator. As may have been mentioned in this document, the editor developed here is to become the basis for a language-independent translator. With a powerful tool for PDL construction, the user will be able to write an algorithm in high-level description language and then request a compilable version of the program in any of a variety of modern programming languages.

This will thus enable the user to design on a high level, using all the facilities of the syntax-directed editor which encourage top-down design. When the algorithm has been refined down to low-level PDL, the coded program can be used as input to one of a few language translators. Thus, the user will finally become independent of implementation language. This is seen as a major breakthrough in design methodology.

If this PDL generator package proves a success, it may be a worthy exercise to implement a coding algorithm to convert a standard text file into a form usable by the package. This will require, not only coding of the keywords and variables, but also extensive parsing to trap and record any errors which may be present in the PDL file.

APPENDIX A: THE TEST ROUTINES

This appendix shows the menus produced by the resource test programs. All the major functions are performed; logical and physical views of the resource structure also being given. The Line Editor test format is also shown. The following figures are true copies of the scroling display presented to the user on the VDU when the test programs are run. They have been chosen to convey the nature, flexibility and power of the individual test routines. The menu will end with a "? :" to which the user must reply. Each time a menu is displayed, a menu choice user response is required. All user responses are highlighted and underscored for easy identification.

A.1 Definition Table

Definition Table Structure And Access Operators

```
          Table Initialise          (I)
               A_Read               (A)
               B_Read               (B)
               Structure Display    (S)

          Quit this program         (Q) ? :I


     TABLE INITIALISED !


          Table Initialise          (I)
               A_Read               (A)
               B_Read               (B)
               Structure Display    (S)

          Quit this program         (Q) ? :A
```

```
This State (Range: 1.. 9) = 6


    The operation was SUCCESSFUL

This Key Definition Line is :
 Globl:  Perma:  Exter:  Local:
   1       2       3       4

                                                CONT    RET
                                                 9      10


                Table Initialise        (I)
                      A_Read            (A)
                      B_Read            (B)
                      Structure Display (S)

                Quit this program       (Q) ? :B


This State (Range: 1.. 9) = 6


This Key (Range: 1..10) = 2


This operation was Successful
Next State (if no error) =  7
Next State (if error)    =  7
Key Code =   32
Prompt Code (if no error) =  6
Prompt Code (if error)    =  6
```

```
                Table Initialise          (I)
                      A_Read              (A)
                      B_Read              (B)
                      Structure Display   (S)

                Quit this program         (Q) ? :S


                        STATE =  1

DD Seg  Algo
------- ------ ------ ------ ------
                                                              RET
                                ------ ------ ------ ------ ------

KEY  KEY CODE  NEXT STATE1  NEXT STATE2  PROMPT CODE1  PROMPT CODE2
 1       50          3           3            1             1
 2       60          8           8            8             8
 3        0          0           0            0             0
 4        0          0           0            0             0
 5        0          0           0            0             0
 6        0          0           0            0             0
 7        0          0           0            0             0
 8        0          0           0            0             0
 9        0          0           0            0             0
10     4000          9           9            0             0

The Key Flags are as follows:
T   T   F   F   F   F   F   F   F   T

PRESS ANY KEY TO CONTINUE



                        STATE =  2

Yes             No
------- ------ ------ ------ ------
                                ------ ------ ------ ------ ------

KEY  KEY CODE  NEXT STATE1  NEXT STATE2  PROMPT CODE1  PROMPT CODE2
 1     1001          1           1            5             5
 2        0          0           0            0             0
 3     2000          3           3            1             1
 4        0          0           0            0             0
 5        0          0           0            0             0
 6        0          0           0            0             0
 7        0          0           0            0             0
 8        0          0           0            0             0
 9        0          0           0            0             0
10        0          0           0            0             0

The Key Flags are as follows:
T   F   T   F   F   F   F   F   F   F

PRESS ANY KEY TO CONTINUE
```

STATE = 3

Const:	Var:	Type D				
					COMT	RET

KEY	KEY CODE	NEXT STATE1	NEXT STATE2	PROMPT CODE1	PROMPT CODE2
1	1	4	4	1	1
2	2	4	4	1	1
3	3	3	3	3	3
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	3000	4	3	1	4
10	4000	2	2	2	2

The Key Flags are as follows:
T T T F F F F F T T

PRESS ANY KEY TO CONTINUE

... etc ...

PRESS ANY KEY TO CONTINUE

Table Initialise (I)
A_Read (A)
B_Read (B)
Structure Display (S)

Quit this program (Q) ? :Q

A.2 Key Code Table

Key Code Table Structure And Access Operators

```
            Table Initialise          (I)
                  Read                (R)
                  Structure Display   (S)

            Quit this program         (Q) ? :I


        TABLE INITIALISED!


            Table Initialise          (I)
                  Read                (R)
                  Structure Display   (S)

            Quit this program         (Q) ? :R

  Key Code = 12


      The operation was SUCCESSFUL

Key Word        = Integer:
Key Word Length =  8

Indentation     =  5

Edit Flag       = FALSE
```

```
Table Initialise          (I)
      Read                (R)
      Structure Display   (S)

Quit this program         (Q) ? :S
```

Maximum Table Size = 52

Key Code	Indentation	Edit Flag	Key Word
1	0	FALSE	Constants:
2	0	FALSE	Variables:
11	5	FALSE	Boolean:
12	5	FALSE	Integer:
13	5	FALSE	Real:
14	5	FALSE	Character:
21	8	FALSE	Single:
22	8	FALSE	Array:
31	11	FALSE	Global:
32	11	FALSE	Permanent:
33	11	FALSE	External:
34	11	FALSE	Local:
0	0	FALSE	
40	14	TRUE	
100	0	TRUE	If
101	2	FALSE	then:
102	2	FALSE	else:
103	0	FALSE	End If:
200	0	TRUE	While
201	0	FALSE	Do:
202	0	FALSE	End While:
300	0	FALSE	Repeat:
301	0	TRUE	Until
400	0	TRUE	Case
401	0	FALSE	Of:
402	0	FALSE	End Case:
900	0	TRUE	<CONSTRUCT>
901	2	TRUE	<CONSTRUCT>
902	4	TRUE	<CONSTRUCT>
910	0	TRUE	<CONDITION>
920	0	TRUE	<VARIABLE>
950	0	TRUE	
951	2	TRUE	

... etc ...

Press any key to continue

```
Table Initialise          (I)
      Read                (R)
      Structure Display   (S)

Quit this program         (Q) ? :Q
```

A.3 Prompt Table

Prompt Table Structure And Access Operators

```
        Table Initialise          (I)
              Read                (R)
              Structure Display   (S)

        Quit this program         (Q) ? :I


    TABLE INITIALISED!




        Table Initialise          (I)
              Read                (R)
              Structure Display   (S)

        Quit this program         (Q) ? :R

This Code = 3


   The operation was SUCCESSFUL
This Prompt = This function key is as yet undefined...




        Table Initialise          (I)
              Read                (R)
              Structure Display   (S)

        Quit this program         (Q) ? :S
```

```
Maximum Table Size = 20

Code          Prompt
----          ------
 1    Define the Data Item using the Line Editor and Function Keys.
 2    Do you want to abandon this definition ?
 3    This function key is as yet undefined...
 4    Use the function keys to define this field.
 5    Definition ABANDONED !
 6    Do you want to accept this definition ?
 7    Definition Accepted.
 8    Choose one of the following Templates using the Function Keys.
 9    Remember to replace any outstanding Placeholders
10
11
12
13
14
15
16
17
18
19
20

     Press any key to continue



                Table Initialise          (I)
                      Read                (R)
                      Structure Display   (S)

                Quit this program         (Q) ? :Q
```

A.4 Line Linked List

Due to the complexity of this resource, a comment is given when each test operation is performed. These comments (displayed as PDL comments) should lead the reader through a useful exercise in linked list manipulation. For the sake of clarity the linked list has a capacity of only 10 data items.

```
  Line Linked List Data Structure and Access Operators
  ----------------------------------------------------

          (Forward and Backward Pointers)

          List Initialise         (I)
          Get a Record            (>)
          Return a Record         (<)
          Write a Record          (W)
          Read a Record           (R)
          Move Pointer            (M)
          Display List            (D)
          List Structure          (S)

          Quit this Program       (Q) ? I

*List is Initialized*


    LIST INITIALISED!









          List Initialise         (I)
          Get a Record            (>)
          Return a Record         (<)
          Write a Record          (W)
          Read a Record           (R)
          Move Pointer            (M)
          Display List            (D)
          List Structure          (S)

          Quit this Program       (Q) ? >

*A blank record is inserted into the link list*


    STATUS = 0  RECORD INSERTED INTO LINK LIST!
```

```
          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? W
```

An "I" is written in the new record

```
 DATA ITEM = I


 STATUS = 0 DATA ITEM WRITTEN INTO LIST!
```

```
          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? >
```

** A further two characers (a "T" and an "S") are written into the list **

```
    STATUS = 0  RECORD INSERTED INTO LINK LIST!
```

```
          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? W
```

```
DATA ITEM = T


STATUS = 0 DATA ITEM WRITTEN INTO LIST!




        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? >


   STATUS = 0  RECORD INSERTED INTO LINK LIST!




        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? W

DATA ITEM = S


STATUS = 0 DATA ITEM WRITTEN INTO LIST!
```

```
            List Initialise         (I)
            Get a Record            (>)
            Return a Record         (<)
            Write a Record          (W)
            Read a Record           (R)
            Move Pointer            (M)
            Display List            (D)
            List Structure          (S)

            Quit this Program       (Q) ? M

*The list pointer (now at position 3) is moved back by 8
 positions*

 MOVE INCREMENT = -8


List Bounds were OVERSHOT
 STATUS = 3 LLP IS OUTSIDE OF LIST!









            List Initialise         (I)
            Get a Record            (>)
            Return a Record         (<)
            Write a Record          (W)
            Read a Record           (R)
            Move Pointer            (M)
            Display List            (D)
            List Structure          (S)

            Quit this Program       (Q) ? D

*A logical display of the list is requested*



     Logical List Pointer =  0
     Maximum Records      = 10

 LOCATION :  1    2    3
DATA ITEM :  I    T    S
```

```
        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? S
```

A physical display of the list is requested

```
   MAXIMUM ITEMS          =   10     LINK LIST SIZE          =    3
   LINK LIST ROCK         =    1     SPACE LIST ROCK         =    4
   LOGICAL LIST POINTER   =    0     PHYSICAL LIST POINTER   =    0

LOCATION :          1   2   3   4   5   6   7   8   9  10
FORWARD POINTER :   2   3  11   5   6   7   8   9  10  11
BACKWARD POINTER:   0   1   2   3   4   5   6   7   8   9
DATA ITEM :         I   T   S
```

```
        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? R
```

A record is read (but the pointer is still at position zero!)

```
STATUS = 3 LLP IS OUTSIDE OF LIST!
```

```
          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? M
```

The list pointer is moved forward by two positions

```
 MOVE INCREMENT = 2


 STATUS = 0 LIST POINTER NOW AT =  2
```

```
          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? R
```

A read operation is performed and it can be seen that the list pointer is now at position 2

```
 STATUS = 0 READ DATA ITEM = T
```

```
List Initialise          (I)
Get a Record             (>)
Return a Record          (<)
Write a Record           (W)
Read a Record            (R)
Move Pointer             (M)
Display List             (D)
List Structure           (S)

Quit this Program        (Q) ? D
```

Confirmation of the above operation is seen via the logical display

```
    Logical List Pointer =  2
    Maximum Records      = 10

 LOCATION :  1    2    3
DATA ITEM :  I    T    S
```

```
List Initialise          (I)
Get a Record             (>)
Return a Record          (<)
Write a Record           (W)
Read a Record            (R)
Move Pointer             (M)
Display List             (D)
List Structure           (S)

Quit this Program        (Q) ? >
```

** With the pointer at position 2, a "!" character is inserted **

```
STATUS = 0  RECORD INSERTED INTO LINK LIST!
```

```
List Initialise          (I)
Get a Record             (>)
Return a Record          (<)
Write a Record           (W)
Read a Record            (R)
Move Pointer             (M)
Display List             (D)
List Structure           (S)

Quit this Program        (Q) ? W
```

```
 DATA ITEM = !

 STATUS = 0 DATA ITEM WRITTEN INTO LIST!


          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? D
```

Again the logical display is requested to observe the insert operation (Note the position of the "!" inserted character and the list pointer)

```
     Logical List Pointer =  3
     Maximum Records      = 10

 LOCATION :  1    2    3    4
DATA ITEM :  I    T    !    S


          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? <
```

** Now, returning the record should delete the "!" character (Note again the resulting list pointer position) **

```
 STATUS = 0 RECORD RETURNED TO SPACE LIST!
```

```
        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? D


    Logical List Pointer =  2
    Maximum Records      = 10

 LOCATION :  1    2    3
DATA ITEM :  I    T    S
```

```
        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? I
```

** Initializing the list will empty the list logically, but not physically, as the records are returned to the space list **

```
    LIST INITIALISED!
```

```
        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? D
```

```
LINK LIST IS EMPTY!


          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? S


   MAXIMUM ITEMS          =  10    LINK LIST SIZE          =   0
   LINK LIST ROOT         =  11    SPACE LIST ROOT         =   1
   LOGICAL LIST POINTER   =   0    PHYSICAL LIST POINTER   =   1

LOCATION :          1  2  3  4  5  6  7  8  9 10
FORWARD POINTER :   2  3  4  5  6  7  8  9 10 11
BACKWARD POINTER:   0  1  2  3  4  5  6  7  8  9
DATA ITEM :         I  T  S  '


          List Initialise        (I)
          Get a Record           (>)
          Return a Record        (<)
          Write a Record         (W)
          Read a Record          (R)
          Move Pointer           (M)
          Display List           (D)
          List Structure         (S)

          Quit this Program      (Q) ? Q

*Exit the test program*
```

A.5 File Linked List

A similar exercise to the one of section A.4 can be performed with this resource. The only difference is in the definition of the multiple fields to identify a PDL line.

```
File Linked List Data Structure and Access Operators
-----------------------------------------------------
        (Forward and Backward Pointers)

        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? I

  LIST INITIALISED!

        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? >

  STATUS = 0  RECORD INSERTED INTO LINK LIST!

        List Initialise         (I)
        Get a Record            (>)
        Return a Record         (<)
        Write a Record          (W)
        Read a Record           (R)
        Move Pointer            (M)
        Display List            (D)
        List Structure          (S)

        Quit this Program       (Q) ? W
```

INDENTATION CODE = 2

KEY CODE 1 = 900

KEY CODE 2 = 901

TEXT LINE = X := X + 1

STATUS = 0 DATA ITEM WRITTEN INTO LIST!

```
List Initialise        (I)
Get a Record           (>)
Return a Record        (<)
Write a Record         (W)
Read a Record          (R)
Move Pointer           (M)
Display List           (D)
List Structure         (S)

Quit this Program      (Q) ? R
```

```
STATUS = 0 THE CODES AND TEXT LINE ARE :
          2          900          901
X := X + 1
```

```
List Initialise        (I)
Get a Record           (>)
Return a Record        (<)
Write a Record         (W)
Read a Record          (R)
Move Pointer           (M)
Display List           (D)
List Structure         (S)

Quit this Program      (Q) ? >
```

STATUS = 0 RECORD INSERTED INTO LINK LIST!

```
          List Initialise          (I)
          Get a Record             (>)
          Return a Record          (<)
          Write a Record           (W)
          Read a Record            (R)
          Move Pointer             (M)
          Display List             (D)
          List Structure           (S)

          Quit this Program        (Q) ? W

 INDENTATION CODE = 6

 KEY CODE 1 = 100

 KEY CODE 2 = 110

 TEXT LINE = Line Length := 0


 STATUS = 0 DATA ITEM WRITTEN INTO LIST!




          List Initialise          (I)
          Get a Record             (>)
          Return a Record          (<)
          Write a Record           (W)
          Read a Record            (R)
          Move Pointer             (M)
          Display List             (D)
          List Structure           (S)

          Quit this Program        (Q) ? D


      Logical List Pointer =   2
      Maximum Records      =  10

  LOCATION:      1     2
 INDEN CODE:     2     6
 KEY CODE 1:   900   100
 KEY CODE 2:   901   110
 The Text Lines below are in order of location

 X := X + 1
 Line Length := 0


 Hit any Key to Continue...
```

```
          List Initialise          (I)
          Get a Record             (>)
          Return a Record          (<)
          Write a Record           (W)
          Read a Record            (R)
          Move Pointer             (M)
          Display List             (D)
          List Structure           (S)

          Quit this Program        (Q) ? S


    MAXIMUM ITEMS          =  10    LINK LIST SIZE          =   2
    LINK LIST ROCK         =   1    SPACE LIST ROCK         =   3
    LOGICAL LIST POINTER   =   2    PHYSICAL LIST POINTER   =   2

 LOCATION :          1    2    3    4    5    6    7    8    9   10
 FORWARD POINTER:    2   11    4    5    6    7    8    9   10   11
 BACKWARD POINTER:   0    1    2    3    4    5    6    7    8    9
 INDEN CODE:         2    6    0    0    0    0    0    0    0    0
 KEY_CODE 1:       900  100    0    0    0    0    0    0    0    0
 KEY_CODE 2:       901  110    0    0    0    0    0    0    0    0

X := X + 1
Line Length := 0




Hit any key to Continue...




          List Initialise          (I)
          Get a Record             (>)
          Return a Record          (<)
          Write a Record           (W)
          Read a Record            (R)
          Move Pointer             (M)
          Display List             (D)
          List Structure           (S)

          Quit this Program        (Q) ? Q
```

A.6 Line Editor

The display of the Line Editor test program is shown here. Only the Window Screen (enclosed in a box) is editable. The prompts preceding this point are required user *responses needed* for initializing the Line Editor. User responses are underscored and highlighted. In the Window Screen, the key word is highlighted and underscored here.

When the Line Editor is entered, the cursor is positioned under the "T" of "CONDITION". In its place, a "v" is typed, and the ENTER key is then depressed. (The cursor, Cursor 2, is shown as a highlighted underscored character) The editor is then exit. The lines following the Window Screen are outputs generated by the test program to give the user an indication of the final value of certain variables.

****THE LINE EDITOR****

KEY_TEXT = Until
IN_STRING = <CONDITION>
WSP = 15
START_COL = 3

```
|-------------------------------------------------------------|
|  Until <CONDIvION>                                          |
|-------------------------------------------------------------|
```

OUT_STRING = <CONDIvION>
LENGTH OUT_STRING = 11
OUT_KEY = -1
WINDOW SCREEN CURSOR POSITION = 16

APPENDIX B: FILENAMES AND DOCUMENTATION DETAILS

The package routines are stored on a number of diskettes. The disk number, together with its directory (if relevant) is specified below. Each routine on the *disk* is listed, giving the filename used; the routine name and the routine function. A list of filename extensions and their associated meanings follows:

.DIR -- A directory for the PDL generator routines
.PAS -- A Pascal routine for the PDL generator
.*SYS* -- A *system* table for the PDL generator
.COD -- A coded file used by the PDL generator
.PDL -- A formatted file created by the PDL generator
.BAT -- A batch file used by the Disk Operating System
.OBJ -- An object file created after compilation of a Pascal module
.EXE -- An executable file created after linking of a Pascal module
.DOC -- A document file containing the documentation of the PDL generator and created with IBM-Wordstar
.BAK -- A *backup file for the documentation* also created by IBM-Wordstar.

B.1 Resources

Terminal_Resource Disk: R1; Directory: TERMINAL.DIR

File Name	Routine Name	Function
TERM_RES.PAS	TERMINAL_RESOURCE	The resource module
TERM_OPS.PAS	--	The resource operators
KEYBOARD.PAS	KBD_GET	*Gets a character without echo*
BELL .PAS	BELL	Sounds terminal bell
BLINK_ON.PAS	BLINK_ON	Turns blinking on
BOLD_ON .PAS	BOLD_ON	Turns highlighting on
RVID_ON .PAS	RVID_ON	Turns reverse video on
UDSC_ON .PAS	UDSC_ON	Turns underscoring on
RESTORE .PAS	RESTORE	Reverts attributes to normal
SET_CP .PAS	SET_CP	Sets cursor position
READ_CP .PAS	READ_CP	Reads cursor position
HOME .PAS	HOME	Sends cursor HOME
CLR_SCR .PAS	CLR_SCR	Clears screen
CLR_LINE.PAS	CLR_LINE	Clears line
UP_SCR .PAS	UP_SCR	Cursor up screen
DN_SCR .PAS	DN_SCR	Cursor down screen
CUR_RIGH.PAS	CUR_RIGHT	Cursor right
CUR_LEFT.PAS	CUR_LEFT	Cursor left
SCR_FORM.PAS	*SCR_FORMAT*	Formats the screen
MS_CLEAR.PAS	MS_CLEAR	Clears Main Screen
MS_C_ON .PAS	MS_CUR_ON	Main Screen cursor on
MS_C_OFF.PAS	MS_CUR_OFF	Main Screen cursor off
MS_WRITE.PAS	MS_WRITE	Main Screen write
WS_CLEAR.PAS	WS_CLEAR	Clears Window Screen

Terminal Resource Disk: R1; Directory: TERMINAL.DIR

File Name	Routine Name	Function
WS_CPSET.PAS	WS_ASET_CP	Window Screen set cursor
WS_RV_WR.PAS	WS_RV_WRITE	W.S. reverse video write
WS_HL_WR.PAS	WS_HL_WRITE	W.S. highlighted write
WS_LO_WR.PAS	WS_LO_WRITE	W.S. normal write
PS_CLEAR.PAS	PS_CLEAR	Clears Prompt Screen
PS_WRITE.PAS	PS_WRITE	Prompt Screen write
FS_CLEAR.PAS	FS_CLEAR	Clears Function Screen
FS_WRITE.PAS	FS_WRITE	Function Screen write

Definition Tables Disk: R1; Directory: DEF_T.DIR

File Name	Routine Name	Function
DT_RES .PAS	DT_RESOURCE	The resource module
DT_OPS .PAS	--	The resource operators
DT_INIT .PAS	DT_INIT	Initializes Definition Table
DT_A_RE .PAS	DT_A_READ	A Reads the Definition Table
DT_B_RE .PAS	DT_B_READ	B Reads the Definition Table
DT_TEST .PAS	DT_TEST	Definition Table test program
DT_STRUC.PAS	DT_STRUCTURE	D.T. test structure routine
DT .SYS	--	Definition Table file

Key Code Table Disk: R1; Directory: KEY_C_T.DIR

File Name	Routine Name	Function
KCT_RES .PAS	KCT_RESOURCE	The resource module
KCT_OPS .PAS	--	The resource operators
KCT_INIT.PAS	KCT_INIT	Initializes Key Code Table
KCT_READ.PAS	KCT_READ	Reads the Key Code Table
KCT_TEST.PAS	KCT_TEST	Key Code Table test program
KCT_STRU.PAS	KCT_STRUCTURE	K.C.T. test structure routine
KCT .SYS	--	Key Code Table file

Prompt Table Disk: R1; Directory: PROMPT_T.DIR

File Name	Routine Name	Function
PT_RES .PAS	PT_RESOURCE	The resource module
PT_OPS .PAS	--	The resource operators
PT_INIT .PAS	PT_INIT	Initializes Prompt Table
PT_READ .PAS	PT_READ	Reads the Prompt Table
PT_TEST .PAS	PT_TEST	Prompt Table test program
PT_STRUC.PAS	PT_STRUCTURE	P.T. test structure routine
PT .SYS	--	Prompt Table file

Line Linked List Disk: R2; Directory: LINE_LL.DIR

File Name	Routine Name	Function
LIST_RES.PAS	LINE_LL_RESOURCE	The resource module
LIST_OPS.PAS	--	The resource operators
LSBAA .PAS	LIST_INITIALISE	Initializes Line Linked List
LSBAB .PAS	LIST_GET_RECORD	Fetches record
LSBAC .PAS	LIST_RETURN_RECORD	Returns record
LSBAD .PAS	LIST_WRITE_RECORD	Writes record
LSBAE .PAS	LIST_READ_RECORD	Reads record
LSBAF .PAS	LIST_MOVE_POINTER	Moves list pointer

Line Linked List Disk: R2; Directory: LINE_LL.DIR

File Name	Routine Name	Function
LSBAJ .PAS	LIST_LOG_INFO	Returns Logical List Pointer
LSBAK .PAS	LIST_LOG_STRING	Returns the ordered list
LSBTST .PAS	LSBTST	List test program
LSBAH .PAS	LIST_DATA_DISPLAY	Logical display test routine
LSBAI .PAS	LIST_STRUCTURE_DISPLAY	Physical displ. test routine

File Linked List Disk: R2; Directory: FILE_LL.DIR

File Name	Routine Name	Function
FLL_RES .PAS	FLL_RESOURCE	The resource module
FLL_OPS .PAS	--	The resource operators
LSBAA .PAS	FLL_INITIALISE	Initializes File Linked List
LSBAB .PAS	FLL_GET_RECORD	Fetches a record
LSBAC .PAS	FLL_RETURN_RECORD	Returns a record
LSBAD .PAS	FLL_WRITE_RECORD	Writes a record
LSBAE .PAS	FLL_READ_RECORD	Reads a record
LSBAF .PAS	FLL_MOVE_POINTER	Moves list pointer
LSBAJ .PAS	FLL_LOG_INFO	Returns Logical List Pointer
LSBTST .PAS	LSBTST	List test program
LSBAH .PAS	LIST_DATA_DISPLAY	Logical display test routine
LSBAI .PAS	LIST_STRUCTURE_DISPLAY	Physical displ. test routine

B.2 Processes

Line Editor Disk: P1; Directory: LINE_ED.DIR

File Name	Routine Name	Function
LINE_ED .PAS	LINE_EDITOR	Line Editor Process module
LOAD .PAS	LOAD	Initializes the Line Editor
CUR_FWD .PAS	MOVE_CUR_F	Moves Cursor 2 forward
CUR_BWD .PAS	MOVE_CUR_B	Moves Cursor 2 backward
CUR_HOME.PAS	WS_HOME	Moves Cursor 2 HOME
END_O_L .PAS	END_OF_LINE	Moves Cursor 2 to end of ln
ENT_TEXT.PAS	ENTER_TEXT	Text enter mode
INSERT .PAS	INSERT	Character insert routine
DELETE .PAS	DELETE	Character delete routine
DEL_EOL .PAS	ERASE_EOL	Erases to end of line
DUMP .PAS	DUMP	Dumps Line Editor contents
TEST .PAS	TEST	Line Editor test program

Base Level Disk: P1; Directory: FILE_ED.DIR

File Name	Routine Name	Function
FL_SCROL.PAS	[illegible]LL	Base Level Process module
CUR_UP .PAS	[illegible]	Moves Cursor 1 up
CUR_DOWN.PAS	[illegible]N	Moves Cursor 1 down
PAGE_BWD.PAS	[illegible]_BWD	Scrolls back one page
PAGE_FWD.PAS	PAGE_FWD	Scrolls forward one page
TOP_O_FL.PAS	TOP_OF_FILE	Moves to top of file
BOT_O_FL.PAS	BOT_OF_FILE	Moves to bottom of file
CUR_T_LN.PAS	CUR_TO_LINE	Moves cursor to given line
EDIT_LN .PAS	EDIT_LINE	Editing via Line Editor
INSERT .PAS	INSERT	Insert mode gateway

Base Level Disk: P1; Directory: FILE_ED.DIR

File Name	Routine Name	Function
FORMATER.PAS	FORMATTER	Formatter process

Data item Insert mode Disk: P1; Directory: DATA_DES.DIR

File Name	Routine Name	Function
DATA_DES.PAS	DATA_DESCRIPTION	Data Description Insert module
ARR_LOAD.PAS	ARRAY_LOAD	Initializes definition array
CONDIT .PAS	CONDITIONS	Conditions of Defn. T. choice
PLACEMNT.PAS	PLACEMENT	Places new data item in file
FORMATER.PAS	FORMATTER	Formatter process

Construct Insert mode Disk: P1; Directory: ALGO.DIR

File Name	Routine Name	Function
ALGORITH.PAS	ALGORITHM	Construct Insert module
GET_CON .PAS	GET_CONSTRUCT	*Fetches construct from table*
INDENT .PAS	INDENT	Calculates indentation
FORMATER.PAS	FORMATTER	Formatter process
CUR_UP .PAS	CUR_UP	Moves Cursor 1 up
CUR_DOWN.PAS	CUR_DOWN	Moves Cursor 1 down
PAGE_BWD.PAS	PAGE_BWD	Scrolls back one page
PAGE_FWD.PAS	PAGE_FWD	Scrolls forward one page

Line Insert mode Disk: P1; Directory: INS_LN.DIR

File Name	Routine Name	Function
INS_LN .PAS	INSERT_LINE	Line Insert module
INDENT .PAS	*INDENT*	*Calculates indentation*
FORMATER.PAS	FORMATTER	Formatter process

Main Program Disk: M1

File Name	Routine Name	Function
MAIN .PAS	MAIN	Main Program
FORMATER.PAS	FORMATTER	Formatter process
FIL_LOAD.PAS	FILE_LOAD	File load routine
FOR_DUMP.PAS	FILE_F_DUMP	Formatted file dump
UNF_DUMP.PAS	FILE_UF_DUMP	Unformatted file dump

B.3 *Documentation*

Preface Disk: D1

File Name	Function
FRONT .DOC	The front pages of the document
PAPER .DOC	A 14 page summary document

Literature Survey Disk: D1

File Name	Function
LIT .DOC	A 26 page document surveying editors

User's Manual Disk: D2

File Name	Function
CHAP_1 .DOC	Chapter 1
CHAP_2 .DOC	Chapter 2
CHAP_3 .DOC	Chapter 3
CHAP_4 .DOC	Chapter 4
APP_A .DOC	Appendix A
APP_B .DOC	Appendix B
APP_C .DOC	Appendix C

Designer's Reference Disk: D3

File Name	Function
CHAP1 .DOC	Chapter 1
CHAP2 .DOC	Chapter 2
CHAP3 .DOC	Chapter 3
CHAP4 .DOC	Chapter 4
CHAP5 .DOC	Chapter 5
CHAP6 .DOC	Chapter 6
CHAP7 .DOC	Chapter 7
APP_A .DOC	Appendix A
APP_B .DOC	Appendix B
APP_C .DOC	Appendix C
APP_D .DOC	Appendix D
REFS .DOC	References

APPENDIX C: SYSTEM TABLES

C.1 Definition Table

The 80-character key definition line has been split into two equal halves for the sake of convenience. Section 3.2.3 explains the method of coding used below.

DD	Seg	Constr	Ins	Ln	RET
50	1	1	3	3	
60	8	8	8	8	
70	10	10	10	10	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
4000	0	0	9	9	
Yes			No		

1001	5	5	1	1	
0	0	0	0	0	
2000	1	1	3	3	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	
0	0	0	0	0	

Const:	Var:	Type D	CONT	RET
1	1	4	4	
2	1	4	4	
3	3	3	3	
0	0	0	0	
0	0	0	0	
0	0	0	0	
0	0	0	0	
0	0	0	0	
3000	1	4	3	
4000	2	2	2	

Boole:	Integ:	Real:	Chara: CONT	Others RET
11	1	5	5	
12	1	5	5	
13	1	5	5	
14	1	5	5	
15	3	4	4	
0	0	0	0	
0	0	0	0	
0	0	0	0	

```
3000  1     4     5     4
4000  1     1     3     3
  Singl:  Array:
                            CONT    RET
21    1     1     6     6
22    1     1     6     6
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
3000  1     4     6     5
4000  1     1     4     4
  Globl:  Perma:  Exter:  Local:
                          CONT    RET
31    6     6     7     7
32    6     6     7     7
33    6     6     7     7
34    6     6     7     7
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
3000  6     4     7     6
4000  1     1     5     5
  Yes               No

1000  7     7     1     1
0     0     0     0     0
2000  2     2     2     2
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
0     0     0     0     0
  If-t    If-t-e  While   Repeat  Case
  Case-e  Cobegn  Get     Put     RET
51    9     9     1     1
52    9     9     1     1
53    9     9     1     1
54    9     9     1     1
55    9     9     1     1
56    9     9     1     1
57    9     9     1     1
58    9     9     1     1
59    9     9     1     1
4000  5     5     1     1
  Page B  Page F  Top F   Bot F   To lin
  Edit l  Insert
61    0     0     9     9
62    0     0     9     9
63    0     0     9     9
64    0     0     9     9
65    0     0     9     9
70    0     0     9     9
80    0     0     1     1
0     0     0     0     0
0     0     0     0     0
```

```
0      0      0      0      0
  End

70     0      0      1      1
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
0      0      0      0      0
```

C.2 Key Code Table

See Section 3.3.3 for the coding method used below.

```
N 1      0 Constants:*
N 2      0 Variables:*
N 11     5 Boolean:*
N 12     5 Integer:*
N 13     5 Real:*
N 14     5 Character:*
N 21     8 Single:*
N 22     8 Array:*
N 31    11 Global:*
N 32    11 Permanent:*
N 33    11 External:*
N 34    11 Local:*
N 0      0 *
Y 40    14 *
Y 100    0 If *
N 101    2 then:*
N 102    2 else:*
N 103    0 End If:*
Y 200    0 While *
N 201    0   Do:*
N 202    0 End While:*
N 300    0 Repeat:*
Y 301    0 Until *
Y 400    0 Case *
N 401    0   Of:*
N 402    0 End Case:*
Y 900    0 <CONSTRUCT>*
Y 901    2 <CONSTRUCT>*
Y 902    4 <CONSTRUCT>*
Y 910    0 <CONDITION>*
Y 920    0 <VARIABLE>*
Y 950    0 *
Y 951    2 *
Y 952    4 *
N 500    0 Cobegin:*
N 501    0 Coend:*
N 600    0 Get:*
N 601    0 End Get:*
N 700    0 Put:*
N 701    0 End Put:*
```

```
Y -1    0 Program *
Y -2    0 <PROGRAM NAME>*
Y -11   0 Procedure *
Y -12   0 <PROCEDURE NAME>*
Y -21   0 Inputs: *
Y -22   0 Outputs: *
Y -30   0 External Procedures: *
Y -31   0 <PARAMETER/(S)>*
N 814   0 Begin:*
N 815   0 End:*
N 816   0  * None **
N 817   0 End Program:*
N 818   0 End Procedure:*
```

C.3 Prompt Table

See Section 3.4.3 for the coding of the Prompt Table.

```
Define the Data Item using the Line Editor and Function Keys.
Do you want to abandon this definition ?
This function key is as yet undefined...
Use the function keys to define this field.
Definition ABANDONED !
Do you want to accept this definition ?
Definition Accepted.
Choose one of the following Templates using the Function Keys.
Remember to replace any outstanding Placeholders at a later stage
Insert required lines of text by successive Carriage Returns.
One line was deleted.
```

C.4 Construct Table

Two lines are used to represent a construct. The first line contains a single key code which distinguishes a construct from another. This is the code which is generated when the function key corresponding to this template is depressed in Construct Insert mode. The second line, used to describe the construct, contains a series of key words which make up the template. An asterisk between key words indicates a new line in the template. A maximum of three codes are permitted on a single line (an initial key word followed by a placeholder followed by a second key word). To distinguish the end of a construct, a 9999 code is used. In this way, any combination of key words can be put together and new templates thus constructed. In the table shown below, for example, the first template represents an If-then construct.

```
51
100 910*101*902*103*9999
52
100 910*101*902*102*902*103*9999
53
200 910 201*901*202*9999
54
300*901*_  910*9999
```

```
55
400 920 401*902*402*9999
56
400 920 401*902*102*902*402*9999
57
500*901*501*9999
58
600*901*601*9999
59
700*901*701*9999
60
-1 -2*-30 -31*814*901*815*817*9999
61
-11 -12*-21 -31*-22 -31*-30 -31*814*901*815*818*9999
```

In the Definition Table, State 8 is the Construct Insert mode. From here, it can be seen that each construct has a key code which corresponds to one of the key codes listed in the Construct Table above. The last two templates, however, are used on startup for a program and a procedure block respectively.

APPENDIX D: STORAGE FILES

Figure 2 shows a coded file (ie. with extension ".COD") output by the PDL generator package, while figure 1 is the same file, but in prettyprinted or formatted style. The coded file is always twice as long as the formated file due to its use of two lines of *data to represent a single formatted text line.*

The third line of the PDL program in figure 1 (also pointed to in figure 2) is taken as an illustrative example. The coded file contains firstly an indentation level. This integer value, when multiplied by the current tab setting (set in the program as two places) yields a number of spaces. (ie. 1 x 2 = 2) The first key code is searched for in the Key Code Table (see Appendix C). From the Key Code Table, it can be found that this line is editable (due to the <CONDITION> placeholder); that a relative indentation of zero is associated with it; and that the key word is "While: ".

Now, the indentati[illegible] be calculated by adding the value obtained from the i[illegible]ion level to the absolute indentation found above. Thus, th[illegible]e must be indented by two spaces (ie. 2 + 0 = 2). The key word "While: " then follows. The <CONDITION> placeholder is found in the second line of the coded file as a text string. This string is user-editable and is written alongside the "While: " key word. The final key word is found by searching the Key Code Table for the second key code. Editability and indentation values obtained here are ignored. Only the key word "do:" is used to complete the line in the formatted file.

```
                                                    0   -12     0
                                                Test
                                                    0   814     0
    Procedure  Test
    Begin:                                  -->     1   200   201
-->   While <CONDITION> Do:                 -->  <CONDITION>
        <CONSTRUCT>                                 1   951     0
      End While:                                <CONSTRUCT>
    End:                                            1   202     0
    End Procedure:
                                                    0   815     0
Figure 1: A formatted PDL program
                                                    0   818     0
```

Figure 2: The coded PDL program

REFERENCES

1. Bassanino,A.P. (1984): "Software System Design Aid", A Function-key Driven PDL Generator, a fourth year BSc (Eng) design project (Project 3B/84) submitted for Course 19419 in the Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, November 1984.

2. Bassanino,A.P. (1985a): A Function-key driven Syntax-directed Editor for Software Systems Design, "Literature Survey", a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

3. Bassanino,A.P. (1985b): A Function-key driven Syntax-directed Editor for Software Systems Design, "User's Manual", Version 1.0, a document submitted for an MSc (Eng) degree in the department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985-1986.

4. Caine,S.H. and Gordon,E.K. (1975): "PDL -- a tool for software design", Proceedings of the National Computer Conference, 1975, pp.271-276.

5. Capers,J.T. (1984): "Reusability in Programming: A Survey of the State of the Art", IEEE Transactions on Software Engineering, Vol. SE-10, No.5, September 1984.

6. Data General Corporation (1979a): "Command Line Interpreter User's Maunal", Advanced Operating System (AOS), Third Revision, June 1979.

7. Data General Corporation (1979b): MP/Pascal Programmer's Reference, First Edition, July 1979.

8. Data General Corporation (1979c): "Dasher Display Terminals, Models D100/D200", User Reference Series, First Edition, October 1979.

9. Data General Corporation (1980): "SED Text Editor User's Manual", Advanced Operating System / Virtual Storage (AOS/VS), Revision 1.0, November 1980.

10. IBM (1981): "Pascal Compiler" for the IBM Personal Computer, Computer Language Series, by Microsoft Inc., First Edition, August 1981.

11. IBM (1983): Disk Operating System (DOS) for the IBM Personal Computer, by Microsoft Inc., Version 2.0, May 1983.

12. Jensen,K. and Wirth,N. (1974): PASCAL User Manual and Report, Springer-verlag, Second Edition, November 1974.

13. Master,B.A. (1984): "Software System Design Aid", Computer Assisted Software System Design Aid, a fourth year BSc (Eng) design project (Project 3A/84) submitted for Course 19419 in the Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, November 1984.

14. Miller,E.F. (1984): "Software Testing Technology: An Overview", Chapter 16, Handbook of Software Engineering, edited by Vick,C.R. and Ramamoothy,C.V., Van Nosrand Reinold, 1984.

15. Myers,G.J. (1975): Relisble software through composite design, Pertocelli/charter, New York, 1975.

16. Parnas,D.L. (1972): "On the Criteria to be used for in Decomposing Systems into Modules", Communications of the ACM, Vol.15, No.12, December 1972.

17. Shankar,K.S. (1984): "Data Types: Types, structures and abstractions", Chapter 12, Handbook of Software Engineering, edited by Vick,C.R. and Ramamoothy,C.V., Van Nosrand Reinold, 1984.

18. Sommerville,I. (1982): Software Engineering, Addison Wesley International Computer Science Series, 1982.

19. Vosbury,N.A. (1984): "Process Design", Chapter 25, Handbook of Software Engineering, edited by Vick,C.R. and Ramamoothy,C.V., Van Nosrand Reinold, 1984.

20. Walker,A.J. (1984): Structured Information Processing System Design, Internal publication of the Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1984.

21. Walker,A.J. (1985): A Screen Managed Environment for the Rapid Prototyping and Development of Interactive Application Programs, Internal publication of the Department of Electrical Engineering, University of the Witwatersrand, Johannesburg, 1985.