

Near East University

Faculty of Engineering

Department of Computer Engineering

Air Ticket Reservation System

**Graduation Project
COM-400**

Student: Abdul Manan Qureshi(980859)

Supervisor: Mr. Umit Ilhan

Nicosia - 2002

Acknowledgments

First of all the author would like to thank the Almighty for giving him this chance at higher education and giving him a chance to put his constructive abilities to work.

Second, the author would like to thank his advisor without whom this project couldn't have gone forward and whose valuable advice kept the project on track and on time.

Third of all, the author wishes to thank his parents for their unwavering support and love that saw him through, even in the most difficult of times. They have spent their lifetime working hard to ensure a bright future, may God give the author the ability never to let them down and to be of service to parents, Religion and Country.

Abstract

This project basically deals with the construction of a program that handles reservations for a Air ticketing System. This system would accept search patterns from the user and would plough through its databases to manipulate what is required from it to fulfill the user's request.

This report has been split up into three parts, the first part deals with the history of the coding language in which the program is written and then a brief introduction as to what the VCl language stands for and its properties and its relational importance in this project. The third part deals with the project source code it self, starting of from the peripherals and then going on to step by step analysis of the code functionality.

It has been the author's utmost wish to keep the report simple and to reduce the complexity as much as possible. The report ends with a conclusion, summarizing the insight,, and technical enhancements gained by the author as a result of this project.

Table of Contents

NAME	PAGE #
ACKNOWLEDGEMENTS	i
ABSTRACT	ii
TABLE OF CONTENTS	iii
INTRODUCTION	IV
 CHAPTER ONE: LITERATURE RESEARCH	
1.1.Introduction to Delphi™	02
1.2.Introduction to OOP	02
1.2.1. Components of OOP	03
1.2.2. Program Structure and Syntax	05
1.2.3. The Program Heading	06
1.2.4. The program Uses clause	06
1.2.5. The block	06
1.2.6. Fundamental Syntactic Elements	06
1.2.6.1. Special Symbols	07
1.2.6.2. Identifiers	07
1.2.6.3. Qualified Identifiers	07
1.2.6.4. Directives	08
1.2.6.5. Numerals	08
1.2.6.6. Labels	09
1.2.6.7. Character strings	09
1.2.7. Components and compiler directives	10
1.2.7.1. Operators	10
1.2.7.2. Boolean Operators	11
1.2.7.3. Complete vs. Shortcut Boolean evaluation	11
1.2.7.4. Logic (Bitwise) Operators	12
1.2.7.5. String Operators	12
1.2.7.6. Pointer Operators	13
1.2.8. Libraries and packages	13
1.2.8.1. Calling Dynamically Loadable Libraries	13
1.2.8.2. Static Loading	14
1.2.8.3. Dynamic Loading	14

1.2.8.4. Writing Dynamically loadable Libraries	15
1.2.8.5. The Exports Clause	16
1.2.8.6. Library Initialization Code	17
1.2.8.7. Global variables in a library	18
1.2.8.8. Libraries and System vars.	18
1.2.8.9. Exceptions and runtime errors in Libraries	18
1.3.Devaloping applications Using Delphi	19
1.3.1. Designing applications	20
1.3.2. Devaloping applications	20
1.3.3. Creating projects	20
1.3.4. Editing Code	21
1.3.5. Compiling Applications	21
1.3.6. Debugging applications	22
1.3.7. Deploying applications	22
 CHAPTER TWO: PROGRAM LOGIC VISUALIZATION	
2.1.Introduction to the problem	23
2.2.Databases	23
2.2.1. detailed Account of Eac DB	24
2.2.1.1.1. Aircraft.db	24
2.2.1.1.2. Airlines.db	25
2.2.1.1.3. Class.db	25
2.2.1.1.4. Cost.db	26
2.2.1.1.5. Cust.db	27
2.2.1.1.6. Flights.db	27
2.2.1.1.7. Reservation.db	27
2.2.1.1.8. Routes.db	28
2.2.1.1.9. Seats.db	28
2.2.2. Inter-Table Connections nad relationships	29
2.2.2.1. In-depth look	30
 CHAPTER THREE: SOFTWARE STRUCTURE AND DETAIL	
3.1.Forms and units	33
3.2.Flow of control	40
CONCLUSION	53
REFERENCES	54

Introduction

First of all an air ticket reservation system, is a very complex and highly connected software system to construct, as it takes into account many different factors that affect the outcome of the result.

It consists of many layers of code, logic, and structural design, necessary to ensure a smooth functionality status. Implementation of such a program in different coding languages surely will result in a different outlook for the program.

The author developed the program using the Delphi 6.0 coding language. Delphi 6.0 is actually a RAD tool that consists of a package of software that enables a developer to base his code easily and on a multiplatform basis. Main examples of such a system are KYLIX™. KYLIX™ is a software platform that allows the code to be manipulated in such a way that it can perform optimally in different OS environments.

The author developed the program using Windows based version. Hence negating the need here to further explain the usage of the properties of KYLIX™, but for the readers own good, a brief decryption of the Delphi 6.0, VCL and other topics crucial to understanding the core of the program are explained in the opening chapter.

The first chapter of this report deals with the brief history, understand and the implementation of VCL in modern day programming. It explains the different terms and terminology used in Delphi and its inherent use in developing software packages

The second chapter deals with logic of the programming of the system. During the quest for a solution for a particular system, the main thing to be given special focus to is the logic of the program, or the solution. The logic is the core, the design part is just cosmetics. Because if the logic of the program tackles the problem efficiently then the rest is downhill from there. So in this chapter, a detailed look at the logic of the system has been discussed and with the aid of visual examples, an attempt has been made by the author to make the logic easily understandable.

The third and the last chapter deals with the design phase of the code. As is the case of every software, user friendliness and ease of use make for efficient use of time and energy. In this last chapter, the coding of the program is discussed in detail and with the help of screenshots and coded examples, the author has attempted to explain the design phase of the code.

The project ends with a conclusion highlights the significant gain in programming hours and experienced that the author has gained during the preparation of this program and its productivity for the author.

Chapter 1

Literature research

1.1 Introduction to Delphi™

Delphi is an object-oriented, visual programming environment for rapid application development (RAD). Using Delphi, you can create highly efficient applications for Microsoft Windows 2000, Windows 98, and Windows NT with a minimum of manual coding. Delphi also provides a simple cross-platform solution when used in conjunction with Kylix, Borland's RAD tool for Linux. Delphi provides all the tools one needs to develop, test, and deploy applications, including a large library of reusable components, a suite of design tools, application and form templates, and programming wizards.

Delphi is basically OOP version of the 70s PASCAL programming language, henceforth before delving into further detail a explanation OOP is required and is presented below

1.2 Introduction to OOP

Most modern programming languages support object-oriented programming (OOP). Languages are based on three fundamental concepts: encapsulation (usually implemented with classes), inheritance, and polymorphism (or late binding).

You can write Delphi applications even without knowing the details of Object Pascal. As you create a new form, add new components, and handle events, Delphi prepares most of the related code for you automatically. But knowing the details of the language and its implementation will help you to understand precisely what Delphi is doing and to master the language completely.

A single chapter doesn't allow space for a full introduction to the principles of object-oriented programming and the Object Pascal language. Instead, I will outline the key OOP features of the language and show how they relate to everyday Delphi programming. Even if you don't have a precise knowledge of OOP, the chapter will introduce each of the key concepts so that you won't need to refer to other sources.

The Object Pascal language used by Delphi is an OOP extension of the classic Pascal language, which Borland pushed forward for many years with its Turbo Pascal compilers. The syntax of the Pascal language is known to be quite verbose and more readable than, for example, the C language. Its OOP extension follows the same approach, delivering the same power of the recent breed of OOP languages, from Java to C#.

In this chapter, I'll discuss only the object-oriented extensions of the Pascal language available in Delphi. However, I'll highlight recent additions Borland has done to the core language. These features have been introduced in Delphi 6 and are, at least partially, related to the Linux version of Delphi.

New Pascal features include the `$IF` and `$ELSEIF` directives for conditional compilation, the `$WARN` and `$MESSAGE` directives, and the platform, library, and deprecated hint directives. These topics are discussed in the following sections. Changes to the assembler (with new directives, support for MMX and Pentium Pro instructions, and many more features) are really beyond the scope of this book.

Other relatively minor changes in the language include a change in the default value for the `$WRITEABLECONST` compiler switch, which is now disabled. This option allows programs to modify the value of typed constants and should generally be left disabled, using variables instead of constants for modifiable values. Another change is the support for the `Int64` data type in variants. Finally, you can assign specific values to the elements of an enumeration (as in the C/C++ language), instead of using the default sequence of values.

1.2.1 Program Organization

Programs are usually divided into source-code modules called units. Each program begins with a heading, which specifies a name for the program. The heading is followed by an optional uses clause, then a block of declarations and statements. The uses clause lists units that are linked into the program; these units, which can be shared by different programs, often have uses clauses of their own.

The uses clause provides the compiler with information about dependencies among modules. Because this information is stored in the modules themselves, Object Pascal programs do not require makefiles, header files, or preprocessor "include" directives. (The Project Manager generates a makefile each time a project is loaded in the IDE, but saves these files only for project groups that include more than one project.)

1.2.1 Components of OOP

The compiler expects to find Pascal source code in files of three kinds:

- unit source files (which end with the `.pas` extension)
- project files (which end with the `.dpr` extension)
- package source files (which end with the `.dpk` extension)

Unit source files contain most of the code in an application. Each application has a single project file and several unit files; the project file—which corresponds to the "main" program file in traditional Pascal—organizes the unit files into an application. Borland development tools automatically maintain a project file for each application.

If you are compiling a program from the command line, you can put all your source code into unit (.pas) files. But if you use the IDE to build your application, you must have a project (.dpr) file.

Package source files are similar to project files, but they are used to construct special dynamically linkable libraries called packages.

In addition to source-code modules, Borland products use several non-Pascal files to build applications. These files are maintained automatically and include

- form files, which end with the .dfm (Delphi) or .xfm (Kylix) extension,
- resource files, which end with the .res extension, and
- project options files, which end with the .dof (Delphi) or .kof (Kylix) extension.

A form file is either a text file or a compiled resource file that can contain bitmaps, strings, and so forth. Each form file represents a single form, which usually corresponds to a window or dialog box in an application. The IDE allows you to view and edit form files as text, and to save form files as either text or binary. Although the default behavior is to save form files as text, they are usually not edited manually; it is more common to use Borland's visual design tools for this purpose. Each project has at least one form, and each form has an associated unit (.pas) file that, by default, has the same name as the form file.

In addition to form files, each project uses a resource (.res) file to hold the bitmap for the application's icon. By default, this file has the same name as the project (.dpr) file. To change an application's icon, use the Project Options dialog.

A project options (.dof or .kof) file contains compiler and linker settings, search directories, version information, and so forth. Each project has an associated project options file with the same name as the project (.dpr) file. Usually, the options in this file are set from Project Options dialog.

Various tools in the IDE store data in files of other types. Desktop settings (.dsk or .desk) files contain information about the arrangement of windows and other configuration options; desktop settings can be project-specific or environment-wide. These files have no direct effect on compilation.

Figure 1.2.2.a shows a simple program generated in Delphi, explicitly showing the types of files generated when using OOP . i.e. DELPHI 6.0

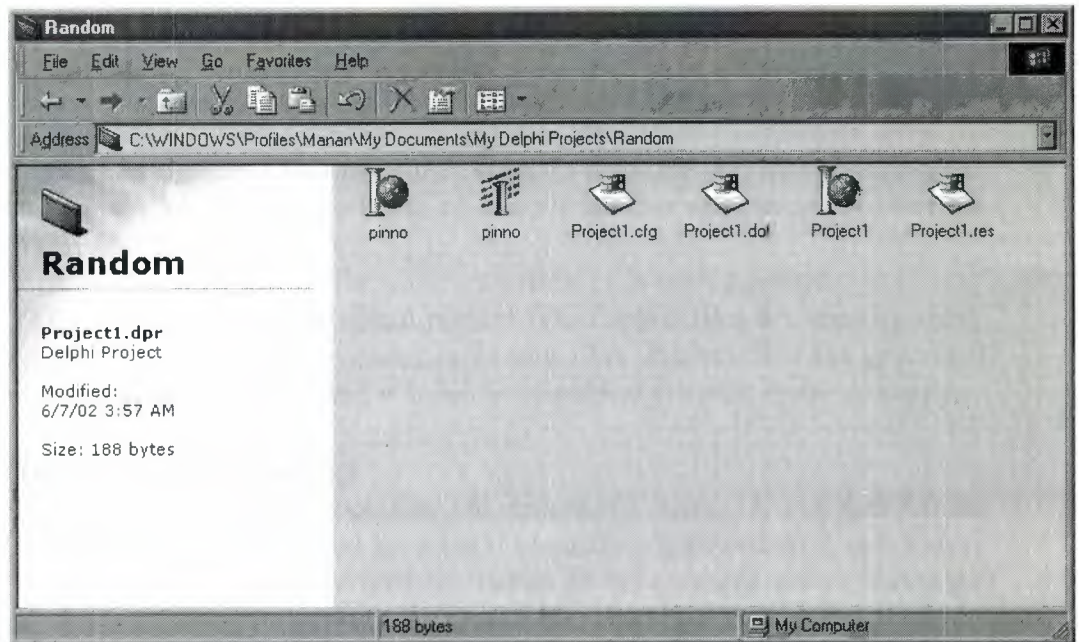


figure 1.2.1.a

A figure showing a simple program when saved to a folder, generates the files necessary to continue functionality.

1.2.2 Program structure and syntax

A program contains

- a program heading,
- a uses clause (optional), and
- a block of declarations and statements.

The program heading specifies a name for the program. The uses clause lists units

used by the program. The block contains declarations and statements that are

executed when the program runs. The IDE expects to find these three elements in a

single project (.dpr) file.

The example below shows the project file for a program called Editor.

```

1 program Editor;
2
3 uses
4 Forms, {change to QForms in Linux}
5 REAbout in 'REAbout.pas' {AboutBox},
6 REMain in 'REMain.pas' {MainForm};
7
8 {$R *.res}
9
10 begin
11 Application.Title := 'Text Editor';
12 Application.CreateForm(TMMainForm, MainForm);

```



```
13 Application.Run;
```

```
14 end.
```

Line 1 contains the program heading. The uses clause is on lines 3 through 6. Line 8 is

a compiler directive that links the project's resource file into the program.

Lines 10 through 14 contain the block of statements that are executed when the program runs.

Finally, the project file, like all source files, ends with a period.

This is, in fact, a fairly typical project file. Project files are usually short, since most of a program's logic resides in its unit files. Project files are generated and maintained automatically, and it is seldom necessary to edit them manually.

1.2.3 The program heading

The program heading specifies the program's name. It consists of the reserved word `program`, followed by a valid identifier, followed by a semicolon. The identifier must match the project file name. In the example above, since the program is called `Editor`, the project file should be called `EDITOR.dpr`.

In standard Pascal, a program heading can include parameters after the program

```
name:
```

```
program Calc(input, output);
```

Borland's Object Pascal compiler ignores these parameters.

1.2.4. The program uses clause

The uses clause lists units that are incorporated into the program. These units may in turn have uses clauses of their own. For more information about the uses clause.

1.2.5. The block

The block contains a simple or structured statement that is executed when the program runs. In most programs, the block consists of a compound statement—bracketed between the reserved words `begin` and `end`—whose component Object Pascal uses the ASCII character set, including the letters A through Z and a through z, the digits 0 through 9, and other standard characters. It is not case sensitive.

The space character (ASCII 32) and the control characters (ASCII 0 through 31—including ASCII 13, the return or end-of-line character) are called blanks. Fundamental syntactic elements, called tokens, combine to form expressions, declarations, and statements. A statement describes an algorithmic action that can be executed within a program. An expression is a syntactic unit that occurs within a statement and denotes a value. A declaration defines an identifier (such as the name of a function or variable) that can be used in expressions and statements, and, where appropriate, allocates memory for the identifier.

1.2.6. Fundamental syntactic elements

On the simplest level, a program is a sequence of tokens delimited by separators. A token is the smallest meaningful unit of text in a program. A

separator is either a blank or a comment. Strictly speaking, it is not always necessary to place a separator between two tokens; for example, the code fragment

```
Size:=20;Price:=10;
```

is perfectly legal. Convention and readability, however, dictate that we write this as

```
Size := 20;
```

```
Price := 10;
```

Tokens are categorized as special symbols, identifiers, reserved words, directives, numerals, labels, and character strings. A separator can be part of a token only if the token is a character string. Adjacent identifiers, reserved words, numerals, and labels must have one or more separators between them.

1.2.6.1.Special symbols

Special symbols are nonalphanumeric characters, or pairs of such characters, that have fixed meanings. The following single characters are special symbols. # \$ & ' () * + , - . / : ; < = > @ [] ^ { } The following character pairs are also special symbols. (* (. *) .) .. // := <= >= <> The left bracket—[—is equivalent to the character pair of left parenthesis and period—(.; the right bracket—]—is equivalent to the character pair of period and right parenthesis—.) . The left-parenthesis-plus-asterisk and asterisk-plus-rightparenthesis —(* *)— are equivalent to the left and right brace—{ }. Notice that !, " (double quotation marks), %, ?, \, _ (underscore), | (pipe), and ~ (tilde) are not special characters.

1.2.6.2.Identifiers

Identifiers denote constants, variables, fields, types, properties, procedures, functions, programs, units, libraries, and packages. An identifier can be of any length, but only the first 255 characters are significant. An identifier must begin with a letter or an underscore (_) and cannot contain spaces; letters, digits, and underscores are allowed after the first character. Reserved words cannot be used as identifiers.

Since Object Pascal is case-insensitive, an identifier like CalculateValue could be written in any of these ways:

```
CalculateValue
```

```
calculateValue
```

```
calculatevalue
```

```
CALCULATEVALUE
```

On Linux, the only identifiers for which case is important are unit names. Since unit names correspond to file names, inconsistencies in case can sometimes affect compilation.

1.2.6.3.Qualified identifiers

When you use an identifier that has been declared in more than one place, it is sometimes necessary to qualify the identifier. The syntax for a qualified identifier is

```
identifier1.identifier2
```

where identifier1 qualifies identifier2. For example, if two units each declare a variable called CurrentValue, you can specify that you want to access the CurrentValue in Unit2 by writing

Unit2.CurrentValue

Qualifiers can be iterated. For example,

Form1.Button1.Click

calls the Click method in Button1 of Form1.

If you don't qualify an identifier, its interpretation is determined by the rules of scope described in "Blocks and scope" on page 4-27.

Reserved words

The following reserved words cannot be redefined or used as identifiers.

Table 4.1 Reserved words

and	downto	in	or	string
array	else	inherited	out	then
as	end	initialization	packed	threadvar
asm	except	inline	procedure	to
begin	exports	interface	program	try
case	file	is	property	type
class	finalization	label	raise	unit
const	finally	library	record	until
constructor	for	mod	repeat	uses
destructor	function	nil	resourcestring	var
dispinterface	goto	not	set	while
div	if	object	shl	with
do	implementation	of	shr	xor

In addition to the words in Table 4.1, private, protected, public, published, and automated act as reserved words within object type declarations, but are otherwise treated as directives. The words at and on also have special meanings.

1.2.6.4.Directives

Directives are words that are sensitive in specific locations within source code. Directives have special meanings in Object Pascal, but, unlike reserved words, appear only in contexts where user-defined identifiers cannot occur. Hence—although it is inadvisable to do so—you can define an identifier that looks exactly like a directive.

absolute	dynamic	message	private	resident
abstract	export	name	protected	safecall
assembler	external	near	public	stdcall
automated	far	nodefault	published	stored
cdecl	forward	overload	read	varargs
contains	implements	override	readonly	virtual
default	index	package	register	write
deprecated	library	pascal	reintroduce	writeln
dispid	local	platform	requires	

1.2.6.5.Numerals

Integer and real constants can be represented in decimal notation as sequences of digits without commas or spaces, and prefixed with the + or - operator to indicate sign. Values default to positive (so that, for example, 67258 is equivalent to +67258) and must be within the range of the largest predefined real or integer type. Numerals with decimal points or exponents denote reals, while other numerals denote integers. When the character E or e occurs within a real, it

means “times ten to the power of”. For example, 7E-2 means 7×10^{-2} , and 12.25e+6 and 12.25e6 both mean 12.25×10^6 .

The dollar-sign prefix indicates a hexadecimal numeral—for example, \$8F. For the Integer type (16-bit integer), the sign of a hexadecimal is determined by the leftmost (most significant) bit of its binary representation. For all other types, you must use a prefixed + or - operator to indicate sign.

For more information about real and integer types, see Chapter 5, “Data types, variables, and constants”. For information about the data types of numerals, see “True constants” on page 5-39.

1.2.6.6.Labels

A label is a sequence of no more than four digits—that is, a numeral between 0 and 9999. Leading zeros are not significant. Identifiers can also function as labels. Labels are used in goto statements. For more information about goto statements and labels, see “Goto statements” on page 4-18.

1.2.6.7.Character strings

A character string, also called a string literal or string constant, consists of a quoted string, a control string, or a combination of quoted and control strings. Separators can occur only within quoted strings.

A quoted string is a sequence of up to 255 characters from the extended ASCII character set, written on one line and enclosed by apostrophes. A quoted string with nothing between the apostrophes is a null string. Two sequential apostrophes in a quoted string denote a single character, namely an apostrophe. For example,

```
'BORLAND' { BORLAND }
'You'll see' { You'll see }
'' { ' }
'' { null string }
' ' { a space }
```

A control string is a sequence of one or more control characters, each of which consists of the # symbol followed by an unsigned integer constant from 0 to 255 (decimal or hexadecimal) and denotes the corresponding ASCII character. The control string

```
#89#111#117
```

is equivalent to the quoted string

```
'You'
```

You can combine quoted strings with control strings to form larger character strings. For example, you could use

```
'Line 1'#13#10'Line 2'
```

to put a carriage-return-line-feed between “Line 1” and “Line 2”. However, you cannot concatenate two quoted strings in this way, since a pair of sequential apostrophes is interpreted as a single character. (To concatenate quoted strings, use the + operator described in “String operators” on page 4-9, or simply combine them into a single quoted string.)

A character string’s length is the number of characters in the string. A character string of any length is compatible with any string type and with the PChar type. A character string of length 1 is compatible with any character type, and, when extended syntax is enabled ($\{\$X+\}$), a character string of length $n=1$ is compatible with zero-based arrays and packed arrays of n characters. For more

information about string types, see Chapter 5, "Data types, variables, and constants".

1.2.7. Comments and compiler directives

Comments are ignored by the compiler, except when they function as separators (delimiting adjacent tokens) or compiler directives.

There are several ways to construct comments:

```
{ Text between a left brace and a right brace constitutes a comment. }
(* Text between a left-parenthesis-plus-asterisk and an
asterisk-plus-right-parenthesis also constitutes a comment. *)
// Any text between a double-slash and the end of the line constitutes a comment.
```

A comment that contains a dollar sign (\$) immediately after the opening { or (* is a compiler directive. For example,

```
{ $WARNINGS OFF }
```

tells the compiler not to generate warning messages.

Expressions

An *expression* is a construction that returns a value. For example,

X	{ variable }
@X	{ address of a variable }
15	{ integer constant }
InterestRate	{ variable }
Calc(X,Y)	{ function call }
X * Y	{ product of X and Y }
Z / (1 - Z)	{ quotient of Z and (1 - Z) }
X = 1.5	{ Boolean }
C in Range1	{ Boolean }
not Done	{ negation of a Boolean }

The simplest expressions are variables and constants (described in Chapter 5, "Data types, variables, and constants"). More complex expressions are built from simpler ones using operators, function calls, set constructors, indexes, and typecasts.

1.2.7.10 Operators

Operators behave like predefined functions that are part of the Object Pascal language. For example, the expression (X + Y) is built from the variables X and Y—called operands—with the + operator; when X and Y represent integers or reals, (X + Y) returns their sum. Operators include @, not, ^, *, /, div, mod, and, shl, shr, as, +, -, or, xor, =, >, <, <>, <=, >=, in, and is.

The operators @, not, and ^ are unary (taking one operand). All other operators are binary (taking two operands), except that + and - can function as either unary or binary. A unary operator always precedes its operand (for example, -B), except for ^, which follows its operand (for example, P^). A binary operator is placed between its operands (for example, A = 7).

Some operators behave differently depending on the type of data passed to them. For example, not performs bitwise negation on an integer operand and logical negation on a Boolean operand. Such operators appear below under multiple categories. Except for ^, is, and in, all operators can take operands of type Variant. For details, see "Variant types" on page 5-30.

The sections that follow assume some familiarity with Object Pascal data types. For information about data types, see Chapter 5, "Data types, variables,

and constants”. For information about operator precedence in complex expressions, see “Operator precedence rules” on page 4-12.

Arithmetic operators

Arithmetic operators, which take real or integer operands, include `+`, `-`, `*`, `/`, `div`, and `mod`.

Table 4.3 Binary arithmetic operators

Operator	Operation	Operand types	Result type	Example
<code>+</code>	addition	integer, real	integer, real	<code>X + Y</code>
<code>-</code>	subtraction	integer, real	integer, real	<code>Result - 1</code>
<code>*</code>	multiplication	integer, real	integer, real	<code>P * InterestRate</code>
<code>/</code>	real division	integer, real	real	<code>X / 2</code>
<code>div</code>	integer division	integer	integer	<code>Total div UnitSize</code>
<code>mod</code>	remainder	integer	integer	<code>Y mod 6</code>

The following rules apply to arithmetic operators.

- The value of `x/y` is of type Extended, regardless of the types of `x` and `y`. For other arithmetic operators, the result is of type Extended whenever at least one operand is a real; otherwise, the result is of type Int64 when at least one operand is of type Int64; otherwise, the result is of type Integer. If an operand’s type is a subrange of an integer type, it is treated as if it were of the integer type.

- The value of `x div y` is the value of `x/y` rounded in the direction of zero to the nearest integer.

- The `mod` operator returns the remainder obtained by dividing its operands. In other words, `x mod y = x - (x div y) * y`.

- A runtime error occurs when `y` is zero in an expression of the form `x/y`, `x div y`, or `x mod y`.

1.2.7.2.Boolean operators

The Boolean operators `not`, `and`, `or`, and `xor` take operands of any Boolean type and return a value of type Boolean.

Table 4.5 Boolean operators

Operator	Operation	Operand types	Result type	Example
<code>not</code>	negation	Boolean	Boolean	<code>not (C in MySet)</code>
<code>and</code>	conjunction	Boolean	Boolean	<code>Done and (Total > 0)</code>
<code>or</code>	disjunction	Boolean	Boolean	<code>A or B</code>
<code>xor</code>	exclusive disjunction	Boolean	Boolean	<code>A xor B</code>

These operations are governed by standard rules of Boolean logic. For example, an expression of the form `x and y` is True if and only if both `x` and `y` are True.

1.2.7.3.Complete versus short-circuit Boolean evaluation

The compiler supports two modes of evaluation for the `and` and `or` operators: complete evaluation and short-circuit (partial) evaluation. Complete evaluation means that each conjunct or disjunct is evaluated, even when the result of the entire expression is already determined. Short-circuit evaluation means strict left-to-right evaluation that stops as soon as the result of the entire expression is determined. For example, if the expression `A and B` is evaluated under short-circuit mode when `A` is False, the compiler won’t evaluate `B`; it knows that the entire expression is False as soon as it evaluates `A`.

Short-circuit evaluation is usually preferable because it guarantees minimum Execution time and, in most cases, minimum code size. Complete evaluation is Sometimes convenient when one operand is a function with side effects that alter the execution of the program. Short-circuit evaluation also allows the use of constructions that might otherwise result in illegal runtime operations. For example, the following code iterates through the string S, up to the first comma.

```
while (I <= Length(S)) and (S[I] <> ',') do
begin
  f
  Inc(I);
end;
```

In a case where S has no commas, the last iteration increments I to a value which is greater than the length of S. When the while condition is next tested, complete evaluation results in an attempt to read S[I], which could cause a runtime error.

Under short-circuit evaluation, in contrast, the second part of the while condition— (S[I] <> ',')—is not evaluated after the first part fails. Use the \$B compiler directive to control evaluation mode. The default state is {\$B-}, which enables short-circuit evaluation. To enable complete evaluation locally, add the {\$B+} directive to your code. You can also switch to complete evaluation on a project-wide basis by selecting Complete Boolean Evaluation in the Compiler Options dialog. Note If either operand involves a variant, the compiler always performs complete evaluation (even in the {\$B-} state).

1.2.7.4.Logical (bitwise) operators

The following logical operators perform bitwise manipulation on integer operands. For example, if the value stored in X (in binary) is 001101 and the value stored in Y is 100001, the statement

Operator	Operation	Operand types	Result type	Examples
not	bitwise negation	integer	integer	not X
and	bitwise and	integer	integer	X and Y
or	bitwise or	integer	integer	X or Y
xor	bitwise xor	integer	integer	X xor Y
shl	bitwise shift left	integer	integer	X shl 2
shr	bitwise shift right	integer	integer	Y shr 1

The following rules apply to bitwise operators.

- The result of a not operation is of the same type as the operand.
- If the operands of an and, or, or xor operation are both integers, the result is of the predefined integer type with the smallest range that includes all possible values of both types.
- The operations x shl y and x shr y shift the value of x to the left or right by y bits, which is equivalent to multiplying or dividing x by 2^y; the result is of the same type as x. For example, if N stores the value 01101 (decimal 13), then N shl 1 returns 11010 (decimal 26). Note that the value of y is interpreted modulo the size of the type of x. Thus for example, if x is an integer, x shl 40 is interpreted as x shl 8 because an integer is 32 bits and 40 mod 32 is 8.

1.2.7.5.String operators

The relational operators =, <>, <, >, <=, and >= all take string operands (see

“Relational operators” on page 4-10). The + operator concatenates

1.2.7..6.Pointer operators

The relational operators <, >, <=, and >= can take operands of type PChar (see “Relational operators” on page 4-10). The following operators also take pointers as The ^ operator dereferences a pointer. Its operand can be a pointer of any type except the generic Pointer, which must be typecast before dereferencing.

P = Q is True just in case P and Q point to the same address; otherwise, P <> Q is True. You can use the + and – operators to increment and decrement the offset of a character pointer. You can also use – to calculate the difference between the offsets of two character pointers. The following rules apply.

- If I is an integer and P is a character pointer, then P + I adds I to the address given by P; that is, it returns a pointer to the address I characters after P. (The expression I + P is equivalent to P + I.) P – I subtracts I from the address given by P; that is, it returns a pointer to the address I characters before P.
- If P and Q are both character pointers, then P – Q computes the difference between the address given by P (the higher address) and the address given by Q (the lower address); that is, it returns an integer denoting the number of characters between P and Q. P + Q is not defined. You can use the + and – operators to increment and decrement the offset of a character pointer. You can also use – to calculate the difference between the offsets of two character pointers. The following rules apply.
- If I is an integer and P is a character pointer, then P + I adds I to the address given by P; that is, it returns a pointer to the address I characters after P. (The expression I + P is equivalent to P + I.) P – I subtracts I from the address given by P; that is, it returns a pointer to the address I characters before P.
- If P and Q are both character pointers, then P – Q computes the difference between the address given by P (the higher address) and the address given by Q (the lower address); that is, it returns an integer denoting the number of characters between P and Q. P + Q is not defined.

1.2.8.Libraries and packages

A dynamically loadable library is a dynamic-link library (DLL) on Windows or a shared object library file on Linux. It is a collection of routines that can be called by applications and by other DLLs or shared objects. Like units, dynamically loadable libraries contain sharable code or resources. But this type of library is a separately compiled executable that is linked at runtime to the programs that use it. To distinguish them from standalone executables, on Windows files containing compiled DLLs are named with the .DLL extension. On Linux, files containing shared object files are named with a .so extension. Object Pascal programs can call DLLs or shared objects written in other languages, and applications written in other languages can call DLLs or shared objects written in Object Pascal.

1.2.8.1.Calling dynamically loadable libraries

You can call operating system routines directly, but they are not linked to your application until runtime. This means that the library need not be present when you compile your program. It also means that there is no compile-time validation of attempts to import a routine. Before you can call routines defined in a shared

object, you must import them. This can be done in two ways: by declaring an external procedure or function, or by direct calls to the operating system. Whichever method you use, the routines are not linked to your application until runtime. Object Pascal does not support importing of variables from shared libraries.

1.2.8.2.Static loading

The simplest way to import a procedure or function is to declare it using the external directive. For example,

On Windows: procedure DoSomething; external 'MYLIB.DLL';

On Linux: procedure DoSomething; external 'mylib.so';

If you include this declaration in a program, MYLIB.DLL (Windows) or mylib.so (Linux) is loaded once, when the program starts. Throughout execution of the program, the identifier DoSomething always refers to the same entry point in the same shared library. Declarations of imported routines can be placed directly in the program or unit where they are called. To simplify maintenance, however, you can collect external declarations into a separate “import unit” that also contains any constants and types required for interfacing with the library. Other modules that use the import unit can call any routines declared in it. For more information about external declarations, see “External declarations” on page 6-6.

1.2.8.3.Dynamic loading

You can access routines in a library through direct calls to OS library functions, including LoadLibrary, FreeLibrary, and GetProcAddress. In Windows, these functions are declared in Windows.pas; on Linux, they are implemented for compatibility in SysUtils.pas; the actual Linux OS routines are dlopen, dlclose, and dlsym (all declared in Kylix’s Libc unit; see the man pages for more information). In this case, use

procedural-type variables to reference the imported routines.

For example, on Windows or Linux:

uses Windows, ...; {On Linux, replace Windows with SysUtils }

type

TTimeRec = **record**

Second: Integer;

Minute: Integer;

Hour: Integer;

end;

TGetTime = **procedure**(var Time: TTimeRec);

THandle = Integer;

var

Time: TTimeRec;

Handle: THandle;

GetTime: TGetTime;

f

begin

Handle := LoadLibrary('libraryname');

if Handle <> 0 **then**

begin

@GetTime := GetProcAddress(Handle, 'GetTime');

if @GetTime <> nil **then**

begin

GetTime(Time);

with Time **do**

WriteLn('The time is ', Hour, ':', Minute, ':', Second);


```

end;
FreeLibrary(Handle);
end;
end;

```

When you import routines this way, the library is not loaded until the code containing the call to LoadLibrary executes. The library is later unloaded by the call to FreeLibrary. This allows you to conserve memory and to run your program even when some of the libraries it uses are not present. This same example can also be written on Linux as follows:

```

uses Libc, ...;
type
TTimeRec = record
Second: Integer;
Minute: Integer;
Hour: Integer;
end;
TGetTime = procedure(var Time: TTimeRec);
THandle = Pointer;
var
Time: TTimeRec;
Handle: THandle;
GetTime: TGetTime;
begin
Handle := dlopen('datetime.so', RTLD_LAZY);
if Handle <> 0 then
begin
@GetTime := dlsym(Handle, 'GetTime');
if @GetTime <> nil then
begin
GetTime(Time);
with Time do
WriteLn('The time is ', Hour, ':', Minute, ':', Second);
end;
dlclose(Handle);
end;
end;

```

In this case, when importing routines, the shared object is not loaded until the code containing the call to dlopen executes. The shared object is later unloaded by the call to dlclose. This also allows you to conserve memory and to run your program even when some of the shared objects it uses are not present.

1.2.8.4. Writing dynamically loadable libraries

The main source for a dynamically loadable library is identical to that of a program, except that it begins with the reserved word library (instead of program). Only routines that a library explicitly exports are available for importing by other libraries or programs. The following example shows a library with two exported functions, Min and Max.

```

library MinMax;
function Min(X, Y: Integer): Integer; stdcall;
begin
if X < Y then Min := X else Min := Y;
end;
function Max(X, Y: Integer): Integer; stdcall;
begin
if X > Y then Max := X else Max := Y;
end;
exports
Min,

```



```
Max;
begin
end.
```

If you want your library to be available to applications written in other languages, it's safest to specify `stdcall` in the declarations of exported functions. Other languages may not support Object Pascal's default register calling convention. Libraries can be built from multiple units. In this case, the library source file is frequently reduced to a `uses` clause, an `exports` clause, and the initialization code. For

example,

```
library Editors;
uses EdInit, EdInOut, EdFormat, EdPrint;
exports
  InitEditors,
  DoneEditors name Done,
  InsertText name Insert,
  DeleteSelection name Delete,
  FormatSelection,
  PrintSelection name Print,
f
  SetErrorHandler;
begin
  InitLibrary;
end.
```

You can put `exports` clauses in the interface or implementation section of a unit. Any library that includes such a unit in its `uses` clause automatically exports the routines listed the unit's `exports` clauses—without the need for an `exports` clause of its own. The directive `local`, which marks routines as unavailable for export, is platform-specific and has no effect in Windows programming. On Linux, the `local` directive provides a slight performance optimization for routines that are compiled into a library but are not exported. This directive can be specified for standalone procedures and functions, but not for methods. A routine declared with `local`—for example,

```
function Contraband(I: Integer): Integer; local;
—does not refresh the EBX register and hence
```

- cannot be exported from a library.
- cannot be declared in the interface section of a unit.
- cannot have its address taken or be assigned to a procedural-type variable.
- if it is a pure assembler routine, cannot be called from another unit unless the caller sets up EBX.

1.2.8.5. The exports clause

A routine is exported when it is listed in an `exports` clause, which has the form `exports entry1, ..., entryn`; where each entry consists of the name of a procedure, function, or variable (which

must be declared prior to the `exports` clause), followed by a parameter list (only if exporting a routine that is overloaded), and an optional name specifier. You can qualify the procedure or function name with the name of a unit.

(Entries can also include the directive `resident`, which is maintained for backward compatibility and is ignored by the compiler.)

On Windows only, an index specifier consists of the directive `index` followed by a numeric constant between 1 and 2,147,483,647. (For more efficient programs,

use low index values.) If an entry has no index specifier, the routine is automatically assigned a number in the export table.

Note Use of index specifiers, which are supported for backward compatibility only, is discouraged and may cause problems for other development tools.

A **name** specifier consists of the directive **name** followed by a string constant. If an entry has no **name** specifier, the routine is exported under its original declared **name**, with the same spelling and case. Use a name clause when you want to export a routine under a different **name**. For example,

exports

DoSomethingABC **name** 'DoSomething';

When you export an overloaded function or procedure from a dynamically loadable library, you must specify its parameter list in the exports clause. For example,

exports

Divide(X, Y: Integer) **name** 'Divide_Ints';

Divide(X, Y: Real) **name** 'Divide_Reals';

On Windows, do not include index specifiers in entries for overloaded routines.

An exports clause can appear anywhere and any number of times in the declaration part of a program or library, or in the interface or implementation section of a unit. Programs seldom contain an exports clause.

1.2.8.6. Library initialization code

The statements in a library's block constitute the library's initialization code. These statements are executed once every time the library is loaded. They typically perform tasks like registering window classes and initializing variables. Library initialization code can also install an exit procedure using the ExitProc variable, as described in "Exit procedures" on page 12-4; the exit procedure executes when the library is unloaded.

Library initialization code can signal an error by setting the ExitCode variable to a nonzero value. ExitCode is declared in the System unit and defaults to zero, indicating successful initialization. If a library's initialization code sets ExitCode to another value, the library is unloaded and the calling application is notified of the failure. Similarly, if an unhandled exception occurs during execution of the initialization code, the calling application is notified of a failure to load the library. Here is an example of a library with initialization code and an exit procedure.

library Test;

var

SaveExit: Pointer;

procedure LibExit;

begin

f // library exit code

ExitProc := SaveExit; *// restore exit procedure chain*

end;

begin

f // library initialization code

SaveExit := ExitProc; *// save exit procedure chain*

ExitProc := @LibExit; *// install LibExit exit procedure*

end.

When a library is unloaded, its exit procedures are executed by repeated calls to the address stored in ExitProc, until ExitProc becomes nil. The initialization parts of all units used by a library are executed before the library's initialization code,

and the finalization parts of those units are executed after the library's exit procedure.

1.2.8.7.Global variables in a library

Global variables declared in a shared library cannot be imported by an Object Pascal application.

A library can be used by several applications at once, but each application has a copy of the library in its own process space with its own set of global variables. For multiple libraries—or multiple instances of a library—to share memory, they must use memory-mapped files. Refer to the your system documentation for further information.

1.2.8.8.Libraries and system variables

Several variables declared in the System unit are of special interest to those programming libraries. Use `IsLibrary` to determine whether code is executing in an application or in a library; `IsLibrary` is always `False` in an application and `True` in a library. During a library's lifetime, `HInstance` contains its instance handle. `CmdLine` is always `nil` in a library.

The `DLLProc` variable allows a library to monitor calls that the operating system makes to the library entry point. This feature is normally used only by libraries that support multithreading. `DLLProc` is available on both Windows and Linux but its use differs on each. On Windows, `DLLProc` is used in multithreading applications; on Linux, it is used to determine when your library is being unloaded. You should use finalization sections, rather than exit procedures, for all exit behavior. (See "The finalization section" on page 3-5.)

To monitor operating-system calls, create a callback procedure that takes a single integer parameter—for example,

procedure `DLLHandler`(Reason: Integer);

—and assign the address of the procedure to the `DLLProc` variable. When the procedure is called, it passes to it one of the following values.

On Linux, these are defined in the `Libc` unit. In the body of the procedure, you can specify actions to take depending on which parameter is passed to the procedure.

1.2.8.9.Exceptions and runtime errors in libraries

When an exception is raised but not handled in a dynamically loadable library, it propagates out of the library to the caller. If the calling application or library is itself written in Object Pascal, the exception can be handled through a normal `try...except` statement.

Note Under Linux this is only possible if the library and application have both been built with the same set of (base) runtime packages (which contains the EH code) or if both link to `ShareExcept`.

If the calling application or library is written in another language, the exception can be handled as an operating-system exception with the exception code `$0EEDFACE`. The first entry in the `ExceptionInformation` array of the operating-system exception record contains the exception address, and the second entry contains a reference to the Object Pascal exception object.

Generally, you should not let exceptions escape from your library. On Windows,

Delphi exceptions map to the OS exception model; Linux does not have an exception model.

If a library does not use the SysUtils unit, exception support is disabled. In this case, when a runtime error occurs in the library, the calling application terminates. `DLL_PROCESS_DETACH` Indicates that the library is detaching from the address space of the calling process as a result of a clean exit or a call to `FreeLibrary` or `(dlclose` on Linux). `DLL_THREAD_ATTACH` Indicates that the current process is creating a new thread (Windows only). `DLL_THREAD_DETACH` Indicates that a thread is exiting cleanly (Windows only). Because the library has no way of knowing whether it was called from an Object Pascal program, it cannot invoke the application's exit procedures; the application is simply aborted and removed from memory.

1.3.Developing Applications Using Delphi

Borland Delphi is an object-oriented, visual programming environment for rapid development of 32-bit applications for deployment on Windows and Linux. Using Delphi, you can create highly efficient applications with a minimum of manual coding. Delphi provides a comprehensive class library called the Visual Component Library (VCL), Borland Component Library for Cross Platform (CLX), and a suite of Rapid Application Development (RAD) design tools, including application and form templates, and programming wizards. Delphi supports truly object-oriented programming:

- the VCL class library includes objects that encapsulate the Windows API as well as other useful programming techniques (Windows)
- the CLX class library includes objects that encapsulate the Qt library (Windows or Linux)

This chapter briefly describes the Delphi development environment and how it fits into the development life cycle. The rest of this manual provides technical details on developing general-purpose, database, Internet and Intranet applications, and includes information on creating ActiveX and COM controls and writing your own components. Integrated development environment When you start Delphi, you are immediately placed within the integrated development environment, also called the IDE. This environment provides all the tools you need to design, develop, test, debug, and deploy applications. Delphi's development environment includes a visual form designer, Object Inspector, Object TreeView, Component palette, Project Manager, source code editor, and debugger among other tools. Some tools may not be included in all versions of the product. You can move freely from the visual representation of an object (in the form designer), to the Object Inspector to edit the initial runtime state of the object, to the source code editor to edit the execution logic of the object. Changing code-related properties, such as the name of an event handler, in the Object Inspector automatically changes the corresponding source code. In addition, changes to the source code, such as renaming an event handler method in a form class declaration, is immediately reflected in the Object Inspector. The IDE supports application development throughout the stages of the product life cycle—from design to deployment. Using the tools in the IDE allows for rapid prototyping and shortens development time. A more complete overview of the

development environment is presented in the Quick Start manual included with the product. In addition, the online Help system provides help on all menus, dialogs, and windows.

1.3.1.Designing applications

Delphi includes all the tools necessary to start designing applications:

- A blank window, known as a form, on which to design the UI for your application.
- Extensive class libraries with many reusable objects.
- An Object Inspector for examining and changing object traits.
- A Code editor that provides direct access to the underlying program logic.
- A Project Manager for managing the files that make up one or more projects.
- Many other tools such as an image editor on the toolbar and an integrated debugger on menus to support application development in the IDE.
- Command-line tools including compilers, linkers, and other utilities.

You can use Delphi to design any kind of 32-bit application—from general-purpose utilities to sophisticated data access programs or distributed applications. Delphi's database tools and data-aware components let you quickly develop powerful desktop database and client/server applications. Using Delphi's data-aware controls, you can view live data while you design your application and immediately see the results of database queries and changes to the application interface. Many of the objects provided in the class library are accessible in the IDE from the Component palette. The Component palette shows all of the controls, both visual and nonvisual, that you can place on a form. Each tab contains components grouped by functionality. By convention, the names of objects in the class library begin with a T, such as TStatusBar. One of the revolutionary things about Delphi is that you can create your own components using Object Pascal. Most of the components provided are written in Object Pascal. You can add components that you write to the Component palette and customize the palette for your use by including new tabs if needed. You can also use Delphi for cross platform development on both Linux and Windows by using CLX. CLX contains a set of classes that, if used instead of those in the VCL, allow your program to port between Windows and Linux.

1.3.2.Developing applications

As you visually design the user interface for your application, Delphi generates the underlying Object Pascal code to support the application. As you select and modify the properties of components and forms, the results of those changes appear automatically in the source code, and vice versa. You can modify the source files directly with any text editor, including the built-in Code editor. The changes you make are immediately reflected in the visual environment as well.

1.3.3.Creating projects

All of Delphi's application development revolves around projects. When you create an application in Delphi you are creating a project. A project is a collection of files that make up an application. Some of these files are created at design time. Others are generated automatically when you compile the project source code. You can view the contents of a project in a project management tool

called the Project Manager. The Project Manager lists, in a hierarchical view, the unit names, the forms contained in the unit (if there is one), and shows the paths to the files in the project. Although you can edit many of these files directly, it is often easier and more reliable to use the visual tools in Delphi. At the top of the project hierarchy, is a group file. You can combine multiple projects into a project group. This allows you to open more than one project at a time in the Project Manager. Project groups let you organize and work on related projects, such as applications that function together or parts of a multi-tiered application. If you are only working on one project, you do not need a project group file to create an application.

Project files, which describe individual projects, files, and associated options, have a .dpr extension. Project files contain directions for building an application or shared object. When you add and remove files using the Project Manager, the project file is updated. You specify project options using a Project Options dialog which has tabs for various aspects of your project such as forms, application, compiler. These project options are stored in the project file with the project.

Units and forms are the basic building blocks of a Delphi application. A project can share any existing form and unit file including those that reside outside the project directory tree. This includes custom procedures and functions that have been written as standalone routines. If you add a shared file to a project, realize that the file is not copied into the current project directory; it remains in its current location. Adding the shared file to the current project registers the file name and path in the uses clause of the project file. Delphi automatically handles this as you add units to a project. When you compile a project, it does not matter where the files that make up the project reside. The compiler treats shared files the same as those created by the project itself.

1.3.4.Editing code

The Delphi Code editor is a full-featured ASCII editor. If using the visual programming environment, a form is automatically displayed as part of a new project. You can start designing your application interface by placing objects on the form and modifying how they work in the Object Inspector. But other programming tasks, such as writing event handlers for objects, must be done by typing the code. The contents of the form, all of its properties, its components, and their properties can be viewed and edited as text in the Code editor. You can adjust the generated code in the Code editor and add more components within the editor by typing code. As you type code into the editor, the compiler is constantly scanning for changes and updating the form with the new layout. You can then go back to the form, view and test the changes you made in the editor and continue adjusting the form from there. The Delphi code generation and property streaming systems are completely open to inspection. The source code for everything that is included in your final executable file—all of the VCL objects, CLX objects, RTL sources, all of the Delphi project files can be viewed and edited in the Code editor.

1.3.5.Compiling applications

When you have finished designing your application interface on the form, writing additional code so it does what you want, you can compile the project

from the IDE or from the command line. All projects have as a target a single distributable executable file. You can view or test your application at various stages of development by compiling, building, or running it:

- When you compile, only units that have changed since the last compile are recompiled.
- When you build, all units in the project are compiled, regardless of whether or not they have changed since the last compile. This technique is useful when you are unsure of exactly which files have or have not been changed, or when you simply want to ensure that all files are current and synchronized. It's also important to use Build when you've changed global compiler directives, to ensure that all code compiles in the proper state. You can also test the validity of your source code without attempting to compile the project.
- When you run, you compile and then execute your application. If you modified the source code since the last compilation, the compiler recompiles those changed modules and relinks your application. If you have grouped several projects together, you can compile or build all projects in a single project group at once. Choose Project|Compile All Projects or Project|Build All Projects with the project group selected in the Project Manager.

1.3.6. Debugging applications

Delphi provides an integrated debugger that helps you find and fix errors in your applications. The integrated debugger lets you control program execution, monitor variable values and items in data structures, and modify data values while debugging. The integrated debugger can track down both runtime errors and logic errors. By running to specific program locations and viewing the values of variables, the functions on the call stack, and the program output, you can monitor how your program behaves and find the areas where it is not behaving as designed. The debugger is described in online Help. You can also use exception handling to recognize, locate, and deal with errors. Exceptions in Delphi are classes, like other classes in Delphi, except, by convention, they begin with an E rather than the initial T for other classes.

1.3.7. Deploying applications

Delphi includes add-on tools to help with application deployment. For example, InstallShield Express (not available in all versions) helps you to create an installation package for your application that includes all of the files needed for running a distributed application. Refer to Chapter 13, "Deploying applications" for specific information on deployment.

Note Not all versions of Delphi have deployment capabilities.

TeamSource software (not available in all versions) is also available for tracking application updates.

Chapter 2.

Project Logic Visualization

2.1 Introduction to the problem

The main logic behind the program is the construction of a system that can handle a linkage variance of different types of stored data and then through the use of data aware controls manipulate these controls and perform basic reservation, calculation, functions.

i.e. Reservation and confirmation of a seat, Search the flight database, Addition or removal of different datasets in different tables. The first thing required by such a system is a network of interlinked tables that can be made use of through internal interconnections.

2.2 Databases

Any such problem must have a storage base upon which data can be written and stored when need be. All the tables made were constructed in *Paradox* format. And the program used for their construction was *Database Desktop* that comes with the complete *Borland Delphi 6.0 package*.

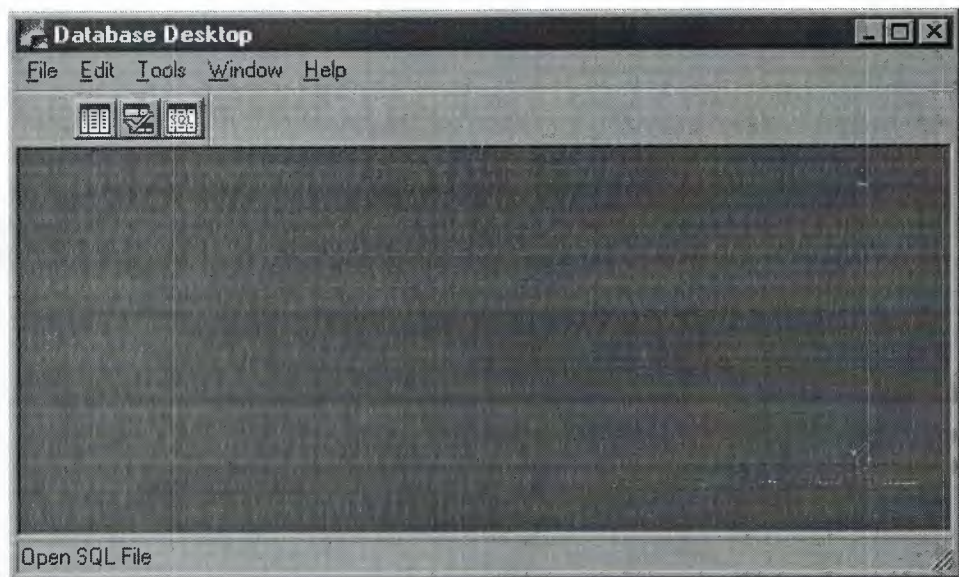


figure 2.2.a . Database Desktop Initial Launch status

Now, Let us examine the Databases required for the program they are listed below:

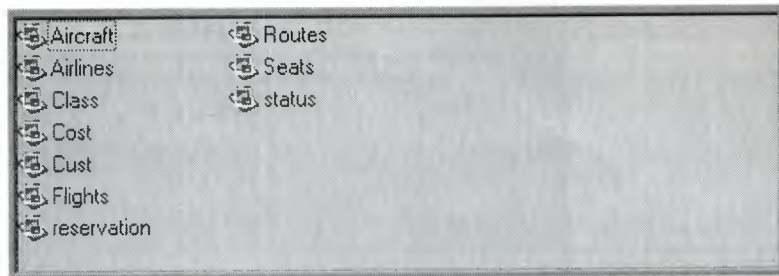


fig 2.2 b Visual List of DBs

- *Aircraft.db*
- *Airlines .db*
- *Class .db*
- *Cost.db*
- *Cust.db*
- *Flights.db*
- *Reservation.db*
- *Routes.db*
- *Seat.db*
- *Status.db*

2.2.1 Detailed Account of Each Database

Now let us examine each and every database in detail as to understand what it consists of and why have the fields that comprise the table are henceforth chooses

2.2.1.1. Detailed Account of Each Database

2.2.1.1.1. Aircraft .db

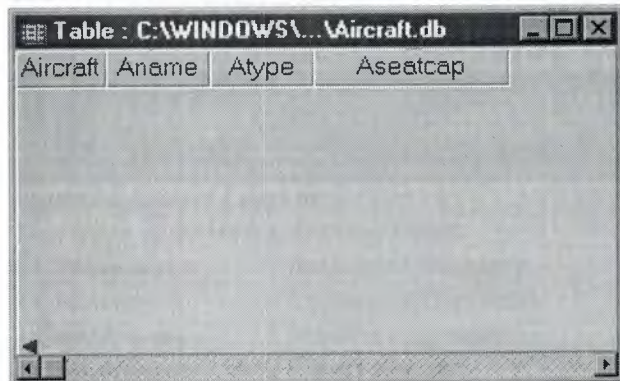


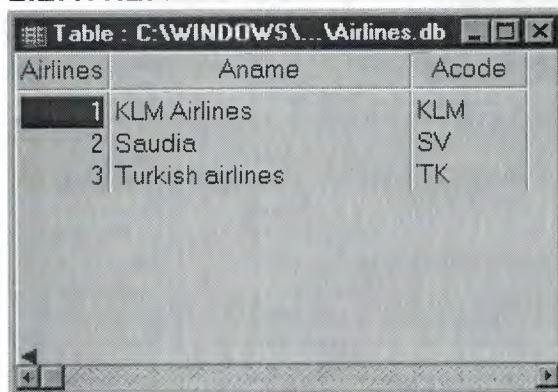
figure2.2.1.1.1.a Aircraft.db

Airlines. Db consists of the following files

1. Aname
2. Atype
3. AseatCap

Aname represents the name of the airline , Atype represents the type of airline, and the 3rd filed represents the seating capacity of the aircraft.

2.2.1.1.2. Airlines .db



Airlines	Aname	Acode
1	KLM Airlines	KLM
2	Saudia	SV
3	Turkish airlines	TK

figure2.2.1.1.1.b Airlines.db

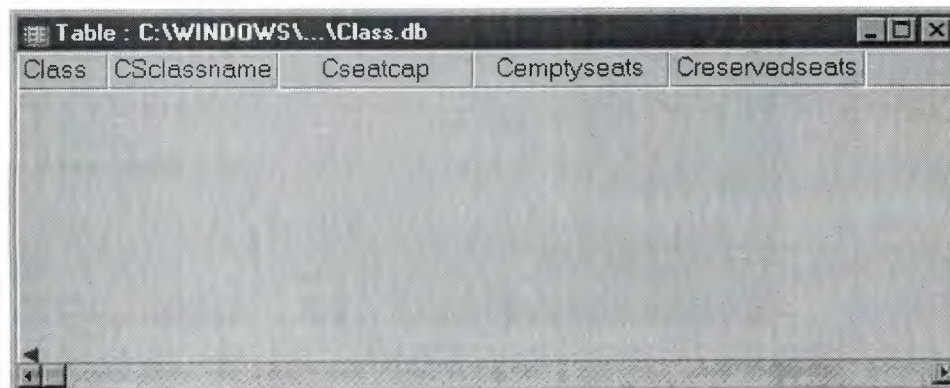
This table consists of the following Fields

- 1.Aname
- 2.Acode

Aname represents the name of the airlines as show in the diagram and Acod represents the code of the airlines used in the program,

2.2.1.1.3. Class .db

figure2.2.1.1.1.b Airlines.db



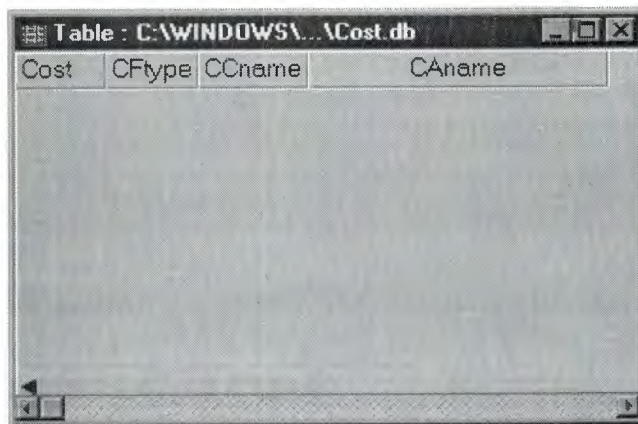
Class	CSclassname	Cseatcap	Cemptyseats	Creservedseats
-------	-------------	----------	-------------	----------------

figure2.2.1.1.1.c Class.db

This table includes the flowing fields.

- 1.CSclassname // deals with the name
- 2.Cseatcap // Capacity of the seats
- 3.Cemptyseats // No of empty seats
- 4.Creseredseats // Reservedseats

2.2.1.1.5. Cost .db



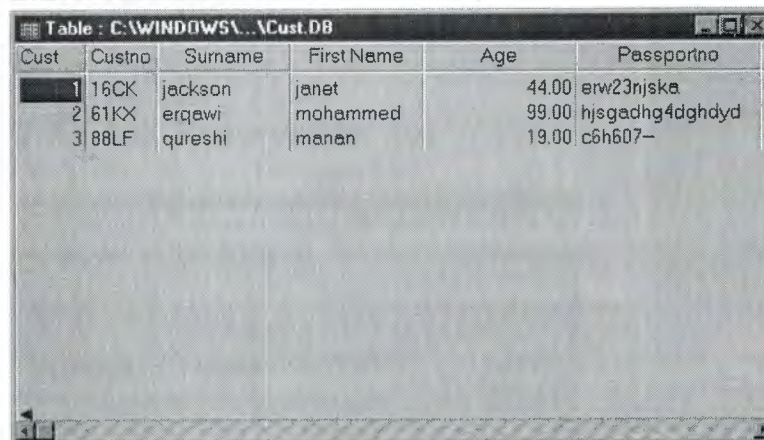
Cost	CFtype	CCname	CAname
------	--------	--------	--------

figure2.2.1.1.d Class.db

Tins contains the following

- 1.CFtype
- 2.CCname
- 3.CAname

2.2.1.1.6. Cust .db



Cust	Custno	Surname	First Name	Age	Passportno
1	16CK	jackson	janet	44.00	erw23njska
2	61KX	erqawi	mohammed	99.00	hjsqadhg4dghdyd
3	88LF	qureshi	manan	19.00	c6h607-

figure2.2.1.1.f Cust.db

this database consists of the following fields

1. Custno
2. Surname
3. age
4. passportno
5. streedadd
6. postalcode
7. htell
8. Htel2
9. Mob
10. email
11. fax

2.2.1.1.7. Flights .db

Flights	Fno	Fdeptime	Farrtime	Fduration	Fdepsource	Farsdestin	Faircraft	Ftype	Frouta	Fairlines
1	pk111				isb	jed		d	isb-jed	pia
2	pk112				ecn	ist		d	ecn-ist	pia
3	sv216				jed	ist		d	ist-jed	saudia
4	sv217				ecn	ist		d	ecn-ist	saudia
5	tk1233				ecn	ny		d	ecn-ny	turkish
6	tk1264				ecn	ist		d	ecn-ist	turkish

figure2.2.1.1.1.g Fligts.db

This table consists of the following fields:

1. Fno
2. Fdeptime
3. Farrtime
4. Fduration
5. Fdepsource
6. Farsdestin
7. Faircraft
8. Ftype
9. Froute
10. Fairlines

2.2.1.1.8. Reservation .db

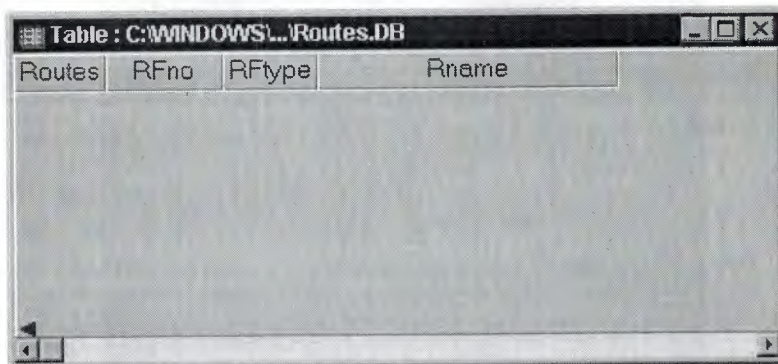
reservation	Rnc	Rpinno	Rcustno	RFno
1		ECL63V8	88JG	pk111
2	1	HDH67I7	16CK	tk1264
3	2	BMW38V	61KX	tk1264
4	3	ECL63V8	88JG	pk111
5	4	CVJ40F2	35QU	tk1264
6	5	SDE340	45XK	pk112
7	6	RDH26V	08SP	pk112
8	7	NJR54D	00VW	pk112
9	8	HFH58JE	50MS	tk1264

figure2.2.1.1.1h Reservation.db

This table Consists of the following fields

1. Rno
2. Rpinno
3. Rcustno
4. RFno

2.2.1.1.9. Routes .db



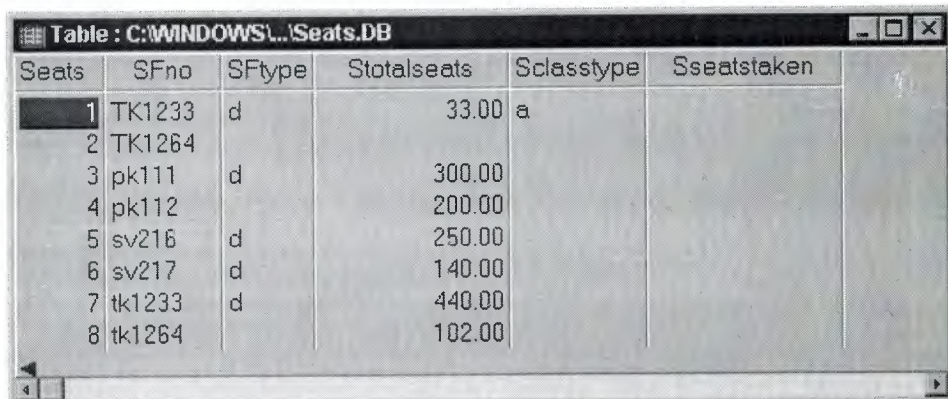
Routes	RFno	RFtype	Rname
--------	------	--------	-------

figure2.2.1.1.i Routes.db

This table consists of the following elements

1. RFno
2. RFtype
3. Rname

2.2.1.1.10. Seats .db



Seats	SFno	SFtype	Stotalseats	Sclasstype	Sseatstaken
1	TK1233	d	33.00	a	
2	TK1264				
3	pk111	d	300.00		
4	pk112		200.00		
5	sv216	d	250.00		
6	sv217	d	140.00		
7	tk1233	d	440.00		
8	tk1264		102.00		

figure2.2.1.1.j Seats.db

This table consists of the following Fierlds

1. SFno
2. SFtype
3. Stotalseats
4. Sclasstype
5. Sseatstaken

2.2.1.1.11.Status .db

Table : C:\WINDOWS\...\status.DB				
status	SRcustno	SRpinno	Sstatus	StFno
1	00VW	NJR54D7	T	pk112
2	08SP	RDH26V1	T	pk112
3	16CK	HDH6717	T	tk1264
4	35QU	CVJ40F2	T	tk1264
5	45XK	SDE34O3	T	pk112
6	50DN	BUS34A6	T	sv217
7	50MS	HFH58J8	T	tk1264
8	61KX	BMW38V0	T	tk1264

figure2.2.1.1.k Status.db

This table consists of the following elements

1. SRcustno
2. SRpinno
3. Sstatus
4. StFno

2.2.2. Inter-Table Connections and relationships (Flow of control)

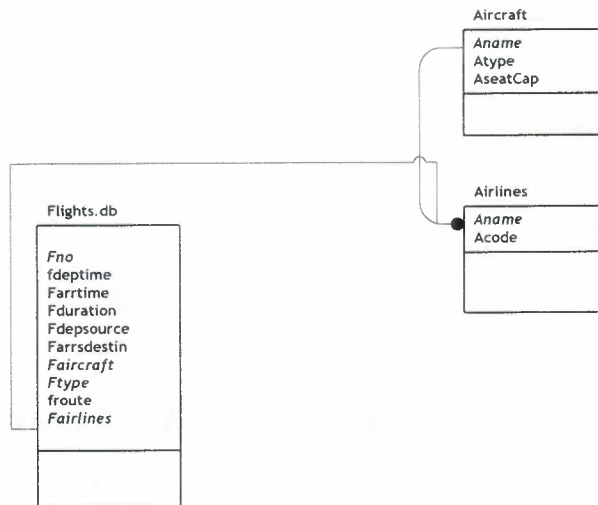


figure2.2.2b Flow of Control and Interconnections

The above diagram show three tables *flights.db*, *Aircraft.db*, *Airlines.db* respectively. Now a closer look reveals that there are some fields common to all three of the tables.i.e. *Fairlines*, *Aname*, *Aname*. Now this is due to the modularity of the process that has been followed a to keep the data as widely and as modular as possible.

Now a mechanism has been developed in the program that allows database editing in such a way that editing one linked record automatically updates the same filed in the linked status.

More detailed account of this will follow in the coming chapters.

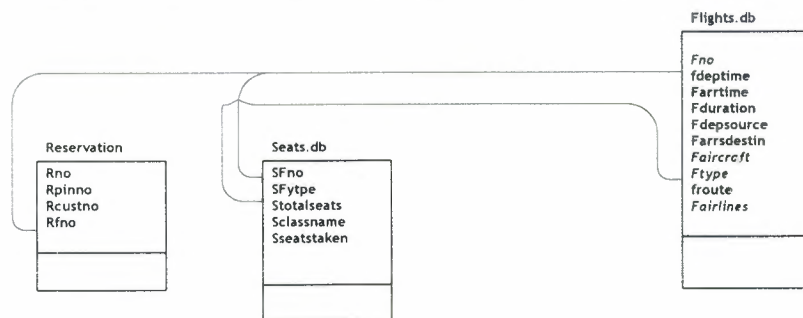


figure2.2.2c Flow of Control and Interconnections

++As the above figure reveals that All these three tables share atleast one field,. In the case of the last two, the number of shared fields is two. This

modular distribution of data Is intentionally kept simple as there is no need to further complicate the already very shady logic.
A few more examples are given below ;

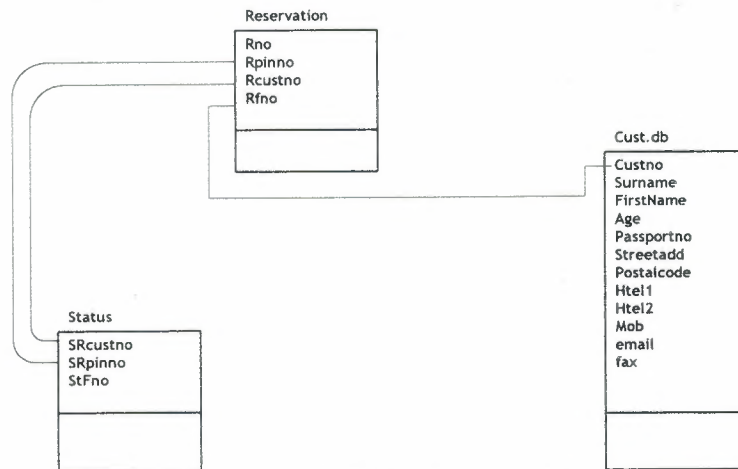


figure2.2.2d Flow of Control and Interconnections

A word of caution must be uttered here, during the programming phase of the project which will be covered later in detail it was discovered that modularity is both productive and counterproductive, in every sense of the word.

modular distribution of data Is intentionally kept simple as there is no need to further complicate the already very shady logic.
A few more examples are given below ;

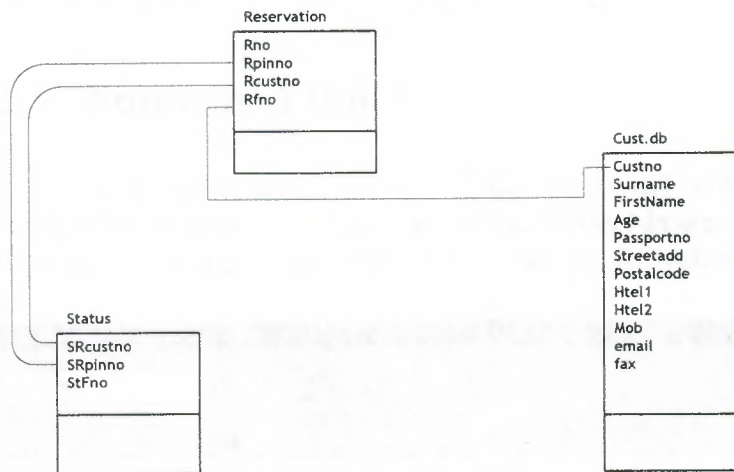


figure2.2.2d Flow of Control and Interconnections

A word of caution must be uttered here, during the programming phase of the project which will be covered later in detail it was discovered that modularity is both productive and counterproductive, in every sense of the word.

Chapter 3

Software Structure and detail

3.1 Forms and Units

First of all let's take a look at all the forms and units used in this project, we start off by showing the figure below the **Project Inspector** which as the name implies keeps a track of the number of units and the inherent information about them.

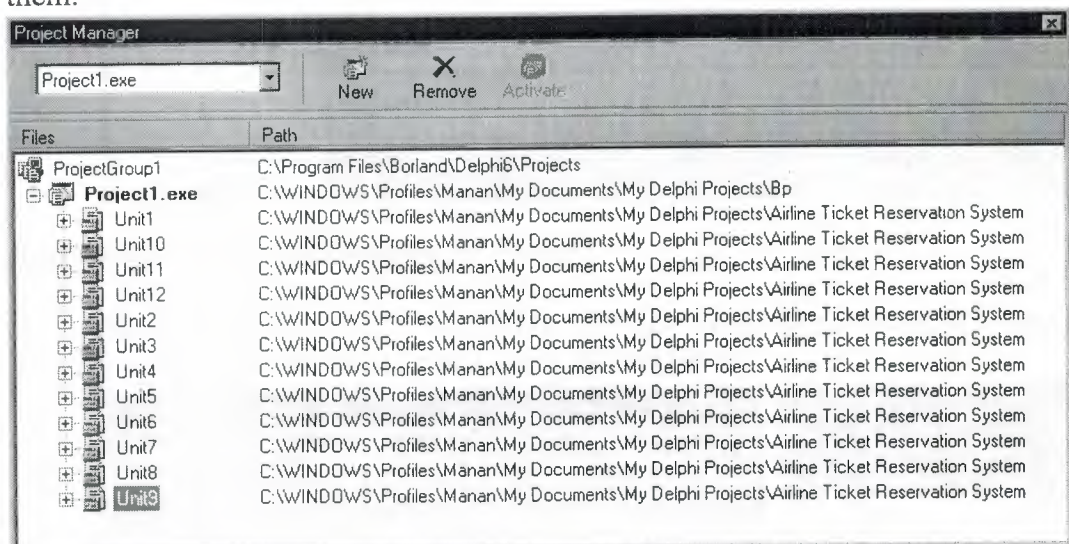


figure 3.1.i Object Manager

The object Inspector can be reached by clicking the *View* menu and then clicking on the OBJECT INSPECTOR menu or by using a keyboard shortcut namely Ctrl+Alt+F11.

Moving on to the forms, When the program icon is double-clicked by the user the form that appears is the main form or the form that is the first in the order of appearance,

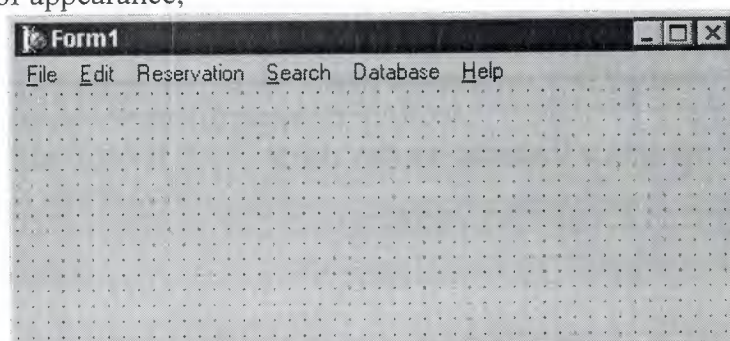


figure 3.1.ii Form1

the source code window of the main form is given below

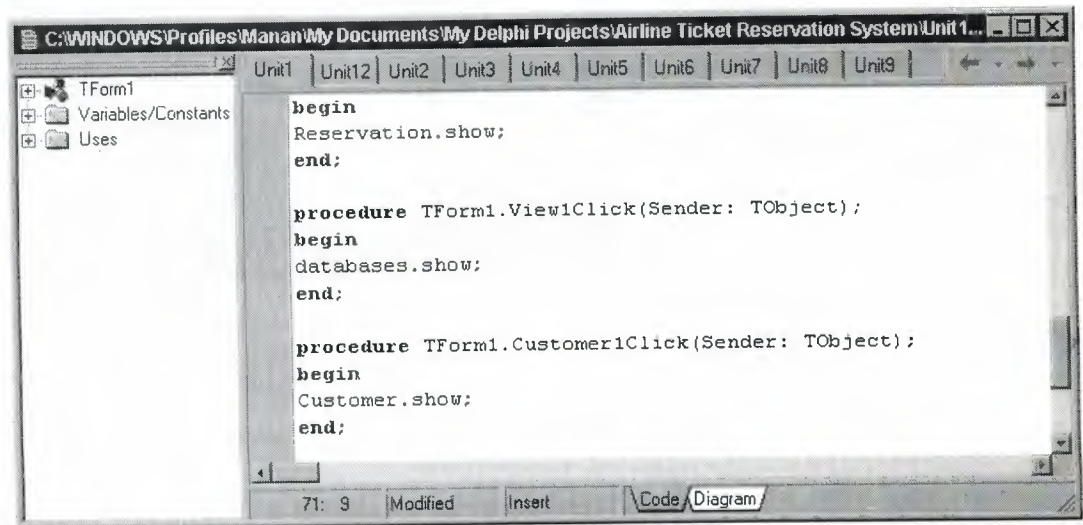


figure 3.1.iii Source Code window for FORM1

Now lets look at the form in detail, The form is a simpleone with a single menu and the menu can be accessed using the following

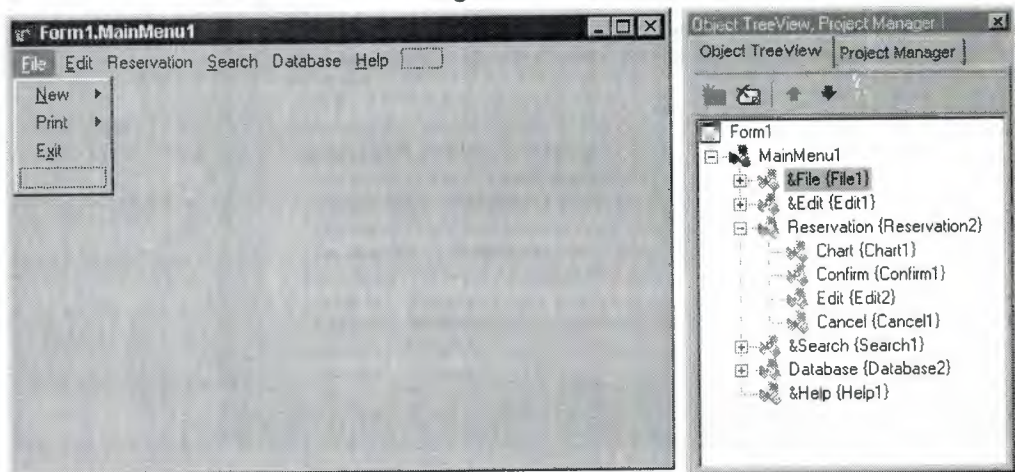


figure 3.1.iv Main Menu and Object Inspector for Main Menu

The above shows the Exact dimensions and contents of the Menu, which holds simple menus for simple operations.

The second form which can be accessed using the Reservation | Chart menu is shown below

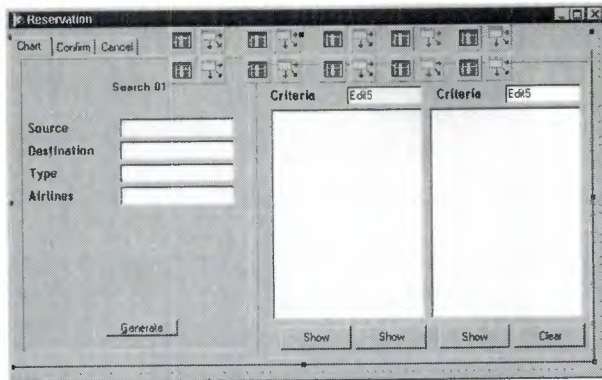


figure 3.1.v Form2

The source code for the above form is given below,

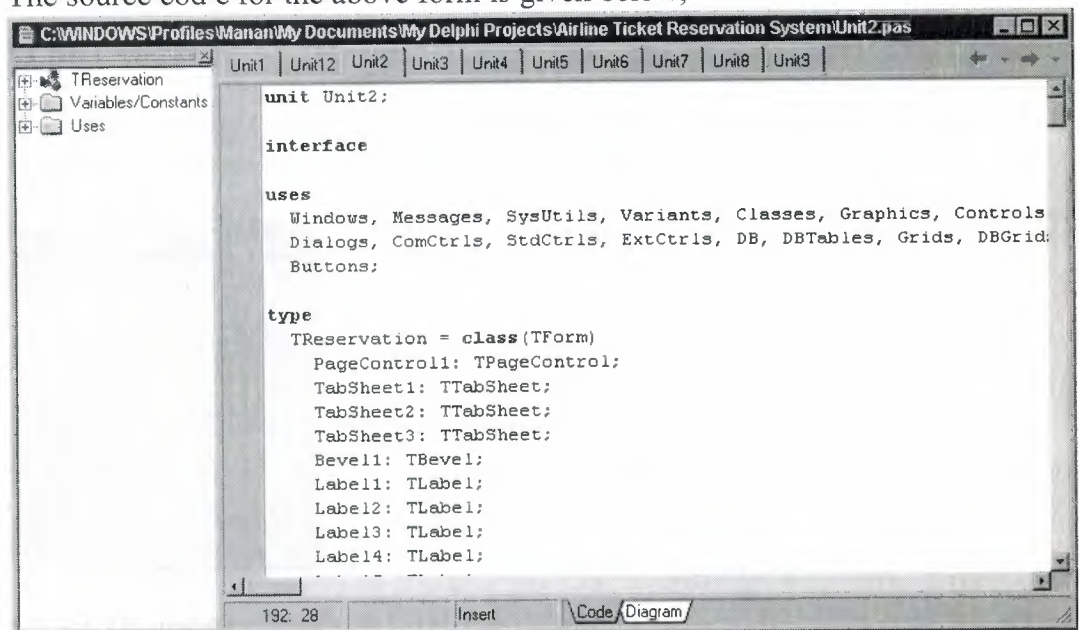


figure 3.1.vi Source code for Form2

as evident from diagram the form has many components and notice the tab component has 3 tabs but only the Chart component is highlighted as it's the only component that should be highlighted as the chart command was chosen.

The next for in the list is the Generate form, which is shown below,

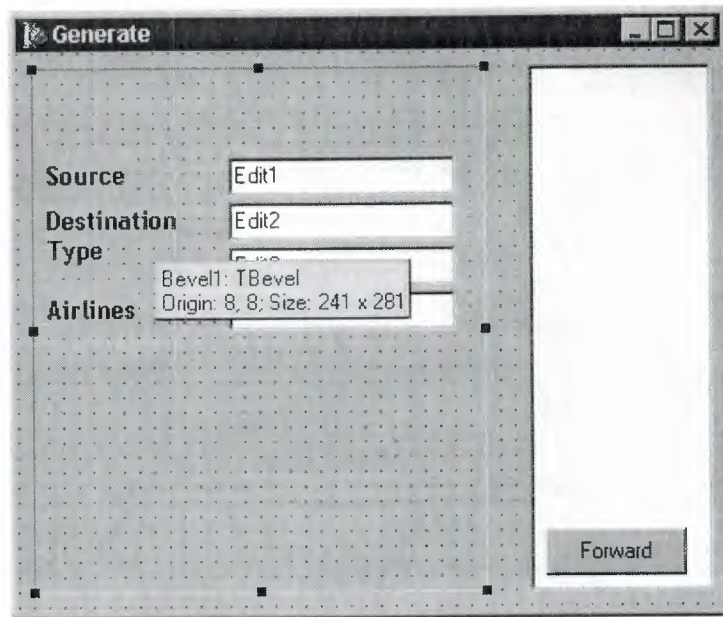


figure 3.1.vii Generate

the next form in the list is the Databases form , a prototype picture of which is provided below

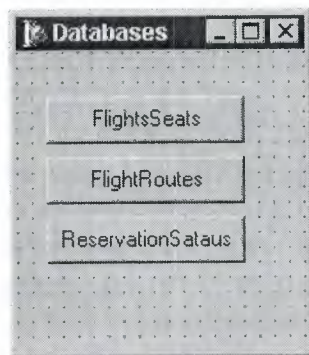


figure 3.1.viii Databases

The next form in the list one of the DB manipulation forms which deals with direct in put of data from the user.()

The image shows two database entry forms side-by-side. The left form is titled 'FlightsSeats' and the right form is titled 'FlightsRoutes'. Both forms have a toolbar at the top with various navigation icons. The 'FlightsSeats' form contains the following fields: Fno (EditFno), Fdeptime (EditFdeptime), Farrtime (EditFarrtime), Fduration (EditFduratio), Fdepsource (EditF), Farrsdestin (EditF), Faircraft (EditFai), Ftype (Ed), Froute (EditFroute), Fairlines (EditFairlines), SFno (EditSFn), SFtype (Ed), Stotalseats (EditStotalse), Sclasstype (Ed), and Sseatstaken (EditSseatsta). The 'FlightsRoutes' form contains the following fields: Fno (EditFno), Fdeptime (EditFdeptime), Farrtime (EditFarrtime), Fduration (EditFduratio), Fdepsource (EditF), Farrsdestin (EditF), Faircraft (EditFai), Ftype (Ed), Froute (EditFroute), Fairlines (EditFairlines), RFno (EditRFn), RFtype (Ed), and Rname (EditRname).

figure 3.1.ix Database Entry Subsets

As its evident from the picture that, as mentioned in the previous chapters, there are some linked filed appearing here, and also as mentioned befor ... fields wioll be updated as one of them is filled, namely the primary one.

The next form in the list is the Generate01 form, that primarily deals with the hardest part of all thatis the generation of Customer Pin numbers for the customer and the manipulation of different table datas. A pictorial description of the table is given below.

figure 3.1.x Generate01

The form is the Customer Info entry form that intakes info Simultaneous in steps along with the Generate01 form

figure 3.1.xi Customer Information

The next form is also a DB input form and the name is Form 10

The screenshot shows a window titled "ReservationStatus" with a standard Windows-style title bar (minimize, maximize, close buttons). Below the title bar is a toolbar with several icons. The main area of the window contains six input fields, each with a label to its left and an "Edit" button to its right:

- Rno: Ed
- Rpinno: EditRpin
- Rcustno: EditR
- SRcustno: EditS
- SRpinno: EditSRpi
- Sstatus: Ed

figure 3.1.xii Database Entry Subset

Form 11 in the list is the search form, and a the picture s given below

The screenshot shows two windows. On the left is a window titled "Customer" with a menu bar containing "File", "Edit", and "Help". The main area of the window is a grid. On the right is a window titled "Object TreeView" showing a hierarchical tree structure:

- Customer
 - MainMenu1
 - File {Customer1}
 - Edit {edit1}
 - Help {Help1}

figure 3.1.xiii Main Menu and Object Inspector for Main Menu

The above forms deals with the Customer and Flight data, and the Details of the menu item inside is also given in the above diagram.

Form 12 in this list of forms is the **Customer list Form** that deals with the customer info , the details of which will be provided in the coming sections.

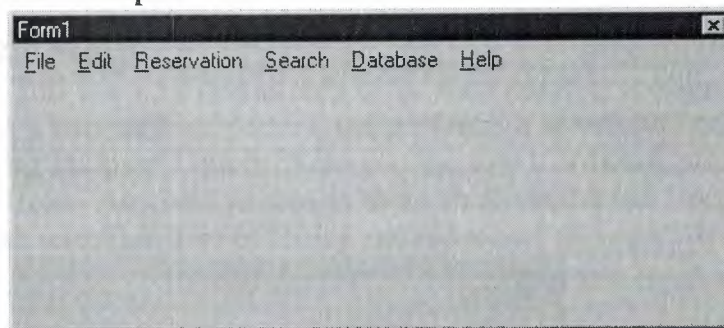
3.2 Flow of control (Procedures and Functions)

As in any program, this software has many concrete steps , sort of small leaps, that are taken each time, user activates a handle that in turn activates a function that produces a certain result, as per to the requirements of the user.\

Before going into the depth of the code involved in various stages of the software, the authors deems necessary that a pictorial presentation of the flow of the program be made, so it is shown below;

(**Note:** The pictorial Representation provided below is just a n example of a reservation procedure, due to the lack of space all procedures will not be accommodated))

3.2.1. Step 1.



3.2.2. Step 2

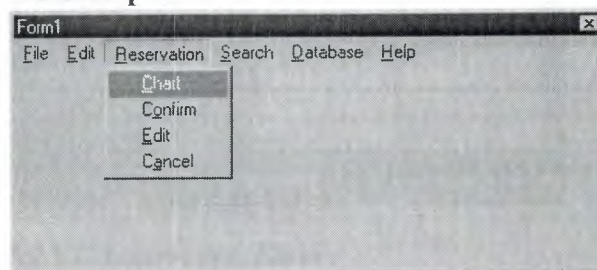


figure 3.2.2.i Form1

Procedures and Functions Involved:

In this step when the Reservation | Chart menu is selected , the following piece of code is executed: and we jump to step no 3.

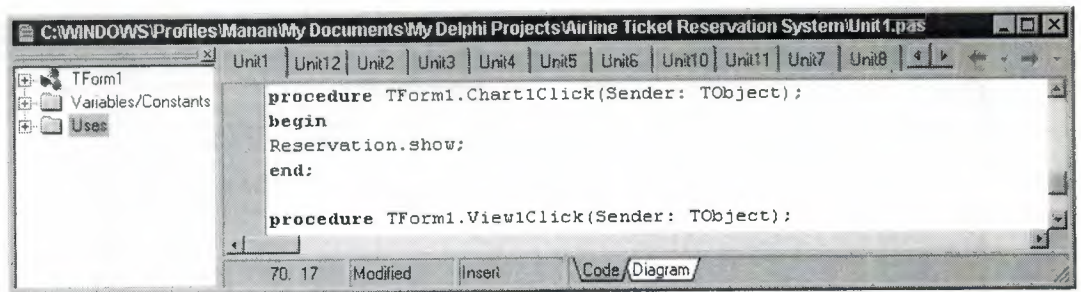


figure 3.2.2.ii Source Code for Reservation | Chart

3.2.3. Step 3

This is a very crucial step in the reservation procedure of an air ticket, as it accepts a user defined pattern and performs a search and comes up with results matching the user's query. Note that the tow List boxes present here Represent, Help Boxes, Any text written in them shall yield a list f results from which the user can select his item of choice and move on .

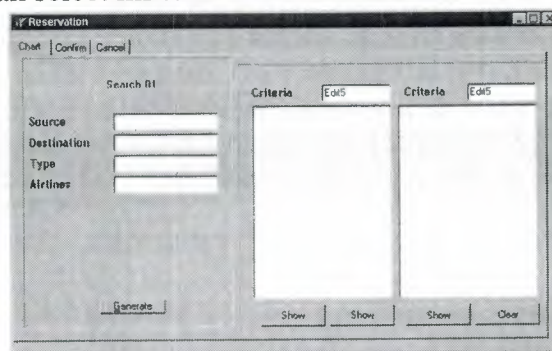


figure 3.2.3.i Reservation Form

Procedures and Functions Involved:

As it can be seen from the above diagram that this from has 5 buttons
Each button performs a specific task, Let us study each and every one separately

3.2.3.1 Generate button Procedure

The generate button accepts data form the 4 edit boxes in the form and performs a LOCATE search in the respective databases, the details of which can be seen in the diagram below;

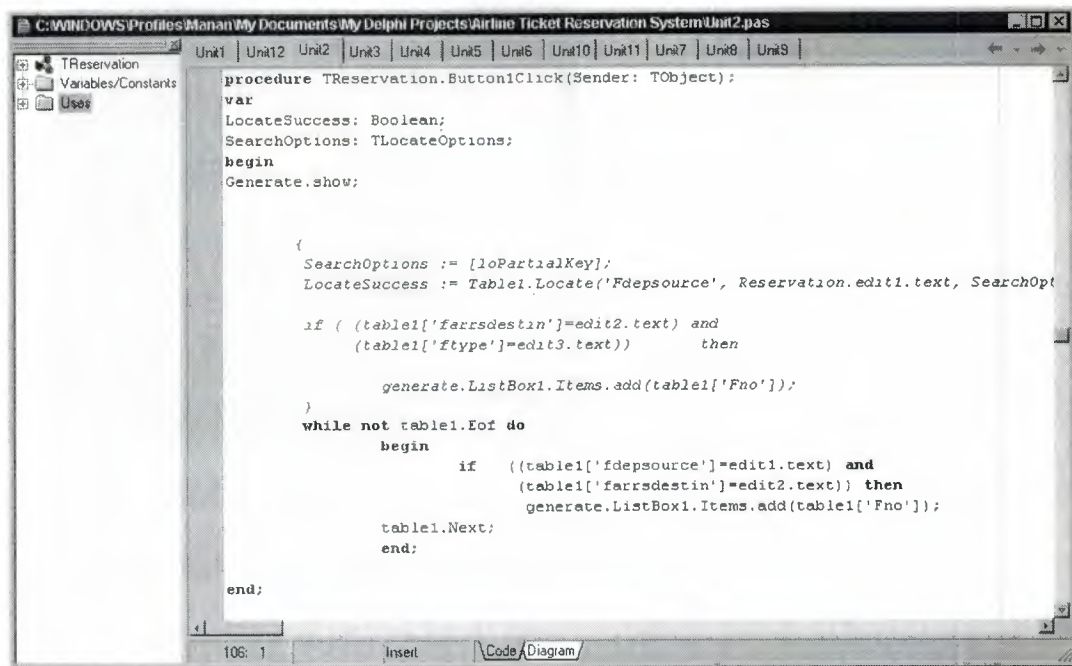


figure 3.2.3.1.i Generate Button Procedure

3.2.3.2 Show button Procedure

There are 2 show buttons in the form and each one of them perform user defined search and comes up with answers that aid the user in progressing further in the program. Only one of the two Show buttons shall be explained due to the usage of the same procedures with minor differences of search parameters.

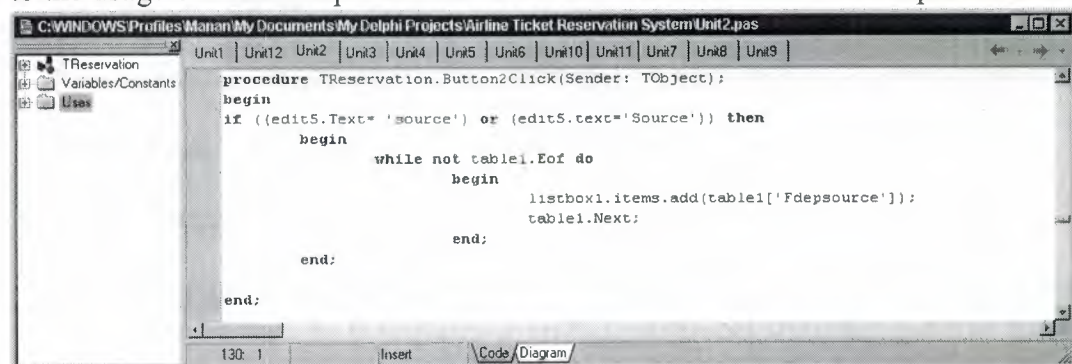


figure 3.2.3.2.i

The above procedure is a simple procedure , it only lists the data required by the user in an indexed form in the specific list box.

3.2.4. Step4

In this step, The Text boxes are static meaning that they only convey usage info from the previous operation, and the search results that the search patterns churn out are listed in the List box on the right of the form.

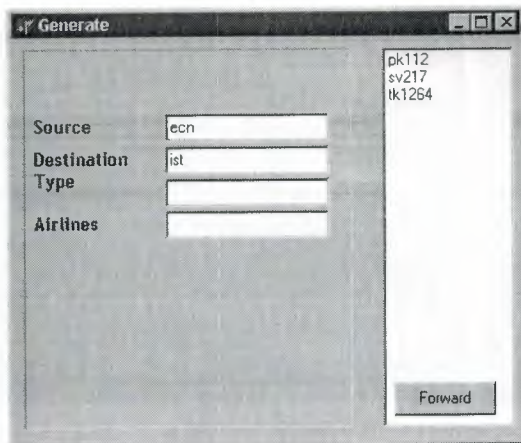


figure 3.2.4.i Generate Form

Procedures and Functions Involved:

3.2.4.1. Forward Button Procedure

The forward button is just a simple item selection procedure it basically selects an item clicked by the user and forwards it as the name implies to the next step of the solution.

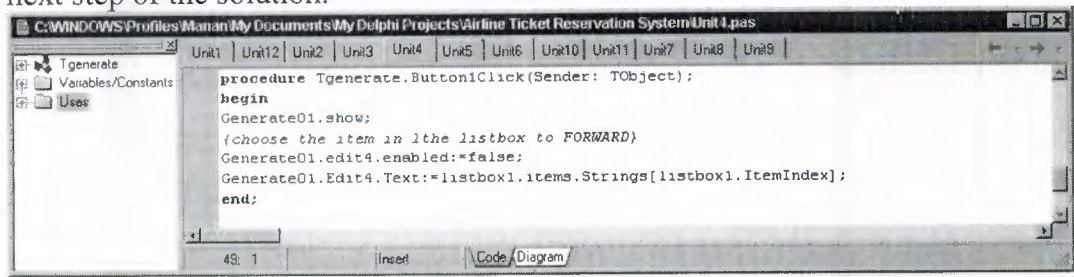


figure 3.2.4.1.i Source Code for Forward Button

3.2.5.. Step5.

This is the most crucial stage of the solution, and a very hectic stage on the part f the author, as it requires a multidimensional thinking patterns to be established , because many different things happen here, which affect many different forms and thus tremors are felt in many different places.

A mistake here could have been very costly as the complexity would have made it difficult to pin point the source of discontent.

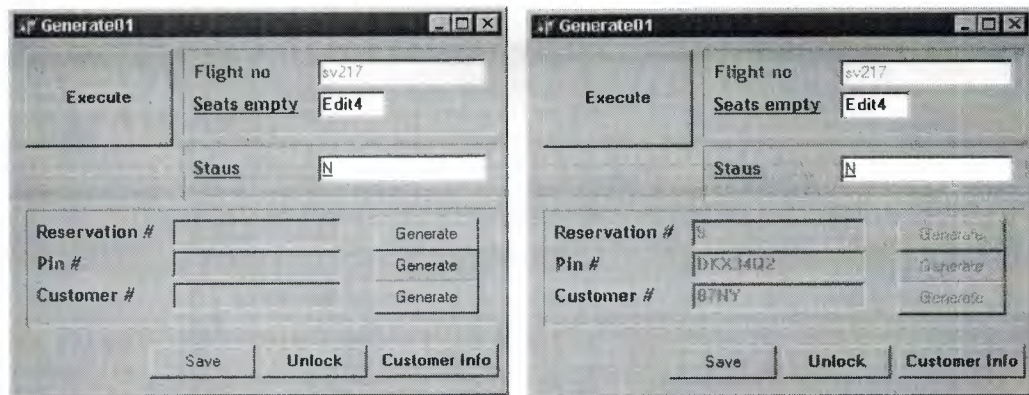


figure 3.2.5.i

One has to bear in mind that, when this form is activated, the user has not yet confirmed or temporarily booked his/her flight, therefore the status of the reservation should by default be N, and to do this, the following procedure is undertaken,

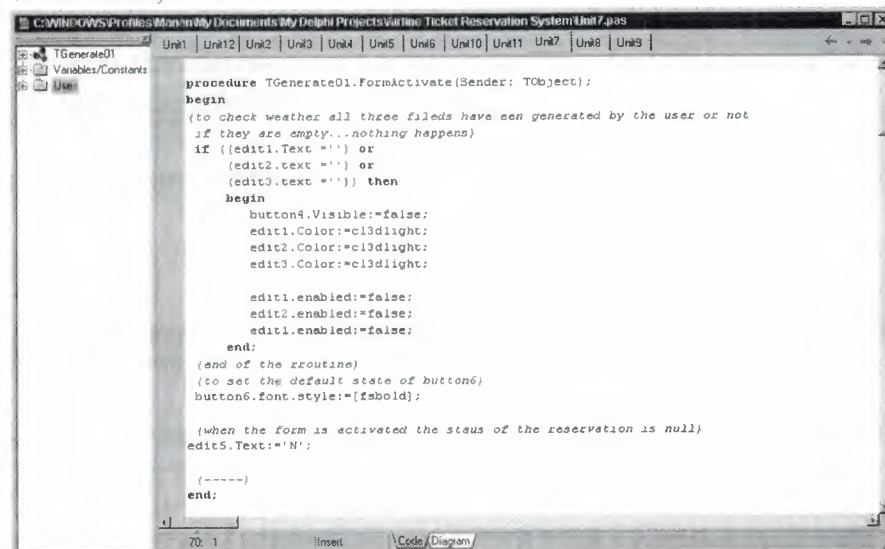


figure 3.2.5.ii Source Code for Form Activate Procedure

And now let's scrutinize the button by button procedures.

3.2.5.1. Generate button (Reservation #)

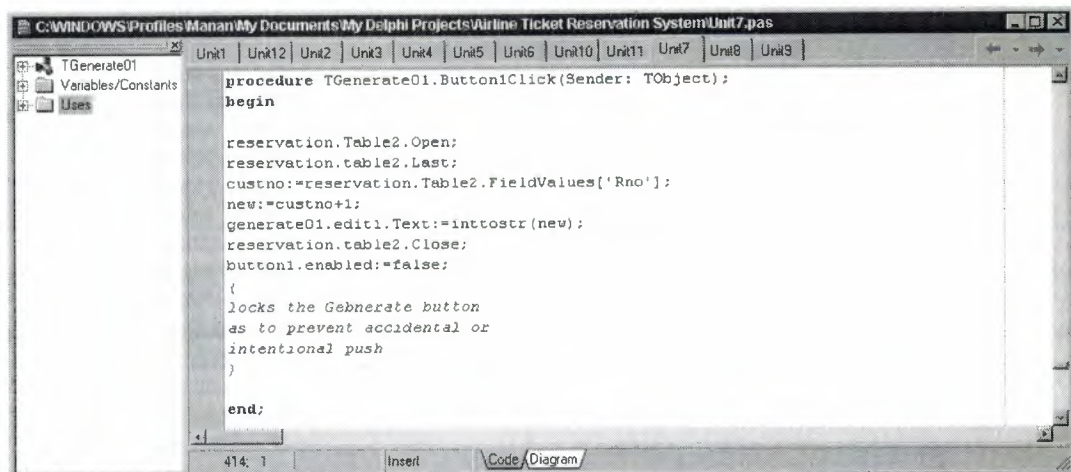


figure 3.2.5.1.i Generate Button (Reervation #)

This is actually a very simple look and tell procedure, it basically goes to the cust table and checks weather a Reservation exits, if one does that it automatically numbers the current record as one after that. And this accomplished with the help of statements evident in the diagram.

3.2.5.2. Generate button (Cust #)

This procedure generates a Randomly Generated customer # specific to that specific customer and the format of the no is NNAA, where N stands for a number between 0-9 and A stands for a Uppercase Alphabetic Character between A-Z.

The coding of this procedure is achieved using the following code.(the graphic could not be shown due to the length of the procedure.)

```

procedure TGenerate01.Button3Click(Sender: TObject);
var
    i: Integer;
    r1,r2,r3,r4,r5,r6,r7: integer;
    d01,d02,d03,d06: char;
begin
    {random PIN generator-----}
    randomize;

    for i:=1 to 9 do
        r4:=random(i);

    for i:=1 to 9 do
        r5:=random(i);

    {----- for d01-----}
    for i:=1 to 26 do
    begin
        r1:=Random(i);

```

case r1 of

```
1: d01 := 'A';
2: d01 := 'B';
3: d01 := 'C';
4: d01 := 'D';
5: d01 := 'E';
6: d01 := 'F';
7: d01 := 'G';
8: d01 := 'H';
9: d01 := 'I';
10: d01 := 'J';
11: d01 := 'K';
12: d01 := 'L';
13: d01 := 'M';
14: d01 := 'N';
15: d01 := 'O';
16: d01 := 'P';
17: d01 := 'Q';
18: d01 := 'R';
19: d01 := 'S';
20: d01 := 'T';
21: d01 := 'U';
22: d01 := 'V';
23: d01 := 'W';
24: d01 := 'X';
25: d01 := 'Y';
26: d01 := 'Z';
```

END;

end;

```
{-----}
{----- for d02-----}
```

for i:=1 to 26 do

begin

r2:=Random(i);

case r2 of

```
1: d02 := 'A';
2: d02 := 'B';
3: d02 := 'C';
4: d02 := 'D';
5: d02 := 'E';
6: d02 := 'F';
7: d02 := 'G';
8: d02 := 'H';
9: d02 := 'I';
10: d02 := 'J';
11: d02 := 'K';
12: d02 := 'L';
13: d02 := 'M';
14: d02 := 'N';
15: d02 := 'O';
16: d02 := 'P';
17: d02 := 'Q';
18: d02 := 'R';
19: d02 := 'S';
20: d02 := 'T';
21: d02 := 'U';
```

```

22: d02 := 'V';
23: d02 := 'W';
24: d02 := 'X';
25: d02 := 'Y';
26: d02 := 'Z';
END;
end;
{-----}

```

```

edit3.Text:=inttostr(r4)+inttostr(r5)+d01+d02;
{end of generatror-----}
{normalization of Edit1 field}
button3.enabled:=false;
edit3.Enabled:=false;
edit3.Color:=cl3dlight;
edit3.font.Style:=[fsBold];
{-----}

```

```

end;

```

3.2.5.3. Generate button (Pinno #)

This procedure generates a Randomly Generated customer # specific to that specific customer and the format of the no is AAANNANA, where N stands for a number between 0-9 and A stands for a Uppercase Alphabetic Character between A-Z.

The coding of this procedure is achieved using the following code.(the graphic could not be shown due to the length of the procedure.)

```

procedure TGenerate01.Button2Click(Sender: TObject);

```

```

var

```

```

  i: Integer;

```

```

  r1,r2,r3,r4,r5,r6,r7:integer;

```

```

  d01,d02,d03,d06:char;

```

```

begin

```

```

  {random PIN generator-----}

```

```

  randomize;

```

```

  {----- for d01-----}

```

```

  for i:=1 to 26 do

```

```

  begin

```

```

    r1:=Random(i);

```

```

  case r1 of

```

```

    1: d01 := 'A';

```

```

    2: d01 := 'B';

```

```

    3: d01 := 'C';

```

```

    4: d01 := 'D';

```

```

    5: d01 := 'E';

```

```

    6: d01 := 'F';

```

```

    7: d01 := 'G';

```



```

8: d01 := 'H';
9: d01 := 'I';
10: d01 := 'J';
11: d01 := 'K';
12: d01 := 'L';
13: d01 := 'M';
14: d01 := 'N';
15: d01 := 'O';
16: d01 := 'P';
17: d01 := 'Q';
18: d01 := 'R';
19: d01 := 'S';
20: d01 := 'T';
21: d01 := 'U';
22: d01 := 'V';
23: d01 := 'W';
24: d01 := 'X';
25: d01 := 'Y';
26: d01 := 'Z';
END;
end;
{-----}
{----- for d02-----}
for i:=1 to 26 do
begin
r2:=Random(i);

case r2 of

1: d02 := 'A';
2: d02 := 'B';
3: d02 := 'C';
4: d02 := 'D';
5: d02 := 'E';
6: d02 := 'F';
7: d02 := 'G';
8: d02 := 'H';
9: d02 := 'I';
10: d02 := 'J';
11: d02 := 'K';
12: d02 := 'L';
13: d02 := 'M';
14: d02 := 'N';
15: d02 := 'O';
16: d02 := 'P';
17: d02 := 'Q';
18: d02 := 'R';
19: d02 := 'S';
20: d02 := 'T';
21: d02 := 'U';
22: d02 := 'V';
23: d02 := 'W';
24: d02 := 'X';
25: d02 := 'Y';
26: d02 := 'Z';
END;
end;
{-----}
{----- for d01-----}
for i:=1 to 26 do

```

```

begin
r3:=Random(i);

case r3 of

    1: d03 := 'A';
    2: d03 := 'B';
    3: d03 := 'C';
    4: d03 := 'D';
    5: d03 := 'E';
    6: d03 := 'F';
    7: d03 := 'G';
    8: d03 := 'H';
    9: d03 := 'I';
    10: d03 := 'J';
    11: d03 := 'K';
    12: d03 := 'L';
    13: d03 := 'M';
    14: d03 := 'N';
    15: d03 := 'O';
    16: d03 := 'P';
    17: d03 := 'Q';
    18: d03 := 'R';
    19: d03 := 'S';
    20: d03 := 'T';
    21: d03 := 'U';
    22: d03 := 'V';
    23: d03 := 'W';
    24: d03 := 'X';
    25: d03 := 'Y';
    26: d03 := 'Z';

END;
end;
{-----}
for i:=1 to 9 do
    r4:=random(i);

for i:=1 to 9 do
    r5:=random(i);

    {----- for d01-----}
for i:=1 to 26 do
begin
r6:=Random(i);

case r6 of

    1: d06 := 'A';
    2: d06 := 'B';
    3: d06 := 'C';
    4: d06 := 'D';
    5: d06 := 'E';
    6: d06 := 'F';
    7: d06 := 'G';
    8: d06 := 'H';
    9: d06 := 'I';
    10: d06 := 'J';
    11: d06 := 'K';
    12: d06 := 'L';
    13: d06 := 'M';

```

```

14: d06 := 'N';
15: d06 := 'O';
16: d06 := 'P';
17: d06 := 'Q';
18: d06 := 'R';
19: d06 := 'S';
20: d06 := 'T';
21: d06 := 'U';
22: d06 := 'V';
23: d06 := 'W';
24: d06 := 'X';
25: d06 := 'Y';
26: d06 := 'Z';
END;
end;
{-----}

for i:=1 to 9 do
  r7:=random(i);

edit2.Text:=d01+d02+d03+inttostr(r4)+inttostr(r5)+d06+inttostr(r7);
{end of generatr-----}
{normalization of Edit1 field}
button2.enabled:=false;
edit2.font.Style:=[fsBold];
{-----}

```

end;

3.2.5.3. Customer Information Button

This button launches the Customer Information form, for the entry of customer information and cosequential storage in the cust database.

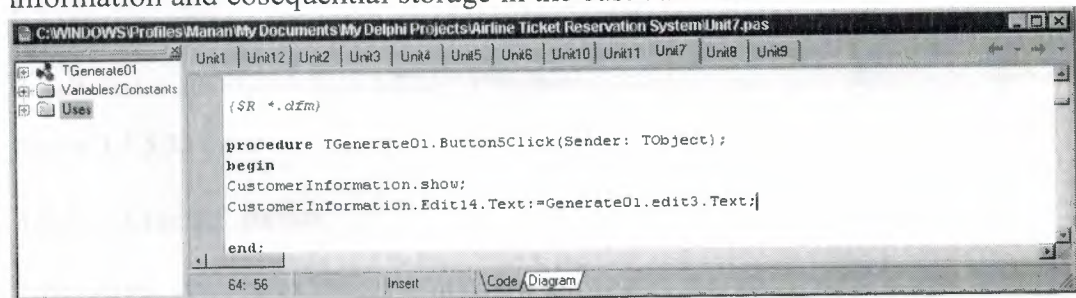


figure 3.2.5.3.i Customer Information Button

3.2.5.4. Unlock Button

As mentioned before, In order to prevent from accidental double clicking of the 2 generate buttons, they are one touch only and will be disabled after being clicked one time, this procedures lifts that restriction temporarily and opens those two buttons

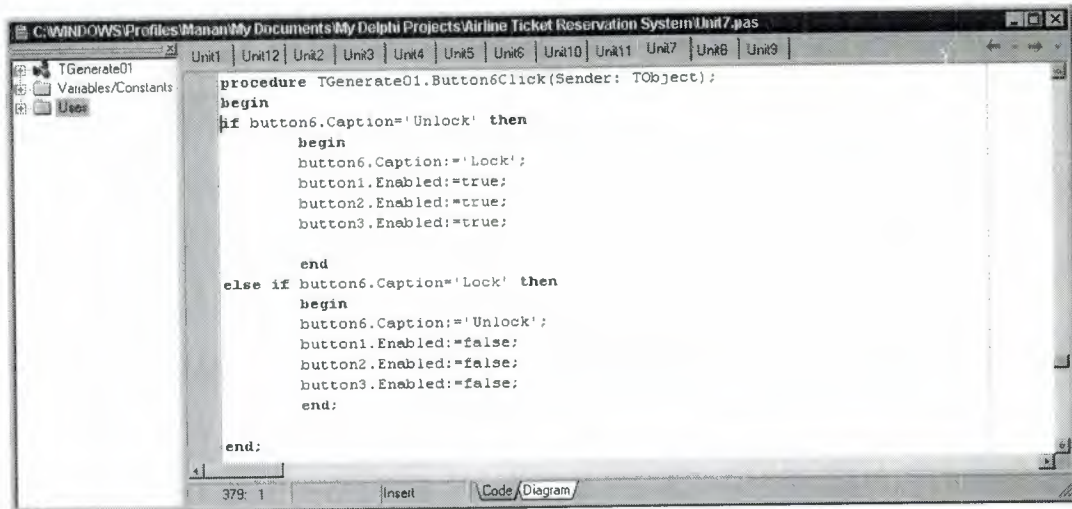


figure 3.2.5.4.i Unlock Button

3.2.5.5. Save Button

This button is saves the generated random patterns to a database, this is achieved by the following.

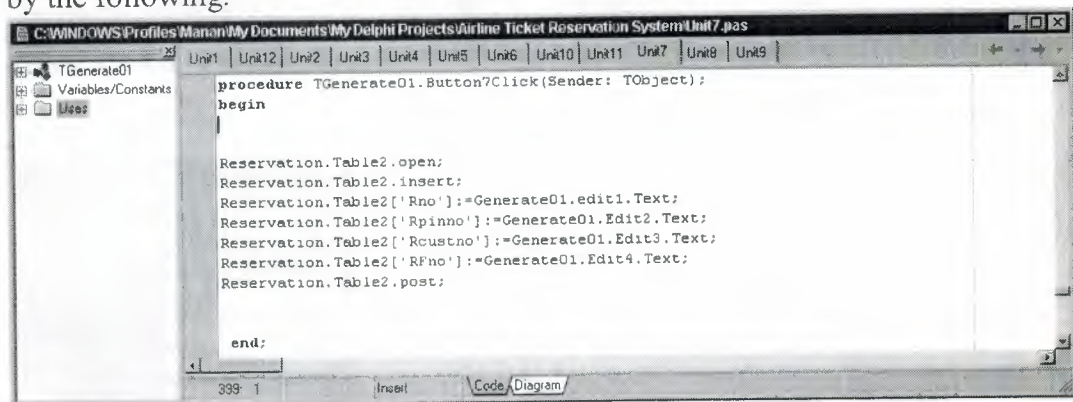


figure 3.2.5.5.i Save

3.2.5.6. Execute Button

This button does three things, it makes a temporary reservation for a user, enters it into status.db and also it reduces the no of empty seats in the seats.db by one for each reservation. And it posts a T instead of a N in status box of the form and the database.

This is done miraculously by the following code

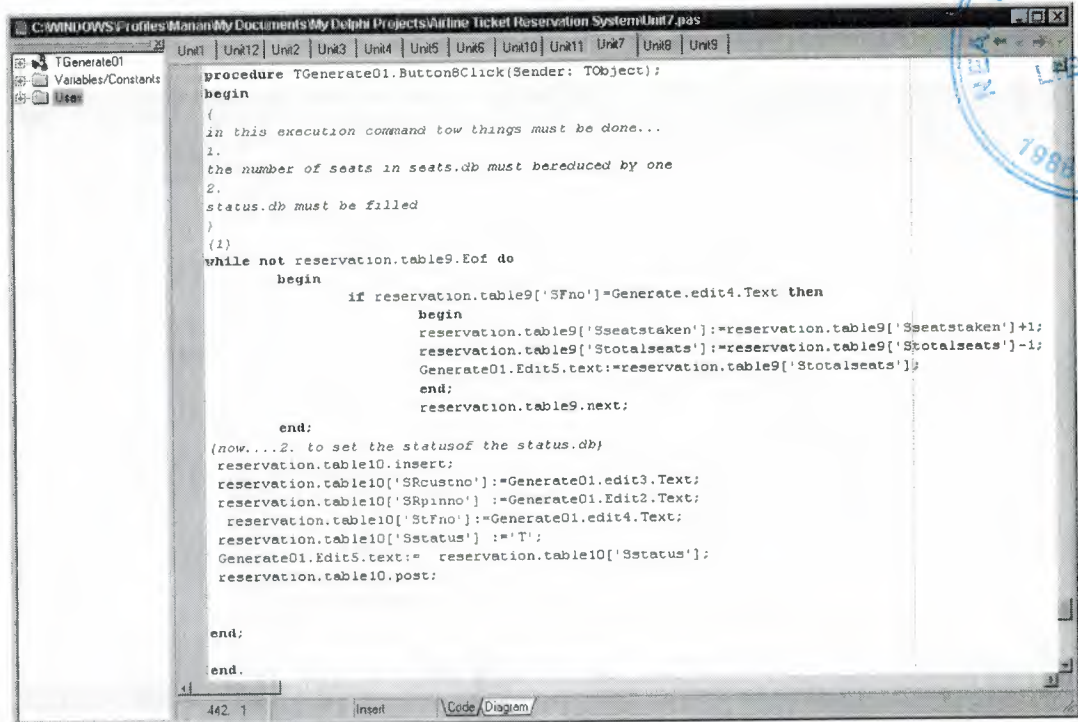


figure 3.2.5.6.i Execute Button

3.2.6. Step 6

Customer Information

Customer Info

Customer No: 874

Name:

Surname:

Age: Sex:

Passport #:

Street Address:

Postal Code:

Home Tel. 01:

Home Tel. 02:

Mobile:

Email:

Fax:

Ok Clear Cancel

figure 3.2.6.i Customer Info

This part deals with storage of personal info of the customer in database. The Saving is done by the following procedure:

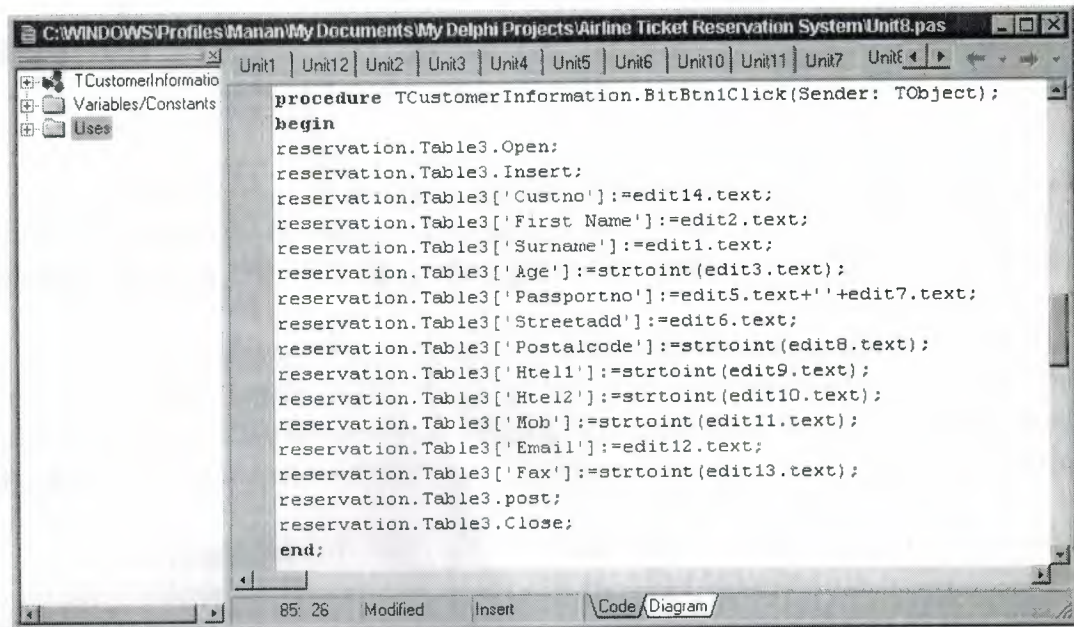


figure 3.2.6.ii OK button

Conclusion

During the course of the preparation of this system, the author has made many improvements in the way he perceive data flow in complex data system. The complexity of such big data systems amazingly reduces the logic construction time.

One thing that was of rime concern to the author, during the preparation of the project was the handling and distribution of data. At the end it was clear that modular non linear distribution of data is a very effective way of bringing down the system size and considerably reducing the risk of cogging or jam during operation.

The development of big data system which have a lot of parallel functionality is done by distributing tasks, in our case, distribution of functionality is done by assigning one execution element to perform many duties at one certain time.

Another very interesting fact that the author accidentally stumbled upon was the fact that large databases are very easy to navigate and manipulate when the commands of manipulation are distributed.

The analysis of the system source code reveals and fortifies all what has been said In the paragraphs above.

REFERENCES

[1] Borland Software Corporation, 100 Enterprise Way, Scotts Valley, CA 95066-3249, **Borland®Delphi™ 6for Windows**