

# **A WYSIWYG EDITOR FOR SQL ENABLED HYPERTEXT MARKUP LANGUAGE (HTML)**

**A major project report submitted  
in partial fulfillment of the requirements  
for the award of the degree of**

**Master of Technology  
in  
*Artificial Intelligence***

**BY**

**Sushish Saha  
&  
N. Raja Subrahmanyam**



**Department of Computer/Information sciences  
School of Mathematics & Computer/Information Sciences**

**University of Hyderabad**

**Hyderabad - 500 046, INDIA.**

**January 1997.**

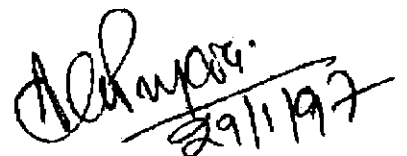
# CERTIFICATE

This is to certify that the thesis entitled **A WYSIWYG editor for SQL enabled Hypertext Markup Language (HTML)** being submitted by Sushish Saha & N.Raja Subrahmanyam in partial fulfillment of the requirements for the award of **Master of Technology in Artificial Intelligence** at University of Hyderabad is a record of the bonafide work carried out by them under my guidance and supervision. The matter embodied in this thesis has not been submitted in any form to any University or Institution for the award of any degree or diploma.



**Dr. P. V. Reddy**

(Project Supervisor)



**Prof. Arun K. Pujari**

Head of Department

Department of CIS.



**Prof. C. Musili,**

Dean,

School of Mathematics, Computer/Information Sciences,

University of Hyderabad.

# ACKNOWLEDGEMENTS

First and foremost we would like to thank our guide **Dr. P.V.Reddy** for suggesting the problem, and providing necessary guidance.

Our head of the department **Prof. A.K.Pujari** has been a source of inspiration for us. He also provided us with the computing facilities whenever we needed. Our deepest gratitude is for him.

We also wish to thank all the **staff members** of the A.I lab and the computer centre for their constant encouragement.

The constant criticisms and the suggestions of our **friends** and **classmates** have been invaluable for us. We are thankful to them.

And finally we would like to thank our beloved **parents** for showing us the right path through out our life.

**Sushish Saha**

**N. Raja Subrahmanyam**

# ABSTRACT

We present a **WYSIWYG** (What You See Is What You Get) type editor, which stores **WYSIWYG** output in **HTML** format. In the front end user can build up his hypertext documents in **WYSIWYG** mode and in the back end each hypertext page (a hypertext document contains one or more pages) gets stored in the form of **HTML** document. User can also execute his database queries through the same hypertext documents created by the editor.

In this report we present the design and development of four programs: 1. **HTML DOCUMENT EDITOR**, 2. **HTML EDITOR**, 3. **BROWSER**, 4. **DATABASE APPLICATION**.

The description of these programs is as follows.

## **HTML DOCUMENT EDITOR:**

This is a simple text editor, which helps the users in building their **HTML** documents directly.

## **HTML EDITOR:**

This editor works in two modes:

In the first mode of operation, user can create an **HTML** document just by clicking the buttons according to the organization which is provided in the **on-line help** of the editor. In this mode, the editor works in front end, which means no process is in background. User can only see the generated **HTML** code in the current window.

In the second mode of operation, the editor works in front end as well as back end. In the front end, users can create a hypertext document in **WYSIWYG** mode and **WYSIWYG** output will be stored in equivalent **HTML** code in the back end. Users are also allowed to take a hard copy of the hypertext documents created by them.

## **BROWSER :**

The browser interprets HTML documents and turns it into a hypertext document. For generating each page of a hypertext document the browser needs the corresponding HTML document to be interpreted. It also executes SQL queries on the database application through a hypertext document. For this purpose we defined syntax which is used for action by the browser.

## **DATABASE APPLICATION:**

To demonstrate the execution of SQL queries on a database application through hypertext, we developed a database application related to the maintenance of the research grants at the university of Hyderabad.

# CONTENTS

## **Chapter 1 . INTRODUCTION**

<b>1.1 Problem Statement</b>	<b>1</b>
<b>1.1.1 System Requirements</b>	<b>2</b>
<b>1.2 Review of hypertext documents &amp; HTML</b>	<b>2</b>
<b>1.2.1 Hypertext document</b>	<b>2</b>
<b>1.2.2 HTML</b>	<b>4</b>
<b>1.2.3 Relation ship between HTML documents         and a hypertext document</b>	<b>6</b>
<b>1.3 What is a browser?</b>	<b>7</b>
<b>1.4 WYSIWYG output</b>	<b>7</b>
<b>1.5 Motivation</b>	<b>7</b>
<b>1.6 Our achievements</b>	<b>9</b>
<b>1.7 Outline of the thesis</b>	<b>9</b>

## **Chapter 2 . DELPHI SYSTEM OVERVIEW**

<b>2.1 Introduction</b>	<b>11</b>
<b>2.2 What is Delphi ?</b>	<b>11</b>
<b>2.3 Elements in Delphi's IDE</b>	<b>12</b>
<b>2.4 Structure of the object Pascal unit</b>	<b>13</b>
<b>2.5 Help file creation process</b>	<b>14</b>
<b>2.6 Components used in this project</b>	<b>14</b>
<b>2.7 Conclusions</b>	<b>20</b>

## **Chapter 3 . HTML SYNTAX**

<b>3.1 Introduction</b>	<b>21</b>
<b>3.2 Notion of HTML tags</b>	<b>21</b>
<b>3.3 HTML categories and their respective tags</b>	<b>24</b>
<b>3.4 Conclusions</b>	<b>35</b>

## **Chapter 4 . HTML DOCUMENT EDITOR**

<b>4.1 Introduction</b>	<b>36</b>
<b>4.2 MDI Application</b>	<b>36</b>
<b>4.3 Design of the HTML document editor</b>	<b>37</b>
<b>4.4 Implementation details of the editor</b>	<b>38</b>
<b>4.4.1 Components used and their properties</b>	<b>38</b>
<b>4.4.2 Event handlers of the components</b>	<b>40</b>
<b>4.5 Conclusions</b>	<b>42</b>

## **Chapter 5 . HTML EDITOR**

<b>5.1 Introduction</b>	<b>43</b>
<b>5.2 Modes of operation</b>	<b>43</b>
<b>5.3 Design and Implementation Details</b>	<b>44</b>
<b>5.3.1 Design of the editor ( Operational Mode 1)</b>	<b>44</b>
<b>5.3.2 Implementation details ( Operational Mode 1)</b>	<b>45</b>
<b>5.3.3 Design of the editor ( Operational Mode 2)</b>	<b>50</b>
<b>5.3.4 Implementation of the editor ( Operational Mode 2)</b>	<b>53</b>
<b>5.4 Conclusions</b>	<b>61</b>

## **Chapter 6. BROWSER**

<b>6.1 Introduction</b>	<b>62</b>
<b>6.2 Design of the browser</b>	<b>62</b>
<b>6.3 Implementation details</b>	<b>63</b>
<b>6.3.1 Browser's actions according to each tag</b>	<b>63</b>
<b>6.3.2 Components used</b>	<b>67</b>
<b>6.3.3 Event handlers and their main functions</b>	<b>68</b>
<b>6.3.4 Important functions</b>	<b>69</b>
<b>6.4 Query execution</b>	<b>71</b>
<b>6.4.1 Syntax used</b>	<b>71</b>
<b>6.4.2 Browser's action</b>	<b>71</b>
<b>6.5 Conclusions</b>	<b>72</b>

## **Chapter 7. DATABASE APPLICATION**

<b>7.1 Introduction</b>	<b>73</b>
<b>7.2 Problem specification</b>	<b>73</b>
<b>7.2.1 Tables used</b>	<b>73</b>
<b>7.2.2 Functions to be performed on the database</b>	<b>75</b>
<b>7.2.3 Reports to be generated</b>	<b>76</b>
<b>7.2.4 Queries</b>	<b>77</b>
<b>7.3 Design of the database</b>	<b>78</b>
<b>7.4 Implementation Details</b>	<b>78</b>
<b>7.5 Query execution through hypertext</b>	<b>80</b>
<b>7.5.1 Syntax</b>	<b>80</b>
<b>7.5.2 A database query and its execution in Delphi</b>	<b>81</b>
<b>7.5.3 Anatomy of the syntax defined by us</b>	<b>82</b>
<b>7.5.4 General algorithm for execution of         the queries through hypertext</b>	<b>82</b>
<b>7.6 Conclusions</b>	<b>83</b>



# **Chapter 8 . CONCLUSIONS AND FUTURE ENHANCEMENTS**

**8.1 Conclusions 84**

**8.2 Future enhancements 85**

**APPENDIX A RAISE 86**

**APPENDIX B RESULTS 93**

**APPENDIX C DELPHI's IDE 99**

**REFERENCES 100**

## **INTRODUCTION**

### **1.1 Problem Statement**

Our main aim is to develop a tool that assists users in generating hypertext documents in an easy manner. They can as well execute SQL queries on the database applications through their hypertext documents.

To realize our goal we have implemented four modules: HTML document editor, HTML editor, Browser and a database application. The details of the modules are as follows. Firstly, HTML document editor is a text editor which allows the user to create more than one HTML document simultaneously in different windows and can switch over from one window to another.

Secondly, HTML editor works in two modes. In first mode, a HTML document can be created without prior knowledge of HTML syntax. The second mode is the WYSIWYG [1.4] mode. Without specifying the structure, content, and behavior of the hypertext document in HTML, it allows user to generate hypertext document directly in the front end and also take a hard copy of the hypertext pages. This is the most important feature provided by our implementation and hence our title 'A WYSIWYG editor for SQL enabled Hypertext Markup Language(HTML)'.

Thirdly, the browser program interprets HTML documents into a hypertext document. This is not a WWW browser [1.3] but one which serves some of the functions of a HTML browser.

Finally to execute the SQL queries, we have developed a database application related to the maintenance of the research grants at the University of Hyderabad. This

database application is used to demonstrate how database queries can be executed through the hypertext documents by incorporating our syntax for query execution in HTML documents. For specification of the database we resort to RAISE [Appendix A] as a specification language.

### **1.1.1 System requirements**

To solve the above problem we have chosen Delphi Integrated Development Environment (IDE) [chapter 2] as our application development environment. This programming environment supports all the visual components that are required for our solution.

## **1.2 Review of hypertext document and HTML**

### **1.2.1 Hypertext document**

**Hypertext** is a representation of non-structured information, with built-in references called links to related information. This is useful because users can just click on the links and access the required information. The user need not go to each site and collect the information.

What differentiates a hypertext document from a book is how it is organized; a hypertext document is **nonlinear**. With a book, you read from one page to the next. In a hypertext document, you can randomly jump from one topic to another, and you can perform fast searches on keywords to locate a particular topic.

A hypertext document is made up of one or more “pages”, often referred as **hypertext pages**. A page can actually be as long as you want to be—the equivalent of the several printed pages. Each hypertext page dedicates itself to a “topic” that can consist of text, references to other topics, or any number of things.

Fig. 1.1 is a simplified representation of a symbolic hypertext “page”. The hypertext page consists of two parts : **i. The header and ii. The page contents.**

## **i. Header**

The header consists of three items : **label, title and keywords.**

The header contains locator information. Because hypertext is a non-linear system, there must be a way to jump from one topic page to another—in other words, each page must have an “address” where the hypertext system can find it. This address is provided in the form of a unique **label** assigned to each topic page.

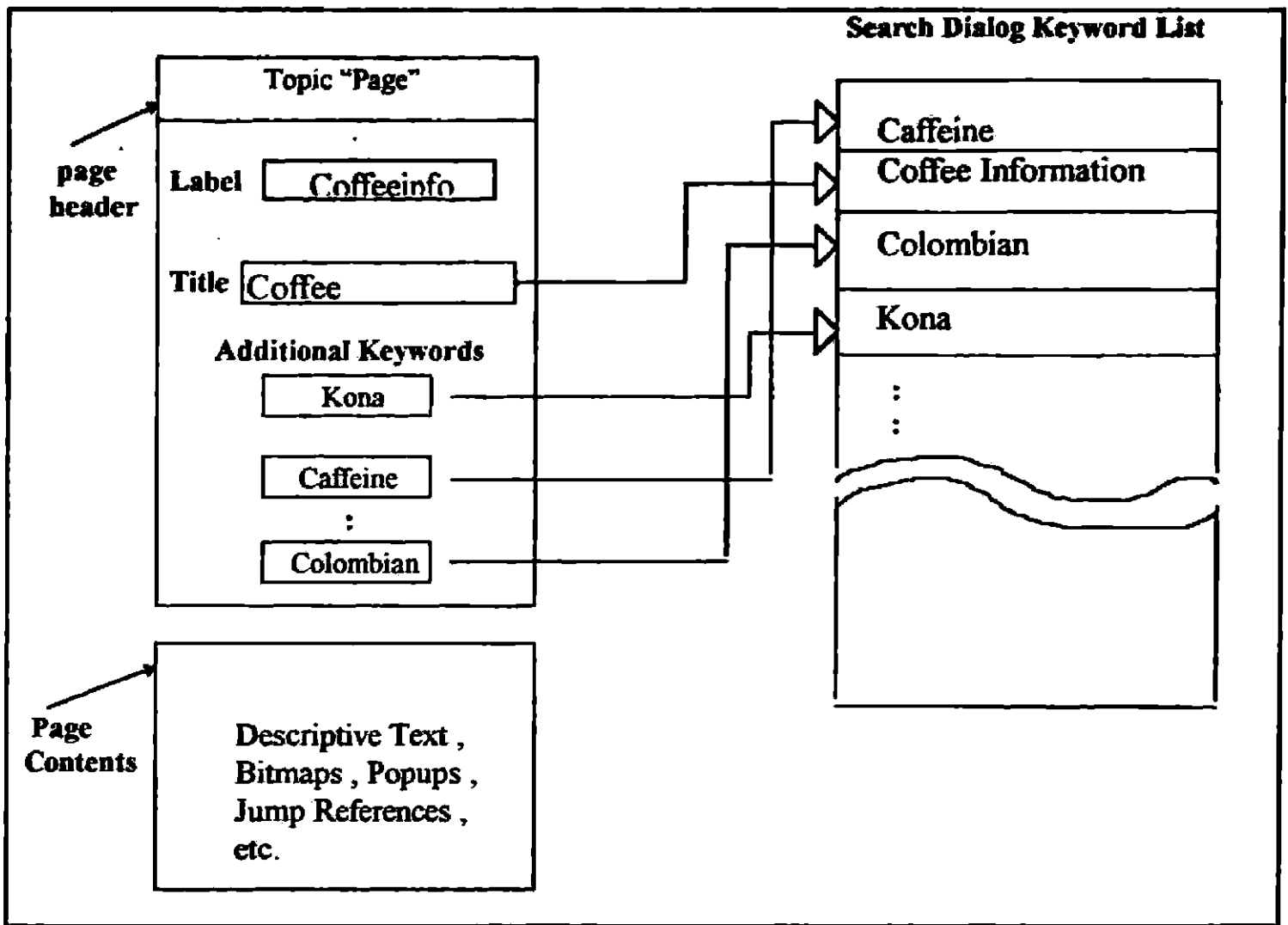
The next item in the header is the page’s title. This title appears at the top of the page when the page is displayed in the Help window, and the same title is also automatically included in the keyword list available to the help system’s search dialog.

The final item in the header is a list containing zero or more **keywords**. These keywords are also made available to the search dialog (which, as the figure shows, always displays items in alphabetical order). Instead of limiting a search to exact topic titles, providing keywords related to topics gives readers a much greater chance of finding what interests them. It is also possible to assign the same keyword to more than one topic.

## **ii. Page contents**

The **page contents**, which appear in the lower portion of the figure, contains the topic message. That may be any combination of descriptive text, bitmaps, references to popup messages, or jump references to other topics.

**Popup messages** are usually used to display a box containing a definition of a term that is highlighted in text. **Jump references** are the most interesting of all. They provide a means to relate a screen “**hot spot**” containing highlighted text to the address (that is, the **label**) of another topic page. By clicking on the highlighted text, the hypertext system brings up the referenced topic.



**Fig. 1.1 Symbolic representation of a topic page.**

### **1.2.2 HTML (Hypertext Markup Language)**

**HTML (Hypertext Markup language)** is a text-based markup language that provides support for one of the most exciting information search and navigation environments ever developed such as **World Wide Web (WWW)** which represents a major step forward in making all kinds of information accessible to any user.

**HTML** is basically needed to create the Web pages. The key to building attractive, readable Web pages depends on knowing how to use **HTML** markup to highlight and organize your content.

**HTML is a display specification language with plenty of structuring and layout controls to manage a document's appearance, and the linkage mechanisms necessary to provide hypertext capabilities. HTML combines instructions within the data to tell a display program, called a browser, how to render the data that the document contains.**

**HTML represents a way to take ordinary text and convert it into hypertext, just by adding special elements that instruct Web browsers how to display its contents. These special elements are called tags. HTML documents are plain-text files that can be created using any text editor (e.g., Emacs or vi on UNIX machines; BBEdit on a Macintosh; Notepad on a windows machine).**

**Example:**

**We provide below a simple HTML document file.**

**HTML document file:**

```
<HTML>  
<HEAD><TITLE> My Title </TITLE>  
</HEAD>  
<BODY>  
My Heading Text  
<P>  
My wonderful text and graphics.  
<P>  
<ADDRESS>  
My Name <BR>  
My Mail Address <BR>  
</ADDRESS>  
</BODY>  
</HTML>
```

**The tags TITLE, BODY, P and ADDRESS describe an object and its components [refer section 3.5 for more information on HTML tags]. It's up to a browser to render these properly on the screen.**

## Hypertext page:

Figure 1.2 shows the corresponding hypertext document for the above HTML document.

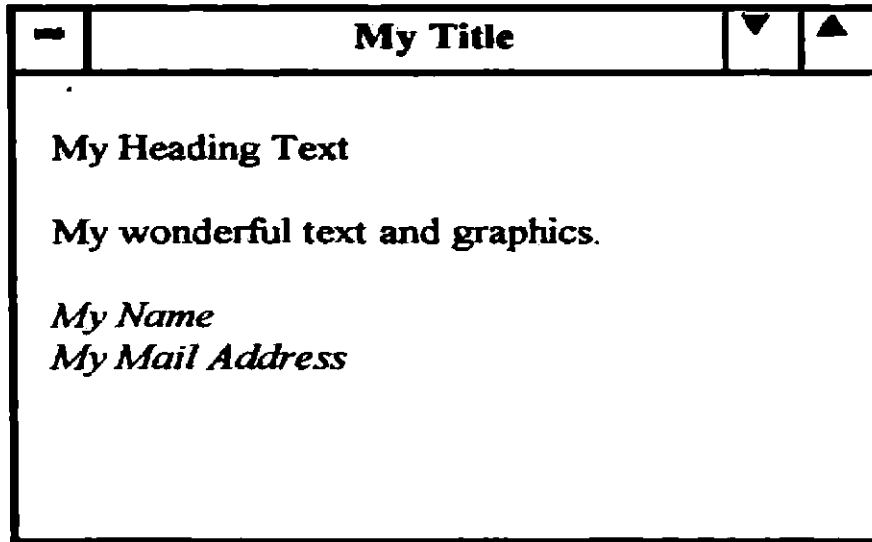


Fig. 1.2 Hypertext page.

What makes HTML particularly interesting is that it's all pure text. In fact, HTML can work completely within the confines of the ASCII 7-bit character set (ISO 646), which contains only 128 distinct viewable characters. This lets HTML display things like accents, umlauts, and other diacritical marks often associated with non-English languages, by including instructions on what characters to represent as a part of the markup. In other words, HTML can provide instructions to a browser even if you can't always see things in the same format when you are writing the HTML. For example, `<OL>` and `<LI>` don't look like a numbered list inside the text files, but they do when the browser interprets them.

### 1.2.3 Relationship between HTML documents and a hypertext document

A hypertext document is made up of one or more "pages". For creation of each hypertext page we should have one HTML document file which contains the display information of that page and can also contain the links to other HTML documents. A browser takes one or more HTML documents and interprets them into a hypertext document.

### **1.3 What is a browser ?**

**Browsing is nothing but searching. Browser facilitates searching through a hypertext document. It is a program that displays the content described by HTML code. It may provides point-and-click interface for accessing multimedia internet resources. Inside the browser software there<sup>is</sup>s a parser that does the work of reading and constructing the display information. A parser reads information in the HTML file and decides which elements are markup and which ones aren't, permitting the browser to take appropriate action. A HTML browser will read the HTML document and turns it into hypertext. These browsers expect specific information because they are programmed according to HTML specifications.**

### **1.4 WYSIWYG Output**

**Although the term WYSIWYG (What You See Is What You Get) has come to define hard-copy quality, to a programmer, producing WYSIWYG output is largely a matter of relative positioning. Unless the display and printer resolutions match exactly, it's not possible to produce the same on-screen and printed graphics, but with careful programming and scaleable TrueType fonts, the results are good enough for most applications.**

**When producing a hard copy of the hypertext document it is much more readable if each page of the hard copy resembles the corresponding hypertext page the user is developing. In our implementation of HTML editor there is a WYSIWYG mode in which the user can see the hypertext document during development itself and also produce a hard copy of the hypertext pages.**

### **1.5 Motivation**

**Windows help file is a good example of a hypertext document.**

**Delphi is a windows based application development environment and hence provides features for developing help files. In Delphi, help files are created [refer chapter 2, section 2.5 for help file creation process] by compiling Rich Text Format (RTF) files with a utility provided by Microsoft (for Windows 3.1, this utility is called**



HC31.exe, a copy of which is in Delphi). Developing the RTF is not an easy task. It involves the following.

- It normally requires a word processor capable of editing or at least exporting RTF files.
- The RTF file must contain special codes (In Delphi RTF file is nothing but macro file [2]. This macro file is needed to specify the contents of the help file. We are using HTML to specify the contents and behaviour of the help file) that describe the structure and interrelationship of all the topic pages that make up the hypertext document.
- In addition to the RTF file, the help compiler requires a project file that contains entries specifying settings for the compiler's options.

Hence in the work presented here, we provide features which overcome these problems besides providing some additional facilities. Hence, in this context, we are motivated to provide the following features to the user.

- We don't need to have a single RTF file which contains the structure and interrelationship of all the topic pages. We can create each topic page separately by using the WYSIWYG editor in an efficient and easy manner.
- We do not go for Windows 3.1 utility i.e., HC31.exe. Our browser program is a Delphi system based HTML interpreter used to turn HTML document into hypertext.
- Generally each hypertext page dedicates itself to a "topic" that can consist of text, references to other topics, or any number of things. But here, we are trying to facilitate the user to execute the database queries through their hypertext documents too. We can achieve this if we define a syntax of our own and put into HTML file which the browser recognizes and takes appropriate action.

## **1.6 Our Achievements**

**We implemented a HTML document editor which helps the user in creating multiple HTML documents at a time.**

**We also implemented a HTML editor to facilitate a naive user in building his hypertext documents easily. This editor works in two operational modes. In the first mode, the user gets a cursory idea of HTML while creating a HTML document, and in the second mode (WYSIWYG mode), the user can create a hypertext document easily without knowing the HTML syntax and can also take a hard copy of the hypertext pages.**

**We have implemented a browser in such a way that the browser program takes HTML documents as the input and interprets them into a hypertext document.**

**We have developed a database application which demonstrates to the user as to how database queries can be executed through their hypertext documents by incorporating our syntax for query execution in HTML documents.**

## **1.7 Outline of the thesis**

**The organization of the chapter is as follows :**

**We have first stated the problem of the thesis and then review the required background of HTML and hypertext document. We have also covered the motivation for our approach.**

**We present below the organization of the remaining part of the thesis.**

**Chapter 2 gives the concepts related to Delphi system. It describes all the principles by which even a layman can understand the intricacies in the Delphi system. It also includes all the features of the components which are used in this project.**

**Chapter 3 talks about the HTML syntax. It describes tags that comprise the HTML syntax and the categorization of these HTML tags.**

**Chapter 4** describes the design and implementation details of the **HTML document editor**. This editor allows users (familiar with HTML syntax) to create their HTML documents directly and generate their hypertext documents by calling the browser program.

**Chapter 5** gives the design and implementation details of the **HTML editor**. This editor works in two modes. In first mode, a HTML document can be created without prior knowledge of HTML syntax. While creating the HTML document user gets some cursory idea about HTML.

The second mode is the **WYSIWYG** mode. In this mode user can create his hypertext document, as is actually seen when using a browser and also produce a hard copy of the hypertext pages.

**Chapter 6** deals with the design and implementation details of the **browser**. The browser takes HTML documents as input and interprets them into a hypertext document. It uses our syntax for query execution through hypertext.

**Chapter 7** deals with the specification and implementation details of the **example database**. It also describes the queries involved in the example database and the execution of queries through hypertext.

Last chapter concludes the project work and suggests the future enhancements to the system.

## DELPHI SYSTEM OVERVIEW

### 2.1 Introduction

In the last chapter we have seen the need of Delphi IDE in this project. We have also seen the RTF file creation process which is needed to generate help file in Delphi. This chapter will give you a brief overview of the Delphi's working principles, the help file creation process and the components involved in developing this project.

The organization of the chapter is as follows. In sec. 2.2, we present the features of Delphi. In sec. 2.3, we cover elements in Delphi's IDE, in sec. 2.4 we give the structure of the object Pascal unit, in sec. 2.5 we discuss the help file creation process in Delphi and in sec. 2.6 we give the overview of all the visual components used in this project. Lastly sec. 2.7 concludes the chapter.

### 2.2 What is Delphi ?

Delphi is a rapid application development environment, suitable for creating windows prototypes and finished applications that rival or exceed the speed and efficiency of programs written in C, C++, Borland Pascal 7.0, Visual Basic. Delphi is a smart code generator, a visual application designer, and a database tool, providing a superb interface that's powerful.

It consists of a front end which is supported by Visual tools and the back end which is supported by object oriented Pascal. Elements of the Delphi programming environment are used to create an application interface. There are several main elements in Delphi's **Integrated Development Environment (IDE)** [A graphical view is shown in **Appendix C**]. Now we will describe them.

## **2.3 Elements in Delphi's IDE**

### **1. Speed Buttons**

These are point and click buttons for selected menu commands. Using these buttons you can save your application's development time.

### **2. Menu bar**

This is a standard windows style menu.

### **3. Component palette**

This palette contains icons that represents components in the Visual Component Library(VCL). By positioning the mouse cursor over any component button you can see the hint box, which tells what component it is. The component can be inserted into the form by clicking on the component icon and clicking on the form at the position where it is desired to be inserted.

### **4. Palette page tabs**

To view other component categories click one of the page tabs below the VCL palette.

### **5. Form**

A form is a visual representation of the program's main window. By selecting a visual component from the component palette and clicking the mouse inside the form, you can insert a component object into the form and you can change the properties and the events to be triggered through the object inspector.

### **6. Object Inspector**

This window shows all of the properties and events of one or more selected components or forms.

### **7. Properties and events page tabs**

Click one of the two page tabs at the bottom of the object inspector window to switch between a component or form's properties and events.

## 8. Unit window

This window shows the Pascal programming text associated with each form in the application. Delphi automatically creates this programming to which we can add Pascal statements that perform actions for events.

In Delphi, the unit window contains the **Object Pascal** code. This object Pascal unit follows a particular structure which is given in Fig. 2.1.

### 2.4 Structure of the object Pascal unit

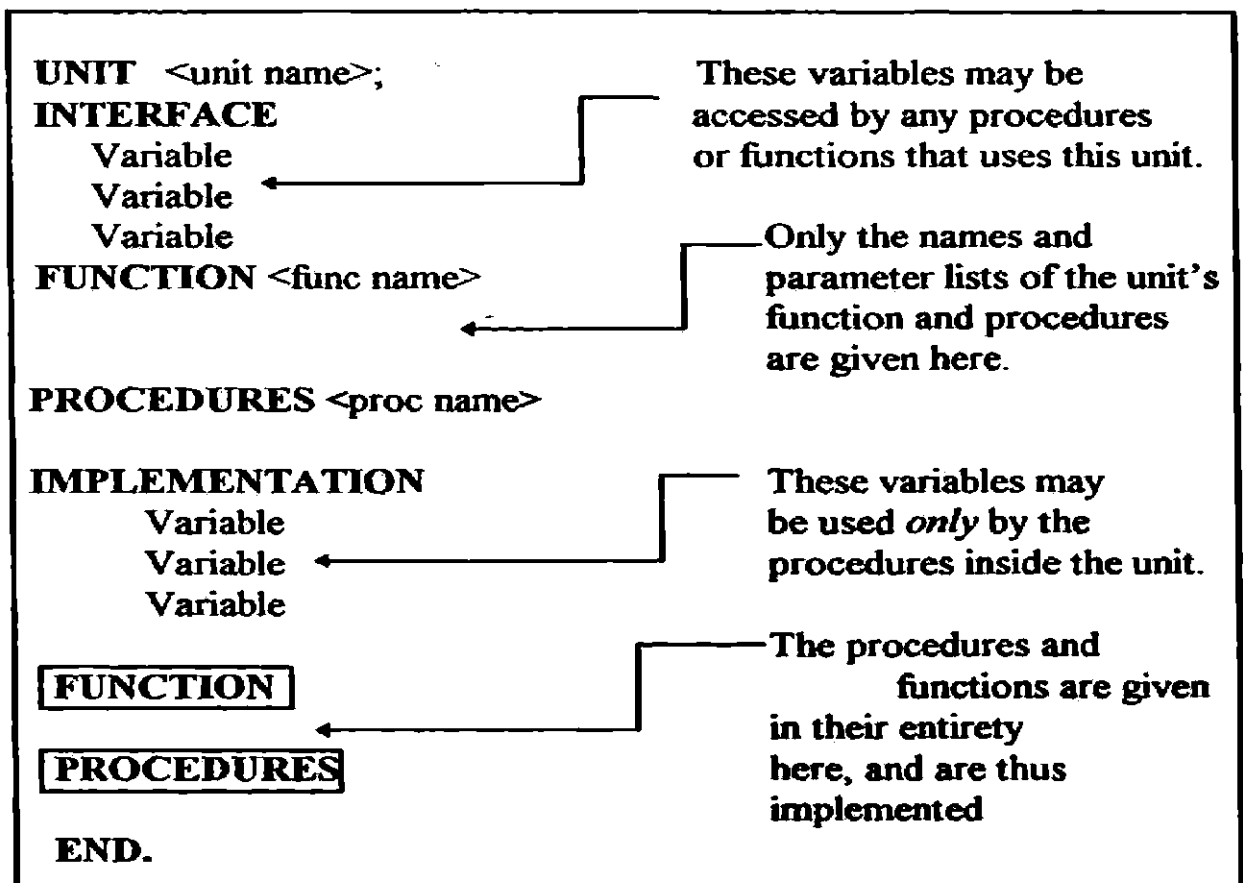


Fig. 2.1 Object Pascal unit.

First and foremost, an Object Pascal unit is a package. It's a way of associating data types and variables with the code that works with them. If you create data types and procedures that act on those data types, a unit helps in keeping everything together in one comprehensible bundle. The fact that a unit is also a physical file makes it a

reusable module. we can define constants, types, variables, procedures, and functions in a unit.

The interface section of a unit consists of everything between the reserved word **INTERFACE** and the reserved word **IMPLEMENTATION**. The interface section is for definitions only. This includes not only obvious things like constant, type, and variable definitions—but also definitions of procedures and functions.

Everything from the reserved word **IMPLEMENTATION** down to the closing **END**. is considered the implementation section of a unit. Two general groups of things are defined in the implementation section of the unit: i. Function and procedure bodies implementing the headers defined in the interface section and ii. Definitions private to the unit as a whole.

Providing a convenient package of data types, variables, and procedures is the obvious job of a unit. Managing the parts of a package of computation to be revealed to the public at large and the parts to remain the secrets of the unit itself, is the subtle job of a unit. Some things are visible, and some things are hidden. By and large, what is defined in the implementation section of a unit remains hidden, and everything else (being part of the interface section) is visible to any unit that uses your unit.

## **2.5 Help file creation process**

Here, help files are created by using the **HC31.EXE** utility, a copy of which is provided with Delphi. Creating a help file is a two step process. These are:

- i. Create the **RTF** (Rich Text Format) file, using the macro file [2].
- ii. Direct **HelpGen** [2] to call the help compiler. It automatically hands over the compiler the project file with the **RTF** file to generate the help file.

## **2.6 Components used in this project**

The **Visual Component Library (VCL)** is made up of objects, most of which are components. Components are visual objects that you can manipulate at design time. Components are the building blocks of Delphi applications. You build an application

by adding components to forms and using the properties, methods, and events of the components.

There are 89 components in the visual component library (21 are not visible in the component palette). We are going to describe only those 25 components which have been used in this project.

### **1. Memo**

Practically a word processor in its own right, the Memo component adds Notepad like capabilities to an application. **Palette : Standard.**

### **2. ScrollBar**

This general-purpose component adds scroll bars to a form, or it can operate independently as a range-selection control. **Palette : Standard.**

### **3. MainMenu**

We use this component to create a windows menu bar, which is always displayed below the top border and caption. If you double-click MainMenu component, the Menu Designer window will appear. The Menu Designer works in concert with the Object Inspector to help you build menus. **Palette : Standard.**

### **4. PopupMenu**

We use this component to create floating pop-up menus, which appear when the user clicks the right mouse button with the mouse cursor over the windows client area. Popup menus are in many ways similar to application main menus, but they normally serve a more limited purpose. Typically, an application will have a single main menu, and each window in the application will have its own local menu, normally implemented as a popup menu. This menu is also accessed by pressing ALT+F10. **Palette : Standard.**

### **5. Panel**

This component provides a platform for segmenting a busy window and for creating toolbars and status panels. Panels never receive the input focus. They are purely visual, and serve as containers for other controls. Components that are placed



on a panel are positioned relative to the panel, which means that if the panel moves, so do the components on it. This makes it much faster to arrange control groupings on a form while keeping them aligned with one another. Panels can also be aligned with the top, left, right, or bottom border of the window, so that when the window is resized, the panel remains in the same position relative to the border. And since the Panel's components are positioned relative to the Panel itself, they move right along it. Panels also have built-in support for fly-by hints, and a **Caption** property that lets you display text on the panel. All in all, the Panel component is an excellent choice as a foundation for a tool bar, tool palette, or status bar. **Palette : Standard.**

## **6. Edit box**

Edit box displays a dialog box into which users can type data. So edit boxes are used to retrieve information from users, they can also display information.

**Palette : Standard.**

## **7. Label**

Labels are used to display text on a form. The text of a label is its value of caption property. Labels are used to display text that users cannot edit.

**Palette : Standard.**

## **8. Button**

It is used for push button control. Users can use these buttons to initiate actions for specified commands. **Palette : Standard.**

## **9. ComboBox**

This standard Windows control combines a **ListBox** with an **Edit** object. Depending on the ComboBox object's style, users can choose from listed entries, which optionally appear in a drop-down box, or they can enter new data into the **Edit** control.

The ComboBox's **Items** property allows you to enter a list of strings that is to be displayed by the dropdown **ListBox** portion of the component. You can build this list of strings into the ComboBox at design time, and you can also modify the list at runtime.

**A ComboBox has a style property that you can set to change the way that the ComboBox behaves. Palette: standard.**

## **10. Listbox**

**It displays a list from which users can select one or more items. The list of items in the list box is the value of its items property. Palette : Standard.**

## **11. BitBtn**

**A BitBtn works like a Button component, but can display a colorful icon, called a glyph, that visually represents the buttons action. This component is having properties such as Glyph, Kind, and Layout that are used to define the button's picture and appearance. Palette : Additional.**

## **12. SpeedButton**

**Speed buttons, implemented with Delphi's SpeedButton component, are special purpose buttons that were designed specifically to be used on a toolbar or tool palette. The SpeedButton is similar to a standard Button component, but includes special internal machinery that allows it to interact with other SpeedButtons, and also with the Panel that contains it. These buttons have graphical images on their faces that users can use to execute commands or set modes. The graphical image that appears on a speed button is the value of its glyph property. Palette : Additional.**

## **13. Notebook**

**This is used to display multiple pages ,each with its own set of controls. This is generally used with Ttabset controls to let users select pages in the Notebook by clicking a tab. Palette : Additional.**

## **14. Tabset**

**This component provides a toolbar of selectable tabs, which you can use in conjunction with other components to provide them with a page turning capability. Palette: Additional.**

## **15. Image**

This is used to display graphical images on a form. So, this component is to picture files what the memo component is to text files. With an Image component you can load and display bitmaps, icons, and Windows metafiles. The internal workings of the Image component are smart enough to figure out what kind of image is loaded, and take care of all the futzing around with display contexts, palettes, and everything else associated with displaying pictures in a Windows program. With Delphi, displaying a picture is as easy as displaying a text files. The image that appears is the value of its picture property. **Palette : Additional.**

## **16. Scrollbox**

This component makes it possible for the user to create scrolling areas on a form that are smaller than the entire form. **Palette : Additional.**

## **17. DataSource**

This component connects dataset components such as Table and Query with data-aware components such as DBEdit and DBMemo. Every database application needs at least one DataSource object. This component determines the source of the data you are using. This enables to connect data-aware controls on different forms to the same Datasource. **Palette : Data Access.**

## **18. Table**

This component usually associated with a DataSource object, which connects the table with data-aware controls. Most database application have atleast one Table object. A Table component accesses every column in a table when you activate it.

**Palette : Data Access.**

## **19. Query**

This component enables Delphi applications to issue SQL statements to a database engine. Query component is used to provide SQL statements that retrieve data from a physical data base table and send data back to a physical data base table.

**Users can view data retrieved in a data-aware component (such as DBEdit or DBGrid). Palette : Data Access.**

## **20. Batchmove**

This component is used to perform operations on groups of records or entire tables. This is done by setting source property to specify a dataset corresponding to an existing source table, setting the destination property to specify a dataset corresponding to a database table and setting the mode property to specify operations to perform. **Palette : Data Access.**

## **21. DBGrid**

This component enables to view and edit all the records in a dataset component (such as Table or Query) in a spreadsheet like form. **Palette : Data Controls.**

## **21. DBNavigator**

This component is a sophisticated database browsing and editing tool. Users click it's buttons to move through the database records , insert records, delete records and perform other navigational operations . **Palette : Data Controls.**

## **22. DBEdit**

A data-aware edit single line text entry component. It can be used to display data from a field in a dataset by specifying a datasource component.

**Palette : Data Control.**

## **23. Opendialog**

This component makes an open dialog available to the user. The main purpose is to let a user specify a file to open. Using the execute method, we can display the open dialog. When the user chooses OK, Users file name selection is stored in the dialog box file name property. **Palette : Dialog.**

## **24. Savedialog**

This component makes available a save file dialog available to the user. The main purpose is to allow a user to specify a file to save. **Palette : Dialog.**

## **25. FontDialog**

This component makes available a font dialog box to the user. The main purpose is to let a user select his font and set attributes of that font. This component is pretty simple. It only has a handful properties, and just one event. Other than **Font**, the only property you're likely to change often is **Options**, which describes the options and types of fonts that should be displayed in the Font dialog box when it's displayed. **Options** is a set property, and double-clicking on it will display all of its sub-properties. Most of these sub properties specify what types of fonts should be displayed in the dialog box. If you set **fdWysiwyg** to true, then only those fonts that are available to both the printer and the screen will be displayed in the dialog box.

**Palette** : Dialog.

## **2.7 Conclusions**

Delphi is a component-based application development system that enables us to write powerful Window-based programs with a minimum of coding. We use elements of the Delphi programming environment to create our own application interface.

Units are the basis of modular programming. We use units to create libraries and to divide large programs into logically related modules. The parts of a unit are unit heading, interface part, implementation part and initialization part.

In Delphi, help files are created by compiling Rich Text Format (RTF) files with a utility HC31.exe provided by Microsoft.

Using the objects and components of VCL, we can develop Windows programs rapidly. Delphi itself was built using VCL. Delphi objects contain both code and data. The data is stored in the properties of the objects, and the code is made up of methods that act upon the property values.

## HTML SYNTAX

### 3.1 Introduction

In introduction chapter we introduced the need for HTML. We came to know that HTML syntax comes in the form of tags, each tag in a certain syntax and a meaning associated with each of them. In this chapter we define and describe in detail all tags used in this project.

The organization of the chapter is as follows. In sec. 3.2, we give the notion of HTML tags, in sec. 3.3, the HTML categories and their respective tags, and finally we give the conclusions of the chapter in sec. 3.4.

### 3.2 Notion of HTML tags

The following example shows syntactic structure of a basic HTML document.

1. <HTML>
2. <HEAD><TITLE>A HTML DOCUMENT</TITLE></HEAD>
3. <BODY>
4. <H1>Table Of Contents</H1>
5. <!-- an unordered list -->
6. <OL>
7. <!-- a list item which is a link to another anchor in the same document  
named 'intro' -->
8. <LI><A HREF="intro">Introduction</A>
9. <LI><A HREF="concepts">Concepts</A>
10. <LI><A HREF="conclusion"><Conclusion</A>
11. </OL>
12. </BODY>
13. </HTML>

We have numbered the lines above for the purposes of reference in the following discussions on different tags in HTML.

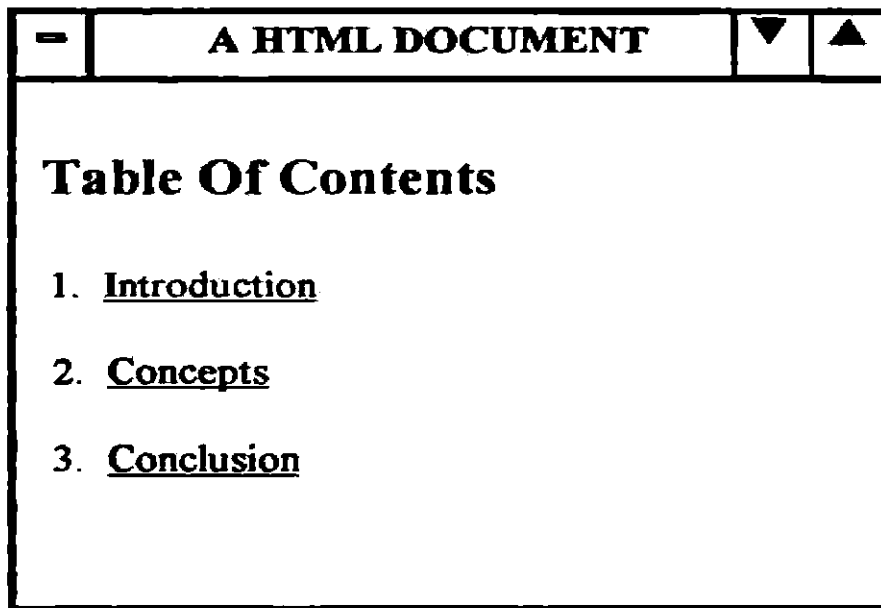
### Function of each tag:

- A pair of tags `<HTML>` and `</HTML>` blocks out the HTML document in its entirety. For an illustration of such a pair, see the lines 1 and 13 of the example.
- A pair of tags, `<HEAD>` and `</HEAD>` is used to physically identify the header of the document. The title of the entire document is specified within the pair of tags, `<TITLE>` and `</TITLE>`. Here the title of the document is: A HTML DOCUMENT. For illustration of the pairs, see line 2.
- `<BODY>` and `</BODY>` tags are used to set off the body of the HTML document. See lines 3 and 12 for illustration of a pair.
- `<H1>` and `</H1>` will tell the browser to print “Table Of Contents” using the first level heading size. See line number 4 for an illustration of the pair.
- `<!-- ... -->` supports author comments. These comments are used for enhancing the readability of the HTML document. See lines 5 and 7 for an illustration of this tag.
- A pair of tags `<OL>` and `</OL>` numbers the elements by order of occurrence. In the above document these tags produce the following output when interpreted by the browser. See lines 6 and 11 for illustration of a pair.
  1. Introduction
  2. Concepts
  3. Conclusion
- `<L1>` tag marks a member item within the ordered list. See lines 8, 9 and 10 for an illustration. In the above document the member items within the ordered list are Introduction, Concepts and Conclusion.

For example, in line 8, `<A HREF=“intro”>Introduction</A>` entry makes the word ‘Introduction’ the hyperlink to the HTML document ‘intro’ on the same machine as this HTML document. This ‘intro’ specifies the complete path of the file and is in this context known as Uniform Resource Locator (URL).

## **Hypertext document:**

HTML represents a way to take ordinary text and convert it into hypertext, just by adding tags that instruct Web browsers how to display its contents. For example, when the above HTML document will be viewed with the Web browser then the following hypertext document will be displayed [When viewed with the browser, the original window displays all the menu options, scroll bars etc. along with the hypertext document. The following figure displays just the contents of the hypertext document.]



## **Properties of HTML tags:**

In the above HTML document we have seen how the text appears between the opening and closing tags as well as we have seen how the tags appear in the document. It is necessary to know some of the properties of the HTML tags to construct a HTML document properly. The properties of the tags are as described below:

- All HTML tags are surrounded by angled brackets ,for example <HEAD> ,
- A forward slash following the left angle bracket denotes a closing tag, for instance </HEAD> ,
- All HTML tags require that the characters in a name be contiguous. No extra blanks can be inserted with in a tag or its surrounding markup with out causing that tag to be ignored,



- But when assigning values to attributes however spaces are OK i.e., when one space is legal, multiple spaces are legal,
- In a tag, if an attribute is present it shows that the value that is present for that tag's attribute can be utilized by it. For instance in the <IMG> tag, the attribute Ismap, if present tells that the map is a clickable one. So the default is if the attributes are not present then the values cannot be utilized.

Sometimes it is necessary to insert one set of markup tags within another set of tags which is called nesting tags. So when nesting tags, the best rule of thumb is to close first what you have opened most recently.

### **3.3 HTML categories and their respective tags**

#### **3.3.1 Comments**

Comments give HTML authors a way to annotate their documents and browsers will not ordinarily display them. Any assumptions, special conditions should be enclosed in comments to help other readers understand what the document is trying to accomplish.

##### **Tag used:**

<!-- ...>, Supports author's comments.

##### **Example:**

<UL>      <! Unnumbered list -->

The comment "Unnumbered list" tells that the tag <UL> is used for an unnumbered list.

#### **3.3.2 Document Structure**

There are numerous tags defined to provide structure to HTML document. They provide an overall HTML label and breakup document into head and body section. They also provide markup to establish links to other documents and to indicate support for electronic indexing capabilities. This markups help in constructing well designed web pages.

Some of the tags used are as follows:

**<HTML> .... </HTML>**, Blocks out an entire HTML document.

**<HEAD>...</HEAD>**, Blocks out a document head.

**<BODY>...</BODY>**, Blocks out a document body.

Example:

**<HTML>**

**<HEAD>**

**<TITLE> ... </TITLE>**

**</HEAD>**

**<BODY>**

.

.

**</BODY>**

**</HTML>**

### **3.3.3 Document Headings**

Headings provide structure for a document's content ,starting with its title, all the way down to sixth level headings. They provide meaningful clues for document navigation and when used in conjunction with a hypertext table of contents can permit readers to quickly jump to other sections.

Some of the tags used are:

**<TITLE>...</TITLE>**, Supplies the title that labels the entire document.

**<H1>...</H1>**, First level heading.

**<H2>...</H2>**, Second level heading.

**<H3>...</H3>** ,Third level heading.

**<H4>...</H4>**,Fourth level heading.

**<H5>...</H5>**,Fifth level heading.

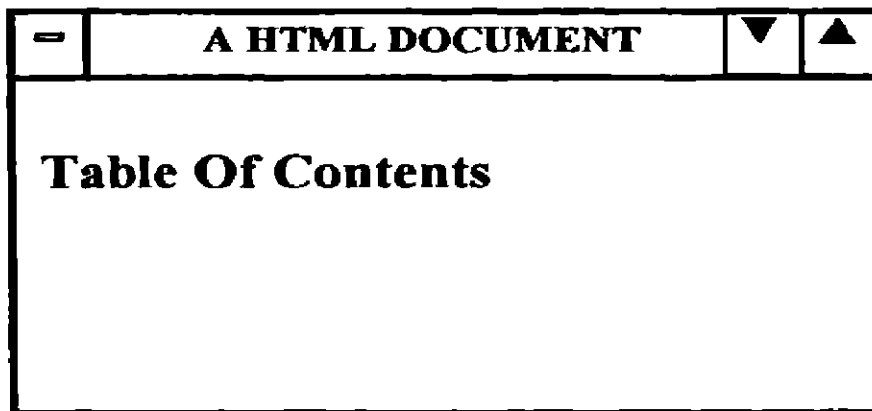
**<H6>...</H6>** , Sixth level heading.

HTML has above six levels of headings, numbered 1 through 6, with 1 being the most prominent. Headings are displayed in larger and/or bolder fonts than normal body text. Do not skip levels of headings in your document. For example, don't start with a level-one heading (<H1>) and then next use a level-three (<H3>) heading.

**Example :**

```
<HTML>
<HEAD><TITLE>A HTML DOCUMENT</TITLE></HEAD>
<BODY>
<H1>Table Of Contents</H1>
</BODY>
</HTML>
```

The output of the above document looks like:



### **3.3.4 Links**

Create links to anchor or another document , or create anchor point for another link .

Tag which come under this is:

**<A>..</A>**, Provides fundamental hypertext link capabilities.

HTML's single hypertext-related tag is **<A>**, which stands for anchor. To include an anchor in your document follow the following steps:

1. Start the anchor with **<A** (include a space after the A)
2. Specify the document you're linking to by entering the parameter **HREF="filename"** followed by a closing right angle bracket (**>**)

**3. Enter the text that will serve as the hypertext link in the current document**

**4. Enter the ending anchor tag: </A> (no space is needed before the end anchor tag)**

**Example:**

Here is a sample hypertext reference in a file called Saha.htm:

```
<A HREF="Saha.htm">Hello</A>
```

This entry makes the word 'Hello' the hyperlink to the document 'Saha.htm', which is in the same machine (in our case) as the first document. Generally, HREF="URL" entry means that URL (Uniform Resource Locator) specifying the location of another network resource, usually another HTML file, but this URL can also be a pointer to services provided by FTP (File Transfer Protocol), Telnet, e-mail (electronic mail) etc. In our case we are considering that the URL only specifies those file names which are in the same machine as the current HTML document (which contains these URL specifications).

### **3.3.5 L a y o u t E l e m e n t s**

Layout elements introduce specific items within the text of a document including, line breaks, lengthy quotes, and horizontal rules to divide up distinct text areas. They include a format for building author information on a page.

**Tags used are:**

**<ADDRESS>...</ADDRESS>**, Author contact information for document.

**<BLOCKQUOTE>...</BLOCKQUOTE>**, Use to set off long quotes or citations.

**<BR>** , Forces a line break into on-screen text flow.

**<HR>** , Draws a horizontal line across the page.

**Example:**

1. **<ADDRESS>**

A beginners guide to HTML / NCSA / [pubs@ncsa.uiuc.edu](mailto:pubs@ncsa.uiuc.edu) /revised April 96

**</ADDRESS>**

**The result is:**

**A beginners guide to HTML / NCSA / pubs@ncsa.uiuc.edu /revised April 96**

**2. <BLOCKQUOTE> “ After spending the last ten years locked in a cell, there was no way that Mr. Saha could conceive of not taking advantage of the sudden earthquake that opened a passage to the outside world, and freedom.”</BLOCKQUOTE>**

**The result is:**

**“ After spending the last ten years locked in a cell, there was no way that Mr. Saha could conceive of not taking advantage of the sudden earthquake that opened a passage to the outside world, and freedom.”**

**3. Super computing applications <BR>  
East Avenue <BR>  
Champaign<BR>**

**The result is:**

**Super computing applications  
East Avenue  
Champaign**

**4. <HR size=2 width=“50%”>**

**The result is:**

**-----**

### **3.3.6 Graphics**

**Graphics enter an HTML file through the <IMG> command. <IMG> points to the graphics source provides a text alternative for non graphical browsing and indicates whether the graphic is a clickable map.**

**Tags used:**

**<IMG>, Inserts a referenced image into a document with alternate text, clickable map, and placement controls.**

To include an image enter:

**<IMG SRC=ImageName>**

We can include two other attributes on **<IMG>** tags to tell the browser the size of the images it is loading with the text. The **HEIGHT** and **WIDTH** attributes let the browser to load the image of that size.

**Example :**

To include a self portrait image in a file along with the portrait's document, enter **<IMG SRC=SelfPortrait.gif HEIGHT=100 WIDTH=65>**.

In this project the browser program is taking care of only .bmp file.

### **3.3.7 F o r m s**

Forms provides the essential mechanism for soliciting reader feedback and input on the web. Form tags covers how forms are set up, provide a variety of graphical and text widgets for soliciting input and supply methods to let readers select options from various types of pick lists.

**Tags used are:**

**<FORM>...</FORM>**, Marks beginning and end of form block.

**<INPUT>** ,Defines type and appearance for input widgets.

**<TEXTAREA>...</TEXTAREA>** , Multiline text entry widget.

**<SELECT>...</ SELECT>**, Creates a menu or scrolling list of input items.

**<OPTION>[...</OPTION>]**, A way of assigning a value or default to an input item.

**Example:**

```
<FORM METHOD= "POST" ACTION= " / cgi / txt-ara">
```

```
<P> Model Number
```

```
<SELECT NAME="mod-num" size='3'>
```

```
<OPTION>102
```

```
<OPTION>103
```

```
< /SELECT>
```

```
<P>Female <INPUT NAME="sex" TYPE ="CHECKBOX" VALUE = "female">
```

**<P><TEXTAREA NAME =“recipe” ROWS=“10” COLS=“25”>**

**Banana Waffles**

**Ingredients**

**</TEXTAREA>**

**</FORM>**

The formatted output is:

Model number	102	▲
	103	
		▼

Female

Banana Waffles Ingredient	▲
	▼

### **3.3.8 Paragraphs**

Break up running text into readable chunks.

**Tag used:**

**<P>** , Breaks up text into spaced regions.

We can center a paragraph by including the **ALIGN=** attribute in our source file.

**Example:**

**<P ALIGN=“CENTER”>**

**This is a centered paragraph.**

**Formatted output:**

**This is a centered paragraph.**

### 3.3.9 Lists

HTML includes numerous styles for building lists ,ranging from numbered to bulleted lists All of these provide useful tools for organizing list of items to improve readability.

Tags used are:

**<DIR>...</DIR>** , Unbulleted list of short elements(less than 20 characters in length).

**<LI>** , Within a list of any type <LI> marks a member item.

**<OL>..</OL>** , Numbered list of elements.

**<UL>..</UL>** , Bulleted list of elements.

**<MENU>...</MENU>** , A pickable list of elements .

**<DL>..</DL>** , A special format for terms and their definitions.

**<DT>** , The term being defined in a glossary list.

**<DD>** , The definition for a term in a glossary list.

Examples:

#### Unnumbered Lists:

Below is a three-item list:

```
<UL>
<LI> apples
<LI> bananas
<LI> grapefruit
</UL>
```

The output is:

```
* apples
* bananas
* grapefruit
```

#### Numbered Lists :

The following HTML code:

```
<OL>
<LI> oranges
<LI> peaches
<LI> grapes
</OL>
```



produces this formatted output:

1. oranges
2. peaches
3. grapes

### Definition Lists :

The following is an example of a definition list:

```
<DL>
<DT> NCSA
<DD> NCSA, the National center for
      supercomputing Applications, is located
      on the campus of the University of
      Illinois at Urban-Champaign.
<DT> Cornell Theory Center
<DD> CTC is located on the campus of Cornell
      University in Ithaca, New York.
</DL>
```

The output looks like:

```
NCSA
      NCSA, the National center for supercomputing
      Applications, is located on the campus of the University
      of Illinois at Urban-Champaign.
Cornell Theory Center
      CTC is located on the campus of Cornell University in
      Ithaca, New York.
```

The `<DT>` and `<DD>` entries can contain multiple paragraphs, lists, or other definition information.

### 3.3.10 Text controls

HTML offers numerous inline controls for adding emphasis or special appearance to text. It provides tools for describing user input and for including samples of computer code, computer output, variables and sample text.

Tags used are:

`<B>...</B>` ,Produces bold faced text.

`<CITE>..</CITE>` , Distinctive text for citations.

`<CODE>..</CODE>` , Used for code samples,

**<DFN>...</DFN>**, Used to emphasis a term about to be defined in the following text.

**<EM>...</EM>**, Adds emphasis to enclosed text.

**<I>...</I>**, Produces italicized text.

**<KBD>...</KBD>**, Text to be typed at keyboard.

**<SAMP>...</SAMP>** Sample in-line text.

**<STRONG>...</STRONG>**, Maximum emphasis to enclosed text.

**<TT>...</TT>** , Produces a typewriter font.

**<VAR>...</VAR>** , Variable or substitution for some other value.

**Example:**

**<B>Hello Friends</B>**

**Result is:**

**Hello Friends**

**<CITE>The Iliad</CITE>** is arguably Homers greatest epic.

**Result is:**

*The Iliad* is arguably Homers greatest epic.

**<CODE>**

crispy rice1,rice2,rice3,rice4;<BR>

rice4=&#40;rice1,rice2,rice3,rice4&#41;;

**</CODE>**

**Result is:**

crispy rice1,rice2,rice3,rice4;

rice4=(rice1,rice2,rice3,rice4);

**<STRONG> no way out </STRONG>**

**Result is:**

*no way out*

**<SAMP>**

African<BR>

Asian<BR>

**</SAMP>**

**Result is:**

African

Asian

**<KBD>Xcopy A:\*. \* C: </KBD>**

**Result is:**

Xcopy A:\*. \* C:

I am failure to **<EM>communicate</EM>**”

**Result is:**

I am failure to *communicate*

**<I> hello friends</I>**

**Result is:**

*hello friends*

**<VAR> filename</VAR>**

**Result is:**

*filename*

**For more information on HTML document and tags refer [4].**

### **3.4 Conclusions:**

**HTML is a markup language that describes the structure of a Web document's content plus some behavioral characters. It is a standard language that all Web browsers are able to understand and interpret. HTML represents a way to take ordinary text and convert it into hypertext, just by adding tags that instruct Web browsers how to display its contents.**

## **HTML DOCUMENT EDITOR**

### **4.1 Introduction**

As we mentioned in section 1.2.4, HTML documents are plain-text files which can be created using any text editor (e.g., Notepad on Ms Windows). Later, in section 1.3 we discussed the difficulties involved in creating RTF files (in our case a RTF file is nothing but a HTML file). It normally requires a word processor capable of editing or at least exporting RTF files. Again the RTF file must contain special codes that describe the structure and interrelationship of all the topic pages that make up the hypertext document.

In this chapter we discuss a text editor which allows the user to create all the HTML files separately at a time which are needed to create a hypertext document. The user can create more than one HTML document simultaneously in different windows and can switch over from one window to another there by providing better visualization of the relationship between the documents which is not possible in Notepad.

To provide the above facility we are using MDI (Multiple Document Interface) as a means for this application to simultaneously open and display two or more HTML files.

The organization of the chapter is as follows. In sec. 4.2 we provide a brief overview of MDI application. In sec. 4.3 we give the design of the editor and in sec. 4.4 we give the implementation details. Lastly, sec. 4.5 concludes the chapter.

### **4.2 MDI Application**

In an MDI application, the main window looks like a normal application window, with a title bar, menu, resizable border, and other characteristics. The client

area though, isn't used for program output. Rather, the client area is a "workspace" in which document windows (or more properly, child windows) display information.

Child windows also look like regular windows, except that they don't have menus. Only the application window has a menu, but those menu items can be applied to the child windows as well. Although many child windows can be simultaneously displayed, only one of them can be active at a time. All child windows are clipped so that they never appear outside of the main window.

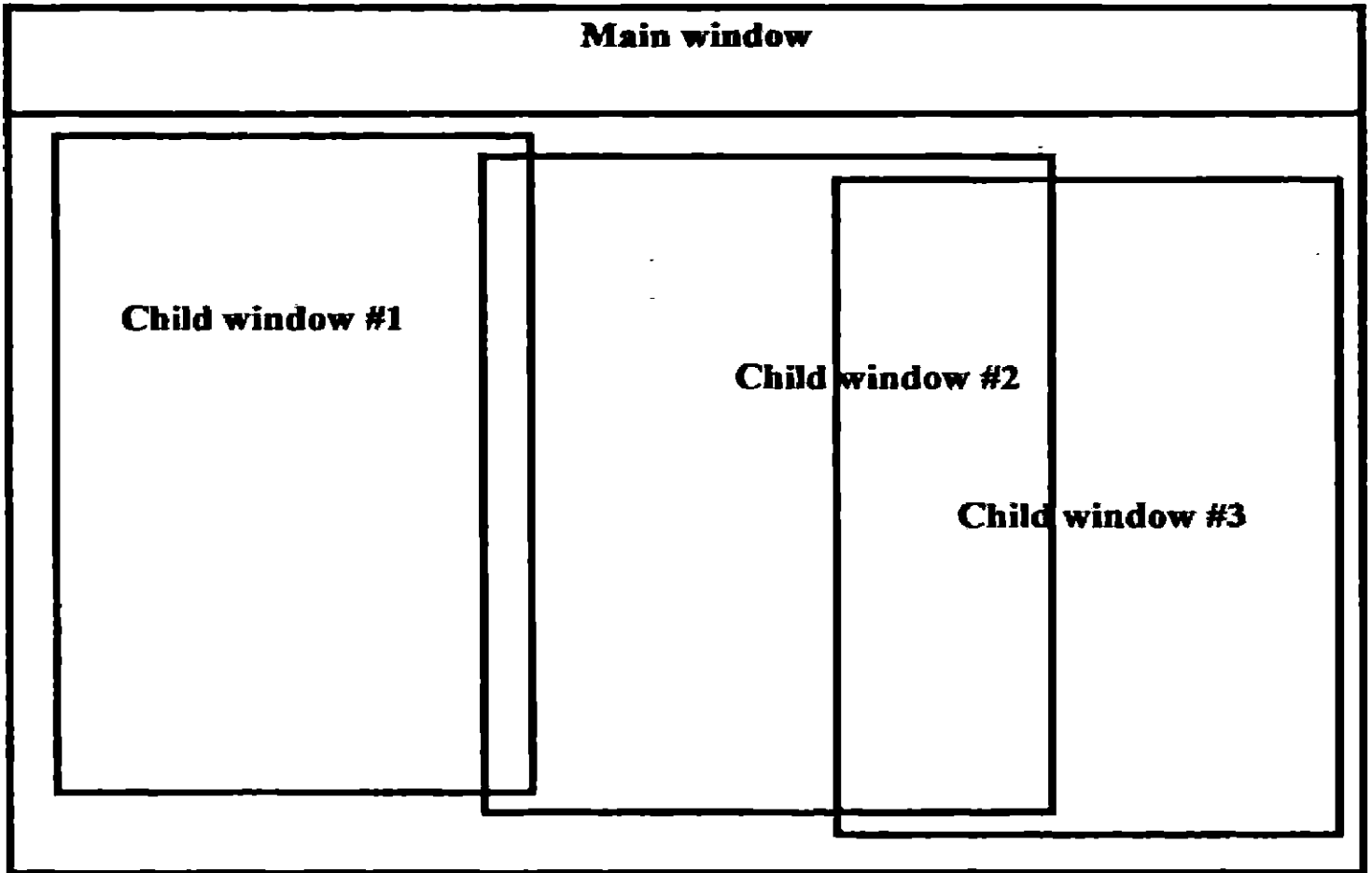
The MDI specification defines what happens when child windows are manipulated (opened, closed, moved, sized, minimized, maximized), and what special keystrokes are used to manipulate child windows. There are also suggested standard menu layouts and many other details defined in the specification. Most of this special behavior is handled internally by Windows when you create an MDI application. With traditional Windows programming tools, you have<sup>t</sup> jump through a number of hoops to get MDI applications to respond correctly. Not so with Delphi. Delphi handles all of the MDI-specific details automatically, leaving you free to concentrate on those things that are specific to your application.

### **4.3 Design of the HTML document editor**

This editor should allow the user to create all the HTML documents simultaneously in separate windows such that the user can visualize the relationships between the documents in a better way. These HTML documents are needed to create a hypertext document. To create each page of a hypertext document, we need to have a HTML document corresponding to that page which contains the structure, content and behavior of that page.

In order to create multiple HTML documents in separate windows, we are having one main window which will allow the user to create and operate on child windows. The user can edit the contents of the HTML documents in those child windows and can operate on those child windows by using the standard window style menu contained in the main window.

The interface window provided by the editor is:



A sample view of the HTML document editor is shown in section A of the appendix C.

## **4.4 Implementation details**

In the following section we give the properties of the components used by this editor. Later we discuss the events to take place by these components.

### **4.4.1 Components used and their properties**

It uses two forms, one is **MainForm** and another is **ChildForm**. Properties of the components are given below:

## **PROPERTIES :**

### **MainForm :**

**Caption** : HTML Document Editor.  
**FormStyle** : fsMDIForm.  
**Name** : mainform.

The mainform uses two components , one is **MainMenu** and **OpenDialog**.

### **MainMenu :**

In MainMenu's Item property we include File and under File we are having New , Open and Exit command.

### **OpenDialog :**

By double-clicking the Filter property we are including the following properties in the FilterEditor.

<b>FilterName</b>	<b>Filter</b>
All Files(*.htm)	*.htm

### **ChildForm :**

**Activecontrol** : memo1  
**Caption** : Untitled  
**Color** : clWindow  
**FormStyle** : fsMDIChild  
**Name** : childform

The childform uses four components. These are **SaveDialog**, **MainMenu**, **FontDialog** and a **Memo**.

### **SaveDialog :**

**Filter** : like OpenDialog component explained in the above.  
**Name** : SaveFileDialog.



## **MainMenu :**

In MainMenu's Item property we include **File , Edit and Character** command. Under **File** we are having **New, Open, Close, Save, Save-as** and **Exit** command. Under **Edit** we are having **Cut, Copy, Paste, Delete** and **Select-all** command. Under **Character** we are having **Font, Left, Right** and **Center** command.

## **FontDialog :**

No property has been changed.

## **Memo :**

**Align** : alClient  
**Alignment** : taLeftJustify  
**BorderStyle** : bsNone  
**Cursor** : crIBeam  
**ParentCtrl3D** : False  
**ParentFont** : False  
**ScrollBars** : ssBoth  
**WordWrap** : True

## **4.4.2 Event handlers of the components**

### **i. Events of the components used by MainForm:**

#### **OnClick event of MainMenu's New command :**

- 1. Create the ChildForm**
- 2. Show the ChildForm**

#### **OnClick event of MainMenu's Open command :**

**If OpenFileDialog is executing then**

- 1. Create the ChildForm**
- 2. Open the ChildForm with the selected file.**
- 3. Show the ChildForm .**

**OnClick event of MainMenu's Exit command :**

**Close the application .**

**ii. Events of the components used by ChildForm :**

**OnClick event of MainMenu's Font command:**

- 1. Assign memo's Font to FontDialog component's Font property .**
- 2. If FontDialog is executing then give FontDialog's Font to memo's Font.**
- 3. Set a rectangle as an edit area, which is memo's client area. So whatever the user will type in the edit area that will be written in that font only.**

**OnClick event of MainMenu's Left, Right and Center command :**

- Left** : Set memo's Alignment property to taLeftJustify.  
**Right** : Set memo's Alignment property to taRightJustify.  
**Center** : Set memo's Alignment property to taCenter.

**OnClick event of MainMenu's Copy , Cut , Paste ,Delete and Select-all command :**

- Copy** : Copy the selected text into clipboard using **CopyToClipboard** method.  
**Cut** : Cut the selected text into clipboard using **CutToClipboard** method.  
**Paste** : Paste the text from the clipboard into the memo component using **PasteFromClipboard** method.  
**Select-all** : Select all of the text using **SelectAll** method.  
**Delete** : Delete the selected text from the memo component using **ClearSelection** method.

**OnClick event of MainMenu's New , Open ,Close , Save , Save-as and Exit command :**

- New** : Call the MainForm's **New** procedure .

- Open** : Call the MainForm's Open procedure .
- Close** : Call the Close method to close the ChildForm.
- Save** : If filename = ' ' (blank space) or filename is read-only call Save-as procedure  
 \* else  
 Create a backupcopy of the file ,  
 Save all the lines of the memo to that file,  
 Set the Modified property of the memo component to false.
- Save-as  
 Filename** : Give the filename to the SaveFileDialog components property ,  
 If SaveFileDialog is executing then Change the filename to the selected filename,  
 Change the Caption property to the current filename ,  
 Call save procedure.
- Exit** : Call the MainForm's Exit procedure .

#### 4.5 Conclusions :

Using this editor, we can create all HTML documents at a time. We can create multiple ChildForms and change the focus from one ChildForm to another. The documents will be saved in HTML format and with the extension .htm (This is possible by setting the Filter property of OpenFileDialog and SaveDialog component properly as discussed in section 4.4.1). This editor uses one form named mainform which acts as a main window. In this editor, a form named childform acts as a child window. Multiple copies of child window can be created by using New menu item, contained in the main window menu bar. Each and every operations on the child windows <sup>is</sup> are performed by the Window style menu item contained in the main window. Each child window contains one HTML document. A hypertext document contains many pages linked together. Each HTML document is needed to generate each hypertext page. The HTML documents thus created are needed by the browser to produce a hypertext document.

[The result is shown in Appendix B.]

# **HTML EDITOR**

## **5.1 Introduction**

In the last chapter we discussed about a text editor which allows the user to create all the HTML documents separately at a time which are needed to create a hypertext document. In section 1.2.7 we mentioned a WYSIWYG editor. In this chapter we discuss the design and implementation details of a WYSIWYG editor which gives the WYSIWYG output as pointed out in the section 1.2.7, in the front end and HTML code at the back end. This editor allows user to create each hypertext page of a hypertext document directly in an easy fashion. The user need not specify the content of the page in HTML code and call the browser to display the hypertext page.

This editor has another mode of operation to provide an overview of the structure of a HTML document to a naive user.

The organization of the chapter is as follows. In sec. 5.2, we discuss the main functions to be performed by this editor in each mode and in sec. 5.3 we give the design and implementation details of the editor for each mode. Lastly in sec. 5.4, we concludes the chapter.

## **5.2 Modes of operation**

This editor works in two operational modes.

In the first operational mode, the HTML code is generated by following a structure of the HTML document (which is provided in the On-line Help) and clicking the buttons according to that structure. It has only the front end i.e., we can see the generated HTML document in the current window. Through this editor a novice can gain some knowledge of HTML syntax as well about the structure of the HTML document.

In the second mode of operation, the editor generates WYSIWYG type of output in the front end and the corresponding HTML code will be generated in the back end. This editor allows the user to create a hypertext document directly in the front end. Each page of a hypertext document will be stored in equivalent HTML code in separate .htm file. This editor also allows the user to take a hard copy of the hypertext pages.

### 5.3 Design and implementation details

#### 5.3.1 Design of the editor (Operational mode 1):

Input specification: Text.

Output specification: HTML code.

The block diagram of the editor is given in Fig. 5.1.

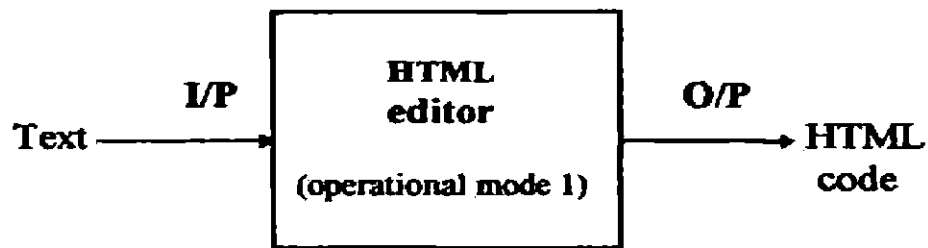


Fig. 5.1.

In this mode, user edits the text and presses buttons to generate its equivalent HTML code. The interface window provided by the editor is shown in Fig.5.2 below.

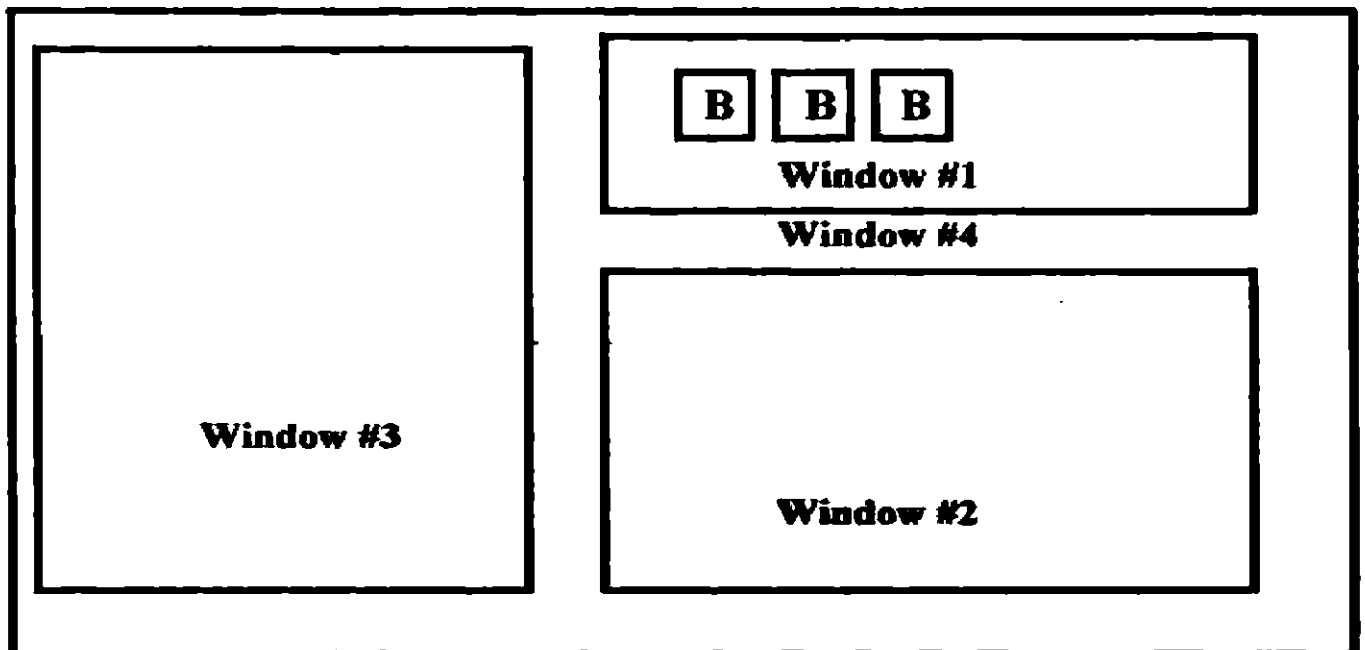


Fig. 5.2.

Window #1 acts as a control panel. It contains the buttons that allow the user to generate a HTML document in window #3. The operations like opening a HTML document, closing the document, saving the document etc. in window #3, are made easy by the buttons in window #1, which operate like the Window style menu items. Beside those buttons, there is one button by clicking which the user gets some cursory idea about the structure of the HTML document and the way of creating a HTML document.

Window #2 allows the user to enter the text which is required by some HTML tags such as H1,...,H6, ADDRESS, A [3.3] etc., which appears between the opening and the closing tags.

Window #3 is required to display the HTML code, generated by using the control panel's (window #1) buttons and to open a HTML document file.

Window #4 acts as a container of other three windows.

### 5.3.2 Implementation details (Operational mode 1):

#### i. Components used and their properties:

We use four forms to implement the above design. The name of the forms are: 1. MainForm, 2. Text\_Entry, 3. Coding, 4. Back

The properties of these forms and the components used by them are given below.

#### **PROPERTIES:**

##### **MainForm:**

This form is used as a control panel. The properties are:

<b>Caption</b>	:	<b>Control Panel</b>
<b>Color</b>	:	<b>clOlive</b>
<b>FormStyle</b>	:	<b>fsStayOnTop</b>
<b>Name</b>	:	<b>MainForm</b>
<b>WindowState</b>	:	<b>Normal</b>
<b>position</b>	:	<b>poDesigned</b>

**This MainForm uses the following components. Their properties are:**

**Notebook :**

**pages :** call notebook editor and add the following lines .

<b>page name</b>	<b>Help context</b>
standard	0
categories	1

**Align :** alTop  
**color :** clOlive

**Tabset :**

**Align :** alTop  
**Background color :** clBtnFace  
**color :** clTeal  
**Tabs :** call string list editor and add the following lines.  
standard  
categories

Under standard tab control we are having **SpeedButton** components. The Name property of each of them is set to **New file, Open file, Close file, Save, Exit, Cut, Paste, Previous page and Next page.**

Under categories tab control also we are having **SpeedButton** components. The Name property of each of them is set to **Comments, Document structure, Document headings, Layout elements, Paragraphs, Graphics, Links, Lists, Text controls, Query, Help and Code\_generator.**

We are using **SpeedButton** component's **Glyph** property to load a **TBitmap** object. We are setting **Hint** property of all the buttons as their **Name** property and setting **ShowHint** property to true. It will show you what each button is for.

**ComboBox :**

**Style :** csDropDown  
**Visible :** False

**BlkBtn :**

**kind** : **bkCustom**  
**Caption** : **OK**

**MainMenu :**

**items** : Using this property we are entering the following items in the Menu Designer.

The structure of the menu designer is :

**Text\_Entry :**

This form is used for entering multiple lines of text. it's properties are :

**Name** : **Text\_Entry**  
**Caption** : **Edit Area**

It uses one Memo component whose properties are :

**Align** : **alClient**  
**Alignment** : **taLeft**  
**BorderStyle** : **bsSingle**  
**Color** : **clWindow**  
**Font** : **Name** → **system**  
**Style** → **Bold**  
**Size** → **10**  
**Color** → **Navy**  
**Name** : **Editing**  
**ScrollBars** : **ssBoth**  
**Visible** : **True**  
**WantReturns** : **True**  
**WantTabs** : **True**  
**WordWrap** : **True**



**Coding :**

This form is used to output the HTML code. It's properties are :

**Caption** : HTML\_code  
**Name** : Coding

It uses one Memo component whose properties are:

**Align** : alClient  
**Alignment** : taLeftJustify  
**BorderStyle** : bsSingle  
**Color** : clGreen  
**Font** : Name → system  
Style → Bold  
Size → 10  
Color → Navy  
**Name** : Code  
**ScrollBars** : ssBoth  
**Visible** : True  
**WordWraps** : True

**Back :**

This form is used as a container of all the above three forms. It's properties are:

**Caption** : HTML Editor (operational mode 1)  
**Color** : clNavy  
**FormStyle** : fsNormal  
**Name** : Back  
**Visible** : False  
**WindowState** : wsMaximized

**ii. Functions to be performed:**

In the first operational mode, on-line help is created by using the **HC31.EXE** utility, a copy of which is provided with Delphi. Including the help file in this project is a three step process.

1. Create the **RTF (Rich Text Format)** file, using the **macro file** as the specification [2].

- 2. Direct HelpGen [2] to call the help compiler. It automatically hands over the compiler the project file with the RTF file to generate the help file.**
- 3. Now include the help file in the project. The procedure for including the help file is given below.**
  - **Create on-click event handler of the Help button (under categories page tab).**
  - **Put the code for calling the help file in that on-click event handler procedure.**

This help file tells the user how the HTML document can be created using this mode of the editor. It also informs the user how HTML document looks like and what the structure of the HTML document is.

Create on-click event handler for all the items of the **MainMenu** component. The algorithm for all the event handlers (except **New**) is the same as discussed in the *implementation details* of the 'HTML document editor' in the last chapter. Here for all the functions, the focus will be on the memo, named **Code**. Here, we are using **setfocus** function to set the focus on the form, named **Coding**.

For all the **SpeedButton** components (except **New File**, **Previous Page** and **Next Page**) under the **standard** page tab, we are creating on-click event handlers and following the same algorithms which are used by the **MainMenu** component. Here, our focus is on 'Code' memo component. When saving the HTML document, the save dialog box appears and asks for the name of the document in which it has to be saved and the contents of the memo component will be saved with the extension of **.htm**. When a new HTML file is created (by clicking **MainMenu**'s **New** command or clicking **New File** button) then the strings **<HTML>** and **<HEAD>** will be added into this memo component (Because each and every HTML document contains those two tags, so these are the default tags). When we want to open a file then only **.htm** file will be opened. We are doing this by setting the **filter** property of the **OpenDialog** component, which is also discussed in the previous chapter. **keep\_track\_previous()** function will keep track of the previous file and **keep\_track\_next()** will keep track of the next file. When on-click event occurs on the **SpeedButtons** whose **Caption**

properties are **Previous Page** and **Next Page** then call the above functions respectively and load the file into this 'Code' memo component.

For all the **SpeedButton** components under the **categories** page tab we are creating on-click event handlers. If user clicks any button (except **Help**, **Query Code\_generator**) from the **categories** page tab then the **ComboBox** will be displayed and tags from the each category will be assigned to the **items** property of the **ComboBox** (So, when user will choose drop down list then all tags under that category will be displayed) . If user selects any tag from the drop down list then **BitBtn** will be displayed. If user selects **OK** (**Caption** property of the **BitBtn** component) then corresponding to that tag, the code will be displayed in the "Code" memo component. If corresponding to that tag some text has to be entered then the focus will be transferred to the "Editing" memo control. After entering some text if user selects **Code\_generator** button then the HTML code for that, will be displayed in "Code" memo control. For **<TITLE>** (category **Document Headings**) and **<A>** (category **Links**) tag , the dialog boxes will appear and prompt the user to enter the 'page title' and to enter the 'hypertext link with the URL' respectively. Here, we are using **InputQuery** function for displaying the dialog boxes.

Now we will see what should happen if user clicks the **Query** button from the **categories** tab control. On-click event handler for **Query** button is given in the following.

1. Open your own dialog box where it asks the name of the database alias, query file and the text which will act as a hyperlink. [ A dialog box is shown in **Appendix B**]
2. Add the equivalent code [refer to the syntax for query execution which is discussed in the next chapter] in 'Code' memo control.

### **5.3.3 Design (Operational mode 2):**

**Input specification** : **Text**.

**Output specification** : **Hypertext document (Front end output).**

**HTML documents (Back end output).**

pages. This technique is analogous to the Depth First Search technique used in Graph traversals.

This editor allows the user to store the equivalent HTML document of each and every page in the back end and also allows to take a hard copy of the hypertext pages. We can also execute the SQL queries on the database application through the hypertext document, created in this manner.

#### 5.3.4 Implementation of the editor (Operational mode 2):

##### i. Components used and their properties :

In this mode, we use two forms. The name of the forms are: 1. ParentForm and 2. ChildForm.

The properties of these forms and the components used by them are given below.

#### ***PROPERTIES :***

##### **ParentForm :**

This form is used as a control panel to allow user to select the HTML categories and to operate on the current ChildForm. The properties are :

<b>Caption</b>	<b>:</b>	<b>HTMLEditor (operational mode 2)</b>
<b>FormStyle</b>	<b>:</b>	<b>fsMDIForm</b>
<b>Name</b>	<b>:</b>	<b>MainForm</b>
<b>WindowState</b>	<b>:</b>	<b>wsMaximized</b>

This ParentForm uses the following components. Their properties are :

##### **Memor1 :**

<b>Align</b>	<b>:</b>	<b>taCenter</b>
<b>BorderStyle</b>	<b>:</b>	<b>bsSingle</b>
<b>Color</b>	<b>:</b>	<b>clWindow</b>
<b>Name</b>	<b>:</b>	<b>Memor1</b>
<b>ScrollBars</b>	<b>:</b>	<b>ssBoth</b>
<b>WantReturns</b>	<b>:</b>	<b>True</b>

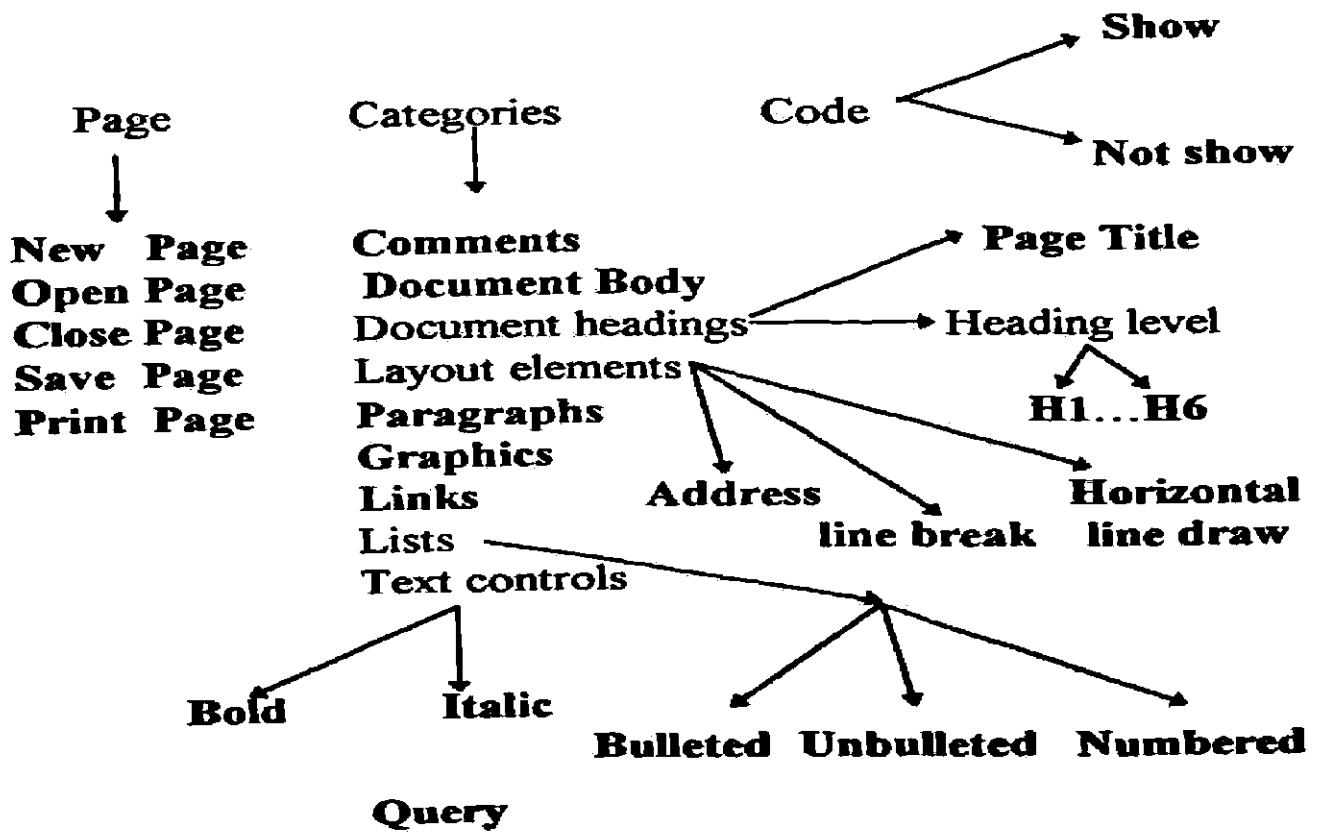
**WantTabs** : **True**  
**WordWrap** : **True**  
**Visible** : **False**

**Memo2 :**

**Align** : **taLeftJustify**  
**Color** : **clTeal**  
**Font** : **Name** → **Arial**  
**Style** → **Regular**  
**Size** → **14**  
**Color** → **Red**  
**Name** : **Memo2**  
**ScrollBars** : **ssBoth**  
**WantReturns** : **True**  
**WantTabs** : **True**  
**WordWrap** : **True**  
**Visible** : **False**

**MainMenu :**

**Items** : Using this we are entering the following structure in the Menu Designer.



### **ChildForm :**

**This form is used to edit and view the hypertext page. This form will operate in WYSIWYG mode. Output of this form is WYSIWYG output and the user can take the hard copy of this output.**

<b>Caption</b>	<b>:</b>	<b>None</b>
<b>FormStyle</b>	<b>:</b>	<b>fsMDIChild</b>
<b>Name</b>	<b>:</b>	<b>ChildForm</b>
<b>WindowState</b>	<b>:</b>	<b>wsNormal</b>
<b>BorderStyle</b>	<b>:</b>	<b>bsSizable</b>
<b>Color</b>	<b>:</b>	<b>clWindow</b>
<b>Position</b>	<b>:</b>	<b>poDefault</b>
<b>Visible</b>	<b>:</b>	<b>True</b>

### **ii. Functions to be performed:**

#### **On-click event handler for New Page:**

- Create the ChildForm by writing the statement **child\_1 := TChildForm.Create(Application)**, where **child\_1** is the type of **TChildForm**.
- Put **<HTML>** and **<HEAD>** tag in **Memo2** by using **Memo2.Lines.Add** method.

#### **On-click event handler for Open Page:**

- Open the ChildForm file. That is converting text files into binary file i.e., **.DFM** format.

#### **On-click event handler for Close Page:**

- Add **</BODY>** and **</HTML>** tag in **Memo2** component respectively.
- Save the ChildForm file (**.DFM** format) into text file (**ASCII** format).

- **Close the active ChildForm.** It means disposing the ChildForm by writing the statement `Action := caFree`, where `Action` is of type `TCloseAction`.
- **Load the previous ChildForm if any.**

#### **On-click event handler for Save Page:**

- **Save the current ChildForm file (.DFM format) into text file (ASCII format).**
- **Return to the current ChildForm.**

#### **On-click event handler for Print Page:**

- **To print a ChildForm, call its print method, which paints the form's client area on an offscreen bitmap. The method uses the Printer object's BeginDoc and EndDoc techniques to print the resulting bitmap.**
- **Call NewPage to eject each page after printing.**
- **Open a page (i.e. another ChildForm) and repeat the above steps.**

#### **On-click event handler for Comments:**

- **Memo1 component will appear (Visible property is set to true) asking for your comments.**
- **If MemoMouseDown then capture the position. Change the mouse cursor.**

```
procedure TParentForm.MemoMouseDown(Sender: TObject; Button
      : TMouseButton; Shift : TShiftState; x, y : Integer) ;
```

```
begin
```

```
    start_pos.x := x ;
```

```
    start_pos.y := y;
```

```
    CursorTo(start_pos.x , start_pos.y) ;
```

```
    cursor := crHsplit ;
```

```
end ;
```

- **Put the corresponding tag and the comments into the Memo2 component at that position.**

### **On-click event handler for Document Body:**

- Put the **<BODY>** tag into the Memo2 component.
- Go on capturing the **OnKeyPress** event and print each character on the canvas of the ChildForm. Here, we have to keep track of the current position on the canvas where we want to print our current **KeyPress** event. Before start to print the character on the canvas, give a font to the canvas (In the following section we will call this font as body's font).

### **On-click event handler for Page Title:**

- Open the **InputQuery** dialog box, asking for the page title.
- ChildForm's **Caption** property is set to the entered text.
- Correctly place the corresponding opening tag, text and closing tag into the Memo2 component.
- Put **</HEAD>** tag in Memo2 component (It implies the closing of the HTML document head).

### **On-click event handler for all levels of Heading (H1, ..., H6):**

- Check the selected heading level with the previous entry in the **heading\_list**. Previous entry will be one level below of the recent selected heading level. (This is for checking whether any heading level skipped or not). If this check returns true then follow the next steps else asks for correct selection.
- Memo1 component will appear asking entry for heading.
- Follow second step of the "On-click event handler for comments" to capture the cursor position to start with.  
(Here, if **ChildForm.FormMouseDown** then capture the position).
- With **ChildForm.Canvas** do
  - i. Give a font to the canvas.
  - ii. Start to print each character on the canvas with that font.



- Add the equivalent HTML code into the Memo2 component (We have to consider the starting position of the cursor in step 3 while adding the code in Memo2).

#### **On-click event handler for ADDRESS:**

- Memo1 component will appear asking for author's address.
- If ChildForm.FormMouseDown then capture the position.
- Contents of the memo1 component will start to print in the ChildForm canvas at the position of (start\_pos.x, start\_pos.y).
- Put the corresponding tag and the address into the memo2 component.

#### **On-click event handler for Line Break:**

- Give a line break on the ChildForm and correspondingly update the value of start\_pos.x and start\_pos.y.
- Put the equivalent HTML code in Memo2 component.

#### **On-click event handler for Horizontal Line Draw:**

- Draw a line on the ChildForm canvas using the method **Canvas.Draw** and modify the start\_pos.x and start\_pos.y value.
- Put the equivalent HTML code in Memo2 component.

#### **On-click event handler for Paragraphs:**

- Memo1 component will appear asking for entering the paragraph.
- procedure TChildForm.FormMouseDown (Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);  
begin  
    if (Shift = [ssRight]) then  
    begin  
        make a paragraph ;  
        update the next starting position on the canvas;  
    end;  
end;

(If you press the right mouse button then the paragraph will be made on the ChildForm and the starting position to print with will be modified).

- Add the corresponding equivalent HTML strings in Memo2 component.

#### **On-click event handler for Graphics:**

- Create a dialog box which will ask you to enter the source file from where you want to load the image.
- Capture the mouse cursor position by following second step of the “On-click event handler for comments” algorithm.
- Put the image at that position on the canvas.(How to put the image, is already discussed in the browser module)
- Add the equivalent HTML code in Memo2 component.

#### **On-click event handler for the type of List items (Bulleted, Unbulleted, Numbered):**

- Memo1 component will appear asking for entering the list items.
- Each line of the Memo1 will be considered as a single item of the list.
- Depending on the type of list we will print lines of the Memo1 on the canvas of the ChildForm. For bulleted output we will create our own image using Delphi’s image editor. We will draw the image on the canvas (at 0,start\_pos.y position) as many as times as the number of lines of Memo1 are. We will print each item of the list and start\_pos.y will be updated. For numbered output we will place a number before each of the item. Number will start from 1 and go on increasing until Memo1.Lines.Count is equal to zero.
- Place the equivalent code in Memo2 component.

#### **On-click event handler for Bold and Italic:**

- Follow the third step of “on-click event handler for all levels of heading” algorithm.  
 For bold : Font.Style := [fsBold]  
 For italic : Font.Style := [fsItalic]

- When we will press the **Text Controls** button for the second time then change the font style of the canvas to regular. It means the next text will be printed using the font which is used by the body.
- Add HTML code into the Memo2 component.

#### **On-click event handler for Links:**

- Open the dialog box to enter the text to be highlighted, which will act as a hyperlink to another page. Next open the dialogbox to enter the name of the page to be linked with.
- After pressing the OK button of the second dialog box, the highlighted text (here highlighting the text means underlining it) will be printed on the canvas. Collect the text boundaries and the page name for this text (procedures for these are given in the browser module) and create a ChildForm with that name. (before creating we have to save the previous ChildForm). After the OK button is pressed, the HTML code will be added into the Memo2 component.
- Now go on making your page by following all the previous algorithms.

For the above link purposes we should have **OnMouseMove** event handler. If the mouse cursor comes within the text boundaries (the highlighted hyperlink) and if **OnMouseClicked** event occurs within this text boundary then display the page which is linked with that highlighted text (the linking procedure is discussed in the browser program).

#### **On-click event handler for Show:**

- Set the visible property of the Memo2 component to true. (This is to view the generated HTML document)

#### **On-click event handler for Not Show:**

- Set the visible property of the Memo2 component to false. (The container of the HTML code will be disappeared)

### **On-click event handler for Query:**

- **Open the dialog box and enter the following :**
  - **text to be highlighted**
  - **database alias and**
  - **query file**
- **After the OK button of the dialog box is pressed, it will print the highlighted text on the canvas. Collect all the above three entries (procedures for this is given in the browser program). After OK button is pressed, the equivalent syntax for the query execution will be added into the Memo2 component.**

To execute the query through the hypertext document we should have **OnMouseMove** event handler. If the mouse cursor comes within the text boundaries (the highlighted hyperlink) and if **OnMouseClicked** event occurs within this text boundary then execute the query associated with that highlighted text (procedure for query execution is discussed in the browser program).

### **5.4 Conclusions:**

This editor works in two operational modes. If the first operational mode is chosen then the user will get some idea of the structure of the HTML document while creating the HTML documents. These HTML documents are needed by the browser module to interpret into a hypertext document. If the second operational mode is chosen then the user can create his hypertext document directly in the front end without specifying the contents and behavior of the document in HTML format. This editor does not allow to incorporate other tags which are not discussed in chapter 3. We used form's canvas property to print on the form in the second mode of the editor. We used Memo component for entering the text and storing the generated HTML code. This Memo component also displays the code by setting it's visible property to true.

**[The results are shown in Appendix B.]**

## BROWSER

### 6.1 Introduction

In this chapter we will discuss the design and implementation details of the browser that reads the HTML documents and generates hypertext document. Each HTML document is needed to generate each page of a hypertext document. This browser, except the parser part which is taking care of executing queries (for executing the SQL queries on a database application we are using our own specification), is programmed according to HTML specifications.

The organization of the chapter is as follows. In sec. 6.2, we provide the design, in sec. 6.3, we discuss the implementation details of the browser, in sec. 6.4, we discuss about query execution through hypertext and finally sec. 6.5 concludes the chapter.

### 6.2 Design of the browser

Input specification : HTML documents.

Output specification : Hypertext document.

The following diagram shows the block diagram of this module.



**Fig. 6.1. Block diagram of the browser**

HTML document contains all the tags and the text. The text comes between the opening and the closing tags. Each tag has a different meaning and a unique syntax. Browser program interprets each tag and performs an operation on the text which is enclosed by the opening and the closing tags.

## 6.3 Implementation details

### 6.3.1 Browser's actions according to each tag:

We use a function named `processtags` which has one string type argument. Depending on the type of arguments, there is an action performed. We summarize below in a table all actions for different types of tags.

#### Tags

#### Actions

<b>HTML</b>	Recognizes that it is a HTML document.
<b>HEAD</b>	Recognizes that the document is having a header. Here <code>textvis</code> (Boolean variable) is set to true.
<b>TITLE</b>	<code>textvis</code> is set to false because for this tag we don't want to print the text which is enclosed by <code>&lt;TITLE&gt;</code> and <code>&lt;/TITLE&gt;</code> . Store the text which will be used afterwards.
<b>/TITLE</b>	<code>textvis</code> is set to true and change the form <code>caption</code> property to the stored text.
<b>BODY</b>	Assigning a value to the variable <code>font</code> (an integer variable defined in public declaration). This variable is keeping track of the current font of the current text. How we are giving the font to the text is discussed below. The text which comes after this tag will be printed in that font. The printing procedure is discussed below.

<b>H1 .. H6</b>	For all these six level headings we are assigning a value of the variable font. The text which will come after the tag will be printed in that font.
<b>/H1 .. /H6</b>	When these tags are appearing (closing tags of all the six level heading) then we are assigning body's font to the current text. We need that when the structure is <b>&lt;BODY&gt;</b> <b>&lt;H1&gt;heading1&lt;/H1&gt;</b> <b>my_body .</b> Here, heading1 will be printed in the font of H1 and my_body will be in the font of BODY.
<b>P</b>	This will make a paragraph. It will skip two row and five columns and the text will start printing in the font of body.
<b>!-</b>	This tag supports author's comments. Browser should not display them. So the textvis will be set to false. The syntax of this tag is <b>&lt;!- Author's comment.&gt;</b> .So, the textvis will be false until the next character is <b>'&gt;'</b> . Here, 'Author's comment will not be displayed.
<b>ADDRESS</b>	Display the text in a particular font until you are founding the corresponding closing tag i.e., <b>/ADDRESS</b>
<b>/ADDRESS</b>	Change the font value to body's font value.

<b>BR</b>	Skip the row and set the focus into the next row.
<b>HR</b>	Draw a line which will start at (0 , current_column) and will be extended upto(form's clientwidth, current_column).
<b>DIR</b>	Read each line (length(line) < 20) and print using body's font. This will go on continue until you are finding it's closing tag /DIR. These lines are acting as a unbulleted list of short elements.
<b>UL</b>	Set the value of the Boolean type variable <b>unnumbered_list</b> to true.
<b>OL</b>	Set the value of the Boolean type variable <b>ordered_list</b> to true.
<b>L1</b>	Within a list of any type it marks a member item. Depending upon the above two variables the member items are to be printed in the fashion of bulleted list or numbered list. Bullets can be put by creating the image of bullet and loading that image. How to create and load the image will be discussed below.
<b>DT</b>	This tag represents the term being defined. Print it like a heading but with a font of body and the font size is somewhat bigger than the body's font size. The term will be put into the glossary listbox.



<b>DD</b>	This tag represents the definition for a term. Skip some column and start to write below the term being defined (see the example (for DT and DD) discussed in chapter 3). We will store this definition of the term in a file where the name of the file is same as term. when we will click the glossary from the browser main window, then visible property of the ComboBox will set to be true. After selecting one item from the dropdown list, if we press OK then the file (same as the name of term) will be loaded into the memo component .
<b>B</b>	Go on printing the text using bold style of the font.
/B	Change the font style from bold to normal i.e., change the current font of the text to body's font.
<b>L</b>	Go on printing the text using italic style of the font.
/L	Change the font style from italic to normal i.e., change the current font of the text to body's font.

<p><b>IMG</b></p>	<p><b>Check for a IMG string in a tag. If it is there then call</b>  <b>Image_tag_process function</b>  <b>else</b>  <b>call processtags function.</b></p> <p><b>This Image_tag_process will display the image. How it is displaying the image is discussed below.</b></p>
<p><b>A</b></p>	<p><b>This tag provides fundamental hypertext link capabilities.</b></p> <ul style="list-style-type: none"> <li><b>* First highlight the text between the opening and closing tags.</b></li> <li><b>* If mouse cursor comes within this text then change the cursor shape to upward pointing hand.</b></li> <li><b>* If we click the text then the next page whose path is specified in the tag will be displayed. How we achieve this will be discussed below.</b></li> </ul>

### **6.3.2 Components used :**

The browser module uses only one **Form** (This is for printing the characters on the form's canvas, using form's **Canvas** property), one **Memo** component (used for showing the definition of the term which is defined in the glossary list box) and one **ComboBox** component (used as a glossary list box). Another component, **Query** is used for execution of database queries.

### **4.3.3 Event handlers and their main functions :**

#### **Form OnCreate event handler :**

**i. Call a procedure FillFontStruct which creates a TFont object.**

**For example, to create two objects of this type the code will look like :**

```
for i:= 0 to 1 do           where FontStruct is an
FontStruct[i] := TFont.Create.  array of Tfont.
```

**Now to change the font see the following example .**

```
FontStruct[0].Name := 'ARCHYLE' ; /* name of the font */
FontStruct[0].Size  := 14 ;        /* size of the font */
FontStruct[0].Color := Clred ;     /* color of the font */
FontStruct[0].Style := [fsBold] ;  /* font style */
```

**ii. Setting the screen cursor by calling the function LoadCursor.**

#### **Form OnActivate event handler :**

Show a dialog box by calling the **InputQuery** function. It will ask to enter the path of the HTML document which is needed to display the first page of the hypertext document. Store the file in the variable **html\_file** (this variable is keeping track of the current HTML document).When user will click OK button then call **Display\_Page** function. **Display\_Page** will perform the following functions :

- Load the current **html\_file** into the memo component .
- Perform the actions corresponding to each tag (discussed above)

#### **Form OnMouseMove event handler :**

If the mouse pointer comes within the region of the text which will provide hypertext link capability then change the screen cursor to upward pointing hand (load the cursor which is created by using Delphi's Image editor) else set the cursor as default screen cursor(**Screen.Cursor := crDefault**).To load the cursor, add the following statement.

```
Screen.Cursors[cursor_ID]:=LoadCursor(hInstance,
                                     'HAND_CUR');
```

To load the cursor we have to include the resource file of the cursor just below the Implementation section by specifying {SR SAHA.RES}.

**Form OnMouseDown event handler :**

```
if RightButton pressed then
begin
if PtInRect(links[i].rect , POINT(x,y)) then
begin
html_file := links[i].filename ;
Display_Page ;
end;
end;
```

Where, **links** is an array of **link** and **link** is the type of record.

Above algorithm checks whether right mouse button has been pressed on the hotspot or not. If it has been pressed then display the page which is having link to that hotspot.

The structure of the **link** is

```
link = Record
filename : string ;
rect      : TRect ;
end;
```

The above structure of the algorithm is needed to keep track of all the links. It has to know which file should be linked for a particular highlighted text.

#### **6.3.4 Important functions:**

We have given some important functions in the following which are needed by the browser while processing the tags.

**Image\_tag\_process** function:

This function collects the image source file from the opening tag and stores it in a variable **source** of type string. This function also collects the size of the image from the **IMAGE** tag attribute (they are **HEIGHT** and **WIDTH**). Now it will call

the function `process_source` with the argument `source`, value of `HEIGHT` attribute and value of `WIDTH` attribute.

**process\_source function:**

It will load the image file and the image will be drawn on the canvas of the form. The function will look like :

```
process_source(s:string ; height1:integer ; width1:integer) ;
var
bmp : TBitmap ;
begin
bmp := TBitmap.Create ; /* creating the bitmap */
bmp.height := height1 ; ← specifying the bitmap
bmp.width := width1 ; ← size.
bmp.LoadFromFile(s); /* loading the image file into
Form1.Canvas.Draw(xc, yc, bmp); /* bmp */
bmp.Destroy ; /* destroying the bmp */
end;
```

**print\_char function:**

When `textvis` is true, calling the function `print_char(ch : char)`. `print_char` will read each character and print onto the canvas of the form. Before printing we have to assign the font of the canvas by writing the statement :

```
Form1.canvas.Font := FontStruct[font] ;
```

Now depending on the value of the font, the character will be printed in that font. `print_char` has to keep track of the current X,Y position on the canvas. Here, the character will be displayed at that current position by calling the function

```
Form1.Canvas.Textout(xc , yc, s) ;
```

where (xc,yc) is the current (X,Y) position on the form's canvas. s is the string conversion of the character ch.

Initially the browser program will ask you to enter the path of the HTML document which is needed to display the first page of the hypertext document. When browser program is running, if any requisite HTML document is found missing then it asks if the document has to be created. If you want to create the file then the program will call Notepad by calling `WinExec()` function and allows you to create and save

your file in the same directory as your first HTML document. If you don't want to create the file then the browser program will continue running but, when the hotspot (for which the HTML document is missing) is clicked an error message will be displayed informing that no such file exists with that name which is specified in your HTML document for that hotspot.

## 6.4 Query Execution

### 6.4.1 Syntax used :

We use the following syntax :

```
<SQL Database_alias Query_file> Text_to_be_highlighted </SQL>
```

Corresponding the above syntax the way the database queries will be executed are discussed in chapter 7.

### 6.4.2 Browser's action :

- By recognizing the string SQL it will check the next fields. If there are two entries then it proceeds else displays an error message stating that the syntax is wrong.
- The form's **OnMouseDown** event is the same as discussed before. But here, the structure of the link is

```
link = Record  
filename : string ;  
alias    : string ;  
rect     : TRect ;  
end;
```

Where **filename** is for storing *Query\_file* (name of the file which contains SQL script written using SELECT statement only), **alias** is for storing *Database\_alias* (name of the database alias) and **rect** is for storing the boundary of *Text\_to\_be\_highlighted* (i.e., the boundary of the hotspot).

When user will click (by using the right mouse button) on the hotspot then the queries specified in the *Query\_file* will be executed on the database specified by the *Database\_alias*. The algorithm for executing the queries is given below:

### **Algorithm :**

```
If RightButton pressed then
begin
  if PtInRect(links[i].rect , POINT(x,y)) then
    begin
      query_file := links[i].filename;
      Assign the database alias name to the DatabaseName
      property the Query component;
      (For example, Query1.DatabaseName := Database_alias.)
      Execute the query on the database and show the result;
      (refer to chapter 7) ;
    end;
  end;
```

Where Query1 is a component of TQuery type.

### **6.5 Conclusions:**

In this chapter we have discussed the actions to be performed by the browser module when it interprets the tags. We have implemented the browser in such a way that the browser program takes HTML documents as the input and interprets them into a hypertext document. Thus, we are successful in providing facilities to the user in building the hypertext document, just by specifying the contents of the hypertext document in the HTML syntax. The browser module uses form's Canvas property to print the characters on the form's canvas. While running, depending on the hyperlink and Uniform Resource locator, the browser will go on loading the HTML files in a Memo component. This module also interprets the syntax for query execution defined by us and takes appropriate actions.

[A sample input and output of this browser module is shown in **Appendix B.**]

# **DATABASE APPLICATION**

## **7.1 Introduction**

In introduction chapter we mentioned the need for a database application. Since we want to execute the SQL queries through hypertext on the database. Hence we have chosen a database problem from the finance section of our university. This is a sample database application. The procedure for 'execution of SQL queries on the database through the hypertext' is valid for all the database applications.

The organization of the chapter is as follows. In sec. 7.2, we discuss the problem specification, design of the database in sec. 7.3, implementation details in sec. 7.4, and finally query execution through hypertext in sec. 7.5. Sec. 7.6 concludes the chapter.

## **7.2 Problem specification**

In this database the accounts are to be maintained on the criterion of funding body. Separate accounting for receipts and payments is done. In the following we will give the description the database tables, the functions to be performed, reports to be generated and the queries involved in this database.

### **7.2.1 Tables used**

This database uses five tables :

- 1. Funding agency table.**
- 2. Project table.**
- 3. Budget table.**
- 4. Payment Voucher table.**
- 5. Receipt Voucher table.**

The fields used by the above database tables are given below.



**1. Funding agency table.**

**Fields**

- 1. Funding\_body**
- 2. Funding\_body\_id**

**2. Project table:**

**Fields**

- 1. Project\_id**
- 2. Voucher**
- 3. Funding\_id**
- 4. Project\_investigator**
- 5. Department**
- 6. Date of commencement**
- 7. Duration in years**

**3. Budget table:**

**Fields**

- 1. Project\_id**
- 2. Salary**
- 3. Fellowship**
- 4. Contingency**
- 5. Equipment**
- 6. Books**
- 7. Consumables**
- 8. Overheads**
- 9. Traveling advances(T.A)**
- 10. Miscellaneous advances(M.A)**
- 11. Grant**

**4. Payment Voucher:**

**Fields**

- 1. Project\_id.**
- 2. Funding\_body**
- 3. Voucher\_no**
- 4. Bill\_no**
- 5. Cash\_book / Cheque no.**
- 6. Date**

## 5. Receipt Voucher:

2118

### Fields

1. Project\_id
2. Cheque\_no / DD no.
3. Date
4. Refund

## 7.2.2 Functions to be performed on the database

Separate accounting for receipts and payments is done in this database, where payments are of three types: 1. Direct payment, 2. Advances, and 3. Adjustments.

### **DIRECT PAYMENT**

Direct Payment relates to expenditure booked directly to a project. Under each project there are different subheadings or subaccounts. Therefore all expenditures are classified under each project and under each subheading. These subheads are :

1. Salary
2. Fellowships
3. Contingency
4. Equipment
5. T.A
6. Books
7. Consumables
8. Overheads

### **ADVANCES**

They are of two types: i. Miscellaneous advances, ii. Temporary advances.

**i. Miscellaneous Advances:** This is a payment in advance to the supplier pending receipt of a material, initially booked under advance subhead of a project. After receipt of a material advance subhead is credited and final subhead like equipment, consumables etc. are debited.

**ii. Temporary Advances:** This is an advance paid to the project investigator in cash. Project account is not debited for this amount. Initially it is shown as an advance in

the cash book. Only after account is rendered and bill adjusted it is debited to the project and corresponding subhead.

## ADJUSTMENTS

This relates to rectification's of mispostings through transfer entry. Correct head or sub head is debited by giving credit to other account head.

### 7.2.3 Reports to be generated

The reports to be generated are:

#### REPORT 1 :

Project Account:

Funding body:

Name of the project:

---

Date	Cash.Book_no	Voucher no	Particulars	Subhead	Amount
.	.	.	.	.	.
.	.	.	.	.	.

---

#### REPORT 2

Statement of receipts and expenditure in respect of funding\_body(ex CSIR) project

Entitled " ... "

of Prof. ... School of ...

#### 1 . Receipts:

Year	Amount
------	--------

.

.

**Total :**

## 2.Expenditure:

Year	Salary	Contingency .....	Equipment	Total
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
<b>Total</b>	...	...	...	...

**Grants received :** ...  
**Expenditure incurred :** ...  
**Balance as on specified date :** ...

**Asst Finance Officer(Accounts).**

### 7.2.4 Queries

The queries which are involved in this database are:

1. List of project investigators in a department.
2. List of projects of an agency in a department.
3. List of projects of a project investigator in a department.
4. Monthly payments of a project.
5. Monthly receipts of a project.
6. Monthly balances of a project.
7. Projects for which advances were taken but not adjusted.
8. List of projects where there is no progress in terms of expenditure.

## 7.3 Design of the database

### Specifications of the database as per RAISE:

The RAISE [Appendix A] specification of the above database is given below.

**DATABASE =**

**Class**

**type**

Key, Data, Record, Person, Project, Payment, Receipt, Ttype ,Agency,

Table = Record-set,

Tset = Table-set,

Ttype-set = {Funding agency table, Project table, Voucher table,  
Receipt table, Budget table }

**value**

Table\_type : Table → Ttype,

List\_project\_investigator : Table → Person-set,

List\_project\_agency : Table → Project-set,

Project\_investigator : Person → Project-set,

Project\_monthly\_payment : Project → Payment,

Project\_monthly\_receipt : Project → Receipt,

Project\_advances\_not adjusted : Table → Project,

Project\_no\_progress : Table → Project

**end**

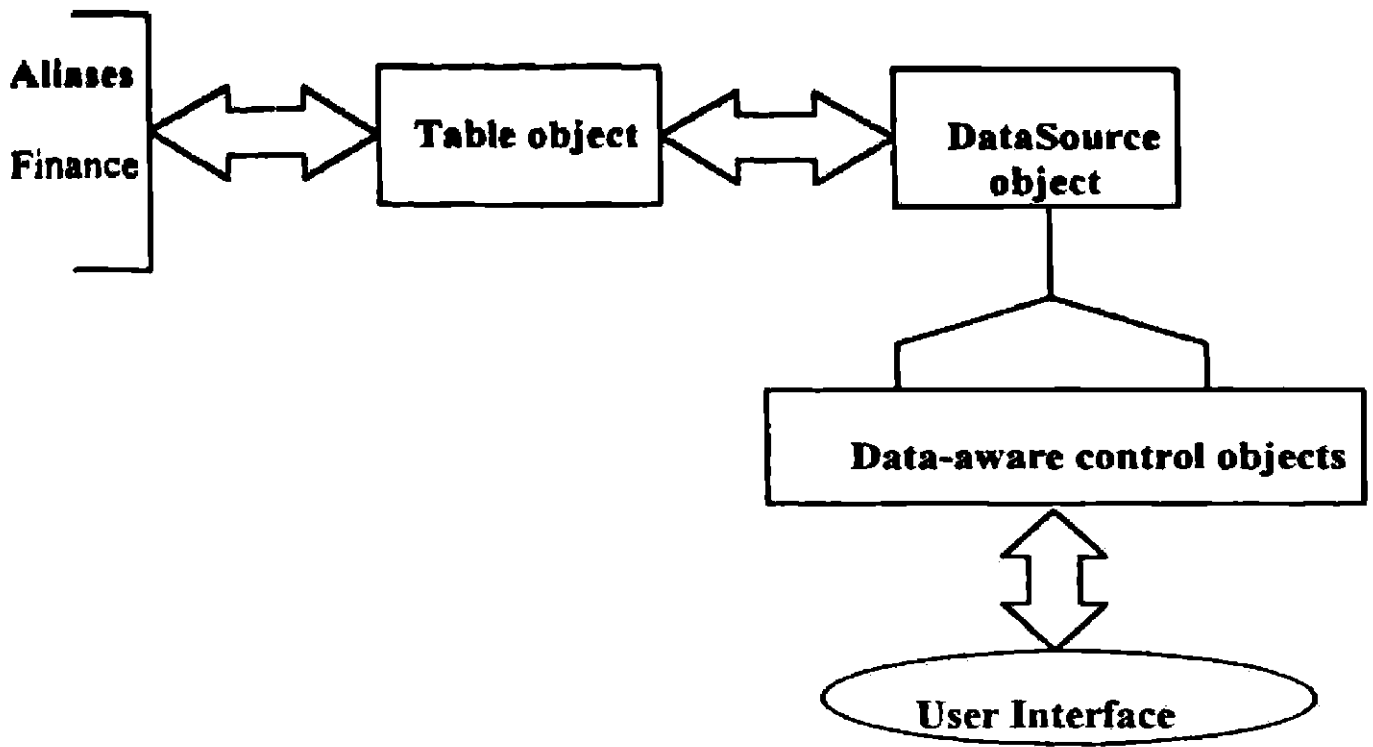
## 7.4 Implementation Details

Delphi's Desktop edition, which includes the Borland Database Engine (BDE), provides a complete set of programming tools for many popular desktop database systems such as dBase and Paradox. Delphi's database components put an object-oriented face on database application development. Even more important, database components standardize access to databases in a variety of formats. This means an

application can access data in dBase files, Paradox tables, Microsoft Access and other Open Database Connectivity (ODBC) systems, or if you have the Client/Server edition, through remote SQL servers. Best of all, you can use all other Delphi components, interface techniques, and Object Pascal programming in your database applications.

The components on the **Data Access Palette** provide access to databases and objects in your application forms. You have to use these components as gateways to database information. In most cases, we will need instances of the two components, **Table** and **Datasource**. The other components perform lookups (**Query**), generate reports via the Reportsmith program (**Report**), and perform global operations such as updating all fields in matching records (**BatchMove**).

The above implementation had been performed by using **Paradox**. The first component to use is **Table**, which creates a bridge between an application and a *database alias* (here, **Finance**) as shown in the figure 7.1. In addition to a table, we'll need a **DataSource** object, which links *data-aware controls* to the database. The **DataSource** object feeds data to and from other objects and the **Table**. The **Table** object handles the actual transactions for the database. All of this takes place courtesy of the **BDE**, which performs the real work of reading and writing data in whatever format you have selected (here, **Paradox**). The data-aware controls might be additionally linked to create interactive data-entry screens. For example, linking a **DBNavigator** object to a **DataSource**, which is connected to a **Table**, creates a browsing toolbar that you can use to *view*, *edit*, *insert*, and *delete* records displayed in **DBEdit** and other control windows.



A table object forms a bridge between the application and a database, identified by a registered alias. A DataSource object links the Table with data-aware controls such as DBEdit and DBNavigator.

Fig. 7.1

## 7.5 Query Execution through hypertext

In order to execute the SQL queries on the database through hypertext, we should have a component which takes care of this query execution. Delphi is providing one such component called **Query** component. We use the following syntax for query execution.

### 7.5.1 Syntax

We defined the following syntax for query execution on the database application through hypertext.

**<SQL Database\_alias Query\_file> Text\_to\_be\_highlighted </SQL>**

Now consider the following query and see how the query is getting executed in Delphi [The browser's action corresponding to the above syntax is discussed in chapter 6 ]

### 7.5.2 A database query and its execution in Delphi

#### Query:

"List of project investigators in a department"

#### Execution procedure of the above query:

Following statements are responsible for executing the above query.

1. Query1.SQL.Clear;
2. Query1.SQL.Add('Select \* from pro\_tab where');
3. Query1.SQL.Add('Department = "'+ department\_name+ '"');
4. Query1.Open;
5. Query1.First;
6. While not Query1.EOF do  
    Begin
7. Listbox1.Items.Add(Query1.FieldByName('project\_investigator').asString);
8. Query1.Next;
- End;

#### Implications of the above statements:

**1:** Clears the contents of the SQL property. You should always call **Clear** before specifying an SQL statement. Otherwise, **Add** or **LoadFromFile** append to the existing statement.

**2,3 :** We can specify the text of the SQL statement in one of the following ways:

- Use the **Add** method of the SQL property. We are using the **Add** method in the statement 2 and 3 to specify the SQL statements.
- Use the **LoadFromFile** method to assign the text in an SQL script file to the SQL property.

**4 :** To execute an SQL statement at run time, issue either an **Open** or an **ExecSQL** method. **Open** for SQL statements return a result set (the **SELECT** statement). **ExecSQL** will be used for all other SQL statements, such as **INSERT**, **UPDATE**, **DELETE**, and so on.

**Query1.Open;** {Returns a result set}



**Query1.ExecSQL;** (Does not return a result set)

**5 :** The **First** method moves the cursor to the first record in the active range of records of the dataset.

**6 :** This is a while loop checking for the **EOF** condition. **EOF** is a Boolean property that indicates whether a dataset is known to be at its last row.

**7 :** **ListBox1.Items.Add** will add the specified string in the **ListBox**. The **FieldByName** method returns the **TField** with the name passed as the argument in **FieldName**. Here it returns the field with the name **project\_investigator** from the query component.

**8 :** The **Next** method moves the cursor forward by one record. If the cursor is already on the last record, it does not move.

### **7.5.3 Anatomy of the syntax defined by us:**

Now to understand the query execution through hypertext we have to understand the syntax <sup>that</sup> what we specified earlier [7.5.1]. We have given the anatomy of the syntax in the following.

**Database\_alias:** This specifies the name of the database alias, on which you want to execute your SQL queries. **Alias** is a name registered with the **BDE** (Borland Database Engine) that hides actual drives and pathnames that locate database files. Always use aliases to refer to databases; never hard code file and pathnames in your applications. By using aliases, you can move your database files to other locations — or transfer them to a network — and all your applications will work without modifications.

**Query\_file:** This file contains SQL statements. The query should be written in this file using the **SELECT** statement only.

**Text\_to\_be\_highlighted:** This is the text which will act as a hyperlink for executing the specified query.

### **7.5.4 General algorithm for execution of the queries through hypertext:**

**If OnClick event occurs on Text\_to\_be\_highlighted then**

1. **Query1.SQL.Clear;**
2. **Query1.DatabaseName := Database\_alias;**
3. **Query1.SQL.LoadFromFile(Query\_file);**
4. **Query1.Open;**

```
8. Query1.First;
6. While not Query1.EOF do
  Begin
    Access the result from the query component;
    Show the result;
    Query1.Next;
  End;
```

## **7.6 Conclusions:**

In this chapter we developed a sample database for maintaining the research grants for the university of Hyderabad. The queries on this database can be executed through a hypertext document. The queries are specified in SQL. The specification for the database has been given in RAISE specification language. This application has been completely implemented using Delphi's built in components. Delphi's Table and Datasource components are used in this application along with Query, ReportSmith and BatchMove components as gateways to database information. Query component is also used for execution of the SQL queries.

## **CONCLUSIONS AND FUTURE ENHANCEMENTS**

### **8.1 Conclusions**

Our objectives are to develop a HTML document editor, a HTML editor, a browser and a database application.

Regarding the first objective of “developing a HTML document editor”, we have implemented the editor in such a way that the user can create multiple HTML documents at a time. This is helpful to the users who are conversant with HTML. Such a user can create HTML documents and use the browser to interpret them into a hypertext document.

Regarding the second objective of “developing a HTML editor”, our intention is to facilitate a naive user in building his hypertext documents easily. And we have addressed this need by implementing this editor such that it works in two operational modes. If the first operational mode is chosen then the user gets a cursory idea of HTML while creating a HTML document. And in the second operational mode, the user can create a hypertext document easily without knowing the HTML syntax in WYSIWYG (What You See Is What You Get) mode.

Regarding the third objective of “developing a browser”, we have implemented the browser in such a way that the browser program takes HTML documents as the input and interprets them into a hypertext document. Thus, we have achieved considerable success in facilitating the user in building the hypertext document, just by specifying the contents of the hypertext document in the HTML syntax.

**Lastly, concerning our fourth objective of “developing a database application”, we have provided the user the support needed to know how the database queries can also be executed through their hypertext documents by incorporating our syntax for query execution in their HTML documents.**

## **8.2 Future enhancements**

**The following are the future enhancements:**

- 1. The browser program doesn't acknowledge all the HTML tags. So operation of those tags can be included in this program.**
- 2. The above extension also applies to HTML editor too.**
- 3. Support for hypermedia application development can be incorporated into the HTML editor. Currently only hypertext is being supported.**
- 4. Extension to the browser module to handle multimedia features.**

## **RAISE**

### **1. Raise and its features**

#### **1.1 Why RAISE ?**

**Aim of RAISE is to develop notations, techniques and tools that would enable industrial usage of formal methods in the construction of large software systems. It provides a sound notation with a semantic and a proof system for capturing requirements and expressing the functionality of software. RAISE aims at providing the software industry with a mature means of developing software correct with respect to its specifications.**

#### **1.2 Features of Raise :**

**RAISE stands for Rigorous Approach to Industrial Software Engineering. RAISE was the name of a CEC funded ESPRIT project, and is now the name for a wide spectrum specification and design language, an associated method, and a commercially available tool set. RAISE contains all features such as parameterizable abstract datatypes, modularity, concurrency, non determinism, subtypes—for full development from abstraction to programming languages like ADA and C++, and for formal correctness proofs.**

**RAISE specification language (RSL) is useful for formal specification, design, and development of software. RSL is the most versatile and comprehensive language of its kind available today. RSL permits abstract, property-oriented specification of sequential as well as concurrent systems. It also permits specification and design of large systems to be modularised and permits separate subsystems to be separately developed. It permits low level operational designs to be expressed, to a level of detail from which extraction of final code is straightforward i.e., most of the construction of a system, from specification to design, may be done using one and the same**

formalism, thus facilitating precise, mathematical arguments for correctness of development steps and of other critical properties.

## 2. RAISE specifications

RAISE specification language includes the description of modules. In general a module definition has the form :

```
id =  
    class  
        declaration 1  
        .  
        .  
        declaration n  
    end
```

where  $n \geq 0$ .

A declaration begins with a keyword indicating the kind of declaration to come, followed by one or more definitions of that kind, separated by commas. The kind of declaration can be **type declaration**, **value declaration**, **axiom declaration**. However axiom declaration can be omitted from the module definition.

Consider the following example to understand how a database can be modelled by using the RSL module.

### Example :

Consider the following requirements for an election database:

‘The database is supposed to support the administration of an election by identifying all the people who are currently registered as voters’.

The database must provide the following functions:

1. *register* : Registers a person in the database.
2. *check* : Checks whether a person has been registered in the database.
3. *number* : Returns the number of people currently registered in the database.

Parts of the informal requirements can be modelled by the following RSL module.

### DATABASE =

```
class
  type
    Person,
    Database = Person-set

  value
    empty : Database,
    register : Person * Database → Database,
    check : Person * Database → Bool

  axiom
    empty = {},
    for all p : Person, db : Database * register(p,db) = {p} u db,
    for all p : Person, db : Database * check(p,db) = p ∈ db

end
```

### TYPE DECLARATION

A *type* is a collection of logically related values. Some types are already built-in, i.e., pre-defined within RSL. An example of a built-in type is **Nat**, which contains all the natural numbers represented by literals: 0,1,2. In addition to the built-in types one is allowed to define one's own types. Types can be named in typed declarations. A type declaration has the form:

```
type
  type_definition 1,
  .
  .
  type_definition n
                                where n ≥ 1.
```

In the above example specification there are two such definitions.

The first type definition has the form :

id

It defines the type *Person* as an abstract type. That is a type with no pre-defined operators for generating and manipulating its values, except for  $\neq$  which compares two values of the type to check whether they are equal.

The fact that *Person* is defined as an abstract type reflects the requirements, where no information is given about how people are identified in terms of their name and the like. An abstract type is also referred to as sort and a definition of such a type is referred to as a sort definition.

The next type definition, which has the form :

$$id = type\_expr$$

is an abbreviation definition where the name *id* is specified to be an abbreviation for the type expression occurring on the right hand side of = .

A database is specified to be a set of people. The type operator *-set* when applied to the type *Person* gives a new type containing as values all (finite) subsets of the set of values in *Person*.

A type obtained by applying a type operator to one or more other types is defined as compound type. Abstract types (like *Person*) are thus not compounds.

## VALUE DECLARATION

Values can be named in value declarations . A value declaration has the form :

Value

Value definition 1,

.

.

Value definition n.

for  $n \geq 1$ .

In the above example specification there are three such definitions.

A value definition has in the simplest case the form :

$$id : type\_expr.$$

That is the identifier *id* is defined to represent a value within the type represented by the type expression. Such a value definition 'defines the value *id*' instead of saying that it 'defines the identifier *id* to represent a value'.

The first value definition defines the constant value *empty* of the type *Database*. This value simply represents the empty database.



The second value definition defines the function *register* that adds a person to the database. Suppose we want to register the person Saha in a database db, then *register*(Saha, db) represents the database after having made the registration.

The type of *register* is represented by the type expression :

$$\text{Person} * \text{Database} \rightarrow \text{Database}$$

The type operator \* (Cartesian product) is thus applied to the pair *Person* and *Database*, and the type operator  $\rightarrow$  (function space) is applied to the pair consisting of the resulting Cartesian product and *Database*.

The Cartesian product of *Person* and *Database* is the type containing as values all pairs (p,db) where *p* : *Person* and *db* : *Database*.

The third value definition defines the function *check*, that, when applied to a person and a database, returns a Boolean value within the built-in type Bool. This type contains two values represented by the literals true and false. The function is supposed to return true if and only if the person is registered in the database.

## AXIOM DECLARATIONS

Axioms express properties of value names. An axiom declaration thus has the form :

```
axiom
    value_expr_1,
    .
    .
    value_expr_n.
for n ≥ 1.
```

In the above example there are three axioms.

The first axiom defines the name *empty* to represent the empty set ( The type of *empty* is *Person-set* ). This axiom states that two value expressions are equivalent, namely *empty* and  $\{ \}$ .

The second axiom in the example expresses that the function *register* adds a person *p* to a database *db* by making the set union of the database, which is a set, and the singleton set containing the person.

The third axiom defines the function *check*. A person is registered if that person belongs to the set representing the database.

The collection of axioms is complete in the sense that for each value identifier the axioms state exactly what value with in its type each identifier represents. For example *empty* is defined to be nothing but the empty set. Likewise, the function *register* represents the one and only one function that adds its first argument to its second argument.

Axioms do not, however, need not be complete. The ultimate extreme is the situation where there are no axioms at all, in which case the value identifier may represent any value with in its type.

An identifier that is not completely specified through the axioms is said to be under-specified. Axioms may be named for documentation purposes and for reference in justifications. The axioms defining *empty*, *register* and *check* can for example be written as follows, where axiom names bracketed with [ and ] precede the axioms:  
**axiom.**

```
[empty_axiom]
empty  $\equiv$  { },
[register_axiom]
for all p : person , db : Database . register(p,db) = { p } u db,
[check_axiom]
for all p : person ,db : database . check(p,db) p  $\in$  db.
```

The three axioms have been named *empty\_axiom*, *register\_axiom* and *check\_axiom*. Axiom namings do not add to the properties of a specification .

**The more generalised form of axiom declaration now is**

**axiom**

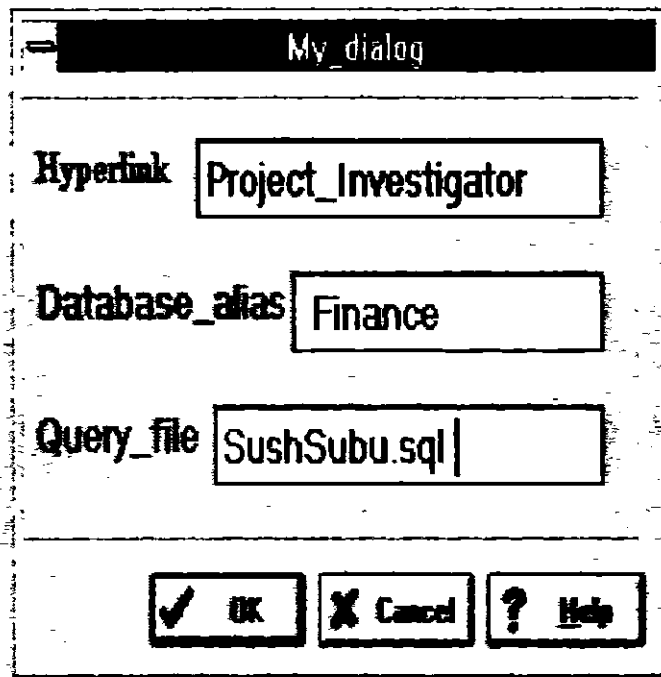
**opt-axiom\_naming 1 value\_expr 1,**

**opt-axiom\_naming n value\_expr n.**

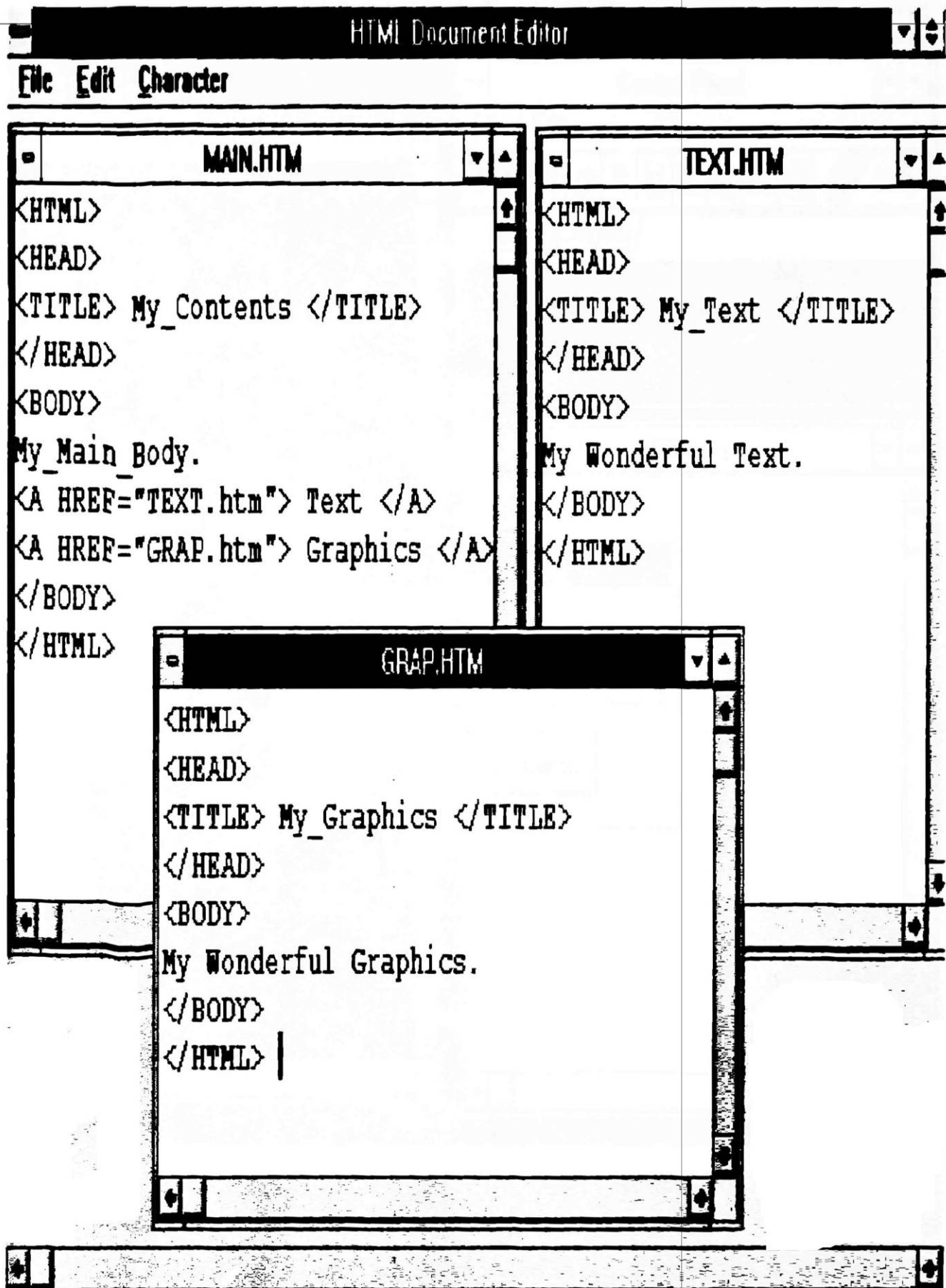
**[ For more information on RAISE refer [3] ]**

**RESULTS**

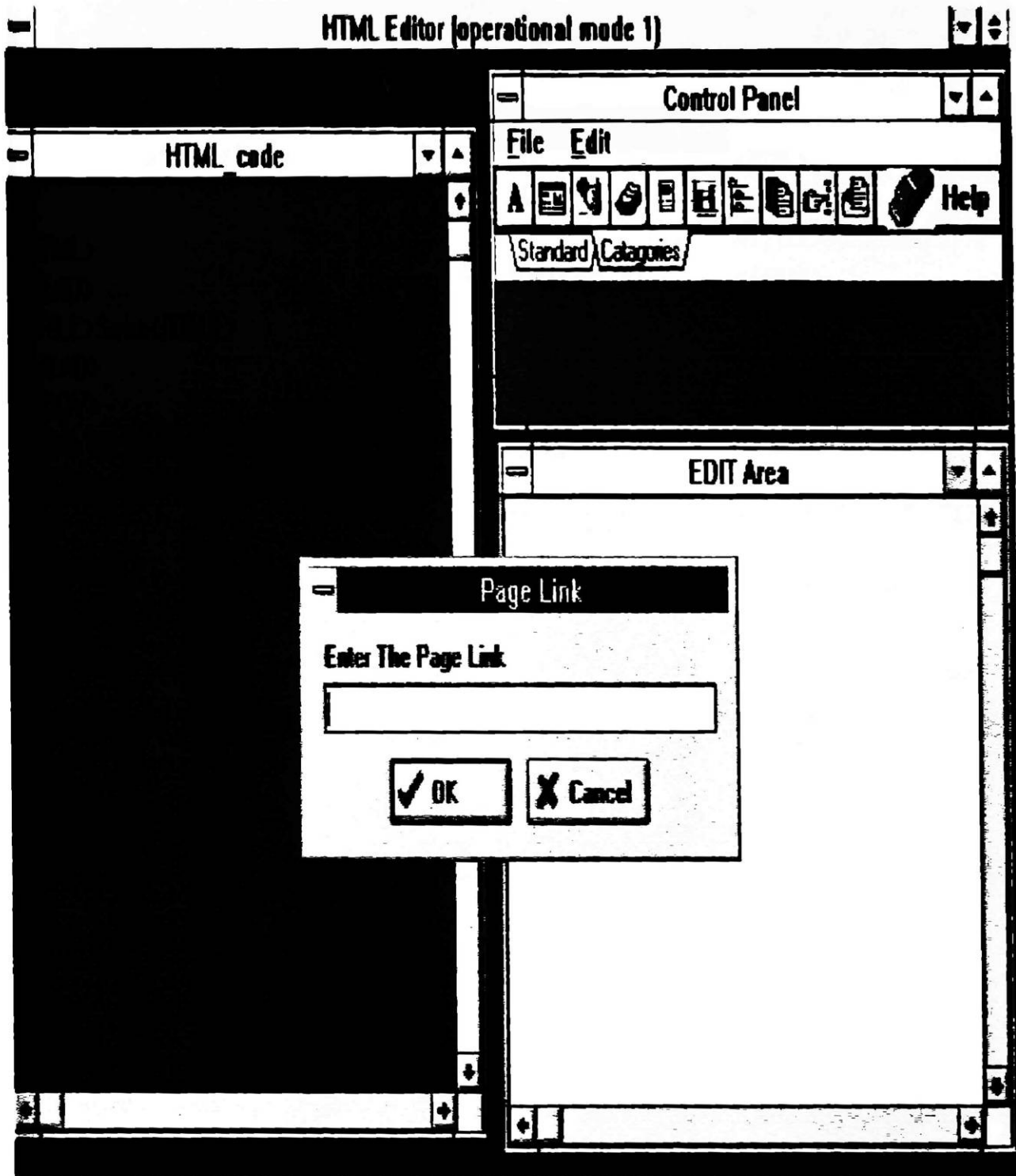
**B.1 MY DIALOG**



## B.2 INTERFACE WINDOW OF HTML DOCUMENT EDITOR



### B.3 INTERFACE WINDOW OF HTML EDITOR (OPERATIONAL MODE 1)



## B.4 INTERFACE WINDOW OF HTML EDITOR (OPERATIONAL MODE 2)

HTML Editor [operational mode 2]

Page Categories Code

Comments

Document Body

Document headings

Layout elements

Paragraphs

Graphics

Links

Lists

Text controls

**SUSHISH**

```
<HTML>
<HEAD>
<TITLE>SUSHISH</TITLE>
</HEAD>
<BODY>
MY BODY.<BR>
MY WONDERFUL
TEXT
AND
GRAPHICS.
<A HREF=" " > TEXT
<P>
<H1> AND </H1>
<P>
<A HREF=" " > GRAPE
</BODY>
</HTML>
```

## B.5 BROWSER'S INPUT AND OUTPUT

### Input :

```
<HTML>
<HEAD>
<TITLE> OUR PAGE (A SAMPLE OUTPUT) </TITLE>
</HEAD>
<BODY>
WE, ARE <B>SUSHISH SAHA</B> AND <B>RAJA
SUBRAHMANYAM DEVELOPED THIS PROJECT. <BR>
WE ARE GOING TO JOIN AS ASSISTANT SYSTEM ANALYST IN
<B>T.C.S.</B> <BR>
<BR>
THIS IS THE SAMPLE OUTPUT BY OUR BROWSER MODULE. <BR>
<BR>
THIS PAGE CONTAINS MAIN MODULES OF OUR PROJECT. THE
MODULES ARE: <BR>
<P>
<OL>
<L1><A HREF="MODULE_1.HTM"> HTML DOCUMENT EDITOR.</A>
<L1><A HREF="MODULE_2.HTM"> HTML EDITOR. </A>
<L1><A HREF="MODULE_3.HTM"> BROWSER. </A>
<L1><A HREF="MODULE_4.HTM"> DATABASE APPLICATION. </A>
</OL>
</BODY>
</HTML>
```



## Output:

### OUR PAGE (A SAMPLE OUTPUT)

WE, **SUSHISH SAHA** AND **RAJA SUBRAHMANYAM** DEVELOPED THIS PROJECT.  
WE ARE GOING TO JOIN AS ASSISTANT SYSTEM ANALYST IN T.C.S

THIS IS THE SAMPLE OUTPUT BY OUR BROWSER MODULE

THIS PAGE CONTAINS MAIN MODULES OF OUR PROJECT. THE MODULES ARE :

1. HTML DOCUMENT EDITOR.
2. HTML EDITOR.
3. BROWSER.
4. DATABASE APPLICATION.

## DELPHI's IDE



# REFERENCES

**1. Foundations of Delphi Application development**  
*by*  
**Tom Swan.**

A Comdex computer publishing.  
A division of Pustak Mahal.

**2. Delphi Programming explorer**  
*by*  
**JEFF Duntemann,**  
**Jim Mischel,**  
**Don Taylor.**

A Comdex computer publishing,  
A division of Pustak Mahal, CORIOLIS Group Books.

**3. The Raise Specification language**  
*by*  
**The Raise language Group.**

Prentice Hall.

**4. HTML for Dummies**  
*by*  
**ED Tittel,**  
**Steve James.**

Comdex Computer publishing,  
A division of Pustak Mahal.