

Developing Programs with
Blackbox Oberon
by

Brett S Hallett
©2002

February 10, 2003

Contents

1	Preamble	9
1.1	Aquiring Blackbox Oberon	11
1.2	Other references	11
1.3	Blackbox Oberon Overview	12
2	Blackbox Oberon Standard Modules	13
2.1	Modules?	13
2.2	Standard Modules	14
2.2.1	Finding information about a MODULE	14
3	Extra Data Types Explained	17
3.1	Special Characters	17
3.2	Extra Data Types	20
3.2.1	Date and Time	20
3.2.2	Currency variables	22
3.3	Declaring Arrays	24
3.3.1	Example Arrays	24
3.3.2	Declaring an array of basic data type	25
3.3.3	Declaring an array of RECORD	26
3.3.4	Using a Multi-Dimensioned Array	27
4	Blackbox Oberon Text handling	29
4.1	Why more than one type of Text?	29
4.2	CHAR & ARRAY OF CHAR	29
4.3	The incompatable Text assignment error	31
4.3.1	Overcoming the incompatable Text assignment error	31
4.3.2	Declare your own Text data type	31
4.3.3	Use MODULE String procedures	32
4.4	Blackbox Oberon View Texts	35
4.4.1	Processing Blackbox Oberon View data	35
4.4.2	Processing Text data using TextMappers.Scanner	38
5	Standard Controls	41
5.1	Radio Buttons	41
5.1.1	Example Output	46
5.1.2	Setting up a Radio Button	47
5.2	Check Box	47

6	ComboBoxes	49
6.1	Comboboxs Example	49
6.2	Setting up a list	50
6.3	Using Embedded Code to load List Controls	51
6.4	The Resources File	56
6.4.1	Resource Key	56
6.5	Using SetResource to load List Controls	57
6.6	An example resources file	61
6.7	Using a List, Selection or Combo Box to retrieve information	62
6.7.1	List Boxes	63
6.7.2	Selection Boxes	64
6.7.3	Combo Boxes	65
6.8	Dynamically loading a List, Selection or ComboBox	66
6.9	Example dropdown list form	67
6.10	Re-building a Dynamic list	68
7	The MySQL Database	79
7.1	Introduction	79
7.2	Blackbox Oberon MySQL general features	80
7.2.1	Full access to SQL	80
7.2.2	Integration of MySQL with Blackbox Oberon	81
7.2.3	Extensibility	81
7.2.4	Separation of program logic and user interface	81
7.3	Aquiring MySQL	81
7.4	The MySQL Book	82
7.5	Installing and running MySQL	82
7.5.1	ODBC & MySQL	82
7.5.2	Create an MySQL database	85
7.5.3	Creating MySQL Tables	85
7.5.4	MySQL Script file to create the tables.	86
7.6	Databases, tables, and interactors	88
7.6.1	Databases verses Tables verses Fields.	89
7.6.2	Database	90
7.6.3	Database Status Response	91
7.6.4	Table Object	91
7.6.5	Read a table row	92
7.6.6	Checking if a SQL statement executed successfully	93
7.6.7	Data Types Supported	94
7.6.8	BlackBox interface to SQL Code	95
7.6.9	Type Row	96
7.6.10	Blobs	96
7.6.11	Asynchronous operation	97
7.6.12	ODBC driver (Windows only)	97
7.6.13	Displaying MySQL tables	97
7.6.14	Table control in its own Window	98
7.6.15	Table control in a form	98
7.6.16	The Table Notifier	99

7.6.17	Linked SQL Controls	99
7.7	Editing MySQL Table Entries	103
7.8	Design rules	105
8	The Report Generator : BxRepGen	107
9	'Karin' a small Blackbox Oberon & MySQL system	133
9.1	The data collecting program	134
9.2	The report output	137
9.2.1	The report program code	137
9.2.2	The generated report program explained	138
10	Modal Form Execution	145
10.1	Snapshots of MODAL operation	146
10.2	How was it done?	148
10.2.1	QuickQuote : Template Procedures	149
11	Programming with Recursion	151
11.1	Printing Cheque Value as Words	151
12	Debugging Oberon Code	161
12.1	Trace Code	162
12.2	ASSERT	166

List of Figures

2.1	Some useful Standard Oberon Modules	14
3.1	Display System Date & Time values	20
3.2	Defining and accessing Currency Variables	23
3.3	Array A of CHAR Array Z of REAL	24
3.4	Multi indexed array of INTEGER	25
3.5	Multi indexed array Of RECORD	26
3.6	Using a multi dimensioned array	27
4.1	Incompatible text assignment	31
4.2	Overcoming incompatible text assignment	32
4.3	Using Module String to overcome assignment error	33
4.4	Copy a Oberon Document to ASCII TEXT File	38
5.1	Calculate Pulleys Program	41
5.2	CalcPulley Program	45
5.3	Example Calculate Pulleys Output	46
6.1	Demo Comboboxes	49
6.2	Declaring List Access	50
6.3	Load Lists from Imbedded Code	55
6.4	Load Lists using SetResources	60
6.5	Example List Box Usage	63
6.6	Example Selection Box Usage	64
6.7	Example Combobox Usage	65
6.8	Dynamic Reload of Dropdown List	67
6.9	Dynamic Loading and Using ListBox	77
7.1	Oberon to Mysql database via ODBC	80
7.2	Example Mysql ODBC alias setup	84
7.3	Summary of MySQL Statements (commands)	89
7.4	MySQL to Blackbox Oberon Data Type Conversion	94
7.5	Gui Form displaying MySQL 1 – > Many record relationship	102
7.6	Edit MySQL Table using 'side bar fields'	103
7.7	Code to Update / Delete selected MySQL Table Entry	104
8.1	BxRepGen - Show available Databases	107
8.2	BxRepGen - Show Tables in selected Database	108
8.3	BxRepGen - Describe Selected Table	109
8.4	BxRepGen - Display Selected Fields in Selected Table	110

8.5	BxRepGen - The Generated Report	111
8.6	The BxRepGen program	131
9.1	Karin : collecting the receipt data form	133
9.2	Karin : data collection program dissected	136
9.3	Karin : reporting the collected receipt data	137
9.4	Karin : report program dissected	143
10.1	Showing two separate program executing (non-modal)	145
10.2	Before pressing Create Template Button	146
10.3	First form waits for user to fill in second form	147
10.4	Upon return from second form, after pressing OK button	147
10.5	Definition of OK Button	148
11.1	Calling Cheque Conversion Library	153
11.2	Convert \$ value to text Words	153
11.3	The Cheque Conversion Module	158
11.4	Example recursion processing	159
12.1	Debugging using Trace Code	162
12.2	Poor Trace Log of Date Conversion	163
12.3	Better Trace Log of Date Conversion	163
12.4	A better method of system date conversion	164

Chapter 1

Preamble

The Blackbox Oberon program is one of the modern programming systems that were developed to encompass the programmer/developer needs in a using GUI interfaces, Reporting facilities, Database access and reliable programming language in one development environment. Others in this class include, Paradox, Delphi, Clarion, etc. Note however these products are not of the '4GL' class and, in general, they do not rely on pre-written solutions to achieve the desired end user result, therefore you will be expected to write code!

Although Blackbox Oberon is a complete development environment , in the authors opinion there is a lack of programmer level information about how to get involved with Blackbox Oberon and how to go about using Blackbox Oberon in real development situations.

Indeed , why use Blackbox Oberon at all ?

This book is written from the perspective of a practicing programmer who while trying to develop programs in Blackbox Oberon , found a lack of useful developer information in the supplied documentation or the few other sources available. Mostly it is a discussion of what I've tried. This book is more *programmers notes* than a reference manual. This is not to say that the other information is not of a good quality, just delivered from a different perspective.¹

The examples used are either full programs (modules) or extractions of code from actual programs developed by the author using Blackbox Oberon . They may not be pretty but they are real programs. With few exceptions all code shown and discussed have been compiled and executed under Blackbox Oberon .²

¹However I will say that the supplied help facility is not very programmer friendly, it simply returns a index to every reference to words entered, while powerful in its own right, is not very useful when trying to find out how to call another GUI form, for example.

²Blackbox Oberon version 1.4 at the time of writing this book

It should be pointed out that the author has no special knowledge of the internal workings of Blackbox Oberon nor access to details other than that available in the normal released version or information available from the internet. Therefore, comments, observations and footnotes about Blackbox Oberon , may be quite incorrect in the technical sense, but I can only write about what I see and believe to be happening and what I would expect you to observe also if you attempt the same task.³

*Also I have not covered the complete Blackbox Oberon development product as I know there are features that I have not used and probably will never use, indeed there are many features (options) in the standard development menu(s) that I dont have a clue about what they are for!*⁴

However, I think that this book will be very useful to ones enjoyment of using the Blackbox Oberon System. I have programmed in too many languages (some 50+) over too many years (since 1964) and have found that Blackbox Oberon is a pleasant environment to program in, not perfect, but still a pleasure – good luck.⁵

Brett S Hallett
Golden Square
February 10, 2003

³in fact many of my observations maybe based on bulldust!

⁴this book will be updated as usege of such features emerge

⁵my prefered GUI development language was *Paradox for Windows*, but its development has 'gone off the rails" since Borland dropped it in favour of Delphi, I still used it from time to time

1.1 Acquiring Blackbox Oberon

Blackbox Oberon is available by writing to :

Oberon Microsystems. Inc,
Technoparkstrasse 1
CH-8006 Zurich
Switzerland

or direct from their web site : www.oberon.ch

There are other versions of Oberon available⁶, the author has had no experience with those and this book only refers to Blackbox Oberon . You will find them via a internet search.

1.2 Other references

The best available introduction to Blackbox Oberon and the Component Pascal Language is the book "Programming with Blackbox" by J. Stanley Warford available via the internet via <ftp://ftp.pepperdine.edu/pub/compsci/prog-bbox/> . This covers much of Blackbox Oberon in detail but does not cover many of the little bits of information that a programmer needs to complete his/her task. His book is some 600 pages long in .pdf format for viewing/printing with *Acrobat Reader*. (available free off the internet).

The book was created for the teaching of Blackbox Oberon and therefore digresses into some programming theory, which I'm not concerned about here, however its interesting reading.

Also that book does not cover developing multi GUI Form systems or SQL databases at all.⁷

⁶which is a major problem in selecting a particular implimentation, I chose Blackbox Oberon as its well suited to Windows development

⁷I'm particularly interested in using MySQL with Blackbox Oberon , hence this book

1.3 Blackbox Oberon Overview

Blackbox Oberon offers the programmer/developer a *development environment* from which to completely develop his/her program. The use of other external tools are generally not required to achieve the desired results, eg: an external GUI library is not required. Blackbox Oberon has a fully integrated GUI tool kit from which one chooses components, or *controls*⁸ to complete the end-user forms with which they interact, an integrated text editor to write the Component Pascal code, and various compile & execute options, all accessed from the Blackbox Oberon gui. Unlike many such systems there is no difference from the end user GUI and the developer GUI, they are one and the same. Indeed the final delivered end-user system is simply a standard Blackbox Oberon with the developer components stripped out.

The Blackbox Oberon language (Component Pascal) is a development of Oberon-2, created by Niklaus Wirth, and is similar to traditional Pascal in structure, although with significant differences. Blackbox Oberon is a simpler, more refined language than Pascal in many ways, but by no means less powerful⁹. The Oberon language is an attempt by N.Wirth to reduce language complexity.¹⁰

Blackbox Oberon is a general-purpose language implementing programming features like, block structure, modules, static typing (with strong type checking across modules), automatic garbage collection, etc. Being general-purpose allows its usage across a wide variety of programming tasks, unlike more specialised languages which eventually restrict their development usage. Too often the programmer has to call procedures written in another language, usually C or Assembler,¹¹ to complete the development.

⁸the word control is given to the gui fields placed by the programmer on a form, these fields have programmable actions which *control* the manner in which the user interacts with that field

⁹however you may discover that many of your favourite features maybe missing, study the Blackbox Oberon Language Report carefully for details

¹⁰Most languages tend to grow in complexity with every release, Delphi, Visual BASIC & Java being prime examples.

¹¹this is not a comment on C or Assembler, just a comment on the chosen development environment

Chapter 2

Blackbox Oberon Standard Modules

2.1 Modules?

A major feature of the Oberon Language design (and therefore Blackbox Oberon) is the MODULE. Modules allow you to program small self contained collections of PROCEDURES, Constants, Type declarations, and Variables and any Component Pascal Code necessary to initialize those variables. Such a MODULE can be separately compiled and called when ever required by the larger task.

At first glance MODULES are similar to SUBROUTINE LIBRARIES,¹ however MODULES are a more secure technique of collection like PROCEDURES into functional groups, eg MODULE StdLog only has PROCEDURES relating to outputing on the Standard Log View of Blackbox Oberon , it does nothing else but those functions.

As you would expect Blackbox Oberon is supplied with a number of useful modules to ease the programming task.

You only need to IMPORT the particular MODULE your current MODULE requires.

eg:

```
MODULE MyTest;
  (* MyTest.Message *)
  IMPORT StdLog;

  PROCEDURE Message*;
  BEGIN
    StdLog.String("A text message appearing on Oberons LOG");
  END Message;
END MyTest.
```

The above module IMPORTS the StdLog module to access and use the preprogrammed PROCEDURES defined there.²

¹a commonly used method of collecting PROCEDURES in C/C++, FORTRAN, Pascal, etc

²in fact StdLog could be the only MODULE you require to product small programs - limited but still useful :-)

2.2 Standard Modules

The table below give the names of the MODULES one needs for *usual* programming tasks. There are many more which a search on the available documentation will show.³

MODULE NAME	Modules General Function
StdLog	Simple Output Procedures for writing to LOG view
Dialog	Interface Procedures for GUI interaction, update GUI form, simple messages, emit Beep sound,etc
Strings	Procedures to manipulate strings (ARRAY OF CHAR), fairly limited set of string functions exist here
SqlDB	The API (application programming interface) for Blackbox Oberon to access SQL databases
SqlControls	The actual PROCEDURES used to access SQL DATABASE TABLES, also need to IMPORT SqlDB as well
Dates	Access System Date & Time information
Math	A collection of useful mathematical functions
Views	A view is a rectangular display object which provides visual presentation of data. A major part of the MVC (Model, View, Controller) concepts underlying Blackbox Oberon . Needed if your program is creating reports, displayed output, etc in a format other than a GUI Form
StdTabViews	For creation & management of Tab (notebook style) views can be programmed but usually developed interactively via dropping a Tab Control onto a GUI Form
StdCmds	Usually used in <i>Menus</i> , but also used to call other GUI Forms as well
TextViews	TextViews are the standard views for text models
TextModels	TextModels are container models which contain sequences of attributed characters and embedded views.
TextControllers	TextControllers are the standard controllers for textviews as defined in TextViews.
TextMappers	TextMappers are mappers that use text riders to scan and format structured text.
TextRulers	TextRulers are text aware views that, if embedded in a text, affect the text setting, especially useful for setting <i>Tabs</i> in report layouts

Figure 2.1: Some useful Standard Oberon Modules

The most complex set of MODULES in the group prefixed *Text* which are used to Read & Write formatted text into Blackbox Oberon Views. They are explained in chapter 4

2.2.1 Finding information about a MODULE

To discover what PROCEDURES exist in a particular MODULE, start Blackbox and left mouse click on *File* then *New*, or *Info* then *Open Log* this will create/open a new Blackbox Oberon

³This is one area that the available information is particularly unhelpful, to my knowledge, no complete summary of MODULES and their usage/relevance exists

view, type the name of the MODULE you wish more information about (eg: StdLog) anywhere in this view , select that text, ie, hold down the left mouse click and move mouse across the word (eg: StdLog) and right mouse click , from pop-up menu left mouse click on "Documentation" to get a full description of the MODULE or left mouse click on "Interface" to get a summary of the available PROCEDURES.

You may also select the MODULE name from a existing line of code on display and display the above information at any time, eg: select StdLog when displaying the MyTest MODULE example above.

Chapter 3

Extra Data Types Explained

When starting to work with Blackbox Oberon I found many simple, but necessary, features were not explained very well or not at all, too often there would be a fleeting mention of a feature in the reference material supplied with Blackbox Oberon , but no actual examples.

3.1 Special Characters

I assume that the reader is familiar with variables, data types, record structures, and the usual fundamentals of programming Pascal style languages, here I include some interesting , sometimes subtle, usage of special characters in Blackbox Oberon . The following characters have their normal meaning when used in their usual context, however, they are also used in special ways.

Namely:

- . (**decimal point**) is normally used in declaring REAL numbers, eg: 32.45, however the decimal point is also used as a *connector*, called 'dot notation', between variable names to qualify exactly the variable being used.

eg:

```
Person : RECORD
  IdNo  : INTEGER;
  Name  : ARRAY 30 OF CHAR;
END;
```

To access the Name variable you program :

```
Person.Name := 'Henry Smith';
```

This dot notation ensures that your program is accessing *Name* in the *Person* RECORD above and not the variable *Name* in another definition. Blackbox Oberon uses this dot notation in linking variables to fields in Forms.

- * (**asterisk**) when placed immediately *after* a variable declaration, that variable becomes accessible globally and may be referenced *and* updated by a referring MODULE or VDU FORM.

eg:

```
Parts* : RECORD
  PartNo      : INTEGER;
  PartsPrice* : Dialog.Currency;
  Description- : ARRAY 40 OF CHAR;
END;
```

In the above example Parts.PartsPrice is accessible, while Parts.PartNo is not.

- (**minus**) when placed immediately *after* a variable declaration, that variable becomes accessible globally and may be referenced *but not updated* by a referring MODULE or VDU FORM

eg: Parts.Description (above)

- \$ (**dollar sign**) when placed immediately *after* an ARRAY OF CHAR variable (not the definition), then only the number of characters up to the 0X char (end of text marker) is copied, printed, etc and not the complete defined field length.

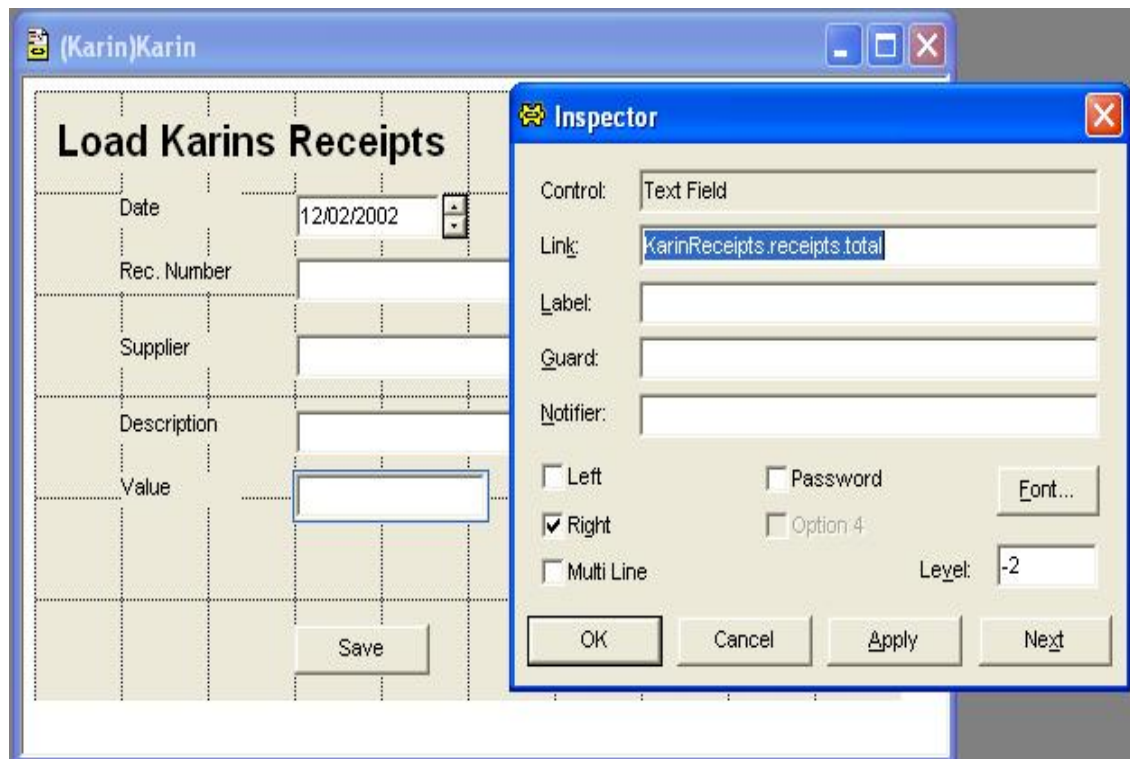
eg: Surname : ARRAY 50 OF CHAR;

```
Surname := 'Hallett';
StdLog.String(Surname$);
```

This will only print the 7 characters 'Hallett' and not the 50 characters actually defined to hold the Surname field.

Misuse, or most often no use, of these characters will result in some confusion by the programmer, especially when trying to display a variables content on GUI Forms. As only (* & -) prefixed variables are accessible to a GUI Form for display, attempting to *Link* non postfixed variables in the controls inspector will simply leave the displayed control uncommitted, ie: its colour will *not be white* on the GUI Form. By default GUI form fields are displayed *grey* before linkage is made.

Also, note that if a variable is in a RECORD structure, then *both* the RECORD variable and the actual variable must be declared with a * if the actual variable is to be visible (accessible).



The above (possibly unreadable!) snapshot shows the Form field "value" being linked to MODULE Karin,s RECORD field "Total". Refer to 9.1 and observe the *receipts :RECORD* field declarations.

3.2 Extra Data Types

I am often amazed at the lack of real world datatypes and appropriate language functions to process them in so called modern languages. Typically Date & Time, Currency¹ are most often ignored, String handling functions are another to miss out also. Blackbox Oberon supplies some data types in addition to the standard ones defined in the Oberon language.

3.2.1 Date and Time

Like many programming languages, the base Oberon Language ignores many useful data types used in normal data processing, eg Date, Time, and Currency types. Blackbox Oberon overcomes this lack to some extent by defining some extra types, not as base types but as extensions to supplied modules, eg: Date and Time in MODULE Dates.

Many programs will need to access the current Date and Time values, for report headings, database logging, etc.

```

MODULE ReportDateTime;
(* ReportDateTime.PrintDateTime *)
IMPORT Dialog, Dates, Strings, StdLog;
PROCEDURE PrintDateTime*;
VAR
    today : Dates.Date;
    now : Dates.Time;
    sdate : ARRAY 50 OF CHAR;
    stime : ARRAY 12 OF CHAR;
BEGIN
    Dates.GetDate( today);
        (* get the systems date into variable today *)
    Dates.DateToString(today, Dates.long, sdate );
        (* convert to string *)
    StdLog.String(sdate$ + " ");
        (* print the date string *)
    Dates.GetTime( now );
        (* get the systems time into variable now *)
    Dates.TimeToString(now, stime );
        (* convert to string *)
    StdLog.String( stime$ );
        (* print the time string *)
END PrintDateTime;
END ReportDateTime.

```

Figure 3.1: Display System Date & Time values

¹COBOL (circa 1960) is the real master at processing currency, its designers realised the importance of MONEY
!!

Notes:

1. The Dates.GetDate and Dates.GetTime system routines return the Operating Systems date and time values into a fields defined in MODULE Dates as :

```
TYPE
Date = RECORD
    year, month, day: INTEGER
END;

Time = RECORD
    hour, minute, second: INTEGER
END;
```

Clearly, such a structure for normal programming purposes is quite difficult to handle, fortunately Blackbox Oberon supplies the necessary conversion tools to operate on these data types in MODULE Dates.

2. The Dates.DateToString(today, Dates.long, sdate); code converts these integer values in 'today' into CHAR variable 'sdate' giving a displayable CHAR form, the actual format is controlled by the second parameter (Dates.long here). See MODULE Dates CONST declarations for other legal values. It is the programmers responsibility to ensure that the receiving variable is defined large enough to accept this conversion.

3.2.2 Currency variables

The base Oberon language ignores many useful data types used in normal data processing, eg Date, Time, and Currency types. Blackbox Oberon overcomes this lack to some extent by defining some extra types, not as base types but as extensions to supplied modules, eg: Currency in MODULE Dialog.

Notes:

1. the data type Currency is defined in MODULE Dialog, thus :

```
Currency = RECORD
    val  : LONGINT;
    scale: INTEGER
END;
```

Clearly, such a structure for normal programming purposes is quite difficult to handle, fortunately once the scale is set you don't need to interact with that variable again and only use the *.val* variable when programming. Currency data is held as a LONGINT data type and therefore processed as FIXED POINT INTEGER values with the decimal place indicated by the scale value.

The advantage of using LONGINT to hold currency values is that there is no loss of accuracy in computations , and unlike REAL , no rounding errors.

LONGINT can hold, -9223372036854775808 .. 9223372036854775807,

a huge range of data values with great accuracy.

If a variable is declared as Dialog.Currency then the programmer must access its contents via the val & scale variables.

In the example below the value of receipts.total is 0 with 2 decimal places of accuracy ie: 0.00. It should also be noted that variable name *receipts.total* should be referred to in access from a GUI Form, it is not necessary to use the full variable name : *receipts.total.val*.

2. it is the .scale value that controls the number of decimal places considered
3. When interfacing Dialog.Currency with the GUI Form and SQL Databases you must consider the preferred SQL structure for holding Currency data ² as there are various ways to define currency fields in database tables.

In MySQL, I use DECIMAL(9,2) as a typical currency field in the MYSQL table definition. Other SQL databases may require another definition of currency fields.

²MySQL is the authors preferred database for use with Blackbox Oberon , although Oberon microsystems recommend Dtf

```
MODULE KarinReceipts;
IMPORT Dialog, Dates;
VAR
    receipts* : RECORD
    recdate* : Dates.Date;
    supplier* : ARRAY 40 OF CHAR;
    goods* : ARRAY 30 OF CHAR;
    total* : Dialog.Currency;
    seq_ : INTEGER;
    recnumber* : ARRAY 20 OF CHAR;
    END;
(* ===== *)

PROCEDURE ClearRec;
BEGIN
    receipts.supplier := " ";
    receipts.goods := " ";
    receipts.total.val := 0; (* clear currency value *)
    receipts.total.scale := 2; (* set decimal place *)
    (* for currency accuracy *)
    receipts.recnumber := " ";

    Dialog.Update(receipts);

END ClearRec;
END KarinReceipts.
```

Figure 3.2: Defining and accessing Currency Variables

3.3 Declaring Arrays

Arrays are a fundamental data structure of much computer processing, Blackbox Oberon offers a very powerful ARRAY type which allows the programmer to define very complex data arrays. In particular Blackbox Oberon allows for the creation of multi-dimensional ARRAY structures which are of great use to mathematical data processing and many mathematical processes require array processing.

With definition of an appropriate RECORD data type very complex data structures may be programmed against with clarity in Blackbox Oberon . If your data structure requires large amounts of data, that might exceed your computer's physical memory, then you might consider using a SQL database to hold the data external to your program and reload/save as necessary when processing.

3.3.1 Example Arrays

Index	Content	Index	Z	A	B	C
0	Fred	0	45.5	12.3	23.44	231.0
1	John	1	56.8	12.3	23.44	231.0
2	Tom	2	47.3	12.3	23.44	231.0
3	Jill	3	87.7	12.3	23.44	231.0
4	Joan	4	23.2	12.3	23.44	231.0
5	Smithy	5	67.0	12.3	23.44	231.0

Figure 3.3: Array A of CHAR

Array Z of REAL

The ARRAY A would be declared : ARRAY 6 , 10 OF CHAR;

Which gives us indexes from 0 to 5 (6 entries !!) containing a CHAR data type of 10 CHARACTERS each.

The ARRAY X would be declared : ARRAY 6, 4 OF REAL;

This ARRAY a multi dimensional ARRAY OF REAL numbers (24 entries). and gives us 4 COLUMNS of 6 REAL NUMBERS in each column.

3.3.2 Declaring an array of basic data type

A basic data type is be considered to be INTEGER, REAL, CHAR , BYTE, etc as defined in Component Pascal Language Report that accompanies your Blackbox Oberon system.

```
MODULE MultiIndexedArrays;
(*  $\Phi$ MultiIndexedArrays.Do*)
IMPORT StdLog;
VAR
    A1 : ARRAY 10 OF ARRAY 10 OF ARRAY 10 OF ARRAY 10 OF INTEGER;
                                     (* defines a 4 dimensioned array of Integers *)
                                     (* A1 : ARRAY 10, 10, 10, 10 OF INTEGER; is also a legal definition *)
PROCEDURE Do*;
BEGIN
    A1[2,2,2,2] := 2 * 2 * 2 * 2;
    StdLog.Int( A1[ 2, 2, 2, 2 ] ); StdLog.Ln;
END Do;
END MultiIndexedArrays.
```

Figure 3.4: Multi indexed array of INTEGER

3.3.3 Declaring an array of RECORD

By using a user defined RECORD data type, very complex structures may be processed by Blackbox Oberon programs. Once defined, the user data type is used exactly as a basic type but with the addition of the RECORD variables name(s) appended to the ARRAY subscripting.

```

MODULE MultiIndexedArrayOfRecords;
(* ΦMultiIndexedArrayOfRecords.Do*)
IMPORT StdLog;

TYPE
  Struc = RECORD
    int1 : INTEGER;
    int2 : INTEGER;
  END;

VAR
  A1   : ARRAY 10 OF ARRAY 10 OF ARRAY 10 OF ARRAY 10 OF Struc;
      (* defines a 4 dimensioned array of RECORD Struc *)

PROCEDURE Do*;
BEGIN

  A1[2,2,2,2].int1 := 2 * 2 * 2 * 2;
  A1[2,2,2,2].int2 := 4 * 4 * 4 * 4;
      (* note how the RECORD fields are accessed via 'dot' notation after subscripts *)

  StdLog.Int( A1[2,2,2,2].int1 ); StdLog.Ln;
  StdLog.Int( A1[2,2,2,2].int2 ); StdLog.Ln;

END Do;

END MultiIndexedArrayOfRecords.

```

Figure 3.5: Multi indexed array Of RECORD

The above program defines a 4 dimension ARRAY *A1* of a user defined RECORD *Struc* which contains 2 variables *int1* & *int2*. The user defined data type can be very complex.

3.3.4 Using a Multi-Dimensioned Array

The example below declares a 2 dimensional ARRAY of INTEGERS with 4 elements, counted from 0 to N-1.³

Notice the use of a CONST to define the ARRAY size, and of LEN to check the array elements limits. Using LEN is the safest way to ensure your code does not fail if you modify the size of your array.

```

MODULE MultiIndexedArrays2;
(* ΦMultiIndexedArrays2.Do*)
IMPORT StdLog;
CONST
    asize = 4;
VAR
    A1   : ARRAY asize, asize OF INTEGER;
                                           (* defines a 2 dimensioned array of Integers *)
    cnt : INTEGER;

PROCEDURE Do*;
VAR
    x1, x2, x3, x4 : INTEGER;
                                           (* indexes for accessing array elements *)

BEGIN
    cnt := 0;
    StdLog.Ln; StdLog.String("#####"); StdLog.Ln;
    FOR x1 := 0 TO LEN( A1 )-1 DO
        FOR x2 := 0 TO LEN( A1)-1 DO
            INC(cnt);
            A1[ x1, x2 ] := cnt;
                                           (* fill array with a computed value *)
            END;
        END;

    FOR x1 := 0 TO LEN( A1 )-1 DO
        FOR x2 := 0 TO LEN( A1 )-1 DO
            StdLog.IntForm(A1[ x1, x2 ], StdLog.decimal, 6, ' ', FALSE); (* print the computed value(s) *)
            END;
            StdLog.Ln;
        END;

    StdLog.String("====="); StdLog.Ln;
END Do;

END MultiIndexedArrays2.

```

Figure 3.6: Using a multi dimensioned array

³I really dislike 0 (zero) based arrays with n-1 limits, it causes unnecessary programming errors if the -1 is forgotten. Counting from 1 to n avoids those problems – its a continuing argument with language developers

Chapter 4

Blackbox Oberon Text handling

4.1 Why more than one type of Text?

Under Blackbox Oberon you are faced with two types of text.

1. text, declared in your code & defined as CHAR or ARRAY OF CHAR.
2. Text, as held in Blackbox Oberon *views* and are processed via special by *Readers & Writers*. They only process *Blackbox Oberon view* held data, not CHAR and ARRAY OF CHAR. These are the PROCEDURES defined in the MODULES prefixed by *Text* listed in the Table of Standard Modules (see 2.2)

The Warford book has a very good explanation of the MVC (Models, View & Controller)¹ Design Pattern. The MVC enables the creation of very complex *documents* containing images, variable text formats, fonts, styles, imbedded views, etc.

For example, when you load a Blackbox Oberon MODULE for editing you normally observe plain text, like a traditional text editor, however every character displayed may have attributes attached to it to describe the way in which that character will be presented to you or to any program processing that text.

Not at all like standard ASCII text files !

4.2 CHAR & ARRAY OF CHAR

These are simple texts² held as separate CHAR or ARRAY OF CHAR variables, defined in your Component Oberon code.

eg:

```
VAR
  description : ARRAY 50 OF CHAR;
prices       : ARRAY 10 OF ARRAY 40 OF CHAR;
(* 10 elements of 40 char *)
yes          : CHAR;
```

¹Models, View & Controller concept as developed by Zerox Corp and which became a fundamental feature of the APPLE MAC Computer Systems and also part of Blackbox Oberon

²Only the first 256 characters of the ASCII standard character set

```
no          : CHAR;  
.. etc ..
```

Values are assigned thus :

```
description := 'A pentium 4 computer';  
yes         := "Y";  
no          := 'N';  
.. etc ..
```

Notice that you may use either double or single quote marks to declare a CHAR value which is very useful because you may use the 'other' quote mark in any text value. You must use the *same* quote mark to start & end the assignment.

```
description := "C++ executes on many computer's";  
.. etc ..
```

ie: you cannot enter :

```
description := 'A pentium 4 computer";  
yes         := 'Y";  
no          := "N';  
.. etc ..
```

4.3 The incompatible Text assignment error

The following small programs show various problems with working with CHAR and ARRAY OF CHAR.

4.3.1 Overcoming the incompatible Text assignment error

```

MODULE BookTestChar1;
(* BookTestChar1.Test *)
IMPORT StdLog, Dialog, Strings;

VAR
    s1, s2: ARRAY 40 OF CHAR;
    s3 : ARRAY 40 OF CHAR;

PROCEDURE Test*;
BEGIN
    s2 := "TestChar1";
    s1 := s2;

    StdLog.String(s1 + " * " + s2 );
    StdLog.String("@ " + s3 + "@");
    s3 := s2; (* incompatible assignment error here *)
END Test;

END BookTestChar1.

```

Figure 4.1: Incompatible text assignment

You might be surprised at receiving an 'incompatible assignment error' when compiling this program. The assignment statement

```
s3 := s2;
```

seems innocent enough, however under the type rules of Blackbox Oberon³ the variables s3 is NOT type compatible with any other CHAR variable in the program, even tho variables s1 & s2 are also defined as ARRAY 40 OF CHAR, they are *different* ARRAY 40 OF CHAR to s3, whereas, S1 & s2 are the *same* ARRAY 40 OF CHAR because they are defined together with the same type declaration.⁴

4.3.2 Declare your own Text data type

By declaring our own data type , called String and using that instead of separate ARRAY 40 OF CHAR declarations, the program is now using compatible ARRAY 40 OF CHAR variable definitions.

³and most of N.Wirths languages

⁴Technically this is correct as ARRAY OF CHAR are referenced by pointers, therefore s1 & s2 refer to the same declaration, however, relaxing the type rules for ARRAY OF CHAR would appear to be a simple and effective change to coding Blackbox Oberon programs, this was done in Borlands Turbo Pascal which greatly enhanced its usage

```

MODULE BookTestChar3;
(* BookTestChar3.Test
Result on Log View :
999
999 * 999@999@
)
IMPORT StdLog, Dialog, Strings;
TYPE
    String = ARRAY 40 OF CHAR;
VAR
    s1, s2: String;
    s3 : ARRAY 10 OF String;
    x : INTEGER;
    res : INTEGER;
PROCEDURE Test*;
BEGIN
    s2 := "999";                                (* note numeric 999 value in CHAR (string) form *)
    s1 := s2$;
                                           (* use Strings Module Procedures to process string variables*)
    Strings.StringToInt (s2,x, res);              (* convert string to Integer *)
    Strings.IntToString (x, s3[5]);              (* convert back to string *)
    s3[2] := 'third row of ARRAY';
    StdLog.Int(x); StdLog.Ln;
    StdLog.String(s1 + " * " + s2 );
    StdLog.String("@ " + s3[5] + "@");
    s3[8] := s2;
END Test;
END BookTestChar3.

```

Figure 4.2: Overcoming incompatible text assignment

There is a data type

String = ARRAY 256 OF CHAR; (* Dialog.String *)

defined in Dialog Module which could be used in the same way , with the advantage of being large enough for most applications needs and as most user written modules will use Dialog anyway its readily available to use.⁵

4.3.3 Use MODULE String procedures

You can overcome many of the above problems by using the procedures defined in Strings Module, esp the Extract and Replace procedures. These procedures will operate on any defined ARRAY nn OF CHAR variables, and can also be used to Copy between variables of different declaration and lengths.

⁵dont confuse ARRAY OF CHAR and ARRAY nn OF CHAR, in most parameter passing in PROCEDURES, see MODULE Dialog PROCEDURES for example, ARRAY OF CHAR is used as a general type for passing CHAR parameters of various lengths, ARRAY nn OF CHAR is a quite specific declaration of a fixed length


```

MODULE BookTestChar4;
(* BookTestChar4.Test
Outputs:
Extract: @A text string * A text string@A text string@
Replace: @A text string * A text string@A text string@
)
IMPORT StdLog, Dialog, Strings;
VAR      s1, s2: ARRAY 25 OF CHAR;
          s3 : ARRAY 10, 40 OF CHAR;

PROCEDURE Test*;
BEGIN
    s2 := "A text string";
    s1 := s2;

    s3[4] := 'data for testing the array';
    s3[6] := s3[4];

    (* copy incompatible text variables *)
    Strings.Extract(s2,0,999,s3[4]);

    StdLog.String("Extract: @" + s1 + " * " + s2 );
    StdLog.String("@ " + s3[4] + "@");          StdLog.Ln;

    Strings.Replace(s3[4],0,999,s2);
    StdLog.String("Replace: @" + s1 + " * " + s2 );
    StdLog.String("@ " + s3[4] + "@");          StdLog.Ln;
END Test;

END BookTestChar4.

```

Figure 4.3: Using Module String to overcome assignment error

There are subtle differences between Strings.Extract & Strings.Replace, ie:

Extract (s: ARRAY OF CHAR; pos, len: INTEGER; OUT res: ARRAY OF CHAR)

Extracts characters from pos, len (MIN(pos+len, Len(s))) characters in *s* and returns it in *res*. The result is truncated if *res* is not large enough. The same actual parameter may be passed for *s* and *res*.

Extract is the *safer* technique to use as a Copy operation, because *res* is given a new value after the extract.

Replace (VAR s: ARRAY OF CHAR; pos, len: INTEGER; IN rep: ARRAY OF CHAR)

Replaces the characters from pos , len (MIN(pos+len, Len(s))) characters in *s* with the string in *rep*. The characters after the replaced range are moved if necessary. The result is truncated if *s* is not large enough.

Note:

If len = 0 then all of rep is inserted in s at position pos.

If LEN(rep\$) = 0 then the characters [pos, MIN(pos+len, LEN(s\$))] are deleted from s.

Replace is *not as safe* as a Copy operation as Extract, because any characters inside s lying outside the range pos,len are moved across as well as the desired character range.

4.4 Blackbox Oberon View Texts

When you write MODULES they are stored as Blackbox Oberon Views, in a folder, called Mod (for MODULE), and although you create the code by using a wordprocessor interface, the text is of a more complex data type than that described on page 29.

It should be understood that Blackbox Oberon text's are not ASCII texts as used in most programming languages, they are a more complex & powerful version of texts, and also more difficult to understand, programming wise. ⁶

A benefit of this complex text data type is that you may write your code in any font, size, color, vertical offsets, etc that suits you, and mix those attributes at will for highlighting code features, ie: all your comments in blue, SQL statements in BOLD. Particularly useful when developing code, you can select a special colour for any new code you are working on.

```
PROCEDURE invDelete*; (* delete current inventory record *)
BEGIN
inv.base.Exec("DELETE FROM inventory WHERE CATEGORY =
:Quikquote.Inventory.CATEGORY AND STK_CODE =
:Quikquote.Inventory.STK_CODE");
    inv.base.Commit();
inv.Exec("SELECT * FROM inventory ORDER BY CATEGORY, STK_CODE");
    Dialog.Update(inv);
END invDelete;
```

If you attempted to compile a syntactally correct program formatted as above with a traditional programming language, eg Pascal, you would be presented with many diagnostics! Why?, well traditionally, programming languages can only process ASCII in its most limited form, and the inclusion of special control characters to make the displayed text **bold** would not be 'understood' by the compiler.⁷

4.4.1 Processing Blackbox Oberon View data

In writing this book I needed to be able to transfer example Blackbox Oberon programs from a Blackbox Oberon Module into a standard ASCII text file to be further processed by the T_EX typesetting system⁸.

Even with all the information available in the supplied documentation, and the free (incomplete) ASCII 'conversion' example Module and Warfords PdoxMappers Module, it was proving difficult to produce a clean ASCII *file* copy of a given Blackbox Oberon Module, after much mucking about however a working program was indeed created. All the example programs displayed their output in a *display view* and did not output the results onto a stand alone file. All the example program code displayed in this book has been processed via this program :

⁶Indeed, *standard ASCII text files* are hardly used in Blackbox Oberon , a bit of a problem when interfacing to other data sources.

⁷actually Blackbox Oberon view texts hold their formatting attributes imbedded in the same file as the view text, so I doubt most programs will be able to process the Blackbox Oberon text. I've tried cutting and pasting from Blackbox Oberon texts to other editors, etc and always have found data errors

⁸Still the best typesetting system available (very biased opinion)

CopyToTeX.⁹ The only difference to *CopyToTeX* is the removal of specialised \TeX command generation, the structure and logic is identical.

```

MODULE CopyToAscii;
(*
Converts a Standard Oberon .odc program module to a ASCII Text File

input file : a Oberon MODULE of type .odc
output file : a ASCII version of type .txt

The output file is a std ASCII file

Brett S Hallett (c) Jan 2002
)

(*  $\Phi$ CopyToAscii.Do *)

IMPORT StdLog, Converters, Files, TextModels, TextMappers, TextViews, Views,
Dates, BookUtils;
(* ----- *)
CONST
    linesperpage = 32;
    PROCEDURE Do*;
VAR
    utmd, inmd: TextModels.Model;          (* utmd = output, inmd = input model *)
    vw: TextViews.View;
    utfm : TextMappers.Formatter;          (* define output formatter *)
    insc : TextModels.Reader;              (* define input scanner *)
    loc: Files.Locator; name: Files.Name; conv: Converters.Converter;
                                          (* parameters for attaching to file(s) *)
    v: Views.View;
    res: INTEGER;                          (* status response variable *)

    linecnt : INTEGER;

BEGIN
    linecnt := 0;

```

Declare the user variables to access the various Text Reders, Formatters & Views required to both read from a Blackbox Oberon document and write the ASCII file out.

```

(* --- attach to input files --- *)
v := Views.Old(Views.ask, loc, name, conv);  (* ask user for input file / document - the view *)

```

The single line above it perhaps the most obscure in the program!, it causes the operating system to display the *standard file select dialog*, you make the appropriate file selection to continue. Normally choose a Blackbox Oberon MODULE .odc file.

⁹Unfortunately, processing that *CopyToTeX* thru itself proved difficult so I decided to show you similar but simpler example program for creating ASCII files from Blackbox Oberon Modules

```

inmd := v(TextViews.View).ThisModel();          (* attach input model to file - the model *)
inmc := inmd.NewReader(NIL);                    (* connect a reader to file view - the reader *)

```

The above code is equivalent to a *File OPEN* command in other languages and makes the internal connection between the Blackbox Oberon Text *Input* Procedures and the *physical file* selected previously.

```

(* — create a output view — *)
utmd := TextModels.dir.New();                    (* create NEW empty output model - the model *)
utfm.ConnectTo(utmd);                            (* connect to the output model - the formatter *)
(* ————— *)

```

The above code is equivalent to a *File OPEN* command in other languages and makse the internal connection between the Blackbox Oberon Text *Output* Procedures and its internal View, note that the physical file has not been selected yet.

```

inmc.Read;                                       (* read first char from input document *)
WHILE ~inmc.eot DO
  IF inmc.char < 100X THEN                       (* only process ASCII chars *)

    IF inmc.char = 0DX THEN                     (* EOL read ? *)
      INC(linecnt);
      utfm.WriteLine;                          (* outputs a Cr/LF on Windows *)
                                              (* its not OK just to copy the EOL char ! *)
    ELSIF inmc.char = 09X THEN (* TAB read ? *)
      utfm.WriteString(" ");
    ELSIF inmc.char < 20X THEN (* ignore all other control chars *) ELSE
                                              (* add other selection ELSIF code here *)
      utfm.WriteChar( inmc.char);              (*copy char to output view *)
    END;
  END;
  inmc.Read;                                    (* read next char from input document *)
END; (* while *)
                                              (* output data via the formatter to the model *)

```

The above code is a simple read characters loop until eot (end of text) is reached and rejecting any characters above the standard ASCII character, processing some of them or simply copying them to the output view.

```

vw := TextViews.dir.New(utmd);          (* attach to model with the written data - the view *)

Views.Register(vw, Views.ask, loc, name, conv, res);
                                         (* ask the user where to save the document *)
END Do;

END CopyToAscii.

```

Figure 4.4: Copy a Oberon Document to ASCII TEXT File

The final two lines make the attachment of the output model to the view for saving by the Views.Register line, this is equivalent to a *File CLOSE* in other languages. The Views.Register will display *the standard operating systems file save dialog* where you can enter a new filename and you **must select a file type of TXT** to finish the conversion process.

This program is reading and writing CHARACTERS and therefore must check *every character* read, and it must be remembered that it simply 'throws away' any characters not in the standard ASCII character set range.¹⁰ This is not the usual way most programmers like to process their data but like to process text files as *lines of text*, not a file of characters.

4.4.2 Processing Text data using TextMappers.Scanner

Blackbox Oberon does not offer a lines of text input/output procedures, but does offer a Text.Scanner facility which allows you to process the input text as *symbols* instead of single characters.

That is as :

char, string, int, real, bool, set, view, tab, line, para, eot, invalid

symbols and automatically skipping white space (*blanks, carriage returns*) until a complete symbol is input, or *eot* reached. By skipping the *carriage returns* any concept of *lines of text* is automatically lost to the processing of the data.

This is a free format style of processing data, it has some advantages, but many disadvantages :

1. not easy to process data arranged as lines of column data, as the end of line (carriage return) is thrown away.
2. if there are a varying number of fields per line then its very easy to lose track of field processing. ie: which field are we processing ?
3. Creating data in freeformat style by user data entry is always prone to data entry errors. Using a Vdu Form for data entry can ensure correct sequence however.

¹⁰primitive but effective!

There is a good example of using this technique in *ObxCount1*, supplied in the Blackbox Oberon examples set and also read the section *after* "Listing 5-25. Code pattern for TextModels.Reader" in the TEXT section of the Blackbox Oberon Help System for a good explanation.¹¹ But be aware that ObxCount1 is designed to read from a selected Blackbox Oberon View, not a operating system file. You have to Open a Blackbox Oberon file in a view, select it (the view) and the execute ObxCount1 and see the results in the Log View.

However, be aware that Scan does not readily process Text files created by other non Blackbox Oberon programs, but is very good for processing Blackbox Oberon document files.¹²

¹¹I would have liked to publish the ObxCount1 example here but Oberon MicroSystems are very reluctant to give permission for anybody to republish their texts. In fact this book is a direct result of their refusal to allow me to modify their SQL help file into a MySQL Help version

¹²I attempted to use Scan, because I did not know any better at the time, on a output ASCII file created by INTERBASE SQL for importing into a MySQL database and ended up writing the conversion program in Paradox, this was before figuring out the CopyToAscii program described in 4.4.1

Chapter 5

Standard Controls

To facilitate the development of GUI (Graphic User Interfaces) Blackbox Oberon offers many *standard controls* which are simply predefined graphic 'pictures' that you can attach your own code, make links to variables defined in your modules, etc.

There are many standard controls available, not all are explained here!

5.1 Radio Buttons

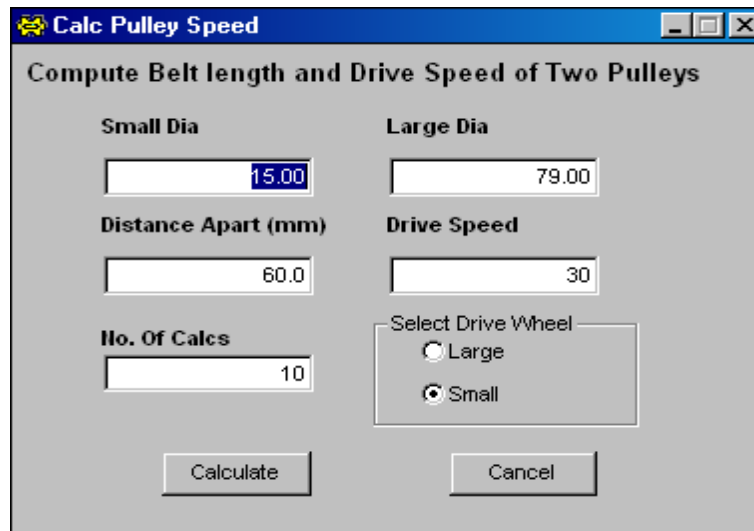


Figure 5.1: Calculate Pulleys Program

Radio Buttons are a very useful control used where you wish the user to make a selection of one, and one only, of the options offered to the user. In the above example the user is allowed to choose either 'Large' or 'Small'. It is not possible to choose both! or more than one option if other buttons were available. Radio Buttons are usually grouped together, and there may be more than one group on a form.

They are usually used where a *fixed, small number of options* are displayed all at once to the user, unlike comboboxes which can display many choices in a drop down list.

The program shown here is a very simple program to compute pulley sizes, belt lengths and final RPM speed of the driven pulley. See section 5.1.1 for example output.

The user can select which pulley (the Large or Small one) that will be considered the driving pulley for the calculations, by clicking on one of the radio buttons, when the Calculate is pressed the program computes and prints results. The report headings are adjusted depending upon the chosen Radio Button.

```

MODULE CalcPulley2;

(*  $\Phi$ "StdCmds.OpenAuxDialog('Calc/Rsrc/CalcPulleys2','Calc Pulley Speed')" *)

(* computes the length of 'O' Ring and speed of final drive of two pulleys *)

    IMPORT StdLog, Math, Dialog, Views, Ports,
    TextModels, TextControllers, TextMappers, TextRulers, TextViews;

CONST large = 0 ; small = 1;                                (* for drive wheel radio buttons *)
VAR

dlg* : RECORD
    smalldia*          : REAL;
    largedia*          : REAL;
    distanceapart*     : REAL;
    drivespeedrpm*     : INTEGER;
    noofcalcs*         : INTEGER;
    drivewheel*        : INTEGER;                                (* for drive wheel radio buttons *)
END;

(* ===== *)

PROCEDURE SetUpTabs( r :TextRulers.Ruler; f: TextMappers.Formatter );
VAR
    x : INTEGER;
BEGIN

FOR x := 1 TO 300 BY 20 DO
    TextRulers.AddTab( r, x * Ports.mm );
END;

TextRulers.SetRight(r, (x + 10) * Ports.mm );

f.WriteView(r);

END SetUpTabs;
(* ===== *)

```

Continues \mapsto

```

PROCEDURE Do*;

VAR
t : TextModels.Model; f : TextMappers.Formatter; v : TextViews.View;
  r : TextRulers.Ruler;
  tmpdia : REAL;
BEGIN
t := TextModels.dir.New();
f.ConnectTo(t);
r := TextRulers.dir.New(NIL);
  SetUpTabs( r,f );
  f.WriteTab;

CASE dlg.drivewheel OF
large : f.WriteString("Large "); f.WriteTab;
        f.WriteString("Small "); f.WriteTab;

| small :      f.WriteString("Small "); f.WriteTab;
              f.WriteString("Large ");      f.WriteTab;
END; (* print pulley sizes headings *)
f.WriteString("Dist. "); f.WriteTab;
f.WriteString("Belt "); f.WriteTab;
f.WriteString("Motor "); f.WriteTab;
f.WriteString("Final "); f.WriteTab;
f.WriteString("Drive");
f.WriteLine;      f.WriteTab;
f.WriteString("Pulley "); f.WriteTab;
f.WriteString("Pulley "); f.WriteTab;
f.WriteString("Apart "); f.WriteTab;
f.WriteString("Length "); f.WriteTab;
f.WriteString("Rpm "); f.WriteTab;
f.WriteString("Rpm"); f.WriteTab;
f.WriteString("Ratio"); f.WriteLine;
tmpdia := dlg.smalldia ;
WHILE ( dlg.smalldia < ( tmpdia + dlg.noofcalcs ) ) DO
  f.WriteTab;
CASE dlg.drivewheel OF
large :      f.WriteRealForm(dlg.largedia, 7, 2, -1, ' ');
              f.WriteTab;
              f.WriteRealForm(dlg.smalldia, 7, 2, -1, ' ');
              f.WriteTab;

| small :
              f.WriteRealForm(dlg.smalldia, 7, 2, -1, ' ');
              f.WriteTab;
              f.WriteRealForm(dlg.largedia, 7, 2, -1, ' ');
              f.WriteTab;

END;

```

(* print pulley sizes *)

Continues ↦

```

f.WriteRealForm(dlg.distanceapart, 7, 2, -1, ' ');
f.WriteTab;
f.WriteRealForm( ( dlg.distanceapart * 2 ) +
( ( dlg.smalldia * Math.Pi() /2 ) +
( dlg.largedia * Math.Pi()/2 ) ) , 7, 2, -1, ' ');
f.WriteTab;                                     (* print belt length *)
f.WriteInt( dlg.drivespeedrpm);
f.WriteTab;
IF dlg.drivewheel = large THEN
    f.WriteRealForm( (dlg.largedia / dlg.smalldia )* dlg.drivespeedrpm, 7, 2, -1, ' ');
ELSE
    f.WriteRealForm( (dlg.smalldia / dlg.largedia )* dlg.drivespeedrpm, 7, 2, -1, ' ');
END;                                           (* print final RPM *)
f.WriteTab;
IF dlg.drivewheel = large THEN
f.WriteString(" 1 : ");
    f.WriteRealForm ( dlg.largedia / dlg.smalldia, 7, 3, -1, ' ');
ELSE
    f.WriteString(" 1 : ");
    f.WriteRealForm ( dlg.smalldia / dlg.largedia, 7, 3, -1, ' ');
END;                                           (* print drive ratio *)

f.WriteLine;
dlg.smalldia := dlg.smalldia + 1;
END;
f.WriteString(" Compute until "); f.WriteInt(dlg.noofcalcs); f.WriteString(" results printed");
f.WriteLine;
f.WriteString(" **** End Report **** ");
v := TextViews.dir.New(t);
Views.OpenAux(v, " Pulley Speed Table ");
dlg.smalldia := tmpdia;                       (* recover original smalldia value *)
Dialog.Update(dlg);
END Do;

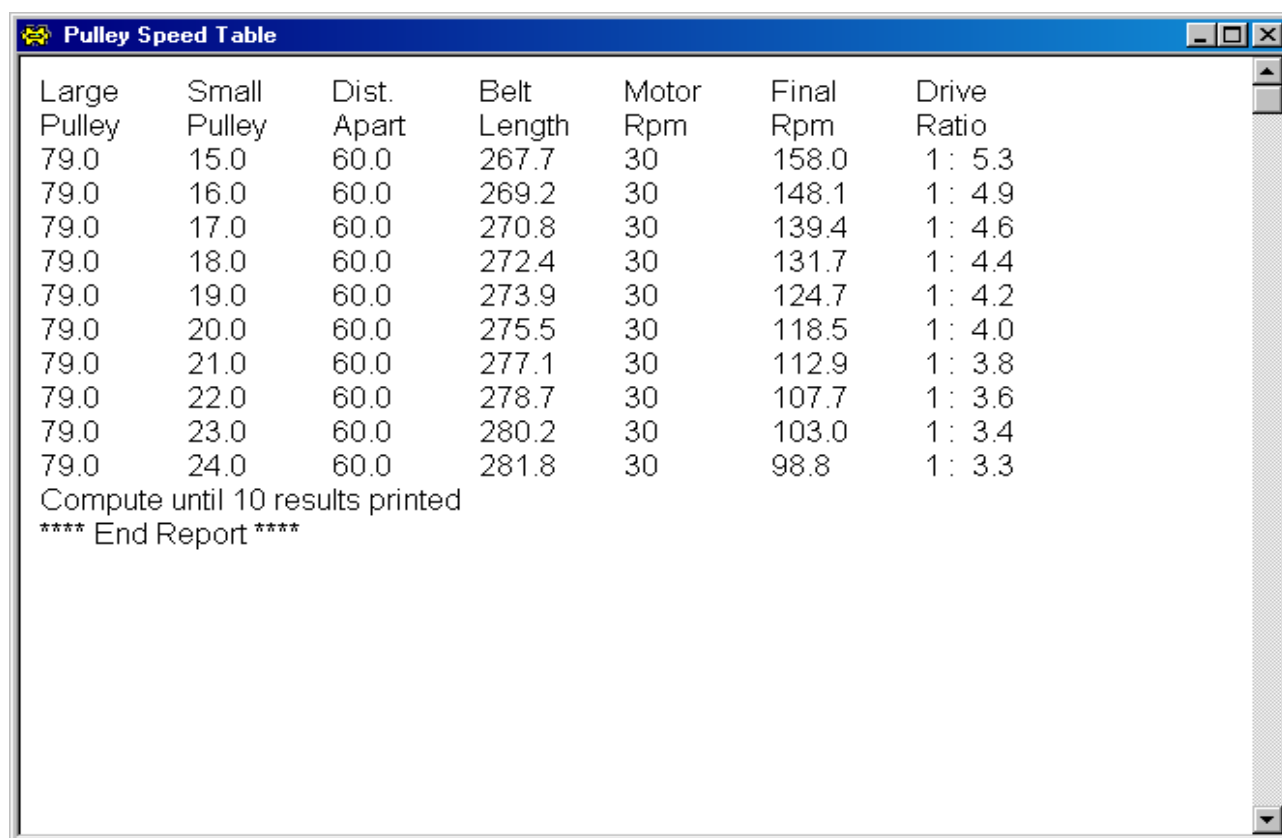
BEGIN
    dlg.smalldia      := 15;                    (* set up default values *)
    dlg.largedia      := 79;
    dlg.distanceapart := 60;
    dlg.drivespeedrpm := 30;                    (* ford XD rear wiper motor rpm *)
    dlg.noofcalcs     := 10;
    Dialog.Update(dlg);

END CalcPulley2.

```

Figure 5.2: CalcPulley Program

5.1.1 Example Output



Large Pulley	Small Pulley	Dist. Apart	Belt Length	Motor Rpm	Final Rpm	Drive Ratio
79.0	15.0	60.0	267.7	30	158.0	1 : 5.3
79.0	16.0	60.0	269.2	30	148.1	1 : 4.9
79.0	17.0	60.0	270.8	30	139.4	1 : 4.6
79.0	18.0	60.0	272.4	30	131.7	1 : 4.4
79.0	19.0	60.0	273.9	30	124.7	1 : 4.2
79.0	20.0	60.0	275.5	30	118.5	1 : 4.0
79.0	21.0	60.0	277.1	30	112.9	1 : 3.8
79.0	22.0	60.0	278.7	30	107.7	1 : 3.6
79.0	23.0	60.0	280.2	30	103.0	1 : 3.4
79.0	24.0	60.0	281.8	30	98.8	1 : 3.3

Compute until 10 results printed
**** End Report ****

Figure 5.3: Example Calculate Pulleys Output

5.1.2 Setting up a Radio Button

Radio Buttons in Oberon are a little awkward to setup! They can return either a Boolean value or a INTEGER value into a module variable defined by the programmer which must be defined globally.

In the example program, observe the INTEGER variable

drivewheel in RECORD dlg

which is declared global (*) and can therefore be referenced by a Oberon FORM control.
The sequence for setup up a Radio Button is :

- Define a global INTEGER variable in your module. Compile the module so that the variable exists for the FORM.
(I assume that you know how to create a FORM)
- Drop a Group Box onto the form
- Drop a Edit Field into the Group Box, link this field to the global INTEGER field eg: `dlg.drivewheel`.
- Delete the Edit Field!
- Drop a Radio Button into the Group Box
 - Select the Radio Button and enter the properties inspector.
 - link this Radio Button to the global INTEGER field eg: `dlg.drivewheel`.
 - enter text into the label field.
 - set LEVEL to the VALUE you wish this Radio Button to return when selected by the user.
 - click on Apply.
 - repeat for every Radio Button required.

When your form is executing, if a user selects a Radio Button its LEVEL VALUE is returned to the global INTEGER field defined, eg `dlg.drivewheel`, and can be tested in IF or CASE statements in your program. See example CASE statement in above program on page 44.

5.2 Check Box

Chapter 6

ComboBoxes

Selection Lists and ComboBoxes are a often used form tool for displaying a list of choices to the user. The standard Blackbox Oberon comboboxes are somewhat limited, but no less useful, than those supplied by languages like Delphi and Visual Basic. Unfortunately the Blackbox Oberon documentation and example programs do not actually explain how a programmer should use them.

Indeed, the example code in ObxDialog does not show you how to access the selected item by the program! Which is of course the main reason for using these controls. A new module, ObxDialogBSH, is a version of ObxDialog that does, and it is instructive to compare the two programs and note the differences.

6.1 Comboboxs Example

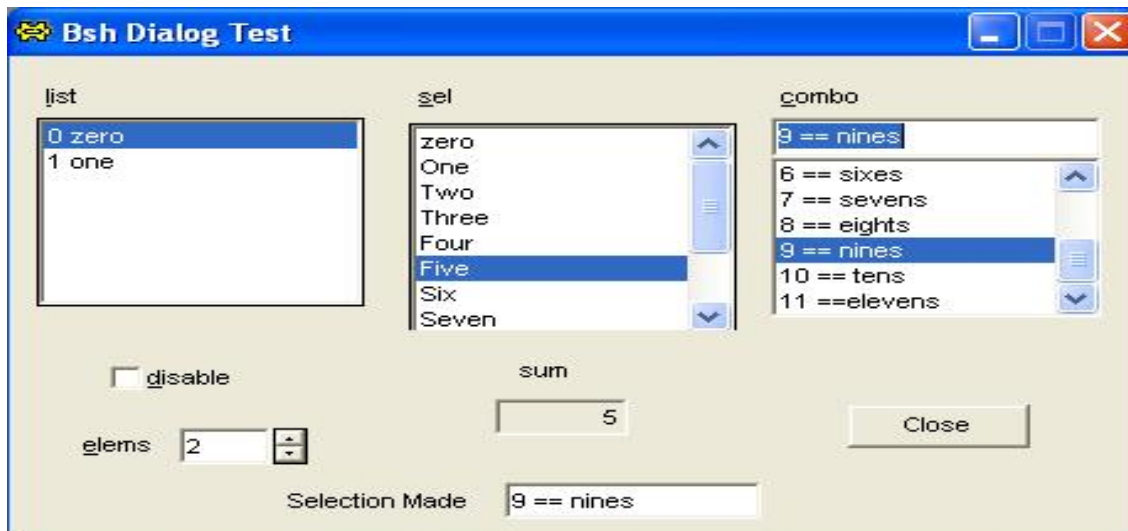


Figure 6.1: Demo Comboboxes

There are a number of aspects of setting up and using these controls:

- You can setup a selectable list either by
 - hard coding entries in program code, or
 - by using the Blackbox Oberon defined file ‘Strings’ and Oberon Command *SetResources* as the data source of entries.
 - read a data file or database and build the list dynamically
- If using SetResource file as the source the displayed entries are read (*once*) from the file upon starting the Oberon Program.
- There are three different types of list/combo controls to choose from :
 - a List Box or
 - a Selection Box or
 - a Combo Box.

They are similar in action but have subtle differences that effect the way in which the user interacts with the displayed data.

6.2 Setting up a list

All the Blackbox Oberon lists (List, Selection, Combo) are special data type of *Dialog* and must be declared before use :

```
MODULE DemoLists;

IMPORT Dialog, StdLog, Strings;

VAR
  dlg*: RECORD
    list*: Dialog.List;
    partslist*: Dialog.Combo;
    sum-: INTEGER;
    disable*: BOOLEAN;
    elems*: INTEGER;
    ans* : Dialog.String;
  END;

  sel*: Dialog.Selection;          (* handled separately to avoid race condition in notifier *)
```

Figure 6.2: Declaring List Access

In the above example there is a declaration for each of the available list types, note the **sel*** declaration for type Dialog.Selection and its attached comment¹

¹A ‘race’ condition is where some resource is in conflict with its usage by another resource depending upon how its called, in the Blackbox Oberon documentation mention is made that the Dialog.Selection’s notifier is

6.3 Using Embedded Code to load List Controls

If you are quite sure of that the list selections are definately not to be changed, then you may decide to place the code to set up the lists in your program. This is useful for mostly small, unchanging lists, such as options lists like colour selection, gender (male, female), etc.

The following example program sets up all its lists using internal procedures with the list elements specifically defined.

```
MODULE ObxDialogBsh;

(*
Modified example code Obx/Mod/Dialog , showing the actual retrieval of selected data from a displayed List, Selection & Combo type.
The original shows the lists working but not how to return the selected data to a program field *)

(* Φ"StdCmds.OpenAuxDialog('Obx/Rsrc/DialogBsh','Bsh Dialog Test')"; *)

IMPORT Dialog, StdLog, Strings;

VAR
  dlg*: RECORD
    list*: Dialog.List;
    combo*: Dialog.Combo;
    sum-: INTEGER;
    disable*: BOOLEAN;
    elems*: INTEGER;
    ans* : Dialog.String;
  END;

  sel*: Dialog.Selection; (* handled separately to avoid race condition in notifier *)
```

Continues ↪

called as often as required to resolve *include/exclude* selections, and presumably this action may cause a race

```
PROCEDURE setupList;
BEGIN

    sel.SetLen(12);
    sel.SetItem(0, "zero 0"); sel.SetItem(1, "one 10"); sel.SetItem(2, "two 2");
    sel.SetItem(3, "three 3"); sel.SetItem(4, "four 4"); sel.SetItem(5, "five 5");
    sel.SetItem(6, "six 6"); sel.SetItem(7, "seven 7"); sel.SetItem(8, "eight 8 ");
    sel.SetItem(9, "nine 9"); sel.SetItem(10, "ten 10"); sel.SetItem(11, "eleven 11");

    END setupList;

PROCEDURE SetList (IN l: Dialog.List; n: INTEGER);
    VAR i: INTEGER;
BEGIN
    l.SetLen(n);
    i := 0;
    WHILE i # n DO
        CASE i OF
            | 0: l.SetItem(i, "0 zero")
            | 1: l.SetItem(i, "1 one")
            | 2: l.SetItem(i, "2 two")
            | 3: l.SetItem(i, "3 three")
            | 4: l.SetItem(i, "4 four")
            | 5: l.SetItem(i, "5 five")
            | 6: l.SetItem(i, "6 six")
            | 7: l.SetItem(i, "7 seven")
            | 8: l.SetItem(i, "8 eight")
            | 9: l.SetItem(i, "9 nine")
            | 10: l.SetItem(i, "10 ten")
            | 11: l.SetItem(i, "11eleven")
        END;
        INC(i)
    END
END SetList;
```

Continues \mapsto

```
PROCEDURE setCombo;  
BEGIN  
  
  dlg.combo.SetLen(12);  
  dlg.combo.SetItem(0, "0 - zero");  
  dlg.combo.SetItem(1, "1 - one");  
  dlg.combo.SetItem(2, "2 - two");  
  dlg.combo.SetItem(3, "3 - three");  
  dlg.combo.SetItem(4, "4 - four");  
  dlg.combo.SetItem(5, "5 - five");  
  dlg.combo.SetItem(6, "6 - six");  
  dlg.combo.SetItem(7, "7 - seven");  
  dlg.combo.SetItem(8, "8 - eight");  
  dlg.combo.SetItem(9, "9 - nine");  
  dlg.combo.SetItem(10, "10 - ten");  
  dlg.combo.SetItem(11, "11 - eleven")  
  
END setCombo;
```

Continues ↪

```
PROCEDURE ListNotifier* (op, from, to: INTEGER);  
BEGIN  
  
  dlg.list.GetItem(dlg.list.index, dlg.ans); (* display selected item *)  
  Dialog.Update(dlg);  
END ListNotifier;
```

Continues ↪

```
PROCEDURE ElemsNotifier* (op, from, to: INTEGER);  
BEGIN  
  IF op = Dialog.changed THEN  
    IF dlg.elems < 0 THEN  
      dlg.elems := 0; Dialog.Update(dlg)  
    ELSIF dlg.elems > 12 THEN  
      dlg.elems := 12; Dialog.Update(dlg)  
    END;  
  
    SetList(dlg.list, dlg.elems); (* rebuild list *)  
    Dialog.UpdateList(dlg.list);  (* update list *)  
  
  END  
END ElemsNotifier;
```

Continues ↪

```

PROCEDURE SelNotifier* (op, from, to: INTEGER);
VAR
    stmp : Dialog.String;
BEGIN

    IF op = Dialog.set THEN    (* recalculate sum of selected numbers *)

        dlg.sum := ( to - from + 1 ) * ( to + from ) DIV 2;

    ELSIF op = Dialog.excluded THEN    (* correct sum after deselection of some numbers *)
        WHILE from <= to DO
            DEC(dlg.sum, from);
            INC(from);
        END
    ELSIF op = Dialog.included THEN    (* correct sum after selection of some numbers *)
        WHILE from <= to DO
            sel.GetItem( from, dlg.ans ); (* display currently selected item *)
            INC(dlg.sum, from);
            INC(from);
        END
    END;

    Dialog.Update(dlg)    (* show new dlg.sum *)
END SelNotifier;

```

Continues \mapsto

```

PROCEDURE ComboNotifier*( op, from, to: INTEGER );
BEGIN
    dlg.ans := dlg.combo.item; (* collect selected combo list item *)

    Dialog.Update(dlg);
END ComboNotifier;

```

Continues \mapsto

```
PROCEDURE ListGuard* (VAR par: Dialog.Par);
BEGIN
    par.disabled := dlg.disable
END ListGuard;

BEGIN

    dlg.elems := 2;
    SetList(dlg.list, dlg.elems);          (* set up list entries (left list) *)

    setupList;                            (* set up 'selection' box list of items ( middle list) *)

    setCombo;                             (* set up 'combo' box ( right hand list)*)

END ObxDialogBsh.
```

Figure 6.3: Load Lists from Imbedded Code

6.4 The Resources File

A resources file is a file defined in the *sub-system* directory *Rsrc* and is always called *Strings*, eg:

Quickquote/Rsrc/Strings

This file is used by the declared Blackbox Oberon sub-system (eg: Quickquote) for resources of a local nature to be used by any program modules in the Quickquote directory, eg special text translations for Buttons displayed texts, error messages, list entries, language translations, etc,

If the list to be displayed is of known entries (values), but may require modification from time to time, or have a large number of entries, then placing them into a resources file is the most appropriate way to maintain the list, as only one line of Blackbox Oberon code for each list is required to load the list. All the list types (list, selection & combo) allow for loading their list from the resource file.

It should be noted that *all* your list controls in a program maybe loaded from the *same* resource file as each list will have its own *key identifier* to separate the list items. You may have as many lists are required in the resource file. There is only one resources file per sub-system.

6.4.1 Resource Key

Entries in the resources file (Strings) are identified by *user defined keys* , the key is matched to that declared in the *SetResources* procedure call and those matching entries are loaded into the appropriate list.

A key definition is : ‘**#Sub_directory colon Key**’

eg: #Quickquote:parts

and used in Blackbox Oberon code like :

dlg.partslist.SetResource(#Quickquote:parts);

The above line of code will search the *Quickquote/Rsrc/Strings* resource file for entries with a key 'parts' and load those entries into dlg.partslist.

In the example code below observe the two PROCEDURES *setupList* & *setupCombo* and note the corresponding key entries in the example resources file on page 61 and find the records (lines) being loaded into their respective lists.

6.5 Using SetResource to load List Controls

```
MODULE ObxDialogBsh;

(*
Modified example code Obx/Mod/Dialog , showing the actual retrieval of selected data from a displayed Lists, Selections & Combo type.
The original shows the lists working but not how to return the selected data to a program field !!! *)

(* Φ"StdCmds.OpenAuxDialog('Obx/Rsrc/DialogBsh','Bsh Dialog Test')"; *)

IMPORT Dialog, StdLog, Strings;

VAR
  dlg*: RECORD
    list*: Dialog.List;
    combo*: Dialog.Combo;
    sum-: INTEGER;
    disable*: BOOLEAN;
    elems*: INTEGER;
    ans* : Dialog.String;
  END;

  sel*: Dialog.Selection; (* handled separately to avoid race condition in notifier *)
```

Continues ↪

```
PROCEDURE setupList;
BEGIN
    sel.SetResources("#Obx:Select"); (* use resources file Obc/Rscr/Srings for data *)

    END setupList;

PROCEDURE SetList (IN l: Dialog.List; n: INTEGER);
    VAR i: INTEGER;
BEGIN
    l.SetLen(n);
    i := 0;
    WHILE i # n DO
        CASE i OF
            | 0: l.SetItem(i, "0 zero")
            | 1: l.SetItem(i, "1 one")
            | 2: l.SetItem(i, "2 two")
            | 3: l.SetItem(i, "3 three")
            | 4: l.SetItem(i, "4 four")
            | 5: l.SetItem(i, "5 five")
            | 6: l.SetItem(i, "6 six")
            | 7: l.SetItem(i, "7 seven")
            | 8: l.SetItem(i, "8 eight")
            | 9: l.SetItem(i, "9 nine")
            | 10: l.SetItem(i, "10 ten")
            | 11: l.SetItem(i, "11eleven")
        END;
        INC(i)
    END
END SetList;
```

Continues ↦

```
PROCEDURE setCombo;
BEGIN

    dlg.combo.SetResources("#Obx:comboentry"); (* use resources file Obc/Rscr/Srings for data *)

    END setCombo;
```

Continues ↦

```
PROCEDURE ListNotifier* (op, from, to: INTEGER);
BEGIN

    dlg.list.GetItem(dlg.list.index, dlg.ans); (* display selected item *)
    Dialog.Update(dlg);
END ListNotifier;
```

Continues ↦

```

PROCEDURE ElemsNotifier* (op, from, to: INTEGER);
BEGIN
  IF op = Dialog.changed THEN
    IF dlg.elems < 0 THEN
      dlg.elems := 0; Dialog.Update(dlg)
    ELSIF dlg.elems > 12 THEN
      dlg.elems := 12; Dialog.Update(dlg)
    END;

    SetList(dlg.list, dlg.elems); (* rebuild list *)
    Dialog.UpdateList(dlg.list);  (* update list *)

  END
END ElemsNotifier;

```

Continues ↦

```

PROCEDURE SelNotifier* (op, from, to: INTEGER);
VAR
  stmp : Dialog.String;
BEGIN
  IF op = Dialog.set THEN  (* recalculate sum of selected numbers *)

    dlg.sum := ( to - from + 1 ) * ( to + from ) DIV 2;

  ELSIF op = Dialog.excluded THEN  (* correct sum after deselection of some numbers *)
    WHILE from <= to DO
      DEC(dlg.sum, from);
      INC(from);
    END
  ELSIF op = Dialog.included THEN  (* correct sum after selection of some numbers *)
    WHILE from <= to DO
      sel.GetItem( from, dlg.ans ); (* display currently selected item *)
      INC(dlg.sum, from);
      INC(from);
    END

  END
END;

Dialog.Update(dlg)  (* show new dlg.sum *)
END SelNotifier;

```

Continues ↦

```
PROCEDURE ComboNotifier*( op, from, to: INTEGER );
BEGIN
    dlg.ans := dlg.combo.item; (* collect selected combo list item *)

    Dialog.Update(dlg);
END ComboNotifier;
```

Continues \mapsto

```
PROCEDURE ListGuard* (VAR par: Dialog.Par);
BEGIN
    par.disabled := dlg.disable
END ListGuard;

BEGIN

    dlg.elems := 2;
    SetList(dlg.list, dlg.elems);          (* set up list entries (left list) *)

    setupList;                             (* set up 'selection' box list of items ( middle list) *)

    setCombo ; (* set up 'combo' box ( right hand list) *)

END ObxDialogBsh.
```

Figure 6.4: Load Lists using SetResources

6.6 An example resources file

STRINGS

Lookup Lookup

ObxConv.ImportTEXT Obx Text File

Off Switch Off

On Switch On

comboentry[0] 0 == zeros

comboentry[1] 1 == ones

comboentry[2] 2 == twos

comboentry[3] 3 == three

comboentry[4] 4 == fours

comboentry[5] 5 == fives

comboentry[6] 6 == sixes

comboentry[7] 7 == sevens

comboentry[8] 8 == eights

comboentry[9] 9 == nines

comboentry[10] 10 == tens

comboentry[11] 11 ==elevens

Select[0] zero

Select[1] One

Select[2] Two

Select[3] Three

Select[4] Four

Select[5] Five

Select[6] Six

Select[7] Seven

Select[8] Eight

Select[9] Nine

Select[10] Ten

Select[11] Eleven

CtrlCol.Prop ObxCtrlCol.InitDialog;

StdCmds.OpenToolDialog('Obx/Rsrc/CtrlCol', 'Properties')

6.7 Using a List, Selection or Combo Box to retrieve information

Having loaded the appropriate data into a List, the List is presented to the end user via a vdu form control, the user may scroll up/down the list using the mouse or typing the first character of a possible entry, eg: type 'e' will scroll the displayed list to the first entry starting with an 'e', a selection is normally made by a mouse click and the program will collect and return the selected value(s) for further processing.

6.7.1 List Boxes

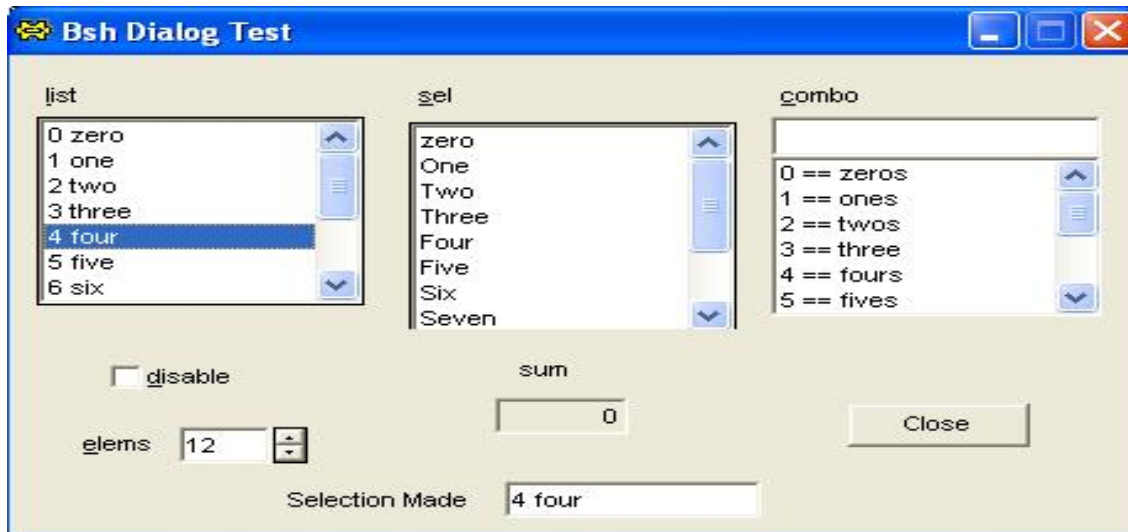


Figure 6.5: Example List Box Usage

List Boxes return a single *index* to the selected item depending upon the selected items position in the list. In the example form above the user has selected entry '4 Four' in the list List, and the program is returned the *index*² value 4, not the selected string value '4 Four'.

Looking at the program code in PROCEDURE listNotifier; on page 58 we see the line of code,

```
dlg.list.GetItem(dlg.list.index, dlg.ans);
```

which uses the *index* set by the user selection, and the Dialog.list objects PROCEDURE Getitem, to place the actual text value '4 Four' into variable dlg.ans, this value is displayed on the form next to 'Selection Made' caption.

²counting from 0 !

6.7.2 Selection Boxes

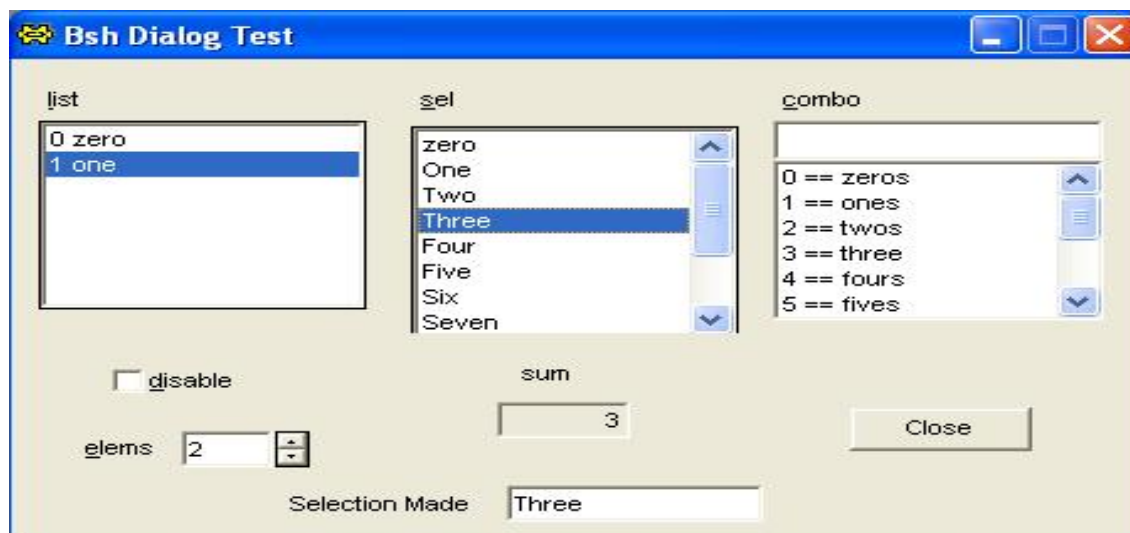


Figure 6.6: Example Selection Box Usage

Selection Boxes return a *set of indices* to all selected items chosen on the displayed list, the use may select multiple choices, and those choices do not have to be consecutive. ie: the user may choose 1,3, 10 thru 15, etc.

6.7.3 Combo Boxes

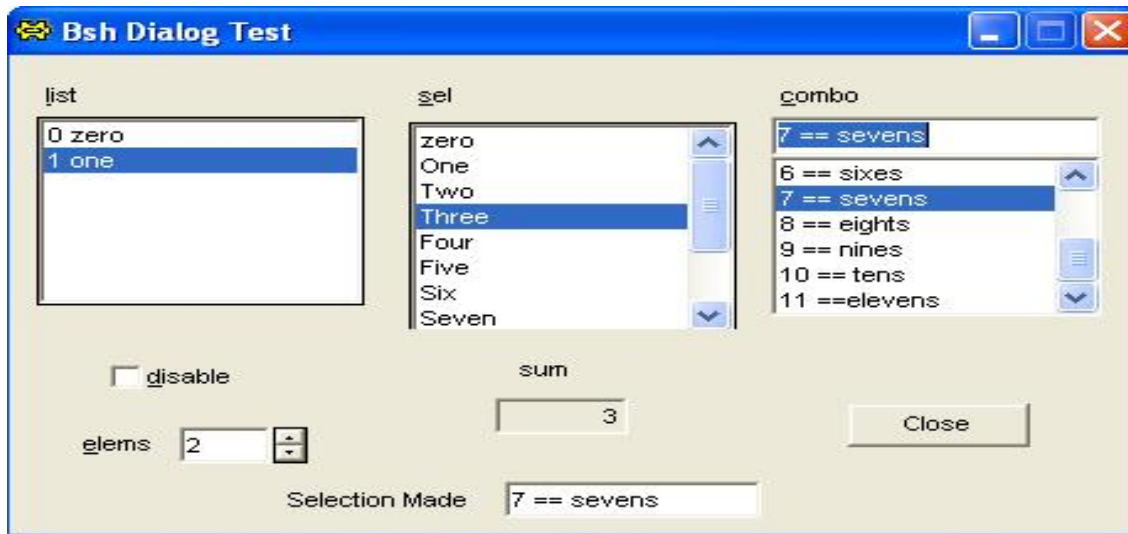


Figure 6.7: Example Combobox Usage

Combo Boxes return the actual (one only) selected string value in a combo box list objects internal variable *item*. Observe the code in PROCEDURE *comboNotifier* on page 59 , and the above example screen where ‘7 == Seven’ has been selected. The following code copies the selected text value into *dlg.ans* for displaying beside caption ‘Selection made’.

```
dlg.ans := dlg.combo.item;
```

to recover the actual selected text into our program for further processing. Notice that we are not told the *index* value of the selected text, although we may use a index to set/get items from the combo list if we wish, refer to Blackbox Oberon documentation on *Dialog.Combo RECORD*.

6.8 Dynamically loading a List, Selection or ComboBox

Often a program will need to display changing lists of information, eg : new part#'s, new books in a library, new accounts, etc, all of which may change in 'real time' as other users access the same information especially in online database situations.

Therefore our programs will need to *update the displayed lists every time a user acesses that list.*

Using this powerful concept is not without its hazards however and care must be taken in the system design to ensure that very large lists are not being re-created continually as the system overheads may eventually cause a major slow down in response times or at worse a total systems crash!

There is little sense in creating lists that contain hundreds of thousands of entries, of parts for example, however lists of categorys of parts may be more useful for an intial access to any part required. A little more work by the programmer should offer a more pleasant end user system.

Note: The following example uses a SQL (MySQL) database as the source of information for building the list. If you are unsure about how Blackbox Oberon interacts with MySQL database(s) then you are advised to read Chapter 7 now, and then return here.³

³I had wanted to place the 'dynamic lists' section in its own chapter after MySQL, but felt it was more useful keeping the use of Lists together.

6.9 Example dropdown list form

The screenshot shows a software window titled "Quotation System" with a menu bar containing: Quotes, Stock, Clients, Load Data, Tax Codes, Translate Table, List Quotes, System, Supplier, and Templates. The "Clients" tab is active. Below the menu bar, there is a search section labeled "Search Clients By:" with a dropdown menu set to "Name" and a text input field containing "Hallett". To the left of the main form area, there is a vertical list of client names: Hallett, Hamilton, Hamilton, Hamilton, Hardiman, Hardinge, Harkin, and Harris. The "Hallett" entry is highlighted. The main form area contains several input fields for client details: Client Name (Hallett), Client Code (2), Surname (Harkin), First Name (Max), Salutation (Mr), Company (Dragon City Bones), Address (P.O. Box 241, Golden Square), State (Vic), Postcode (3555), Tel. No. (54 42 45 47), Fax. No., and Email (dragoncity@origin.net.au). At the bottom of the form, there are buttons for "First", "Previous", "Next", "Last", "Delete", "Update", "New", and "Exit System".

Figure 6.8: Dynamic Reload of Dropdown List

This form shows the user has just selected the client named 'Hallett' from the dropdown LIST, the program has read and displayed the client record for that client.

The List was re-built when the 'Client TAB was clicked upon by the user, this ensures that a fresh version of the current client database is displayed in the dropdown List.

6.10 Re-building a Dynamic list

```
MODULE QuickquoteExtract;
```

```
(*
```

```
NOTE : This is a extract from the program Quickquote and cannot execute !!
```

It is meant to be READ to observe the use of SQL code with LIST boxes, showing the dynamic re-loading of the lookup table displayed by the List box, the List is loaded upon initial startup of the program and every time the Clients TAB is clicked upon to ensure current data is displayed in the List box.

```
Φ"StdCmds.OpenAuxDialog('QuickQuote/Rsrc/QuickQuote','Quick Quote ')"
```

```
QuickQuote : converted from the Delphi / Interbase version into Oberon / MySQL 2001
```

```
June 2002 : added SQL loaded listbox for clients
```

```
)
```

```
IMPORT
```

```
Dialog, Views, Strings, SqlDB, SqlControls, Dates , StdLog, StdTabViews, StdCmds,  
QuickquoteUtils;
```

```
VAR
```

```
clrecno,
```

```
cllastrecno : INTEGER;
```

```
tmpls : ARRAY 10 OF CHAR; (* global conversion fields for SQL row & col display *)
```

```
tmp1, tmp2 : ARRAY 5 OF CHAR;
```

```
template_paras* : RECORD
```

```
code* : ARRAY 11 OF CHAR;
```

```
desc* : ARRAY 51 OF CHAR;
```

```
flag* : BOOLEAN;
```

```
END;
```

```
dlg* : RECORD (* access database variables *)
```

```
id*,
```

```
password*,
```

```
database*,
```

```
driver* : ARRAY 32 OF CHAR;
```

```
END;
```

```
(* ===== *)
```

```
st* : StdTabViews.View; (* TabNotebook anchor *)
```

```
(* ===== SQL Table anchors ===== *)
```

```
cl* : SqlDB.Table; (* client table anchor *)
```

Continues ↪

The above variables are defined for the usage of the Listbox which will contain a list of Client names read dynamically from a MySQL database table.

```
(* ===== local variables ===== *)
total_quote* : RECORD    (* used by SQL SUM *)
    total_quoted_price*  : Dialog.Currency;
    END;
total_buy* : RECORD      (* used by SQL SUM *)
    total_buy_price*    : Dialog.Currency;
    END;
cnt_items* : RECORD      (* used by SQL COUNT *)
    total_items* : INTEGER; (* used to accumulate sub totals on quote(s) form *)
client_reccnt* : RECORD (* used by SQL COUNT *)
    cl_totalrecs* : INTEGER;
    END;          (* current max number of records in client table *)

tvars* : RECORD (* temp variables for client names key manipulation *)
    temp*,
    tempname*,
    tempcode* : Dialog.String;
    END;
```

Continues ↪

```
(* ===== SQL Data Records ===== *)
```

```
Client* : RECORD
  CODE_NAME*      : (* Dialog.String;*) ARRAY 20 OF CHAR; (* key *)
CODE* : ARRAY 10 OF CHAR; (* key *)
FIRST_NAME* : ARRAY 15 OF CHAR;
SURNAME*        : ARRAY 20 OF CHAR;
SALUTATION*     : ARRAY 4 OF CHAR;
COMPANY_NAME*   : ARRAY 30 OF CHAR;
ADDR1*          : ARRAY 30 OF CHAR;
ADDR2*          : ARRAY 30 OF CHAR;
ADDR3*          : ARRAY 30 OF CHAR;
ADDR4*          : ARRAY 30 OF CHAR;
STATE*          : ARRAY 4 OF CHAR;
POSTCODE*       : ARRAY 5 OF CHAR;
TEL_NO*         : ARRAY 15 OF CHAR;
TEL_NO_2*       : ARRAY 15 OF CHAR;
FAX_NO*         : ARRAY 15 OF CHAR;
MOBILE_NO*      : ARRAY 15 OF CHAR;
E_MAIL_ADDR*    : ARRAY 30 OF CHAR;
ADDR2.1*        : ARRAY 30 OF CHAR;
ADDR2.2*        : ARRAY 30 OF CHAR;
ADDR2.3*        : ARRAY 30 OF CHAR;
ADDR2.4*        : ARRAY 30 OF CHAR;
STATE2*         : ARRAY 4 OF CHAR;
POSTCODE2* : ARRAY 5 OF CHAR;
CONTACT_NAME* : ARRAY 40 OF CHAR;
NOTES*         : ARRAY 80 OF CHAR;
ABN* : ARRAY 11 OF CHAR;
CREATE_DATE* : Dates.Date;
  END;

ClientNames* : RECORD  (* List box *)
  nameslist* : Dialog.List;
  END;
```

Continues ↪

```

(* =====
    load list boxes
===== *)
PROCEDURE LoadClientList ( cl : SqlDB.Table );
    (* load listbox for client lookup *)
CONST
    spaces = ' |';
VAR
    reccnt : INTEGER;
    temp,
    tempname,
    tempcode : Dialog.String;
BEGIN
    cl.Exec("Select count(*) from client" );
    cl.Read(0, client_reccnt);      (* get # of records in client table *)

    ClientNames.nameslist.SetLen ( client_reccnt.cl_totalrecs ); (* create a new list *)

    cl.Exec("Select * from client order by CODE_NAME, CODE ");

    FOR reccnt := 0 TO ( client_reccnt.cl_totalrecs - 1) DO
        cl.Read(reccnt , Client);
        temp := Client.CODE_NAME$ ;
        ClientNames.nameslist.SetItem ( reccnt, temp$ );
                                                    (* place in list *)
    END;
    Dialog.ShowStatus( "" );
END LoadClientList;

```

Continues ↦

The PROCEDURE LoadClientList loads the List with the current Client Names held in the MySQL Table Client. This procedure should be called whenever the List needs to be re-freshed, eg, upon program startup, record insertion, deletion. It is up to the programmer to decide when a refresh is appropriate, especially if this may take some time. In this example code the refresh is done only when the user clicks the 'Client' TAB on the TAB Notebook display.

```

(* =====
   List Box Notifiers
   ===== *)

PROCEDURE ClientListNotifier* ( op, from, to : INTEGER );
VAR
  pos : INTEGER;

BEGIN

  ClientNames.nameslist.GetItem( ClientNames.nameslist.index, tvars.temp);  (* collect cho-
sen listbox entry *)
  Dialog.ShowStatus("Client Selected : " + tvars.temp$);

  IF op = Dialog.changed THEN
    cl.Read( ClientNames.nameslist.index , Client);
    Dialog.Update(Client);

    clrecno := ClientNames.nameslist.index;  (* set current recno index *)
  END;

END ClientListNotifier;

```

Continues \mapsto

If the program is expected to process the selected list item, a Notifier must be coded and attached to the ListBox Control (field) on the displayed form. In this example, the program reads and displays (Update) the MySQL database record that has the same 'record number' as the index of the selected List item.


```

(* ===== *)
PROCEDURE NotifierProc* (v : StdTabViews.View; from, to : INTEGER );
BEGIN
(* Standard Tabs processing :
notice that when the user clicks on the Clients TAB the ComboBox List is reloaded to ensure that
current data is displayed to the user for selection ( | 2 : )
)

CASE to OF
| 0 : Dialog.ShowStatus('Quotes ');
| 1 : Dialog.ShowStatus('Stock ');
| 2 :

        LoadClientList( cl );      (* load client lookup list *)
        Dialog.ShowStatus('Clients ');

| 3 : Dialog.ShowStatus('Load Data ');
| 4 : Dialog.ShowStatus('Tax Codes ');
| 5 : Dialog.ShowStatus('Translate Table ');
| 6 : Dialog.ShowStatus('List Quotes ');
| 7 : Dialog.ShowStatus('System ');
| 8 : Dialog.ShowStatus('Supplier');

| 9 : Dialog.ShowStatus('Templates');
ELSE
        StdLog.String('TAB notify CASE failed '); StdLog.Int(to); StdLog.Ln;
END; (* enter TAB switch code here *)

END NotifierProc;

```

Continues ↦

Standard Tabs processing :
notice that when the user clicks on the Clients TAB the List is reloaded to ensure that current data is displayed to the user for selection (| 2 :)

```

(* ===== Client record navigation buttons ===== *)

PROCEDURE clFirst*;
BEGIN
    clReload;
    cl.Read(0, Client);
    Dialog.Update(Client);
    clrecno := 0;

END clFirst;

(* ===== *)

PROCEDURE clLast*;
BEGIN
    cl.Exec("Select count(*) from client order by CODE_NAME, CODE") ;
    cl.Read(0, client_reccnt);      (* get no of records in client table *)
    cllastrecno := client_reccnt.cl_totalrecs -1;
    clrecno := cllastrecno;
                                (* save for table navigation *)
    (* StdLog.Int(clrecno); StdLog.Ln; *)

    clReload;
    cl.Read( cllastrecno, Client);
                                (* read the last record *)
    Dialog.Update(Client);

END clLast;

(* ===== *)

PROCEDURE clNew*;
VAR
    Tstr    : ARRAY 21 OF CHAR;
BEGIN

    ClearClient;

```

Continues ↪

```

    Strings.IntToString( Get_Next_Seq(uniseq, "AUTOINC") , Client.CODE);

    Dialog.Update(Client);
END clNew;

(* ===== *)

PROCEDURE clDelete*;
BEGIN
    cl.base.Exec("DELETE FROM client WHERE :Quickquote.Client.CODE_NAME =
CODE_NAME AND :Quickquote.Client.CODE =CODE ");
    cl.base.Commit();
    clReload;
    cl.Read( clrecno, Client);
    Dialog.Update(Client);
    ShowRecNo;
END clDelete;

(* ===== *)

PROCEDURE clUpdate*;
BEGIN

    cl.Exec("DELETE FROM Client WHERE CODE_NAME = :Quickquote.Client.CODE_NAME
AND CODE = :Quickquote.Client.CODE");
    (* does not matter if DELETE fails, ie record not there to delete !!! *)

    Strings.Replace(Client.CODE_NAME,0,LEN(Client.SURNAME),Client.SURNAME);
    (* StdLog.String(Client.CODE_NAME); StdLog.Ln; *)

    cl.Exec("INSERT INTO Client VALUES (:Quickquote.Client)");
    cl.base.Commit();
    clReload;
    cl.Read( clrecno, Client);

    Dialog.Update(Client);
    ShowRecNo;
END clUpdate;

(* ===== *)

PROCEDURE clPrint*;
BEGIN
    Dialog.Beep;
    Dialog.ShowStatus('Print not implimented yet !');
    Dialog.Beep;
END clPrint;

```

Continues ↪

```

(* ===== *)
PROCEDURE clNext*;
BEGIN
  IF clrecno = cllastrecno THEN
    Dialog.Beep;
    Dialog.ShowStatus('Reached end of table'); Dialog.Beep;
  ELSE
    INC(clrecno);
    cl.Read(clrecno, Client);
    Dialog.Update(Client);
  END;
  ShowRecNo;
END clNext;

(* ===== *)

PROCEDURE clPrevious*;
BEGIN
  IF clrecno > 0 THEN
    DEC(clrecno);
  ELSE
    clrecno := 0;
    Dialog.ShowStatus('Reached beginning of table'); Dialog.Beep;
  END;
  cl.Read(clrecno, Client);
  Dialog.Update(Client);
  ShowRecNo;
END clPrevious;

(* ===== end client navigation buttons ===== *)

```

Continues ↦

The form in section 6.9 has a number of *user navigation buttons* allowing the user to move around the available Client records. It should be understood that MySQL (and most other SQL database systems) does not have the concept of 'next', previous' records ! This is due to the mathematical definition of relational database theory(RDBMS), a subject too complex to explain here. ⁴

In summary, RDBMS's theory allows the database to place stored records in any order and manner deemed applicable by the implimentor, that is, there is NO defined sequence to any records stored!, therefore the concept of 'next', 'previous' record becomes academic, as the 'next' record in the database most likely would not follow any defined ordering method, ie: not alphabetic. However, when a SQL SELECT statement returns records from the query, *those* records are said to form a SET, which maybe ordered by the SELECT statement. These records are accessed by a record number (0,1,2,3,...n) in the Blackbox Oberon SqlTable READ PROCEDURE by the program. Therefore because the SET is ORDERED (SORTED) we can access the 'next', 'previous' records in the SET by controlling the usage of the record number , in the SET.

⁴there are many books explaining this theory, in particular those by C. J. Date are very good

In this example program, notice the usage of the programmer supplied variable *clrecno*, which is used to keep track of which RECORD in the SET the program is currently displaying, and if the user choses a record via the dropdown LIST, then that List entries record *index* becomes the current *clrecno*. Also notice that the PROCEDURES *clnext*, *clprevious*, *clfirst*, *cllast*, etc also update the variable *clrecno*.

```
(* ===== *)  
  
BEGIN  
    dlg.id := "";  
    dlg.password := "";  
    dlg.database := "QuickQuoteMySQL";  
    dlg.driver := "SqlOdbc";  
  
    Dialog.ShowStatus("QuickQuote Ver 1.0 on " + dlg.database);  
    OpenSql;  
  
    LoadClientList( cl );      (* initial load client lookup list *)  
    InitializeDisplay;  
  
END QuickquoteExtract.  
(* ===== *)
```

Figure 6.9: Dynamic Loading and Using ListBox

Chapter 7

The MySQL Database

Blackbox Oberon does not have its own defined a database file system¹, however access to many SQL based database products is possible via the ODBC (Open Database Connection) software supplied with Blackbox Oberon²

The choice of which SQL database system is left to the individual, however reliability should be the main consideration, having a solid reliable product like Blackbox Oberon calling upon a unreliable database will not enhance your final program.

Fortuantely the Blackbox Oberon to SQL interface tools are well founded and much easier to use than typical C code interfaces offered by most other languages wishing to use third party products like databases.³

MySQL is the authors prefered SQL database for usage with Blackbox Oberon .

7.1 Introduction

MySQL is just one of the available SQL database products that may be used with Blackbox Oberon as the 'front-end' user tool. Most SQL databases do not have any knowledge of which computer(s) or computer language is attempting access to the stored data held by the SQL database.

SQL (Structured Query Language) was developed as an interactive tool for more easy access to stored data using the Relational Data Model (RDBMS) of storing data in *Tables*.

MySQL supplies a Command Line Interface to the standard SQL Language Commands, SELECT, UPDATE, etc, for use via a keyboard or using *scripts* held on files and user interfaced from a DOS session when under a Windows Operating System (all versions).⁴ And a API (application programming interface) for use by programming languages.

SQL's design criteria was, and is, for the *Query* part of the operation, pre & post processing of the stored data is left to other programming tools, ie: C/C++, Delphi, Oberon, PL/1, Crystal Reports or whatever.

Over time, SQL based database RDBMS's have evolved to the Client/Server model of storing and access data, ie: a SQL Server maintains all the Tables and data stored therein on a seperate Server Computer, the end-user program wishing to access that data is a Client of that SQL

¹one of the few times you will need to go outside the Blackbox Oberon environment

²at least up to Blackbox Oberon version 1.4

³most languages offer obscure parameter driven interface record structures to make the connection, forcing the programmer to conform to C data type standards, which may entail 'bit' fiddling code to conform

⁴Gui interfaces were not usual when SQL was first designed by IBM in the 1970's

Server. It does not matter what programming language the Client program is written in as the SQL Server will return results of a Query in a SQL standard manner for processing by the Client Program.

You will need to obtain and install your own copy of MySQL to make use of the following information, MySQL is not supplied by Oberon microsystems. Your programs will use the SqlOdbc driver supplied with Blackbox to access MySQL.

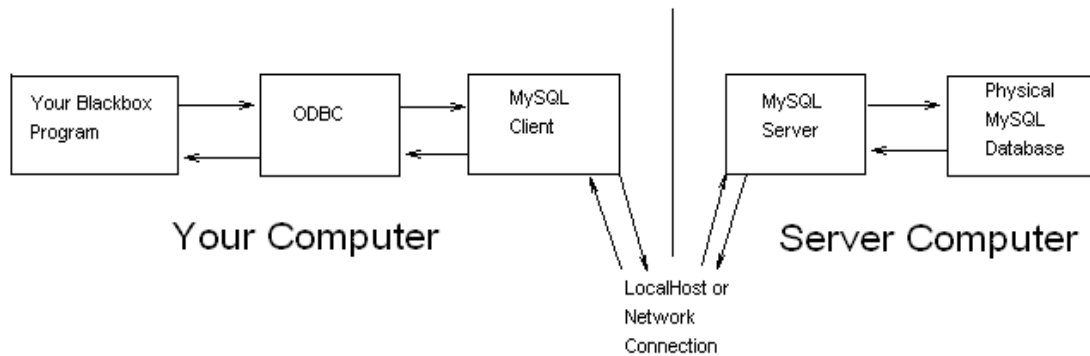


Figure 7.1: Oberon to Mysql database via ODBC

7.2 Blackbox Oberon MySQL general features

It should be kept in mind that SQL is a quite independent program in its own right, and we are using its facilities to return data from its database into Blackbox Oberon for processing or display to a end-user. The Blackbox Oberon to SQL interface has been developed to make this as effective as possible.

Most languages require complicated interfaces to external software not written in the host language, the Blackbox Oberon SQL interface, however, makes good use of modules to overcome this. The central programming interface is Module *SqlDB* which only exports two major and a few minor data types. The major datatypes are *SqlDb.Database* and *SqlDB.Table* which supply the access abstractions for SQL databases and SQL results tables. Data from the results tables can then be read into Component Pascal RECORDS which represent a single (one) row of a SQL Table.

7.2.1 Full access to SQL

Via your Blackbox Oberon program you can have full access to all the features of the SQL language, there is no attempt to define SQL behind special database-aware controls pretending that SQL is a part of the Blackbox Oberon language.

This has both advantages and disadvantages, in particular, the *programmer* must check the syntax of any imbedded MySQL statements to be processed, else unusual results may occur. This

is because the MySQL statements declared inside strings from passing to MySQL for execution. Blackbox Oberon does not attempt to check for syntax or symantic errors in these strings.

7.2.2 Integration of MySQL with Blackbox Oberon

As SQL is a query language, we often need the flexibility to construct SQL statements at run-time. ie: we cannot know in advance exactly what query a end-user may wish to process, on the other hand, we should not expect end-users to understand the SQL command language, hence the use of GUI interface tools like Blackbox Oberon . SQL commands and statements are sent to the SQL Server in plain text.⁵ Blackbox Oberon has a special built-in metaprogramming facilitie which allows direct usage of user defined Component Pascal variables *within* SQL statements.⁶

This feature make for a much easier programming of SQL queries, as hopefully we will see later in this chapter.⁷

7.2.3 Extensibility

We will access MySQL via the ODBC (Open DataBase Connectivity) facility, which offers our Blackbox Oberon programs a degree of independence from the actual version of MySQL we are using. If we install a newer version, our Blackbox Oberon programs will still run as only the ODBC 'connection' will be changed, our programs remain the same, without the need to recompile. In fact, it is possible to use several different ODBC drivers to different databases in the same application concurrently. ie: use MySQL, ACCESS and MS-SQL Server at the same time from the one Blackbox Oberon program via ODBC connections.

7.2.4 Separation of program logic and user interface

A major design feature of Blackbox Oberon is the separation of the user interface (usually the GUI Form) and the actual program code executed by user actions using that form, eg Press a Button. The GUI form has no information about the actions being executed by the button. Therefore the GUI form maybe modified, eg fields moved, without requiring changes to the called code. A Blackbox Oberon program accesses and updates database contents via so-called interactor objects, ie: Component Pascal records. These record variables are used as sources and destinations of MySQL command parameters or result data from MySQL queries. ie: MySQL reads and writes directly into Blackbox Oberon record variables and does any data conversion from SQL as necessary.

7.3 Acquiring MySQL

Versions of MySQL are available for many different Operating Systems, from www.mysql.com. Please observe the liciencing agreement.

There are also many addon utilities, esp. API (Application Programming Interfaces) for numerous programming languages.

⁵They might not be actually , but when you read your SQL statements in Blackbox Oberon code, that's how you will see them.

⁶Almost all other programming languages set up text (Character strings) parameter passing areas to interface with SQL, or C compliant STRUC definitions, a sometimes tedious and error ridden task to code.

⁷We dont need to understand how its done, just *believe* it is done

The SqlOdbc (Open Data Base Connection) supplied with Blackbox is used.

7.4 The MySQL Book

Although the Mysql distribution contains comprehensive documentation and a excellent reference manual, it is recommended that you aquire a copy of the book: "MySQL" by Paul DuBois ⁸
Published by New Riders www.newriders.com
ISBN 0-7357-0921-1

This will save you considerable printing time and paper!. There is also a very informative Appendix A on obtaining and installing the MySQL Software, although the supplied installation information on the distribution files should be considered the latest information.

7.5 Installing and running MySQL

Following the installation documentation for the selected Operating System and MySQL version, you will eventually have a *server task*⁹ (eg Mysqld_nt) running which is normally started up at system boot, also several client tasks are installed at this time. Your Blackbox programs become clients to this server once they have opened a database for access. One of these supplied client tasks is called *mysql*, and is used as a interactive tool to create databases, tables, queries, etc, in command line mode. ie no GUI interface. Most GUI interfaces to SQL databases are done via third party languages like Blackbox, Delphi, Visual Basic, Perl, C/C++, etc. It is unusual for a SQL database to have a native GUI interface tool.

This server task (mysqld on a windows NT machine) is the interface program task between the user program and the ODBC drivers. Blackbox's SqlOdbc & SqlDB modules when called, translate your SQL statements into appropriate ODBC functions, which in turn setup and make system calls to mysqld to access the database files (tables), data is returned to the user program via other SQL statements (see later example code), the mysqld server then 'sleeps' until another user program issues a SQL statement.

The mysql server task is left running even after your program terminates, as all other MySQL interaction will be via this server task. Indeed, once loaded, the mysql server usually stays active until system power off.

7.5.1 ODBC & MySQL

Note that Blackbox's ODBC driver is called SqlOdbc, *the spelling is important*, but the generic word ODBC is mostly used in this text. The word SqlOdbc is only used in the Blackbox Open-Database procedure, as will be shown in the example Blackbox code.

To interface to a Mysql database an ODBC entry must be made by the ODBC administrator program to install an 'alias'¹⁰ so that a connection between MySQL and your Blackbox program via a ODBC entry can be made at your program execution.

1. Install MySQL (see mysql book or reference manual)

⁸a really excellent book indeed, which also contains very useful examples

⁹This server task maybe installed on your computer or a separate computer via a network

¹⁰the use of alais's is useful, it allows us to change our databases name, location, etc, and not have to change our program as long as the alias name remains the same.

2. Install MySQLODBC program.(see mysql book or reference manual)
3. Start the ODBC administrator (assume NT 32 bit version here)
 - (a) Under tab "User DSN", press "Add"
 - (b) choose the Mysql line from the displayed database drivers available.
 - (c) press "Finish"
 - (d) a Mysql configure dialog box should appear (you can ignore most of it !), enter,
 - i. Windows DSN name testmysql
(your database alias)
 - ii. Mysql Host (name or IP) : localhost
(localhost, if not a networked database)
 - iii. Mysql database name : testdb
(the actual database name, and Mysql Sub-Directory name)
 - iv. press OK
4. You should now see an entry for the just created "testmysql" alias in the available ODBC connections.
5. press OK to exit.

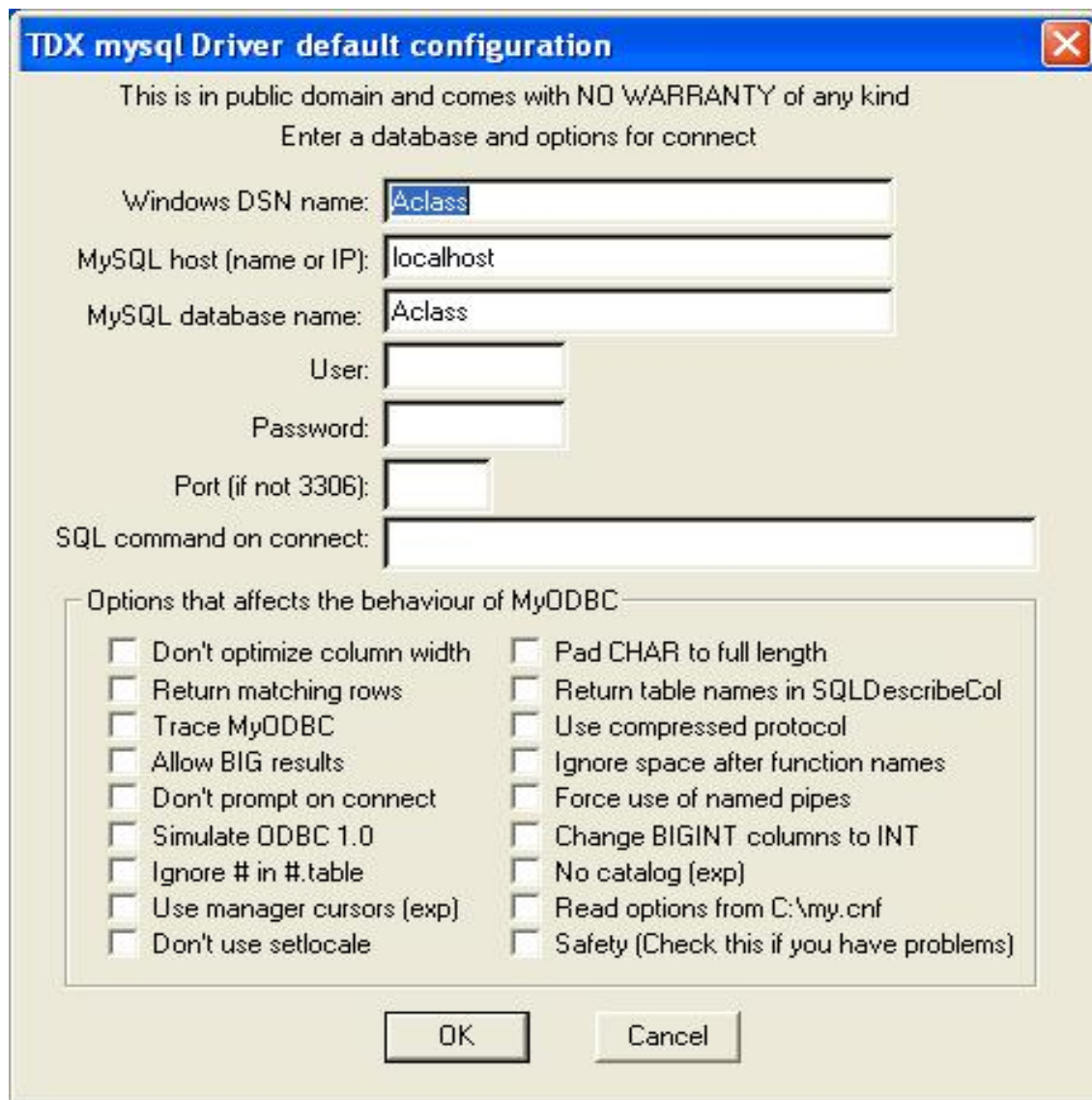


Figure 7.2: Example Mysql ODBC alias setup

The Windows DSN and Mysql Database Name fields do *not* have to be the same name. They both happen to be 'Aclass' in the example above.

Apart from 'localhost' the values for Windows DSN name and Mysql database name above are your entries which allow the ODBC driver to connect to the real database. Your Blackbox program will always refer to the Windows DSN name as its database name in the OpenDatabase call (see later), the ODBC driver will pass messages to the MySQL drivers make the actual connection to the 'real' database , called 'Aclass' here.

Your BlackBox program does not need to know the actual physical location of the database, or the value in MySQL Database Name, and depending upon the MySQL Host Name entrie the

database maybe located on your computer (localhost) or a LAN or WAN, or Internet, etc. if a IP address is entered by you. Whenever Blackbox refers to a 'database' you should use the Windows DSN Name, not the actual database name (Mysql Database Name).

Should the physical location of the database change, no changes are required in your Blackbox program as long as the alias remains unchanged.

The use of the alias is not restricted to usage by your Blackbox program, other development tools, Reports Writers (such as Crystal Reports) , Database Browsers, and other programming languages can make use of the alias to access the same MySQL database at the same time.

You will find a powerful SQL Browser for Windows written in Delphi via the MySQL web site, a very useful database development tool to be used in conjunction with MySQL's command line interface tools.

It is useful to note that you may have both your Blackbox Oberon session running and a MySQL session running (in its own DOS session) at the same time when developing you database. You can develop your SQL commands direct to MySQL and get immediate results, or error reports, rather than have to write special Oberon code to decode MySQL results returned to your program.¹¹

7.5.2 Create an MySQL database

Before any Blackbox program can make use of the just created ODBC alias (testmysql), the real database (testdb) must exist.

Assuming that Mysql has been sucessfully installed, enter the operating systems command line interface, a DOS session under windows, and start up the Mysql client : mysql.

eg while in a DOS Window :

```
C:>mysql
Welcome to the Mysql monitor.
```

Type 'help' for help.

```
mysql>CREATE DATABASE testdb; <press enter>
mysql>exit <press enter>
C:>
```

This will create a database called *testdb* in the mysql data files area which was created when mysql was installed. Neither your Blackbox program or any of the client tools need to know this physical location for any MySQL usage, as they will access this database via the Windows DNS value set up in your ODBC.

7.5.3 Creating MySQL Tables

By itself a SQL database actually holds no data, it is the tables within the database that contain the users data. After creating a database you must use other MySQL commands to create those tables your programming task require. Tables can be quite complex data structures but in general

¹¹this technique is especially useful when developing SELECT statements as any SQL syntax errors are reported directly to you in plain text, also you can observe if the SELECT is returning the data you expect, once you are satisfied, then you place that SQL statement in your Oberon Program

the simpler the design the easier the database will be to program against with your Blackbox code.

There are basically three ways to create tables:

1. interactively by using mysql client.
2. by a batch file.
3. by a Blackbox (or other language) program.

While the third option has some attractions , it suffers from the chicken & egg problem, you need to know how to program MySQL with Blackbox to do it (there is an example of this in Blackbox SQL examples).

The first option is quite a good method, while learning to do simple tables setups, however if you need to modify your table structures later you will have to re-enter all your instructions exactly again.

The second option is the most useful technique as it does not suffer the problems of 1) & 3) , you only need to know enough SQL command language to setup your tables (no Blackbox coding required) and you can re-run the batch file, also called a script file, as often as needed. The script may create as many tables as necessary by one simple run of the script. The main advantage of using a script is that you will not forget some small but important requirement when developing your complete database.

InteractiveMySQL (DOS session) under Windows NT

```
c:>mysql
mysql>CREATE DATABASE testdb;
mysql>CREATE TABLE Companies
(id INTEGER,
name CHAR(255)
ceo CHAR(255)
employees INTEGER);

mysql>CREATE TABLE Ownership
(owner INTEGER,
owned INTEGER,
percent INTEGER);
mysql>exit
c:>
```

This will create two tables, *Companies* & *Ownership* under the database name *testdb*.

Table *Companies* has the fields : id, name, ceo, employees.

Table *Ownership* has the fields : owner, owned, percent.

The fields have MySQL defined data types , INTEGER and CHAR. These are not Blackbox data types, however SqlOdbc will convert the datatypes as necessary during input/output processes.

7.5.4 MySQL Script file to create the tables.

The following MySQL commands are entered into a text file, usually suffixed as .sql, using a text editor, not a word processor, and then executed as if you had typed each line into a interactive

session. It is advisable not use a word processor, eg WORD, WordPRO, etc to create a script file as the script must not contain any special control characters, eg underline, bold, etc in the file, just straight Ascii text is allowed.

eg contents of file : setup_aiclass.sql

```
drop table if exists Aclass;
create table Aclass (
    yacht_name          char(33) not null,
    rego                char(5)  not null,
    version             INTEGER  not null,
    loa                 REAL,
    ohang_fore          REAL,
    ohang_aft           REAL,
    lwl                 REAL,
    qbl_port            REAL,
    qbl_stb             REAL,
    weight              REAL,
    port_freeboard_fore REAL,
    port_freeboard_mid  REAL,
    port_freeboard_aft  REAL,
    stb_freeboard_fore  REAL,
    stb_freeboard_mid   REAL,
    stb_freeboard_aft   REAL,
    draught_measured    REAL,
    mast_H              REAL,
    jib_mast_I          REAL,
    main_luff_A          REAL,
    main_foot_B          REAL,
    jib_foot_J          REAL,
    mast_foot_K          REAL,
    sail_additional_area REAL,
    primary key ( yacht_name , rego , version ) );
```

to use this file:

```
c:>mysql Aclassdb < setup_aiclass.sql
```

if sucessfully executed, this script will create a table called Aclass in a database named Aclassdb. Note the use of the DROP TABLE IF EXISTS statement , this ensures that if the table already exists, from a prior run of the script, or manually created for example, then the script will not abort because you are trying to create an already existing table. This is a MySQL feature and may not be available in other SQL implimentations

7.6 Databases, tables, and interactors

In a Blackbox Oberon environment a database is represented by a SqlDB.Database object. As long as this object exists, MySQL commands can be executed, the actual database may be local, remote or on a server.

7.6.1 Databases verses Tables verses Fields.

In a typical MySQL environment the user can consider the MySQL *system* to be a database *and* the tables in that database. Within the table definitions are the actual fields one wishes to program against.

ie: a MySQL Database has Table(s) which have Field(s)

Within a Database every Table must have a unique name, within a table each fieldname must also be unique, however the same fieldname can be used in any of the tables. Your program must qualify which tablename + fieldname is being referred to in your code to avoid updating the incorrect field

ie : Tablename.fieldname (* note the 'decimal point' connector *)

The SQL language is quite independent of Oberon and was designed as a Query Language, and thus has no knowledge of GUI's, Report Writers, User Programs, etc that may wish to use data held in the database. Most SQL implementations offer various tools for the user to create databases & tables, interact with those tables ie: make a query, enter data or report against those tables.

At first sight SQL appears to be a very simple language, however, if the database design contains many tables with many relationships between them then expecting the end user to enter actual SQL statements on order to extract information will lead to great user dis-satisfaction with your program !.

Oberon is used as a front-end to SQL in such a way that the user is usually offered a GUI Form to interface to the underlying SQL database and this removes the need for the end-user to write SQL programs, your Oberon program will collect information from the user, create or use an appropriate SQL command, execute the SQL command, collect the results and display back to the user.

There are two fundamental SQL statement types, those that do not return a result table and those that do.

Not Returning a Results Table	Returning a Results Table
ALTER	SELECT
CREATE	SHOW
DELETE	DESCRIBE
DROP	EXPLAIN
FLUSH	
GRANT	
INSERT	
KILL	
LOAD DATA	
LOCK TABLES	
REPLACE	
REVOKE	
SET	
UNLOCK	
UPDATE	

Figure 7.3: Summary of MySQL Statements (commands)

The not returning results SQL commands will execute the required task and may return a execution status, but no table (or "answer") result is available for further processing.

The returning results SQL commands (especially a SELECT statement) will return a table of results to Oberon from the SQL for further processing. Note that a query may return a *empty* result table if the query failed to find asked for data,

eg:

```
SELECT * FROM COMPANIES WHERE ID = "IBM",
```

and column ID does not have "IBM" in your table COMPANIES. This is not a failure of the SELECT statement but a "failure" of your question (query) , ie: the answer is "no entries for ID = "IBM", therefore there are no rows to return to your program.

7.6.2 Database

The following Blackbox Oberon declaration describes the SqlDB.Database type:

```
Database = POINTER TO ABSTRACT RECORD
res: INTEGER;
async: BOOLEAN;
showErrors: BOOLEAN;
(d: Database) Exec (statement: ARRAY OF CHAR), NEW, ABSTRACT;
(d: Database) Commit, NEW, ABSTRACT;
(d: Database) Abort, NEW, ABSTRACT;
(d: Database) Call (command: Command; par: ANYPTR), NEW, ABSTRACT;
(d: Database) NewTable (): Table, NEW, ABSTRACT
END;
```

Procedure Exec is used to execute SQL statements which don't return result tables; e.g., DELETE or INSERT statements, such as:

```
database.Exec("DELETE FROM Companies WHERE id = 5" )
```

Transactions are started automatically when the first modifying SQL command is executed. A transaction is terminated either by calling Commit or Abort.

Warning: don't use a database's SQL transaction statements, since they may interfere with Commit and Abort. Mysql versions to and including 3.23.43 do not support transaction statements, later versions may.

A database object is obtained by calling SqlDB.OpenDatabase:

```
PROCEDURE OpenDatabase (protocol, id, password,
                        datasource: ARRAY OF CHAR;
                        async: BOOLEAN;
                        OUT d: SqlDB.Database;
                        OUT res: INTEGER);
```

This procedure opens the database given by the pair (protocol, datasource). where the ODBC Alias (Odbc DSN name) and MySQL Driver name are defined. ("SqlOdbc") If that database is already open from a previous call to OpenDatabase, the same database connection may be used, without considering the id and password information again.

See the example program 'karin' on page 134 for details of using the OpenDatabase procedure.

If an application returns result tables from a database, typically generated by SELECT statements, the program needs to also provide *table objects*, which represent the returned result tables.

The contents of a results table is static, ie: it represents a snapshot of the database contents of the MySQL query at the time the statement is executed. A results table can be considered as a local and independent copy of the database contents.

Several tables can be used simultaneously.

7.6.3 Database Status Response

A status response maybe available to test for correct execution of the command.

eg:

```
.....
  SqlDB.OpenDatabase(driver, id, password, database,
  SqlDB.async, SqlDB.showErrors. db, res);
  IF res = 0 THEN
    table := db.NewTable()
    (* <<<<<<< create a table object <<<<<< *)
  ELSIF res <= 3 THEN
    Dialog.ShowMsg("#Sql:CannotLoadDriver")
  ELSE
    Dialog.ShowMsg("#Sql:ConnectionFailed")
  END
.....
```

The SqlDB.OpenDatabase returns a status response, in res, so that the programmer can check for a successful execution.

7.6.4 Table Object

A table object is obtained by calling its database object's NewTable procedure.

An SqlDB.Table is declared as

```
Table = POINTER TO ABSTRACT RECORD
  base-: Database;
  rows, columns, res: INTEGER;
  strictNotify: BOOLEAN;
  (t: Table) Exec (statement: ARRAY OF CHAR), NEW, ABSTRACT;
  (t: Table) Available (): BOOLEAN, NEW, ABSTRACT;
  (t: Table) Read (row: INTEGER; VAR data: ANYREC), NEW, ABSTRACT;
  (t: Table) Clear, NEW, ABSTRACT;
  (t: Table) Call (command: TableCommand; par: ANYPTR), NEW, ABSTRACT
END;
```

The variables (rows, columns) denotes the number of rows and columns of the most recently returned result table.

The *Table.Exec* can only be used with results returning MySQL statements, usually SELECT, but refer to page 7.6.1. This contrasts with Database.Exec which cannot return a results table.

table.Exec("SELECT * FROM Companies WHERE id = 17")

7.6.5 Read a table row

Read can be used to read a row from the result table into an interactor (i.e. an exported record variable). The interactor can then be manipulated by the Blackbox Oberon program or by its graphical user interface elements. For example, the following statement

```
table.Read(22, company);
```

reads the contents of row 22¹² of table into the variable company, which might be declared as

```
VAR
company*: RECORD
    id*: INTEGER;
    name*, ceo*: ARRAY 32 OF CHAR;
    employees*: INTEGER
END;
```

Record fields and rows of a result table are matched in the order that they are defined in the record or in the database, respectively (MySQL doesn't define an order, but every actual database product does). Record fields to be matched must be exported. If there are non-exported record fields, they are simply ignored.

¹²Unfortunately, Rows are counted from 0, so row 22 is actually row 23

7.6.6 Checking if a SQL statement executed successfully

To add to the programmers confusion, most SQL implementations do not offer any status response to successful execution. However, Oberon's SqlDB interface does return a status value via a predefined variable *res*, which is defined in both the Database and Table definitions.

The following code extraction illustrates a technique of using the Table.rows variable value to check if the preceding SELECT statement was successful. (this should work with any SQL as most SQL's return the number of rows affected by the last SQL command)

```
(* ===== *)
MODULE AclassRating94;      (* in directory Yachting *)
VAR
    .....
    AclassTable* : SqlDB.Table;  (* anchor to Mysql database *)
    .....
    yot* : RECORD
(* this is the MySQL record structure *)
    rego*      : ARRAY 11 OF CHAR;    (* EG : KA219/01 *)  (* key *)
    yacht_name* : ARRAY 33 OF CHAR;
    loa*       : REAL;
    ..... etc .....

PROCEDURE FindSql*;
BEGIN
    AclassTable.Exec("Select * from Aclass where rego =:YachtingAclassRating94.yot.rego ");
    (* attempt read (select) one record from database *)
    IF AclassTable.rows = 1 THEN (* a record was found *)
        AclassTable.Read(0, yot);
        (* now read the actual record into Oberon record "yot" area *)
        (* returned table records are numbered from zero (0) *)
        Dialog.Update(yot);
        (* update display (form) fields *)
    ELSE
        yotLoaded := FALSE;
        (* no record found, or multiple records returned *)
        (* for the required yacht rego number *)
    END;
END FindSql;
(* ===== *)
```

7.6.7 Data Types Supported

The following Component Pascal types are interpreted by SqlDB:

```

BOOLEAN
BYTE, SHORTINT, INTEGER
SHORTREAL, REAL
ARRAY OF CHAR
Dates.Date
Dates.Time
Dialog.Currency
Dialog.List
Dialog.Combo
SqlDB.Blob

```

MySQL Data Type Mapping

(* still working on this , currently still MS-SQL server info *)

How these types are mapped to MySQL data types depends on the actual SQL database product and the Sql driver. The following example table applies to MySQL product which is accessed via ODBC:

MySQL	Component Pascal
{bit, tinyint, smallint, integer, bigint}	{BYTE, SHORTINT, INTEGER, BOOLEAN ⁽¹⁾ , Dialog.List}
{real, float(m) ⁽⁵⁾ , double precision}	{SHORTREAL, REAL}
{char(n) ⁽²⁾⁽⁶⁾ , varchar(n) ⁽³⁾⁽⁶⁾ , long varchar}	{ARRAY OF CHAR ⁽⁶⁾ , Dialog.Combo}
{decimal(m, d) ⁽⁵⁾ , numeric(m, d) ⁽⁵⁾ }	Dialog.Currency
{date, timestamp ⁽⁴⁾ }	Dates.Date
{time, timestamp ⁽⁴⁾ }	Dates.Time

Figure 7.4: MySQL to Blackbox Oberon Data Type Conversion

Note:

1. 0 = FALSE, 1 = TRUE
2. character string of fixed string-length n
3. variable-length character string with a maximum string length n
4. only the date or the time part is used, not both simultaneously note that the ANSI date order is year-month-day (2000-12-25)
5. m & d are precision & scale denoting how many decimal characters are displayed for the field, scale the number of decimal places eg decimal(9,2) = a field of max 9 characters, with 2 decimals. this is not the storage size, only the displayed size.
6. this is ASCII characters not Blackbox Oberon View Characters.

Note: Values of any MySQL data type can be read in a textual form, into an ARRAY OF CHAR. MySQL datetime values can be mapped either to Dates.Date or to Dates.Time, but not to both simultaneously.

7.6.8 BlackBox interface to SQL Code

BlackBox uses a interesting approach which combines the convenience of embedded SQL with the flexibility of declaring Blackbox Oberon variables explicitly in MySQL statements . For this purpose, Blackbox Oberon uses run-time type information, not directly available to a the application programmer, to access global variables, as they occur in an SQL string.

For example, if there is a global and exported integer variable `searchId` in module `Sample`, the following SQL statement can be used:

```
SELECT * FROM Companies WHERE id = :Sample.searchId
```

SqlDB Provides procedures to execute such a string eg `SqlDB.Database.Exec`, `SqlDB.Table.Exec`. and these procedures will replace all variables starting with a colon by the appropriate run-time values, and often converting these to appropriate text representations for processing by MySQL.

For additional convenience, whole record variables can be used in SQL strings. Their fields will be expanded suitably. For example, if there is a global and exported interactor company in module Sample with the fields id, name, ceo, employees¹³ then SqlDB will expand the following MySQL command string

```
INSERT INTO Companies VALUES (:Sample.company)
```

is expanded into

```
INSERT INTO Companies VALUES (555, 'Dragon City', 'Hallett', 1)
```

assuming that company.id = 555, company.name = "Dragon City", company.ceo = "Hallett" and company.employees = 1. Before sending the newly formed string off to MySQL for processing.

7.6.9 Type Row

Normally, a table's Read procedure is performed on an interactor that contains one record field per result table column. This provides automatic mapping between relational data and Component Pascal objects, ie: an object-oriented front-end for SQL databases.

7.6.9.1 Dynamically access a MySQL Tables column names

If you are developing a specialized tools such as a database browsers , a report writer, etc, then you will to extract the exact definition of the tables you will be accessing, and therefore you cannot define the known interactor types in advance. For these special cases, a more general dynamic mechanism is provided.

Instead of passing a Blackbox Oberon variable RECORD interactor to Read, a variable of the special type SqlDB.Row is passed instead. The result of the Read will an array of pointers to strings. Each string contains the textual (actual column name) representation of one table column of the row read. If SqlDB.names is passed as row parameter, the strings will contain the *names* of the table columns, instead of *values*.

```
String = POINTER TO ARRAY OF CHAR;
```

```
Row = RECORD fields: POINTER TO ARRAY OF String END
```

7.6.10 Blobs

Binary Large Objects, or "blobs", allow to store unstructured data as large ARRAY OF BYTE variables. Do not use ARRAY OF CHAR as BYTE allows access to every bit in a BYTE. For example, large image bitmaps (photographs, scanned images), or any other data may be stored in blobs. A blob is represented as a record

```
Blob = RECORD
  len: INTEGER;
  data: POINTER TO ARRAY OF BYTE
END
```

¹³Refer to 7.6.5 for MySQL record structure

The field *len* indicates the number of valid bytes in data. Data is a pointer to the byte array. A *Blob* that is used in several Read operations reuses the same data array if possible, i.e. if the new data takes at most as many bytes as the previous result's data.

7.6.11 Asynchronous operation

There is a detailed section in Blackbox Oberon documentation describing the usage of SQL databases where a discussion on Asynchronous verses Synchronous modes of accessing sql databases. Its interesting reading , but you can basically ignore the problems addressed therein unless you wish to delve deeper into the workings of Client/Server operations.

7.6.12 ODBC driver (Windows only)

Microsoft's Open Database Connectivity (ODBC) is an interface standard for accessing relational SQL databases. There are ODBC drivers for most relational products.

SqlOdbc is an Sql driver, supplied with Blackbox Oberon , which builds a bridge to the ODBC driver manager. Given a suitable 32-bit ODBC driver, this allows Blackbox Oberon access to database(s) via ODBC and Sql.

SqlOdbc supports all features of Sql, except asynchronous operation.

7.6.13 Displaying MySQL tables

As SQL database records are based upon the concept of tables, it is useful to be able to display the results of a MySQL Query in a tabular fashion¹⁴, module *SqlControls* provides appropriate table display controls

A serious limitation of Blackbox Oberon SQL Table Controls, is that the displayed fields (columns & rows) *cannot* be edited, however its possible to *select* a field in a table. See example code 7.7 to see how this is done.

A table control needs to be linked to a global variable of type *SqlDB.Table*.

eg:

```
MODULE SqlBrowser;
VAR
    ....
    table*: SqlDB.Table; (* anchor for database *)
    .....
```

A table control may also denote a notifier with the following signature:

```
TableNotifier = PROCEDURE (t: SqlDB.Table;
                           row, column: INTEGER;
                           modifiers: SET)
```

It is called whenever the user has clicked into a field of the table, indicating the table, its row and column numbers, and the track message's modifier set (-; Controllers.TrackMsg).

A table control is used in one of two ways: either it is opened in its own window, or it is embedded in some container, typically a form view.

¹⁴similar to the now traditional spreadsheet layout

7.6.14 Table control in its own Window

The separate window provides scrollbars if necessary.

See module SqlBrowser for an example of using a separate window for SQL results table.

7.6.15 Table control in a form

See module UpdtCompanies and Form UpdtCompanies for example of a SQL Table control in a form. (UpdtCompanies is a modified version of SqlBrowser)

To insert a SQL Table control on a form :

1. make sure form is in Layout mode (Dev – > Layout Mode)
2. Select menu option, (Controls – > Insert SQL Table), a blank memo icon control will appear on the form, position and adjust the size as required. It is usually wrapped in a scroller view, select SQL Control , then, (Tools– >Add Scroller) which provides the scrollbars.
3. Link the SQL Table Control to the SqlDB.Table via the Sql Table Control Properties Inspector

See below, the SQL Table Control has been selected (left click), then (right click) , then chose Properties from popup menu to open the Inspector Dialog. By filling in the Link & Notifier details the necessary links are made from the SQL Control to the Oberon Modules code.

```
MODULE SqlBrowser;
```

```
.....
```

```
VAR
```

```
    Items*: SqlDB.Table;
           (* anchor for database *)
```

```
.....
```

```
PROCEDURE ItemsNotifier* ( t : SqlDB.Table ;
                           row, col : INTEGER;
                           modifier : SET);
```

(* this procedure executed whenever a user mouse 'clicks' in a SQL table field, the value of row & col will be set to the clicked on row/col number, counting from 0 *)

```
VAR
```

```
    tmp1, tmp2 : ARRAY 5 OF CHAR;
    ans : SqlDB.Row;
```

```
BEGIN
```

```
    t.Read(row, company );
```

(* on user click in a SQLTable row field, read the pointed 'at' record into company update fields *)

```

Strings.IntToString(row, tmp1);
Strings.IntToString(col, tmp2);
Dialog.ShowStatus("Clicked on : " + " Row : " + tmp1 +
                  " Col : " + tmp2 + "-$>$" +
                  company.ceo );

(* display the current field parameters in the window status line *)

(* place your code to process the SQL Table data here *)

END TableNotifier;
(* ===== *)

```

Whenever the program executes a SQL statement that affects the `SqlDB.Table` the results will be displayed in the linked SQL Table control.

7.6.16 The Table Notifier

Whenever the mouse is 'clicked' in a displayed SQL Grid field an event is raised and program executes the Procedure *TableNotifier*, where *Table* is the name **you have defined as a VAR**. In the example below the table name is *Items*.

eg:

```

MODULE ListItems;
.....
VAR
    Items*: SqlDB.Table;
        (* anchor for database *)
.....

```

You also must supply a Procedure called *ItemsNotifier*, which is where processing continues upon detection of the mouse click. There must be a `SqlDB.Table` declaration and a Procedure *TableNotifier* defined for every MySQL Table you wish to have a grid displayed upon a form that user interaction is required.

```

PROCEDURE ItemsNotifier* ( t : SqlDB.Table ;
                          row, col : INTEGER;
                          modifier : SET);

```

7.6.17 Linked SQL Controls

Often a program will need to access multiple tables from a database at the same time. Displaying father – > child (1 – > many) relationships between two (or more) tables displaying data in their own SQL grids is a common feature of user interfaces. eg: Invoices, parts lists, etc.

To achieve this, the program will need to define , for each table, the following :

1. a `SqlDB.Table` variable for each table

2. a Oberon RECORD area for each table
and optionally
3. a SqlDB Table control for any grid results layout required.
4. table Notifiers for each table using a SqlDB Table control.

See the example extract below from program, Quickquote, for examples of this.

```

MODULE Quickquote;
.....
VAR
.....

      (* ===== SQL Table anchors ===== *)
cl* : SqlDB.Table;      (* client table anchor *)
stk*: SqlDB.Table;      (* Stock table anchor *)
      (* -- other anchors -- *)
qhdr* : SqlDB.Table;      (* Quote_Header table anchor *)
qitem*: SqlDB.Table;      (* Quote_Item table anchor *)
.....
      (* ===== Oberon/SQL RECORD LAYOUTS ===== *)
Quote_header* : RECORD      (* Quote Header Record *)
  QTS_CLIENT_CODE* : ARRAY 5 OF CHAR;      (* key *)
  QTS_QUOTE_NO* : INTEGER;      (* key *)
  QTS_QUOTE_DATE* : Dates.Date;
  QTS_MEMO_PAGE* : SqlDB.Blob;
END;

Quote_items* : RECORD      (* Quote Item Record *)
  QU_CLIENT_CODE* : ARRAY 5 OF CHAR;      (* key *)
  QU_QUOTE_NO* : INTEGER;      (* key *)
  QU_ITEM_SEQ* : INTEGER;      (* key *)
  QU_STOCK_CODE* : ARRAY 20 OF CHAR;
  QU_STOCK_DESCRIPTION* : ARRAY 80 OF CHAR;
  QU_BUY_PRICE* : Dialog.Currency;
  QU_QUOTED_PRICE* : Dialog.Currency;
  QU_QUANTITY_REQUIRED* : INTEGER;
END;
.....

```

```

PROCEDURE qhdrNotifier* ( t : SqlDB.Table;
                        row, col : INTEGER;
                        modifier : SET);

(* When the user clicks upon an row/column entry displayed in the Quote Header SQL Table
Control, this code will attempt to find ( by SQL select) the corresponding Quote Item table
records belonging to this quote and display them in their own SQL Table Control. (a typical one
to many display concept, eg father - > child relationship ) *)

BEGIN
    Dialog.ShowStatus("Clicked in qhdr");
    qhdr.Read(row, Quote_header); (* read clicked on row *)
    Dialog.ShowStatus("Client Selected : " +
                      Quote_header.QTS_CLIENT_CODE );

    qitem.Exec("select * from quote_item where QU_CLIENT_CODE=
               :Quickquote.Quote_header.QTS_CLIENT_CODE ");
    (* read header items list using data from Quote Header table *)

    Dialog.Update(qitem);                (* update qitems display *)
END qhdrNotifier;
(* ===== *)
PROCEDURE qitemNotifier* ( t : SqlDB.Table;
                        row, col : INTEGER;
                        modifier : SET);

BEGIN
    Dialog.ShowStatus("Clicked in qitem");
END qitemNotifier;

( *** end code extract *** )

```

The screenshot shows a window titled "Quotation System" with a menu bar containing: Quotes, Stock, Clients, Load Data, Tax Codes, Translate Table, List Quotes, System, Supplier, and Templates. The main area is divided into two sections. The top section, labeled "Quote Header", contains a table with columns: CLIENT_CODE, QUOTE_NO, QUOTE_DATE, and MEMO_PAGE. The table has three rows, with the second row (45, 4545, 2001-04-22) selected. Below this table are buttons: Create Quote, Delete Quote, Copy Quote, and Print Selected Quote. The bottom section contains a table with columns: ITEM_SEQ, STOCK_CODE, STOCK_DESCR, BUY_PRICE, QUOTED_PRICE, and QTY... This table has four rows of item data. At the bottom of the window, there is a summary bar showing "# Items: 4", "631.00", and "694.10", and an "Exit System" button.

CLIENT_CODE	QUOTE_NO	QUOTE_DATE	MEMO_PAGE
60	3360	2001-02-05	
45	4545	2001-04-22	
3799	123	2001-02-05	

ITEM_SEQ	STOCK_CODE	STOCK_DESCR	BUY_PRICE	QUOTED_PRICE	QTY...
282500	CP-IBM-P233MHZ	686 233MMX CPU	88.00	96.80	0
283250	CP-PENTIUM2-266	INTEL PENTIUM II 266 CPU	329.00	361.90	0
284125	CP-PENTIUM2C266	INTEL CELERON 266 CPU NON CACHE CPU	169.00	185.90	0
285062	WIS57558	VIDEO 8900CL TRIDENT 1MD VGA 16BIT	45.00	49.50	0

Items: 4 631.00 694.10

Exit System

Figure 7.5: Gui Form displaying MySQL 1 – > Many record relationship

The user has clicked on Quote Header SQL Table Control, Client Code field "45" and the Quote Items SQL table Control now displays data found for that Client's Quote(4545). Notice the vertical Scroll Bar on the Quote header, these SQL Table Control's allowing viewing by the user of many Quote Header/Items records. ¹⁵

However, it is possible to 'compress' a column completely on the displayed form, thus showing two vertical lines together, a bit primitive but useful. Simply drag one dividing vertical line with your mouse until the displayed information disappears

When a table control is generated by a program, rather than by manual form creation, it is not necessary to link it to a global table pointer variable. Instead, the table pointer can be directly passed as parameter in the call to

SqlControls.dir.NewTableOn(table).

¹⁵As of Blackbox version 1.4 it is not possible to control the layout of the columns, except font size, thus headings, colour, justification, etc cannot be adjusted. The SQL table design however can help with the headings by making them as descriptive as possible to the user (if not to the programmer !)

7.7 Editing MySQL Table Entries

As mentioned in the standard Blackbox Oberon manual, you cannot modify (edit) data in SQL displayed table controls.¹⁶ However we can still allow user the update facility with a little ingenuity. In the example displayed below the fields we wish to allow editing upon are displayed as a 'side bar' to the main MySQL Table display on the left. As the user browses around the MySQL Table, upon a mouse click (currently either left or right click) in any column then, *that rows data* is read into the Modules RECORD area by execution of the *user supplied SQLNotifier Procedure* and displayed in the 'side bar' fields, the user may *edit* any of the displayed fields, and upon pressing the **Update** button that record will be updated and the MySQL table refreshed to show the newly entered values.

The screenshot shows a window titled "Quotation System" with a menu bar containing: Quotes, Stock, Clients, Load Data, Tax Codes, Translate Table, List Quotes, System, Supplier, and Templates. Below the menu is a table with the following columns: CATEGORY, STK_CODE, STK_SEQ, DESCR, and BUY_PRICE. The table lists various cable products. To the right of the table is a "side bar" with input fields for Category, Stock Code, Stock Seq., Buy Price, Tax Rate, Margin \$, and Margin %. At the bottom right are buttons for Delete, Update, and Exit System.

CATEGORY	STK_CODE	STK_SEQ	DESCR	BUY_PRICE
Cables	CAB06305	1998-12-02	CD AUDIO CABLE FOR CREATIVE CARDS	2.00
Cables	CAB06358	2001-12-23	CD AUDIO CABLE TO SUIT YAMAHA CARDS	2.00
Cables	CAB09755	2001-12-23	LAN ETHERNET BNC CRIMP CONNECTOR	2.00
Cables	CAB10002	2002-02-25	CABLE PLATE BP01 DB25M(SERIAL PORT)	13.00
Cables	CAB10108	2001-12-23	CABLE PLATE BP03 DB25M&PS2(SER/PS2)	5.00
Cables	CAB10152	2001-12-23	CABLE PLATE BP04 D25M&D15F(SER/GAM)	5.00
Cables	CAB10258	2002-02-19	CABLE PLATE BP06 USB X 2	12.00
Cables	CAB10355	2001-12-23	CABLE DB25M TO DB25F 10 METRES	12.00
Cables	CAB10452	2001-12-23	CABLE DB25M TO DB25F 3 METRES	66.00
Cables	CAB10505	2001-12-23	CABLE DB25M TO DB25F 5 METRES	33.00
Cables	CAB10752	2001-12-23	CABLE BCRG58 THIN ETHERNET PER/M	1.00
Cables	CAB10805	2001-12-23	CABLE THIN ETHERNET (10M INC.BNC)	15.00
Cables	CAB12708	2001-12-23	CABLE CEN50/CEN50 1 METRE	10.00
Cables	CAB12752	2001-12-23	CABLE CEN50/DB25 1 METRE	10.00
Cables	CAB12805	1998-12-02	CABLE CEN50/SCSI-2 1 METRE	20.00
Cables	CAB14005	1998-12-02	CABLE CHA505 VIDEO 5 BNC TO DB15	35.00
Cables	CAB16952	2001-12-22	CABLE 3.5 FDD DATA ADAPTOR	2.00
Cables	CAB17102	2001-12-22	CABLE DB25 TO SCSI-II 1 METRE (ZIP)	30.00
Cables	CAB17402	2001-12-23	CABLE DC2 DATA (IDE FOR CD ROM)	2.02
Cables	CAB21708E	2001-12-23	CABLE 34 WAY DATA FOR FDD'S	1.00

Side bar fields:

- Category: Cables
- Stock Code: CAB10002
- Stock Seq.: 2002-02-25
- Buy Price: 9.00
- Tax Rate: ☐
- Margin \$: ☐
- Margin %: ☐

Buttons: Delete, Update, Exit System

Figure 7.6: Edit MySQL Table using 'side bar fields'

In the above snapshot, we have changed the price from \$13.00 to \$9.00 for stock item CAB10002, but have not pressed **Update**.

Instead of using a 'side bar' we could just as easily used a modal form, see Chapter 10, where upon clicking on the column, a *new form* with the fields for the row selected would be displayed

¹⁶In my opinion a major failing of the SQL interface under Blackbox Oberon , but there are fairly obvious technical reasons for this.

for editing, upon pressing the `Update` or `Delete` buttons of the called form, the appropriate field validation and MySQL commands would modify the database and refreshed the displayed MySQL Table.

```

PROCEDURE invNotifier* ( t : SqlDB.Table;
                        row, col : INTEGER;
                        modifier : SET);

BEGIN
    inv.Read(row, Inventory);
    Dialog.Update(inv);
END invNotifier;
(* ===== *)
PROCEDURE invUpdate*; (* update current inventory record *)
VAR
BEGIN
    inv.base.Exec("DELETE FROM inventory WHERE
                  :Quickquote.Inventory.CATEGORY = CATEGORY
                  AND :Quickquote.Inventory.STK_CODE = STK_CODE ");
    (* delete currently selected record *)
    QuickquoteUtils.SqlDate(Inventory.STK_SEQ);
    (* get sysdate in Mysql format yyyy-mm-dd, update inventory record *)
    inv.base.Exec("INSERT INTO inventory VALUES
                  (:Quickquote.Inventory)");
    inv.base.Commit();
    (* insert modified record *)

    inv.Exec("SELECT * FROM INVENTORY
             ORDER BY CATEGORY ,STK_CODE");
    Dialog.Update(inv);
    (* redisplay updated table records *)
END invUpdate;
(* ===== *)
PROCEDURE invDelete*; (* delete current inventory record *)
BEGIN
    inv.base.Exec("DELETE FROM inventory
                  WHERE CATEGORY = :Quickquote.Inventory.CATEGORY
                  AND STK_CODE = :Quickquote.Inventory.STK_CODE ");

    inv.base.Commit();
    inv.Exec("SELECT * FROM INVENTORY
             ORDER BY CATEGORY , STK_CODE");
    Dialog.Update(inv);
END invDelete;
(* ===== *)

```

Figure 7.7: Code to Update / Delete selected MySQL Table Entry

7.8 Design rules

The following paragraphs are taken from the Blackbox Oberon SQL reference information, as they are *rules* they are copied verbatim - © Oberon Microsystems.¹⁷

This section gives some rules which should be followed when designing a database application. The reason for each rule is given after the rule in *italics*.

1. Interactors or at least their types must be exported if they should be used as place holders in SQL statements.

Non-exported types are not accessible through metaprogramming, and thus controls and the SQL string translation mechanism could not be used with them.

2. A globally anchored database and all its table pointers must be set to NIL if the database ought to be closed.

If a global pointer variable is not set to NIL, the garbage collector cannot reclaim the data structures anchored in it. Upon garbage collection of a database, the database is closed if there are no more pointers to it. Note that a table contains a pointer to its database object and thus anchors it.

3. A database pointer should not be declared as global variable.

Normally there are global table pointers, which contain references to their databases. A global database pointer would be one more pointer to set to NIL eventually; and anchor views can only set table pointers to NIL.

4. Database.Exec may only execute *non-row-returning* (no results table) statements, e.g. DELETE or INSERT. Note: Table.base.Exec is equivalent to Database.Exec because a Table control points to its Database control.

A result table must always be assigned to a table and via the table to an interactor, otherwise it cannot be accessed by the application.

5. Table.Exec should only execute row-returning (results table) statements, e.g., SELECT.

Note: Table.base.Exec is equivalent to Database.Exec because a Table control points to its Database control.

¹⁷I have added some remarks also.

6. The order and types (but not necessarily the names) of fields in an interactor variable must match those defined in the SQL database. The number of fields must be the same as the number of columns.

eg:

Oberon interactor record

MySQL table definition

inrec* : RECORD

company_owner : INTEGER;

owner_by : INTEGER;

percent_owned : INTEGER;

END;

CREATE TABLE Ownership

(owner INTEGER,

owned INTEGER,

percent INTEGER

PRIMARY KEY (owner));

7. Only complete tables may be assigned to an interactor by Table.Read, no partial assignment is allowed.

There must be a corresponding Oberon field defined for every field in the MySQL table definition, like the example above.

8. The correspondence of result table columns and interactor fields is given by their respective declaration order. Nested arrays, records and pointers are handled recursively.
9. Pointers must not be NIL except for POINTER TO ARRAY OF CHAR which are allocated automatically if the pointer is NIL or if the bound array is too small to receive the corresponding string.
10. Scalar result variables are treated as tables with table.rows = 1 and table.columns = 1.

That is: a SELECT COUNT(ID) FROM ACLASS statement will return the result in a SQL Table with one entry – the count of records found

11. After an SQL statement which may render a row inconsistent with the database, use a SELECT statement to re-establish consistency.

For example, an interactor may still contain the value of a row which has just been deleted through a DELETE statement. Exec on the table will flush the old result table and possibly assign a new one, thus refreshing the displayed MySql records

Chapter 8

The Report Generator : BxRepGen

While the report possibilities using Oberon Views are extremely powerful, they are a bit complex at first approach, while developing MySQL programs the programmer often needs 'quick' reports to check out the current state of tables being used. Therefore, I wrote the program *BxRepGen* (Blackbox Report generator) to solve this problem. The complete code is supplied here, hopefully as an aid to further development by other Oberon users.

The below form snapshots show the steps necessary to generate a basic standard report of a user selected Database Table. This was the tool used to generate the base of the Karin Report shown in 9.2.2

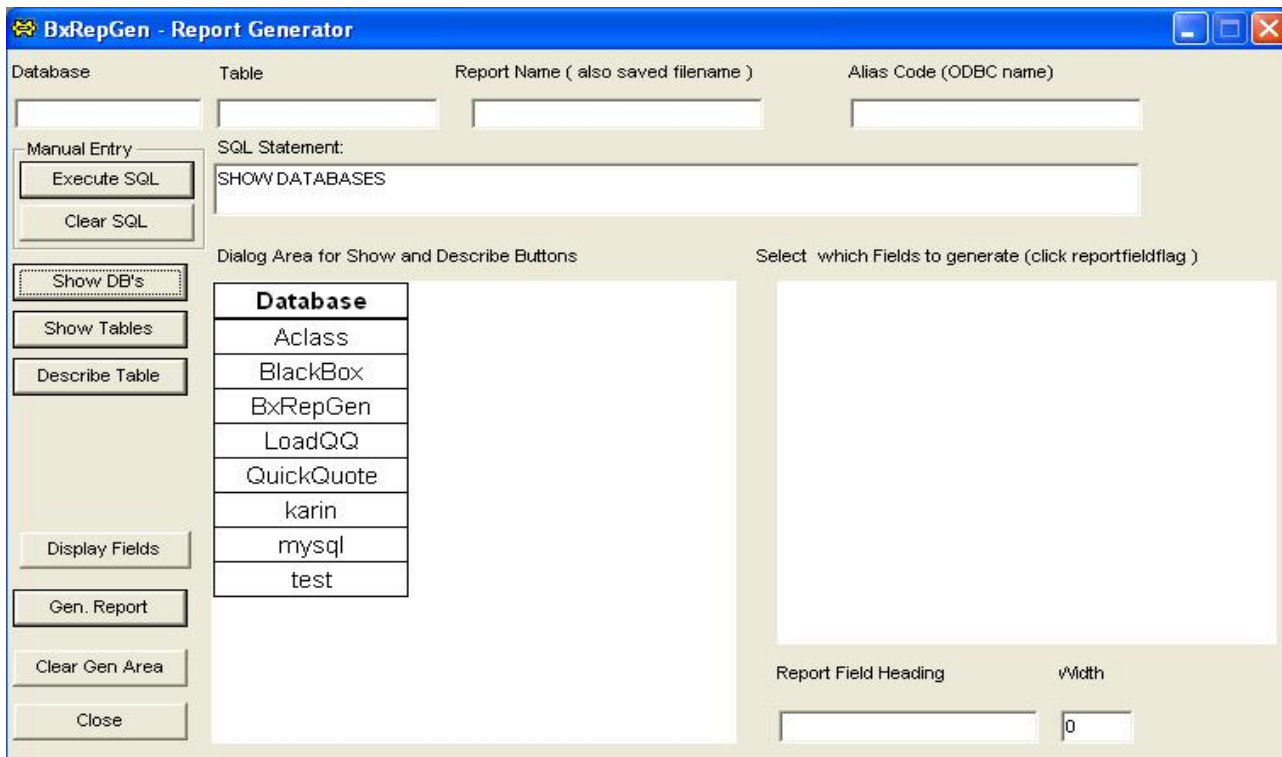


Figure 8.1: BxRepGen - Show available Databases

After the user has pressed the **Show DB** button, BxRepGen displays a list of all the available *databases* accessible by MySQL.

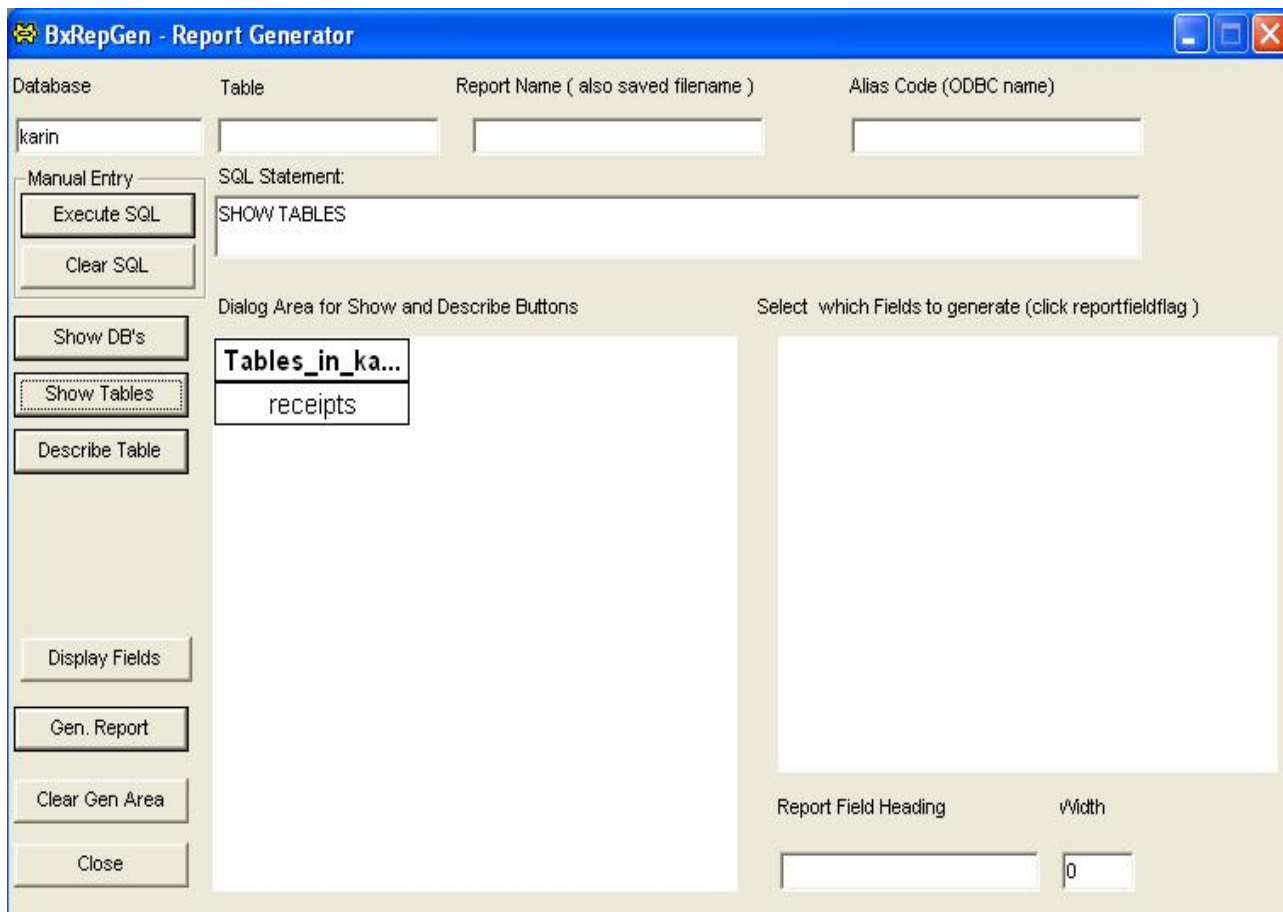


Figure 8.2: BxRepGen - Show Tables in selected Database

By clicking on the desired database in the list displayed and then press **Show Tables** button then the available *Tables* in that database will be displayed.

BxRepGen - Report Generator

Database: Table: Report Name (also saved filename): Alias Code (ODBC name):

Manual Entry:

SQL Statement:

Dialog Area for Show and Describe Buttons:

Display Fields: Gen. Report: Clear Gen Area: Close:

Field	Type	Null
recdate	date	YES
supplier	char(40)	YES
goods	char(30)	YES
total	decimal(9,2)	YES
seq	int(11)	
recnumber	char(10)	YES

Select which Fields to generate (click reportfieldflag)

Report Field Heading: Width:

Figure 8.3: BxRepGen - Describe Selected Table

By clicking on the desired *Table* in the list displayed, then pressing the will display the *Fields* in the chosen Table.

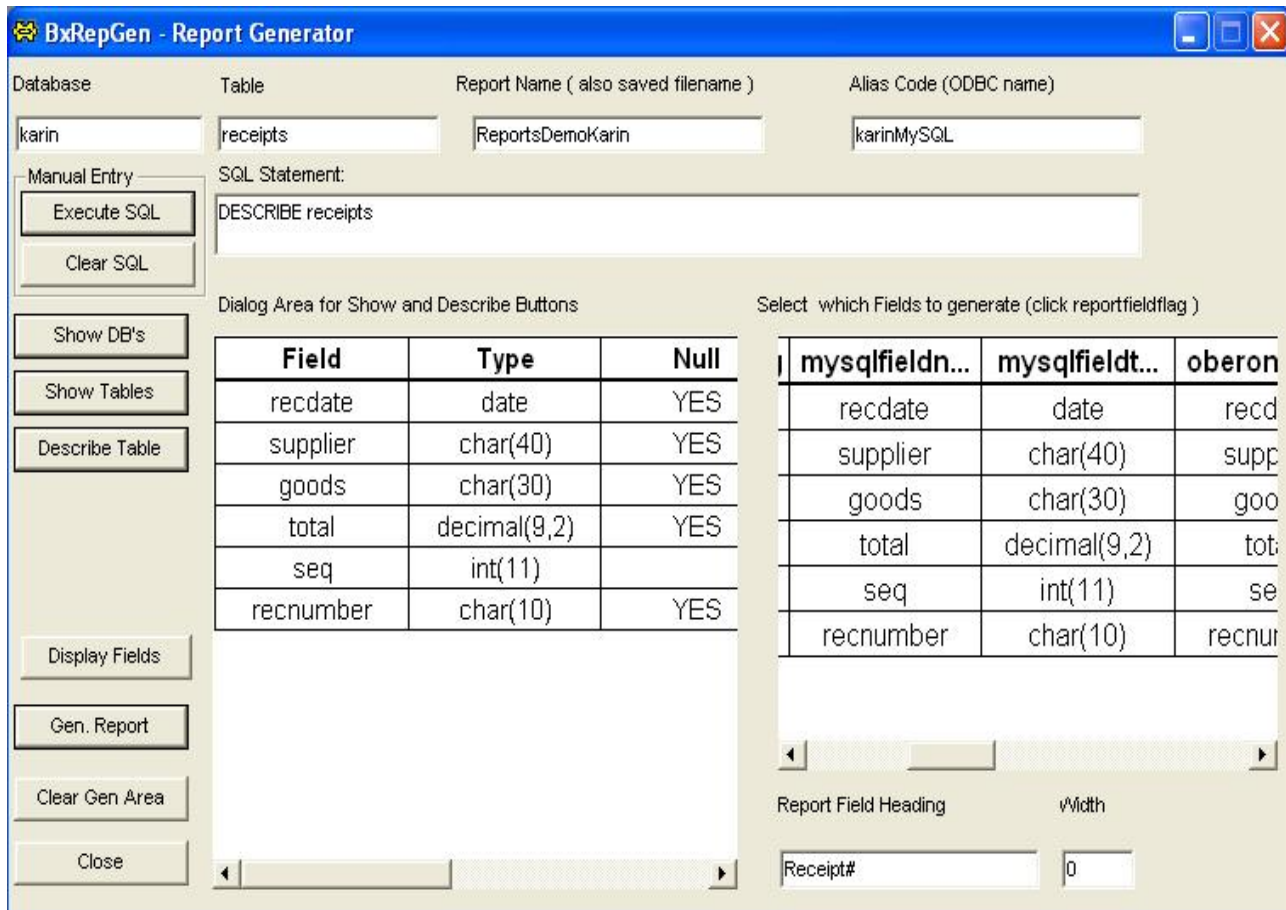


Figure 8.4: BxRepGen - Display Selected Fields in Selected Table

You then click on the required Fields,(visit each displayed field (row) in turn and click once), from the displayed list of *Field Names*, when you have chosen the desired fields, then click on **Display Fields** button to display a new list in the right hand table display.

By entering data in the two fields below this table you can enter Field Heading information and if necessary a field width value. You may revisit any field in this list and turn its selection status to 'Y' or 'N'.

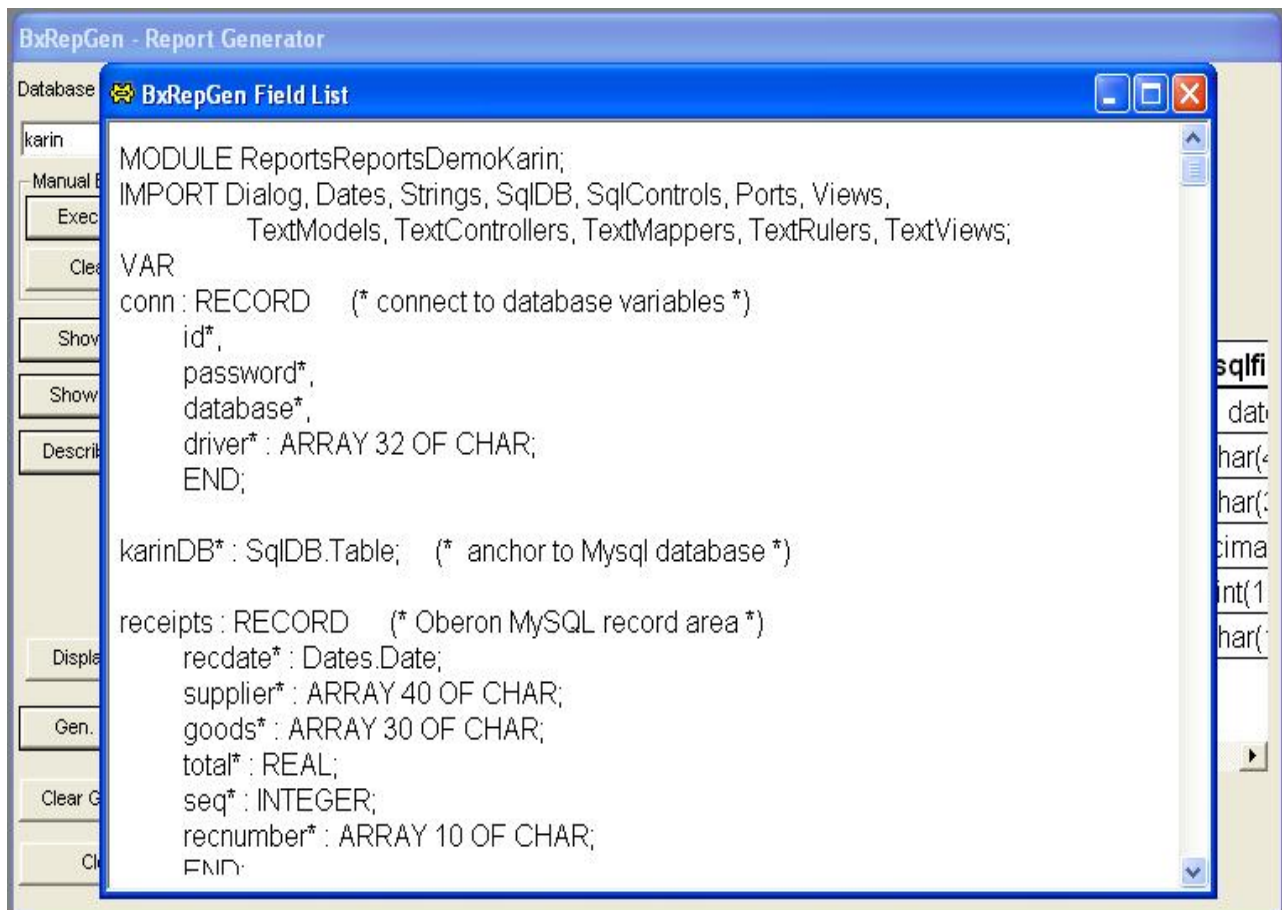


Figure 8.5: BxRepGen - The Generated Report

Enter the Report Name and its ODBC Alias name in the top two fields, then press **Gen. Report** to generate the complete report.

You must save this generated report in the appropriate Blackbox Oberon MOD folder, then compile the report. There should be no errors !

You would usually place a entry for this report in a menu for easy execution, of course the report may be run from a program if desired. Refer to page 134.

Now you may edit the report to suit you exact needs.

```

MODULE Bxrepgen;

(* Φ”StdCmds.OpenAuxDialog('Bxrepgen/Rsrc/Bxrepgen','Report Generator')” *)

(*
simple report generator for MySQL databases

(c) 2001, Brett S Hallett : Dragon City Systems

)

IMPORT
    Dialog, Views, Strings, TextModels, TextControllers, TextMappers, TextViews, Ports, TextRulers,
    SqlDB, SqlControls, StdLog, DcsUtils;

CONST
showdb = 0; showtbls = 1; describetbl = 2; displaysql = 3; (* for selectflag switch *)
CONST
    tchar = 0; tcint = 1; tcreal = 2; tcdate = 3; tctime = 4; (* for field data type switch *)
VAR
    t : TextModels.Model; f: TextMappers.Formatter; v: TextViews.View;
    r : TextRulers.Ruler;      (* connections to text output area *)

    selectflag : INTEGER; (* switch set when user presses particular button*)

    reportfieldheading* : ARRAY 32 OF CHAR;
reportfieldwidth*      : INTEGER;

Bxtable* : SqlDB.Table;  (* anchor for database *)
BxGenTbl* : SqlDB.Table;

lc : RECORD              (* lowercase copies of various fields textual names *)
    databasename : ARRAY 64 OF CHAR;
    tablename : ARRAY 64 OF CHAR;
    reportname   : ARRAY 64 OF CHAR;
END;

dlg* : RECORD
    reportname* : ARRAY 32 OF CHAR;
    aliasname*  : ARRAY 32 OF CHAR;
END;

conn*: RECORD
    id*,
    password*,
    database*,

```



```

        driver* : ARRAY 32 OF CHAR;
        statement* : ARRAY 1024 OF CHAR;
    END;      (* database access fields *)

dbs* : RECORD
    databasename* : ARRAY 32 OF CHAR;
    END;      (* record for database name *)

tbls* : RECORD
    tablename* : ARRAY 32 OF CHAR;
    END;      (* record for table name *)

describeflds* : RECORD
    field*      : ARRAY 64 OF CHAR;
    type*       : ARRAY 20 OF CHAR;
    fnull*      : ARRAY 4 OF CHAR;
    fkey*       : ARRAY 3 OF CHAR;
    default*    : ARRAY 7 OF CHAR;
    extra*      : ARRAY 5 OF CHAR;
    END;      (* data as returned by MySQL describe comand *)

Bxfields* : RECORD
    mysqltablename* : ARRAY 64 OF CHAR;
    fieldseq*      : INTEGER;
    reportfieldflag* : ARRAY 2 OF CHAR;
    mysqlfieldname* : ARRAY 64 OF CHAR;
    mysqlfieldtype* : ARRAY 20 OF CHAR;
    oberonfieldname* : ARRAY 64 OF CHAR;
    oberonfieldheading* : ARRAY 32 OF CHAR;
    oberonfieldtype* : ARRAY 20 OF CHAR;
    oberonfieldcode* : INTEGER; (* for fieldtype case selects *)
    oberonfieldsize_m* : INTEGER; (* scale *)
    oberonfieldsize_d* : INTEGER; (* precision *)
    reportfieldwidth* : INTEGER; (* for report heading/column output only *)
    comments* : ARRAY 255 OF CHAR;

    END; (* selected fieldnames table for code generation and output to Bxfields table*)

(* ===== *)
PROCEDURE setupRulers; (* for code generation *)
CONST
    cm = 10 * Ports.mm; (* universal units *)
VAR
    reccnt : INTEGER;
    x ,y, z : INTEGER;

```

```

BEGIN

  t := TextModels.dir.New(); (* create empty text carrier *)
  f.ConnectTo(t);           (* connect a formatter to text *)
  r := TextRulers.dir.New(NIL); (* set up rulers for the report *)

  TextRulers.AddTab( r, 10 * Ports.mm );
  TextRulers.AddTab( r, 20 * Ports.mm );
  TextRulers.AddTab( r, 30 * Ports.mm );
  TextRulers.AddTab( r, 40 * Ports.mm );
  TextRulers.AddTab( r, 50 * Ports.mm );
  TextRulers.AddTab( r, 60 * Ports.mm );
  TextRulers.AddTab( r, 70 * Ports.mm );
  TextRulers.AddTab( r, 80 * Ports.mm );
  TextRulers.AddTab( r, 100 * Ports.mm );
                                     (* set up TABS for report *)
  TextRulers.SetRight(r, 120 * Ports.mm );

  f.WriteView(r);                  (* write the ruler tabs *)

END setupRulers;
(* ===== *)
PROCEDURE CreateSqlRecord ( IN table : ARRAY OF CHAR ) ;

VAR
  recnt : INTEGER;
BEGIN
  recnt := 0;

  f.WriteString( table$ );
  f.WriteString(" : RECORD (* Oberon MySQL record area *) ");
  f.WriteLine;                    (* Oberons SQL record header *)

  WHILE recnt <= BxGenTbl.rows -1 DO

    BxGenTbl.Read(recnt, Bxfields );
                                     (* read records 1 at a time, starting from first (again) *)

    Strings.ToLower(Bxfields.mysqlfieldname, Bxfields.oberonfieldname);
    f.WriteTab;
    f.WriteString(Bxfields.oberonfieldname$);
    f.WriteString(" * : ");
    f.WriteString(Bxfields.oberonfieldtype$);
    f.WriteLine;
  
```

```

        INC(reccnt);
    END;
    f.WriteTab;
    f.WriteString("END;");
    f.WriteLine;
END    CreateSqlRecord;
(* ===== *)

PROCEDURE CreateSqlClose(IN name : ARRAY OF CHAR);
BEGIN
    f.WriteString("(* ===== *)");
    f.WriteLine;
    f.WriteString("PROCEDURE CloseSQL;");
    f.WriteLine;
    f.WriteString("BEGIN");
    f.WriteLine;
    f.WriteString("IF "); f.WriteString(name$); f.WriteString(" # NIL THEN");
    f.WriteLine;
    f.WriteTab; f.WriteString(name$); f.WriteString(".Clear;");
    f.WriteLine;
    f.WriteTab; f.WriteString(name$); f.WriteString(" := NIL;");
    f.WriteLine;
    f.WriteTab; f.WriteString("END;");
    f.WriteLine;
    f.WriteString("END CloseSQL;");
    f.WriteLine;
    f.WriteString("(* ===== *)");
    f.WriteLine;
END CreateSqlClose;

(* ===== *)
PROCEDURE CreateSqlOpen(IN name : ARRAY OF CHAR );
BEGIN
    f.WriteLine;
    f.WriteString("PROCEDURE OpenSQL;");
    f.WriteLine;
    f.WriteString("VAR db: SqlDB.Database; (* variable db cannot be a global variable ! *)");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("res: INTEGER;");
    f.WriteLine;
    f.WriteString("BEGIN");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("CloseSQL;");

```

```

f.WriteLine;
f.WriteTab;
f.WriteString("SqlDB.OpenDatabase(conn.driver,conn.id,conn.password,conn.database,");
f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString("SqlDB.async, SqlDB.showErrors, db, res);");
f.WriteLine;
f.WriteTab; f.WriteTab; f.WriteTab;
f.WriteString(" (* open the database using variables setup in record conn *)");
f.WriteLine;
f.WriteTab;
f.WriteString("IF res = 0 THEN");
f.WriteLine;
f.WriteTab; f.WriteTab;
f.WriteString(name$); f.WriteString(" := db.NewTable()");
f.WriteLine;
f.WriteTab;
f.WriteString("ELSIF res <= 3 THEN");
f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString('Dialog.ShowMsg("#Sql:CannotLoadDriver")');
f.WriteLine;
f.WriteTab;
f.WriteString("ELSE");
f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString('Dialog.ShowMsg("#Sql:ConnectionFailed")');
f.WriteLine;
f.WriteTab;
f.WriteString("END");
f.WriteLine;
f.WriteString("END OpenSQL;");
f.WriteLine;
f.WriteString("(* ===== *)");
f.WriteLine;
END CreateSqlOpen;

(* ===== *)
PROCEDURE CreateReportTabs; (* for generated report code *)
VAR
  reccnt : INTEGER;
  rc      : INTEGER;
BEGIN
  reccnt := 0;
  WHILE reccnt <= BxGenTbl.rows DO
    BxGenTbl.Read(reccnt, Bxfields );

```

```

(* read records 1 at a time, starting from first *)
IF Bxfields.reportfieldflag$ = 'Y' THEN
    (* only generate print statements for user selected fields *)
    f.WriteTab;

    f.WriteString('TextRulers.AddTab( r , ');
    IF reccnt = 0 THEN
        rc := 1;
        f.WriteInt( rc );
    ELSE
        f.WriteInt( reccnt * 25);
    END;

    f.WriteString(' * Ports.mm ');
    f.WriteLine;

    CASE Bxfields.oberonfieldcode OF
        | tcchar :
        | tcreal, tcint : f.WriteTab;
            f.WriteString('TextRulers.MakeRightTab( r ); ');
            f.WriteLine;
            (* set tab to right justify for numerics *)
        | tcdatetime :
            f.WriteLine;
    END;
END; (* if *)
INC(reccnt);
END; (* while *)
f.WriteLine;
f.WriteTab;
f.WriteString('f.WriteView(r);');
f.WriteLine;
END CreateReportTabs;

(* ===== *)
PROCEDURE CreatePrintDateTime;
VAR
BEGIN
    f.WriteTab;
    f.WriteString('Dates.GetDate( date);');
    f.WriteString('Dates.DateToString(date, 1, sdate );');
    f.WriteString('f.WriteString(sdate$ + " ");');
    f.WriteLine;
    f.WriteTab;
    f.WriteString('Dates.GetTime( now );');
    f.WriteString('Dates.TimeToString(now, stime );');

```

```

f.WriteString('f.WriteString( stime$ );');
f.WriteLine;

END CreatePrintDateTime;
(* ===== *)
PROCEDURE CreatePrintHeadings;
VAR
    reccnt : INTEGER;
BEGIN
    reccnt := 0;
    WHILE reccnt <= BxGenTbl.rows - 1 DO
        BxGenTbl.Read(recnt, Bxfields );
                                (* read records 1 at a time, starting from first *)
        IF Bxfields.reportfieldflag$ = 'Y' THEN
            (* only generate print statements for user selected fields *)
            f.WriteTab;
            f.WriteString('f.WriteTab;');
            f.WriteLine;
            f.WriteTab;
            f.WriteString('f.WriteString(" "); f.WriteString(Bxfields.oberonfieldheading$);
            f.WriteString(""); ');
            f.WriteLine;

            END; (* if *)
            INC(recnt);
        END; (* while *)
        f.WriteLine;
        f.WriteTab;
        f.WriteString('f.WriteLine;');
        f.WriteLine;
    END CreatePrintHeadings ;

    (* ===== *)

PROCEDURE CreatePrintStatements;
CONST
    tcchar = 0; tcint = 1; tcreal = 2; tcdate = 3; tctime = 4;
    (* BxRepGen field data types code for CASE , not a Blackbox type code ! *)
VAR
    reccnt : INTEGER;
BEGIN
    reccnt := 0;
    WHILE reccnt <= BxGenTbl.rows -1 DO

        BxGenTbl.Read( recnt, Bxfields );

```

```

(* read records 1 at a time, starting from first *)
IF Bxfields.reportfieldflag$ = 'Y' THEN
    (* only generate print statements for user selected fields *)

    CASE Bxfields.oberonfieldcode OF
        | tcchar :
            f.WriteTab;f.WriteTab;
            f.WriteString('f.WriteTab;'); f.WriteLine;
            f.WriteTab;f.WriteTab;
            f.WriteString("f.WriteString("); f.WriteString(lc.tablename);
            f.WriteString("."); f.WriteString(Bxfields.oberonfieldname$);
            f.WriteString("$);");
            f.WriteLine;
            f.WriteTab;f.WriteTab;

            f.WriteLine;

        | tcint :
            f.WriteTab;f.WriteTab;
            f.WriteString('f.WriteTab;'); f.WriteLine;
            f.WriteTab;f.WriteTab;
            f.WriteString("f.WriteInt("); f.WriteString(lc.tablename);
            f.WriteString("."); f.WriteString(Bxfields.oberonfieldname$);
            f.WriteString(");");
            f.WriteLine;
            f.WriteTab;f.WriteTab;
            f.WriteLine;

        | tcreal :
            f.WriteTab;f.WriteTab;
            f.WriteString('f.WriteTab;'); f.WriteLine;
            f.WriteTab;f.WriteTab;
            f.WriteString("f.WriteRealForm("); f.WriteString(lc.tablename);
            f.WriteString("."); f.WriteString(Bxfields.oberonfieldname$);
            f.WriteString(",7,");
            f.WriteInt(Bxfields.oberonfieldsize_m);
            f.WriteString(",");
            f.WriteInt(Bxfields.oberonfieldsize_d);
            f.WriteString(", ' ');");
            f.WriteLine;
            f.WriteTab;f.WriteTab;
            f.WriteLine;

        | tcdate :
            f.WriteLine;

```

```

f.WriteTab;f.WriteTab;
f.WriteString('f.WriteTab;'); f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString("Dates.DateToString("); f.WriteString(lc.tablename);
f.WriteString("."); f.WriteString(Bxfields.oberonfieldname$);
f.WriteString(",0,sdate);");
f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString("f.WriteString(sdate$);");
f.WriteLine; f.WriteLine;

| tctime :

f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString('f.WriteTab;'); f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString("Dates.TimeToString("); f.WriteString(lc.tablename);
f.WriteString("."); f.WriteString(Bxfields.oberonfieldname$);
f.WriteString(",0,stime);");
f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString("f.WriteString(stime$);");
f.WriteLine; f.WriteLine;

END; (* case *)
END; (* if *)
INC(reccnt);
END; (* while *)
f.WriteTab;f.WriteTab;
f.WriteString("f.WriteLine;"); f.WriteLine;

END CreatePrintStatements;
(* ===== *)

PROCEDURE CreatePrintAll (IN database, table : ARRAY OF CHAR);
BEGIN

f.WriteString('PROCEDURE PrintAll*; ');
f.WriteLine;
f.WriteString('CONST ');
f.WriteLine;
f.WriteTab;
f.WriteString('cm = 10 * Ports.mm; (* universal units *) ');
f.WriteLine;
f.WriteString('VAR ');
f.WriteLine;

```



```

f.WriteTab;
f.WriteString('t : TextModels.Model; f : TextMappers.Formatter; v : TextViews.View; ');
f.WriteLine;
f.WriteTab;
f.WriteString('r : TextRulers.Ruler; ');
f.WriteLine;
f.WriteTab;
f.WriteString('recCnt : INTEGER; ');
f.WriteLine;
f.WriteTab;
f.WriteString('date : Dates.Date; ');
f.WriteLine;
f.WriteTab;
f.WriteString('now : Dates.Time; ');
f.WriteLine;
f.WriteTab;
f.WriteString('sdate : ARRAY 40 OF CHAR; ');
f.WriteLine;
f.WriteTab;
f.WriteString('stime : ARRAY 16 OF CHAR; ');
f.WriteLine;
f.WriteLine;
f.WriteString('BEGIN');
f.WriteLine;

f.WriteTab;
f.WriteString("t := TextModels.dir.New();");
f.WriteLine;
f.WriteTab;
f.WriteString("f.ConnectTo(t);");
f.WriteLine;
f.WriteTab;
f.WriteString("r := TextRulers.dir.New(NIL);");
f.WriteLine;

CreateReportTabs;

f.WriteTab;
f.WriteString(database); f.WriteString('.Exec("SELECT * FROM ');
f.WriteString(table); f.WriteString("");');
f.WriteLine;
f.WriteTab;
f.WriteString('IF ');f.WriteString(database); f.WriteString(".rows = 0 THEN");
f.WriteLine;
f.WriteTab; f.WriteTab;

```

```

f.WriteString('f.WriteString("No records found"); ');
f.WriteLine;
f.WriteTab; f.WriteTab;
f.WriteString('RETURN; ');
f.WriteLine;
f.WriteTab;
f.WriteString('END; (* check for no records read *) ');
f.WriteLine;

CreatePrintDateTime;
CreatePrintHeadings;

f.WriteTab;
f.WriteString('recnt := 0; ');
f.WriteLine;
f.WriteTab;
f.WriteString('WHILE recnt <= '); f.WriteString(database); f.WriteString(".rows -1 DO ");
f.WriteLine;
f.WriteTab;f.WriteTab;
f.WriteString(database); f.WriteString(".Read(recnt, "); f.WriteString(lc.tablename);
f.WriteString(");");
f.WriteLine;

CreatePrintStatements;

f.WriteTab;f.WriteTab;
f.WriteString('INC(recnt); ');
f.WriteLine;
f.WriteTab;
f.WriteString('END; (* while loop *) ');
f.WriteLine;
f.WriteTab;
f.WriteString('f.WriteString(" ***** End Report ***** "); ');
f.WriteLine;
f.WriteTab;
f.WriteString('v := TextViews.dir.New(t); ');
f.WriteLine;
f.WriteTab;
f.WriteString('Views.OpenAux(v, " '); f.WriteString(lc.tablename$); f.WriteString(' List "); ');
f.WriteLine;

f.WriteString('END PrintAll; ');
f.WriteLine;

```

```

END CreatePrintAll;
(* ===== *)

PROCEDURE CreatePreamble(IN tablename, reportname : ARRAY OF CHAR);
VAR
    tmp1 : ARRAY 64 OF CHAR;
BEGIN
    tmp1 := "defaultreport";
    IF reportname$ # "" THEN
        tmp1 := reportname$;
    END;

    f.WriteString("MODULE Reports");
    f.WriteString(tmp1$ + ";");
    f.WriteLine;

    f.WriteString("IMPORT Dialog, Dates, Strings, SqlDB, SqlControls, Ports, Views,");
    f.WriteLine;
    f.WriteTab;    f.WriteTab;
    f.WriteString("TextModels, TextControllers, TextMappers, TextRulers, TextViews; ");
    f.WriteLine;

    f.WriteString("VAR");
    f.WriteLine;
    f.WriteString("conn : RECORD (* connect to database variables *)");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("id*,");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("password*,");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("database*,");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("driver* : ARRAY 32 OF CHAR;");
    f.WriteLine;
    f.WriteTab;
    f.WriteString("END;");
    f.WriteLine; f.WriteLine;

    f.WriteString(lc.databasesname$); f.WriteString("* : SqlDB.Table;");
    f.WriteString(" (* anchor to Mysql database *)");
    f.WriteLine; f.WriteLine;

```

```
END CreatePreamble;

(* ===== *)
PROCEDURE CreateProgFinal ( IN tablename, reportname : ARRAY OF CHAR );
VAR
    tmp1 : ARRAY 64 OF CHAR;
BEGIN

    tmp1 := "defaultreport";
    IF reportname$ # "" THEN
        tmp1 := reportname$;
    END;

    f.WriteLine;
    f.WriteString("BEGIN");
    f.WriteLine;

    f.WriteTab;
    f.WriteString('conn.id := ""; ');
    f.WriteLine;

    f.WriteTab;
    f.WriteString('conn.password := ""; ');
    f.WriteLine;

    f.WriteTab;
    f.WriteString('conn.database := '); f.WriteString(dlg.aliasname$); f.WriteString('');
    f.WriteLine;

    f.WriteTab;
    f.WriteString('conn.driver := "SqlOdbc"; ');
    f.WriteLine;

    f.WriteTab;
    f.WriteString("OpenSQL;");
    f.WriteLine; f.WriteLine;

    f.WriteString("END Reports"); f.WriteString( tmp1$ ); f.WriteString(".");
    f.WriteLine;

END CreateProgFinal;

(* ===== *)
PROCEDURE CloseSQL*;
```

```

BEGIN
IF Bxtable # NIL THEN
    Bxtable.Clear;
    Bxtable := NIL;
END;

IF BxGenTbl # NIL THEN
    BxGenTbl.Clear;
    BxGenTbl := NIL;
END;

END CloseSQL;
(* ===== *)

PROCEDURE OpenSQL*;
VAR db1, db2 : SqlDB.Database;
    res: INTEGER;
    tmps : ARRAY 5 OF CHAR;
BEGIN

    CloseSQL;

    (* open first database *)
    SqlDB.OpenDatabase(conn.driver, conn.id, conn.password, conn.database,
        SqlDB.async, SqlDB.showErrors, db1, res);

    Strings.IntToString(res, tmps);
    Dialog.ShowStatus("Res db1 is " + tmps);

    IF res = 0 THEN
        Bxtable := db1.NewTable();
    ELSIF res <= 3 THEN
        Dialog.ShowMsg("#Sql:CannotLoadDriver")
    ELSE
        Dialog.ShowMsg("#Sql:ConnectionFailed")
    END;

    (* open second database *)
    SqlDB.OpenDatabase(conn.driver, conn.id, conn.password, conn.database,
        SqlDB.async, SqlDB.showErrors, db2, res);

    Strings.IntToString(res, tmps);
    Dialog.ShowStatus("Res db2 is " + tmps);

    IF res = 0 THEN

```

```

        BxGenTbl := db2.NewTable();
    ELSIF res <= 3 THEN
        Dialog.ShowMsg("#Sql:CannotLoadDriver")
    ELSE
        Dialog.ShowMsg("#Sql:ConnectionFailed")
    END;

END OpenSQL;

(* ===== *)

PROCEDURE executeThis (user_statement: ARRAY OF CHAR);
BEGIN

    IF (user_statement$ # "") & (Bxtable # NIL) THEN
        conn.statement := user_statement$;
        Dialog.Update(conn);
        Bxtable.Exec(user_statement); (* execute the SQL statement entered by the user*)
        Dialog.Update(Bxtable); (* update the SQL table display area *)
    END

END executeThis;

(* ===== *)

PROCEDURE Execute*; (* attached to a button for ahhoc queries *)
BEGIN
    IF Bxtable = NIL THEN
        Dialog.ShowStatus("SQL Database not Opened");
    END;

    executeThis(conn.statement);
    Dialog.Update(conn);
END Execute;

(* ===== *)
PROCEDURE ClearGenArea*;
BEGIN

    BxGenTbl.Exec("DELETE FROM BxFields");
    BxGenTbl.base.Commit;

    setupRulers; (* destroys any currently used textModel area *)

END ClearGenArea;

(* ===== *)

```

```

PROCEDURE ShowDB*; (* press button : display the available database names *)
BEGIN
dbs.databasename := ' '; Dialog.Update(dbs);
tbls.tablename := ' '; Dialog.Update(tbls);

    selectflag := showdb; (* set tablenotifier case switch *)
    Dialog.ShowStatus('Show Databases :' );

    executeThis('SHOW DATABASES');
    Dialog.Update(Bxtable);
END ShowDB;
(* ===== *)

PROCEDURE ShowTables*; (* press button : display the available table for selected database *)
BEGIN

tbls.tablename := ' ';          Dialog.Update(tbls);

IF dbs.databasename$ = ' ' THEN
Dialog.Beep;
    Dialog.ShowStatus('No database selected - click Show DB, then select a Database');
    RETURN;
END;
selectflag := showtbls; (* set tablenotifier case switch *)
    Dialog.ShowStatus('Show Tables for : ' + dbs.databasename$ );
    executeThis('USE ' + dbs.databasename$ );
    executeThis('SHOW TABLES');
    Dialog.Update(Bxtable);
END ShowTables;

(* ===== *)

PROCEDURE DescribeTable*; (* press button : display the fields of the selected table *)
VAR
    row : INTEGER;

BEGIN
IF tbls.tablename$ = ' ' THEN
Dialog.Beep;
    Dialog.ShowStatus('No Table selected - click Show Tables, then select a Table Name');
    RETURN;
END;
selectflag := describetbl; (* set tablenotifier case switch *)
    Dialog.ShowStatus('Describe Tables :' );

```

```

executeThis('DESCRIBE ' + tbls.tablename$ );

Dialog.Update(Bxtable);

END DescribeTable;

(* ===== *)
PROCEDURE DisplaySqlFields*;
BEGIN
IF tbls.tablename$ = ' ' THEN
Dialog.Beep;
    Dialog.ShowStatus('No Table selected - click Show Tables, then select a Table Name');
    RETURN;
END;
    BxGenTbl.Exec('SELECT * FROM Bxfields' );
(*    BxGenTbl.Exec('SELECT oberonfieldname, reportfieldflag FROM Bxfields' ); *)

Dialog.Update(BxGenTbl);
(* note: this SELECT displays in own SQL Table area and does
    not need a case switch to be set see; BxGenTblNotifier *)

END DisplaySqlFields;

(* ===== *)
PROCEDURE GenerateReport*;
(* generate a Oberon Record Structure *)
CONST
    cm = 10 * Ports.mm; (* universal units *)
VAR
    reccnt : INTEGER;
    x ,y, z : INTEGER;
    tmp1      : ARRAY 64 OF CHAR;
BEGIN

tmp1 := "";

    setupRulers; (* destroys any currently used textModel area *)

Strings.ToLower( tbls.tablename$, lc.tablename);

Strings.ToLower( dbs.databasename$, lc.databasename);
    lc.databasename := lc.databasename + "DB";
    lc.reportname := dlg.reportname$;

Dialog.ShowStatus('Generate Structure for :'+ lc.tablename$);

```



```

(* ===== generate print code here ===== *)

BxGenTbl.Exec("SELECT * FROM BxFields ");
    (* select all the records in Bxfields, later code can now simply
       scan over these records using BxGenTbl.Read(row,record) statements *)

IF BxGenTbl.rows = 0 THEN
    StdLog.String("No records found ");
    Dialog.ShowStatus("No records found");
    RETURN;
END;

(* ===== Generate code procedures ===== *)
CreatePreamble ( lc.tablename, lc.reportname );
CreateSqlRecord ( lc.tablename );
CreateSqlClose ( lc.databasesname );
CreateSqlOpen ( lc.databasesname );
CreatePrintAll ( lc.databasesname, lc.tablename );
CreateProgFinal ( lc.tablename, lc.reportname );

v:= TextViews.dir.New(t);    (* create a text view for the generated text t, above *)
Views.OpenAux( v , "BxRepGen Field List");
                                (* open the text view in its own window *)

END GenerateReport;

(* ===== *)

PROCEDURE Commit*;
BEGIN
    Dialog.ShowStatus('Commit:');
    IF Bxtable # NIL THEN
        Bxtable.base.Commit
    END
END Commit;

(* ===== *)

PROCEDURE Clear*;
BEGIN
    Dialog.ShowStatus('Clear:');
    conn.statement := " ";
    Dialog.Update(conn);
END Clear;

(* =====*)

```

```

PROCEDURE TableNotifier* ( t : SqlDB.Table ;
                           row, col : INTEGER;
                           modifier : SET);
(* this procedure executed whenever a user mouse 'clicks' in a SQL table field,
the value of row & col will be set to the clicked on row/col number, counting
from 0 *)
VAR
  tmp1, tmp2 : ARRAY 5 OF CHAR;
  ans : SqlDB.Row;

  reccnt : INTEGER;

BEGIN

CASE selectflag OF (* the case is controlled by selectflag which is set by the
                    appropriate button press on the form *)
  | showdb :

      t.Read(row, dbs );    (* read the database name *)

  | showtbls :

      t.Read(row, tbls );    (* read the table name *)

  | describetbl :

      t.Read (row , describeflds); (* read the table descriptor *)

  Bxfields.mysqltablename      := tbls.tablename$;
  Bxfields.fieldseq             := row;
  Bxfields.reportfieldflag      := "N";
  Bxfields.mysqlfieldname      := describeflds.field$;
  Strings.ToLower( describeflds.field$, Bxfields.oberonfieldname);
  Bxfields.mysqlfieldtype      := describeflds.type$;

  DcsUtils.ConvertMySqlTypeToOberon ( Bxfields.mysqlfieldtype,
                                       Bxfields.oberonfieldtype,
                                       Bxfields.oberonfieldcode,
                                       Bxfields.oberonfieldsizem,
                                       Bxfields.oberonfieldsized );

  Bxfields.comments            := " ";
  BxGenTbl.Exec("DELETE FROM Bxfields WHERE fieldseq = :Bxrepgen.Bxfields.fieldseq");
  BxGenTbl.Exec("INSERT INTO Bxfields VALUES ( :Bxrepgen.Bxfields )");

```

```

        BxGenTbl.base.Commit;

        END; (* case *)

END TableNotifier;
(* ===== *)
PROCEDURE BxGenTblNotifier*( t : SqlDB.Table ;
                             row, col : INTEGER;
                             modifier : SET);
BEGIN
    t.Read (row , Bxfields);

    IF Bxfields.reportfieldflag = 'N' THEN
        BxGenTbl.Exec('UPDATE Bxfields SET reportfieldflag = "Y", oberonfieldheading = :Bxrep-
gen.reportfieldheading, reportfieldwidth = :Bxrepgen.reportfieldwidth WHERE fieldseq = :Bxrep-
gen.Bxfields.fieldseq');
    ELSE
        BxGenTbl.Exec('UPDATE Bxfields SET reportfieldflag = "N", oberonfieldheading = :Bxrep-
gen.reportfieldheading, reportfieldwidth = :Bxrepgen.reportfieldwidth WHERE fieldseq = :Bxrep-
gen.Bxfields.fieldseq');
    END;

    BxGenTbl.base.Commit;

    DisplaySqlFields; (* redisplay to show updated field changes *)

END BxGenTblNotifier;

(* ===== *)

BEGIN
    (* ===== set up defaults in form ===== *)
    conn.id := "";
    conn.password := "";
    conn.database := "BxRepGen";
    conn.driver := "SqlOdbc";

    Dialog.ShowStatus("Blackbox Report Generator ver 1.0");
    Dialog.Update(conn);
    OpenSQL;

END Bxrepgen.

```

Figure 8.6: The BxRepGen program

Chapter 9

‘Karin’ a small Blackbox Oberon & MySQL system

This small example Blackbox Oberon system, called karin, is a complete working Blackbox Oberon program, it consists of a VDU Form, its Blackbox Oberon MODULE code, a Blackbox Oberon Report program and its report output.



The screenshot shows a window titled "Enter Karin Receipts" with a blue title bar. Inside the window, the text "Load Karins Receipts" is displayed. Below this, there are five input fields: "Date" with a date picker showing "11/03/2001", "Rec. Number" with a text box containing "R999", "Supplier" with a text box containing "Dragon City", "Description" with a text box containing "Services", and "Value" with a text box containing "456.29". At the bottom of the window, there are three buttons: "Save", "Print", and "Cancel".

Figure 9.1: Karin : collecting the receipt data form

9.1 The data collecting program

This program was written for Karin to simply enter her taxation receipts into some form of database so that simple reports could be printed to aid in the filling out of her taxation forms. This system does that job and that job alone, a case of the KISS principle¹ at work, the Form simply collects the data with no allowance for Browsing, Deleting, Editing the data entered – just collect it.²

```

MODULE KarinReceipts;
IMPORT Dialog, Dates, Strings, SqlDB, SqlControls, Ports, Views,
      TextModels, TextControllers, TextMappers, TextRulers, TextViews;
VAR
conn : RECORD (* connect to database variables *)
  id*,
  password*,
  database*,
  driver*      : ARRAY 32 OF CHAR;
END;

karinDB* : SqlDB.Table; (* anchor to Mysql database *)

receipts* : RECORD (* Oberon MySQL record area *)
  recdate* : Dates.Date;
  supplier* : ARRAY 40 OF CHAR;
  goods* : ARRAY 30 OF CHAR;
  total* : Dialog.Currency;
  seq- : INTEGER;
  recnumber* : ARRAY 20 OF CHAR;
END;
(* ===== *)

```

The above code defines the RECORD structures used by Blackbox Oberon MySQL interaction via ODBC, *conn* : RECORD has its values set prior to attempting to calling OpenSQL, these values are initialised by the code at the very bottom of this program just above "END KarinReceipts."

The *receipts* : RECORD is the record area that MySQL will READ a single database record into as the table is processed. This RECORD must match exactly the data types, but not necessarily the *variable names* as defined in the MySQL table definition.

¹Keep It Simple - Stupid !

²the complete system took about 30 minutes to create, ie: define the MySQL Database, define the Tables, setup a ODBC entry, write the data entry module and create the Form, then generate a report using my Blackbox Oberon Report generator to build the base report and modify the report, make a menu - finished

```

PROCEDURE CloseSQL;
BEGIN
  IF karinDB # NIL THEN
    karinDB.Clear;
    karinDB := NIL;
  END;
END CloseSQL;
(* ===== *)

PROCEDURE OpenSQL;
VAR db: SqlDB.Database; (* variable db cannot be a global variable ! *)
    res: INTEGER;
BEGIN
  CloseSQL;
  SqlDB.OpenDatabase(conn.driver,conn.id,conn.password,conn.database,
                    SqlDB.async, SqlDB.showErrors, db, res);
  (* open the database using variables setup in record conn *)
  IF res = 0 THEN
    karinDB := db.NewTable()
  ELSIF res <= 3 THEN
    Dialog.ShowMsg("#Sql:CannotLoadDriver")
  ELSE
    Dialog.ShowMsg("#Sql:ConnectionFailed")
  END
END OpenSQL;
(* ===== *)

```

The above code is typical of standard MySQL Open & Close operations, with some minimal error checking, notice the use of the data defined in the *conn : RECORD.*, esp in the *SqlDB.OpenDatabase* statement.

```

PROCEDURE ClearRec;
BEGIN
  receipts.supplier := " ";
  receipts.goods := " ";
  receipts.total.val := 0;
  receipts.total.scale := 2;
  receipts.recnumber := " ";

  Dialog.Update(receipts);
END ClearRec;

(* ===== *)

```

The above code clears the Forms fields and then updates the displayed image.

```

PROCEDURE Save*;
BEGIN
    karinDB.base.Exec("INSERT INTO RECEIPTS VALUES (:KarinReceipts.receipts)" );
        (* insert the complete receipts record *)
    karinDB.base.Commit();
        (* ensure record is written *)
    ClearRec;

END Save;
(* ===== *)

```

The above code is *Linked* to the Button **Save**, using the Object Inspector when creating the Form on page 133, and is executed when the user presses (clicks on) the **Save** button.

The code simply inserts the current form data into the table, clears the form fields then returns to the Oberon 'wait' loop.

```

BEGIN
    conn.id := "";
    conn.password := "";
    conn.database := "karinMYSQL";
    conn.driver := "SqlOdbc";
    OpenSQL;
CLOSE
    CloseSQL;
END KarinReceipts.

```

Figure 9.2: Karin : data collection program dissected

The above code is executed once when the program is executed and sets up the necessary values for MySQL ODBC connections and access to be made.

9.2 The report output

The following screen printout show the receipt records up to the one just entered by the data entry screen³, this neatly illustrates the multi-tasking of MySQL with the Blackbox Oberon program, as soon as the record is committed to MySQL its available for access by other programs, eg: the report program ⁴

Rec Date	Rec #	Supplier	Goods	Total	Seq#
2/02/2002	123	X	xxx	12.23	1
2/02/2002	321	XX	XXX	34.56	2
2/02/2002	231	xxx	XXXXX	1.01	3
2/02/2002	342	xxx	XXX	10.19	4
2/02/2002	666	xxx	xxxxx	0.01	5
12/02/2002	456	max	sleep	33.00	6
12/02/2002	2134	max	stuff	12.12	7
12/02/2002	666	Max	medicine	55.79	8
11/03/2001	R999	Dragon City	Services	456.29	11
				615.20	
**** End Report ****					

Figure 9.3: Karin : reporting the collected receipt data

9.2.1 The report program code

The report program was generate by a small program I wrote, BxrepGen, which allows me to choose MySQL Databases, Tables and Fields , attach Column Headings, and create the necessary Blackbox Oberon code to have a complete simple report running in a few minutes. Most developers would probably use report programs like Crystal Reports⁵,but I preferred to keep my systems in the Blackbox Oberon framework where possible, without ignoring the advantages of using a powerful tool like Crystal Reports.

The main advantage of using BxRepGen is that I do not have to be concerned in remembering⁶ to include the necessary code required to assemble a working report, all the SQL connection code, the Blackbox Oberon to SQL record structure, and of course the record handling logic to select data from the database. The automatic generation of code to drive the Blackbox Oberon

³actually the screen shows just *before* inserting the record displayed , as pressing the Save button clears the screen immediately after the insertion!

⁴I have seen too many SQL Client/Server systems that use complex buffering schemes which hold records in memory so you cant be *sure* you are reporting the latest records eg: Delphi & Interbase!

⁵a vastly more powerful and complex tool than BxRepGen!

⁶I found writing report programs from scratch was always more time consuming than I felt was necessary - hence BxRepGen

TextModels, TextMappers, TextRulers, and TextView interface Modules certainly makes report creation very easy. From this base report I add specialized code to product the exact report required.

It should be noted that there are lots of features not included in BxRepGen , eg: control breaks, header & footers, multi-table selects, and hundreds of features that Crystal Reports offers, however BxRepGen is still a useful tool.⁷

9.2.2 The generated report program explained

The following section is the complete report program which produced the report displayed on 9.2. Each section of the program is explained, with the exception of a few lines extra code the complete program was generated by BxRepGen .

```
MODULE KarinReport;
IMPORT Dialog, Dates, Strings, SqlDB, SqlControls, Ports, Views,
      TextModels, TextControllers, TextMappers, TextRulers, TextViews;
VAR
conn : RECORD (* connect to database variables *)
  id*,
  password*,
  database*,
  driver* : ARRAY 32 OF CHAR;
END;

karinDB* : SqlDB.Table; (* anchor to Mysql database *)

receipts : RECORD (* Oberon MySQL record area *)
  recdate* : Dates.Date;
  supplier* : ARRAY 40 OF CHAR;
  goods* : ARRAY 30 OF CHAR;
  total* : REAL;
  seq* : INTEGER;
  recnumber* : ARRAY 20 OF CHAR;
END;
(* ===== *)
```

The above code defines the RECORD structures used by Blackbox Oberon MySQL interaction via ODBC, *conn* : RECORD has its values set prior to attempting to calling OpenSQL, these values are initialised by the code at the very bottom of this program just above "END KarinReport."

The *receipts* : RECORD is the record area that MySQL will READ a single database record into as the table is processed. This RECORD must match exactly the data types as defined in the MySQL table definition, naturally BxRepGen does this as automatically.

⁷the generated report is a *complete* Blackbox Oberon Module that can be easily modified by the user as BxRepGen creates Blackbox Oberon source code

```

PROCEDURE CloseSQL;
BEGIN
  IF karinDB # NIL THEN
    karinDB.Clear;
    karinDB := NIL;
  END;
END CloseSQL;
(* ===== *)

PROCEDURE OpenSQL;
VAR db: SqlDB.Database; (* variable db cannot be a global variable ! *)
    res: INTEGER;
BEGIN
  CloseSQL;
  SqlDB.OpenDatabase(conn.driver,conn.id,conn.password,conn.database,
                    SqlDB.async, SqlDB.showErrors, db, res);
  (* open the database using variables setup in record conn *)
  IF res = 0 THEN
    karinDB := db.NewTable()
  ELSIF res <= 3 THEN
    Dialog.ShowMsg("#Sql:CannotLoadDriver")
  ELSE
    Dialog.ShowMsg("#Sql:ConnectionFailed")
  END
END OpenSQL;

```

The above code is typical of standard MySQL Open & Close operations, with some minimal error checking, notice the use of the data defined in the *conn : RECORD.*, esp in the *SqlDB.OpenDatabase* statement.

```

(* ===== *)
PROCEDURE Tabs( f :TextMappers.Formatter; notabs : INTEGER );
VAR
  cnt : INTEGER;
BEGIN
  FOR cnt := 1 TO notabs DO f.WriteTab; END;
END Tabs;

```

The Procedure *Tabs*, above, is a simple routine that is used to position the printing "cursor" position when writing to the report output *TEXTVIEW*. Very similar to *TAB* stops in a *WordProcessor* or *Typewriter*.

```
(* ===== *)
PROCEDURE PrintAll*;
CONST
  cm = 10 * Ports.mm; (* universal units *)
VAR
  t : TextModels.Model; f : TextMappers.Formatter; v : TextViews.View;
  r : TextRulers.Ruler;
  recnt : INTEGER;
  date : Dates.Date;
  now : Dates.Time;
  sdate : ARRAY 40 OF CHAR;
  stime : ARRAY 16 OF CHAR;

  reptotal : REAL;
  stmp : ARRAY 15 OF CHAR;
```

The above code is the start of PrintAll procedure and defines the local variables used by the report.

Notice the variables defined as t,f,v & r these are variables that *link* the parts of the Blackbox Oberon Text subsystem ie: t is a TextModel, f the Formatter which connects to t, and v the View which allows the report output to be seen (and printed). r is a TextRuler.(see below)

```
BEGIN
  t := TextModels.dir.New();
  f.ConnectTo(t);
  r := TextRulers.dir.New(NIL);

  TextRulers.AddTab( r , 25 * Ports.mm );
  TextRulers.MakeRightTab(r); (* make previous TAB stop right justified *)
  TextRulers.AddTab( r , 50 * Ports.mm );
  TextRulers.MakeRightTab(r); (* make previous TAB stop right justified *)
  TextRulers.AddTab( r , 60 * Ports.mm );
  TextRulers.AddTab( r , 90 * Ports.mm );
  TextRulers.AddTab( r , 140 * Ports.mm );
  TextRulers.MakeRightTab(r); (* make previous TAB stop right justified *)
  TextRulers.AddTab( r , 150 * Ports.mm );

  f.WriteView(r);
```

The above code creates the TextModel *t* and *f* connected to *t*, also a TextRuler *r*. The *AddTab* code places a TAB position into the TextRuler, the *MakeRightTab* code set the current *AddTab* as *right justified output*. This is the easiest way to line up Currency fields. Note that *left justified alignment* from the tab is the default

The *f.WriteView(r);* code places the TABS onto the output VIEW, or on the printed page if you prefer.

```

karinDB.Exec("SELECT * FROM receipts");
IF karinDB.rows = 0 THEN
    f.WriteString("No records found");
    RETURN;
END; (* check for no records read *)

```

The above code makes the actual call to MySQL to find and return the records need from the table *receipts* held in database Karin. The code also check if no records were found and if so stops the report

```

f.WriteString("Karins Receipts ");
Tabs(f,3);
Dates.GetDate( date);
Dates.DateToString(date, 1, sdate );
f.WriteString(sdate$ + " ");
Dates.GetTime( now );
Dates.TimeToString(now, stime );
f.WriteString( stime$ );

```

The above code writes the report heading, extracts and reformats the system Date & Time. Notice the conversion process required to transform the actual system Date & Time values into CHAR for printing. We have to get the value, convert it to a CHAR form and then we can print it.

```

f.WriteLine; f.WriteLine;
f.WriteTab;
f.WriteString("Rec Date");
f.WriteTab;
f.WriteString("Rec #");
f.WriteTab;
f.WriteString("Supplier");
f.WriteTab;
f.WriteString("Goods");
f.WriteTab;
f.WriteString("Total");
f.WriteTab;
f.WriteString("Seq#");

f.WriteLine; f.WriteLine;

```

The above code prints the column headings, the actual texts were entered when building the report with BxRepGen . Note the use of the WriteTab which will ensures correct placement of the heading above the data values to be printed later. (The heading and data printing will use the same Tab positioning to achieve this.)

```

reccnt := 0; reptotal := 0;
WHILE reccnt <= karinDB.rows -1 DO
    karinDB.Read(reccnt, receipts);

    f.WriteTab;
    Dates.DateToString(receipts.reccdate, 0, sdate );
    f.WriteString(sdate$ );

    f.WriteTab;
    f.WriteString(receipts.recnumber$);

    f.WriteTab;
    f.WriteString(receipts.supplier$);

    f.WriteTab;
    f.WriteString(receipts.goods$);

    f.WriteTab;
    f.WriteRealForm(receipts.total,7,11,-2,' ');

    f.WriteTab;
    f.WriteIntForm(receipts.seq, 10, 6, ' ', FALSE);

    f.WriteLn;
    INC(reccnt);
    reptotal := reptotal + receipts.total;
    (* accumulate report total *)
END; (* while loop *)
f.WriteLn;

Tabs(f, 5);
f.WriteRealForm(reptotal,7,11,-2,' ');
f.WriteLn;
f.WriteString(" **** End Report **** ");
v := TextViews.dir.New(t);
Views.OpenAux(v, " receipts List ");
END PrintAll;

```

The above code is the actual 'driving loop' which reads every record returned by the SQL SELECT above. As each record is printed, it is counted & a total accumulated of receipts.total. The lines of code in **bold** were the lines added by me after BxRepGen had completed its generation task.

BxRepGen automatically generates the correct *Write* call for the particular data type for the variable returned from MySQL. The WriteRealForm or WriteIntForm procedures are generated as they offer the most control over the output format.⁸

⁸According to Warford setting up these Write parameters is cause for much confusion to beginner Blackbox Oberon programmer

```
(* ===== *)  
BEGIN  
  conn.id := "";  
  conn.password := "";  
  conn.database := "karinMYSQL";  
  conn.driver := "SqlOdbc";  
  OpenSQL;  
  
END KarinReport.
```

The above code is executed once when the report is executed and it sets up the necessary values for MySQL ODBC connections and access to be made

Figure 9.4: Karin : report program dissected

Chapter 10

Modal Form Execution

Although Blackbox Oberon is a *non-modal* system, in that almost every GUI form displayed can be 'put aside' by the user¹ and returned to when desired. This is particularly nice way for the user to operate as they can select other programs from the menu, use them and then return to exactly where they were before selecting the other program, without loss of previously entered data.

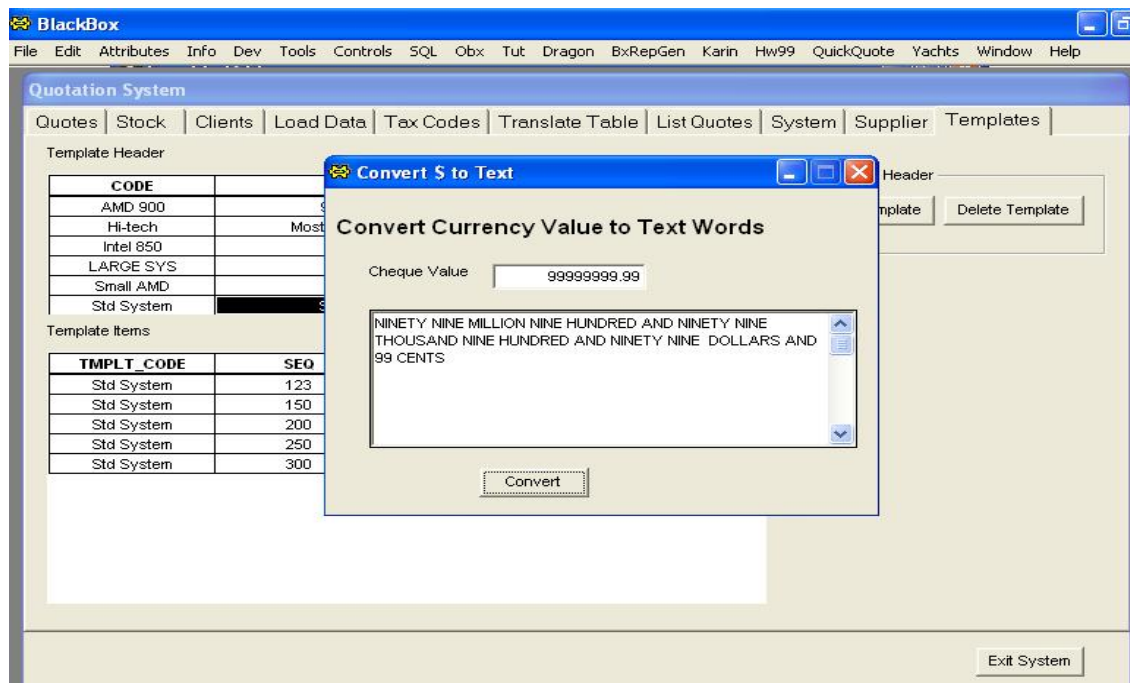


Figure 10.1: Showing two separate program executing (non-modal)

In the above example the user was running the *QuickQuote system*, and simply executed the *Cheque Conversion* program by selecting it from a menu, *QuickQuotes* form display is not effected by this other execution.

In fact Blackbox Oberon has no standard method of making forms modal. ie: writing your own form that when called will expect the calling form to wait for the called form to return.

¹some exceptions are Operating System Dialogs where the user is *expected* to make a choice before continuing procesing ie: select a input file dialog, chose a printer, etc, where modal operation make sense.

However it is possible without too much effort to achieve this.

The following example is extracted from a larger Blackbox Oberon program, called Quick-Quote, that allows the user to create a new 'Template Header' record by calling a data entry form from the currently displayed form by pressing the **Create Template** button, the caller *appears* to wait for the data entry form to return before continuing processing the returned data, ie: like a modal form.

This is not the case in reality, however the effect is similar.

10.1 Snapshots of MODAL operation

The screenshot shows a window titled "Quotation System" with a menu bar containing: Quotes, Stock, Clients, Load Data, Tax Codes, Translate Table, List Quotes, System, Supplier, and Templates. The "Templates" menu is active, displaying two sections:

Template Header

CODE	DESCRIPTION
AMD 900	900 mHz Cpu based system
Hi-tech	Most currenty powerful configuration
Intel 850	850 mHz based system
LARGE SYS	LARGE POWER SYSTEM
Small AMD	Started configuration
Std System	Standard small configuration

To the right of the Template Header table is an "Edit Template Header" box containing "Create Template" and "Delete Template" buttons.

Template Items

TMPLT_CODE	SEQ	STOCK_CODE	QTY_REQ
Std System	123	CP-IBM-P233MHZ	0
Std System	150	VC-AGP-TRI-4MB	0
Std System	200	CC-SCSI-1505A	0
Std System	250	WMS61105	0
Std System	300	MO-SAMSUNG-700S	0

At the bottom right of the window is an "Exit System" button.

Figure 10.2: Before pressing **Create Template** Button

The screenshot shows the 'Quotation System' application window. The 'Templates' tab is active. A modal dialog box titled 'Create Template Header' is open in the center. The dialog has two input fields: 'Template Code' with the text 'LARGE SYS' and 'Description' with the text 'LARGE CONFIGURATION'. There are 'OK' and 'Cancel' buttons at the bottom of the dialog. In the background, the 'Template Header' table is visible with columns 'CODE' and 'DESCRIPTION'. It contains entries for AMD 850, AMD 900, Hi-tech, Intel 850, Small AMD, and Std System. To the right of the table are buttons for 'Edit Template Header', 'Create Template', and 'Delete Template'. At the bottom right of the main window is an 'Exit System' button.

Figure 10.3: First form waits for user to fill in second form

The screenshot shows the 'Quotation System' application window after the dialog box has been closed. The 'Template Header' table now includes a new entry: 'LARGE SYS' with the description 'LARGE CONFIGURATION'. The 'Template Items' table below it has columns 'TMPLT_CODE', 'SEQ', 'STOCK_CODE', and 'QTY_REQ'. The 'Exit System' button remains at the bottom right.

Figure 10.4: Upon return from second form, after pressing **OK** button

Notice the new entry in the top MySQL table 'LARGE SYS', which was passed back to the caller from the called data entry form for insertion into the Template table.

10.2 How was it done?

The modal effect was achieved by using the Blackbox Oberon event loop to our advantage. If you study the supplied Blackbox Oberon documentation you will see mention to an *event loop*, this is an internal loop coded to wait for an event to happen, and then react to that event, ie; keyboard press, mouse click, etc,

What this means to your program in effect, is that after executing a Procedure, either system or user supplied, Blackbox Oberon *waits* for the next event to happen before continuing processing, so therefore if the **last** command in a procedure is to call another form the calling program will in effect *wait* !.

When the user pressed `Create Template`, the called form is displayed, the user enters data into the displayed data entry fields, which are linked to fields defined in the caller

eg: `QuickQuote.templateParas.CODE`, `QuickQuote.templateparas.DESC`.

User presses `OK` button which executes the procedure, `templateOK*`, in the caller, and then closes the called form. Note that all the procedures are defined in the caller (`QuickQuote`), or are Blackbox Oberon supplied standard procedures, ie: `StdCmds.CloseDialog`, `QuickQuoteTemplate` is only the called form, and has no program code.

The `OK` button is defined as :

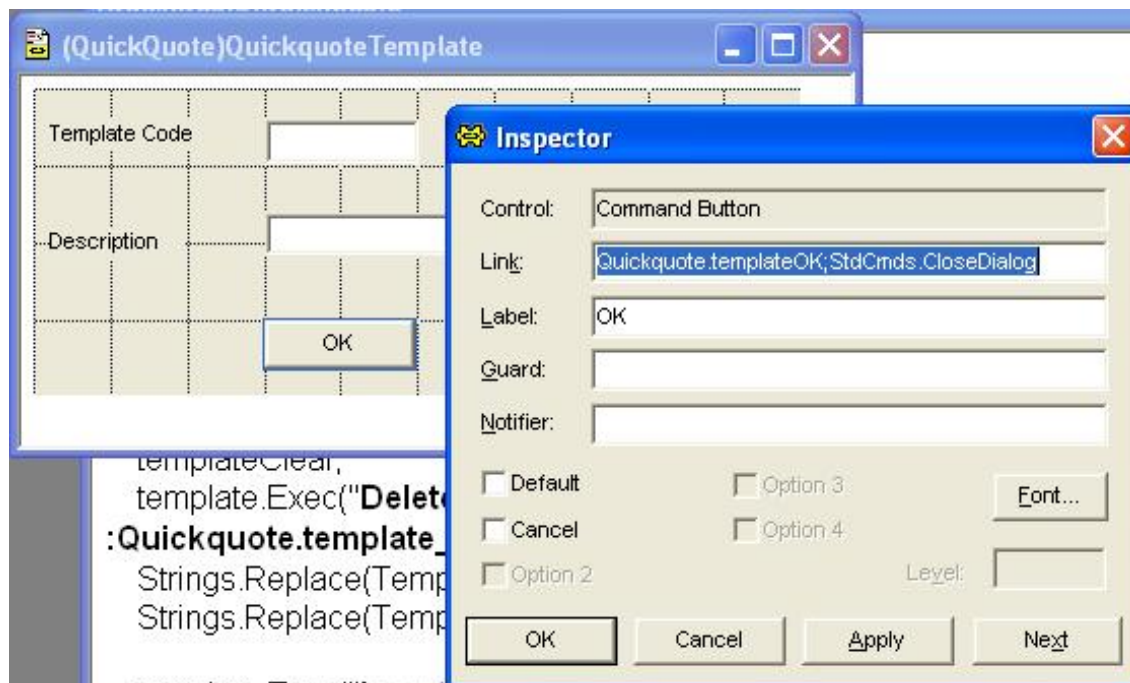


Figure 10.5: Definition of `OK` Button

The 'trick' is to get the called form to execute a procedure in the caller, and then close the called form,

Link: `Quickquote.templateOK;StdCmds.CloseDialog`

Blackbox Oberon does not know that its been tricked into processing a form in a 'modal' fashion !

The `Cancel` button simply excutes a

Link: StdCmds.CloseDialog

command and Blackbox Oberon simply closes the form, and enters the *event loop*. The calling program does not have to check for any data entry, rejected usage of the form, etc, as the forms are actually fully non-modal and considered by Blackbox Oberon to be separate. They just appear to be modal forms to the end-user.

10.2.1 QuickQuote : Template Procedures

```
(* ===== Template procedures ===== *)
PROCEDURE templateClear;
BEGIN
  Template.CODE := " ";
  Template.DESCR := " ";
END templateClear;
(* ===== *)
PROCEDURE templateReload;
BEGIN
  template.Exec("SELECT * FROM TEMPLATE order by CODE");
END templateReload;
(* ===== *)
PROCEDURE templateCreate*;
BEGIN

template_paras.flag := FALSE;
template_paras.code := " ";
template_paras.desc := " ";

StdCmds.OpenToolDialog('QuickQuote\Rsrc\QuickquoteTemplate','Create Template Header');
  (* NOTE : the call to QuickquoteTemplate *must* be the last executable statement in this procedure
to make Quickquote & Quickquote Template to act like MODAL forms !!!
  BSH 4/1/02 *)
END templateCreate;
(* ===== *)
PROCEDURE templateDelete*;
BEGIN
  template.Exec("Delete from Template where CODE = :Quickquote.Template.CODE");
  template.base.Commit();
  templateReload;
  Dialog.Update(Template);

END templateDelete;
(* ===== *)
```

continues ↔

```

PROCEDURE templateOK*;
(* templateOK is called from form QuickquoteTemplate OK button to emulate MODAL form actions
between Quickquote & QuickquoteTemplate , BSH 4/1/02*)
BEGIN
    template_paras.flag := TRUE;
    Dialog.ShowStatus("Create NEW template : " + template_paras.code + " lim " + tem-
plate_paras.desc);

    templateClear;
    template.Exec("Delete from template where CODE = :Quickquote.template_paras.code");
    Strings.Replace(Template.CODE,0,LEN( template_paras.code), template_paras.code);
    Strings.Replace(Template.DESCR,0,LEN( template_paras.desc), template_paras.desc);

    template.Exec("Insert into Template VALUES ( :Quickquote.Template)");
    template.base.Commit();
    templateReload;
END templateOK;
(* ===== *)
PROCEDURE templateNotifier* ( t : SqlDB.Table;
                             row, col : INTEGER;
                             modifier : SET);

BEGIN
Strings.IntToString(row, tmp1); Strings.IntToString(col, tmp2);
    Dialog.ShowStatus("Clicked in template : row : " + tmp1 + " col : " + tmp2 );

    template.Read(row, Template); (* read clicked on row *)

    Dialog.ShowStatus("Selected CODE : " + Template.CODE);

    Strings.Extract(Template.CODE,0,LEN(Template.CODE) ,template_paras.code);
    Strings.Extract(Template.DESCR,0,LEN(Template.DESCR),template_paras.desc);

    Dialog.Update(template_paras);

    templitems.Exec("select * from template_items where TMPLT_CODE = :Quick-
quote.Template.CODE ");
    (* read THIS headers ITEMS list from second table *)
    Dialog.Update(templitems); (* update template items display *)
END templateNotifier;
(* ===== *)

```

Chapter 11

Programming with Recursion

Recursive procedures are ones which call themselves from within their own code body and while considered 'elegant' by some computer scientists as it leads to much shorter written code¹, they are often very hard to debug because of the automatic² use of the systems 'stack' as each call is processed to save the current values of variables as the next call is to be processed.³

Most computer books give the example of computing Factorial to show off the recursion technique⁴, however as I have never needed to use a factorial in my programming career the example I will give is a real one from a commercial data processing system I developed many years ago for a client.⁵. Another classic recursion example, with useful application, is the famous QuickSort algorithm which is not only clever but very fast as well. Recursion is also well suited to searching lists & tree structures.

11.1 Printing Cheque Value as Words

My client needed to print cheques in a format acceptable for processing by their Bank and the format had to conform to quite strict layout. The most difficult part being the printing of the Cheques monetary VALUE in Text Words as one would hand write the cheque.

eg:
input : = 20000
output : = "TWENTY THOUSAND DOLLARS AND 0 CENTS"

input : 20001.99
output : "TWENTY THOUSAND AND ONE DOLLARS AND 99 CENTS"

and NOT
input : 345.00
output : "THREE FOUR FIVE 00 CENTS"⁶

¹because *you* are calling *yourself* and *you* already have been written !

²this is the *clever* part of recursion

³Warford's book see 1.2 has a *very* good chapter on recursion and its mechanics, so I'll assume you've read that

⁴boring!

⁵In Paradox for Windows, based upon a AWK program example published in "The AWK Programming Language" by Aho, Kernighan & Weinberger in 1988. My version is more complex due to Australian English being used rather than American English. I translated into Blackbox Oberon without too much effort

Translating a single numeric digit to a text Word is simply looking up an Array , using the numeric digit as the array index, where the real problems occur is the 'correct' grammar and the placement of the 'AND' word and plural meanings, ie ONE DOLLAR not ONE DOLLARS, TWENTY THOUSAND AND ONE DOLLAR not TWENTY THOUSAND ONE DOLLAR, etc.

The main 'trick' to remember with recursive coding is , **there must be a well defined end to the recursion!**. In the case of Factorial when the value 1 is reached, then the calculation is complete, and the program will start returning from each call passing the answer back to the caller. In the example of Cheque Value to Text Words, recursion is complete when we have processed the last dollar unit.

There are two program shown here, a small testing program that calls the NumToWords Procedure in MODULE ChqLib, and Procedure IntoWords which is the recursive procedure and is defined local to MODULE ChqLib and is not accessible to external procedures. This is a typical example of information hiding that MODULES offer the application programmer, only the procedures that the user (in this case another programmer) needs to access are allowed to be seen externally to the MODULE, thus only the procedure NumToWords is available to the caller.

Developing recursive procedures is perhaps more natural to Mathematical Programmers than Commercial Programmers as Mathematicians tend to think in small functions calling other functions. integrations, etc.⁷

As pointed out by Warford recursion can cause large *overheads* in processing time and stack manipulations, however this example does not incur great overheads as the *depth* of the recursion is generally quite small and the number of variables needed to be kept on the stack per recursion is also small. And remember it only the procedures variables, and some system information, that is placed on the stack, not the procedure itself!

⁷so dont waste your time trying to discover a *reason* to use recursion. I recall many years ago I was developing a Text Editor in SIMULA and was coding away and at one point I needed to call a 'goto line' procedure, and I was currently coding that procedure! - a truly recursive happening.


```

MODULE TestDollarLib;
(*
Φ "StdCmds.OpenAuxDialog('Book/Rsrc/TestDollarLib','Convert $ to Text')" ""
)
IMPORT ChqLib, StdLog, Dialog , Math;

VAR
dlg* : RECORD
    cheqvalue* : Dialog.Currency;
    words* : ARRAY 200 OF CHAR;
END;

PROCEDURE Do*;
BEGIN
    ChqLib.NumToWords( dlg.cheqvalue, dlg.words );
    Dialog.Update( dlg );
END Do;

BEGIN
    dlg.cheqvalue.scale := 2;
END TestDollarLib.

```

Figure 11.1: Calling Cheque Conversion Library

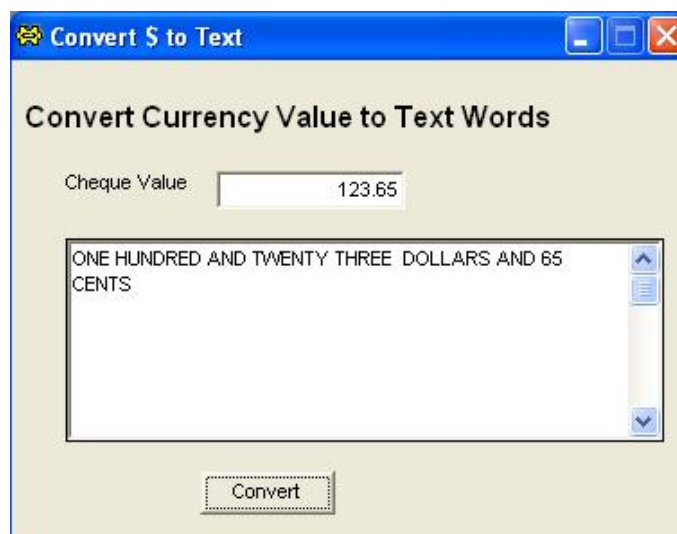


Figure 11.2: Convert \$ value to text Words

The Module above accepts a currency value from the user form, and when Convert button is pressed, calls NumToWords, which will return the cheque value in text variable *words*.

```

MODULE ChqLib;
(* a set of two procedures to convert a Currency value into a string of words
for printing cheques, etc

To use :
    your program must call NumToWords, which sets up the environment for the
    recursive calls to IntoWords to convert the CURRENCY value into TEXT WORDS
    suitable for printing on a Cheque ( for example).

Note: the the output Text Words is in the style of a typical handwritten cheque,
with correct grammer, not just the subsitution by character position.

Typical results:
    eg:
        input : (dialog.Currency) = 20000
        output : (string) = "TWENTY THOUSAND DOLLARS AND 0 CENTS"

        input : (dialog.Currency) = 20001
        output : (string) = "TWENTY THOUSAND AND ONE DOLLARS AND 0 CENTS"

        input : 20001.99
        output : "TWENTY THOUSAND AND ONE DOLLARS AND 99 CENTS"

and NOT
        input : 345.00
        output : "THREE FOUR FIVE 00 CENTS
        as used by some lesser cheque writting algorithms.

*)

IMPORT Dialog, Math, Strings , StdLog;
TYPE
    string = ARRAY 20 OF CHAR;
    bigString = ARRAY 200 OF CHAR; (* build cheque valuse text sentence here *)
VAR

    units : ARRAY 22 OF string; (* 20 entries of 10 characters *)
    tens : ARRAY 12 OF string; (* 10 entries of 10 characters *)
    (* array of numeric word equivalences *)

PROCEDURE IntoWords ( dols : LONGINT; OUT CheqWords : ARRAY OF CHAR );
VAR
    str1, str2 : bigString;
    t_dols, t_rem : LONGINT;
BEGIN

```

Continues ↦

```

str1 := "" ; str2 := "";

IF dols >= 1000000 THEN
  t_dols := ( dols DIV 1000000 );
  t_rem := (dols MOD 1000000 );
  IntoWords( t_dols, str1 );
  IntoWords( t_rem, str2 );

  CheqWords := str1 + "MILLION " + str2 ;
  RETURN
END;

(* process values 1 MILLION DOLLARS and above*)

IF dols >= 100000 THEN
  t_dols := ( dols DIV 100000 );
  t_rem := ( dols MOD 100000 );
  IntoWords( t_dols, str1 );
  IntoWords( t_rem, str2 );
  IF t_rem >= 1000 THEN
    CheqWords := str1 + "HUNDRED AND " + str2 ;
    RETURN;
  ELSE
    IF t_rem > 0 THEN
      CheqWords := str1 + "HUNDRED THOUSAND AND " + str2 ;
      RETURN;
    ELSE
      CheqWords := str1 + "HUNDRED THOUSAND " + str2 ;
      RETURN
    END
  END
END;

(* process values 1 HUNDRED THOUSAND DOLLARS and above *)

IF dols >= 1000 THEN
  t_dols:= ( dols DIV 1000 );
  t_rem := ( dols MOD 1000 );
  IF dols DIV 10000 >= 20 THEN
    Strings.Extract( tens[ (dols DIV 100) +1 ], 0, 99, str1 );
    (* pick up string equivalent of numeric value *)
    IntoWords( t_rem, str2 );
    IF t_rem >= 100 THEN
      CheqWords := str1 + "THOUSAND " + str2;
      RETURN
    ELSE
      IF t_rem > 0 THEN

```

Continues ↦

```

        CheqWords := str1 + "THOUSAND AND " + str2;
        RETURN
    ELSE
        CheqWords := str1 + "THOUSAND " + str2;
        RETURN
    END;
END;
ELSE
    IntoWords( t_dols, str1 );
    IntoWords( t_rem, str2 );
    IF t_rem >= 100 THEN
        CheqWords := str1 + "THOUSAND " + str2;
        RETURN
    ELSE
        IF t_rem > 0 THEN
            CheqWords := str1 + "THOUSAND AND " + str2;
            RETURN
        ELSE
            CheqWords := str1 + "THOUSAND " + str2;
            RETURN
        END;
    END;
END;
END; (* process values 1 THOUSAND DOLLARS and above *)

IF dols >= 100 THEN
    t_dols := ( dols DIV 100 );
    t_rem := ( dols MOD 100 );
    IntoWords( t_dols, str1 );
    IntoWords( t_rem , str2 );
    IF t_rem > 0 THEN
        CheqWords := str1 + "HUNDRED AND " + str2;
        RETURN
    ELSE
        CheqWords := str1 + "HUNDRED " + str2;
        RETURN
    END;
END; (* process values 1 HUNDRED DOLLORS and above *)

IF dols >= 20 THEN
    t_dols := dols DIV 10;
    Strings.Extract(tens[ t_dols + 1 ], 0, 99, str1);
    (* get string name for number *)
    t_rem := (dols MOD 10); IntoWords( t_rem , str2);
    CheqWords := str1 + "" + str2;

```

Continues \mapsto

```

    RETURN
END;

CheqWords := units[dols+1] + "";
RETURN
END IntoWords;

(* ===== *)

PROCEDURE NumToWords* ( dollars : Dialog.Currency ;
OUT dollarsInWords : ARRAY OF CHAR);
VAR
tmpprice : REAL;
int_dollars : LONGINT;
int_cents : LONGINT;
str_cents : ARRAY 4 OF CHAR;

BEGIN

    tmpprice := dollars.val / Math.IntPower (10, dollars.scale);
                                                    (* convert Currency to REAL for processing *)
    int_dollars := ENTIER(tmpprice);
int_cents := dollars.val MOD 100;
                                (* extract dollars & cents *)
    Strings.IntToString(int_cents, str_cents);

    IF int_dollars = 0 THEN
        dollarsInWords:= "ZERO DOLLARS AND " + str_cents$
                        + " CENTS ";
        RETURN
    END;

    IF int_dollars = 1 THEN
dollarsInWords:= "ONE DOLLAR AND " + str_cents$ + " CENTS ";
RETURN
    ELSE
        IntoWords ( int_dollars , dollarsInWords );
        dollarsInWords := dollarsInWords + " DOLLARS AND " + str_cents$ + " CENTS ";
    END;

END NumToWords;

BEGIN
    (* conversion tables loaded ONCE when MODULE first called *)

```

Continues ↪

```
units[1] := " ";
units[2] := "ONE ";
units[3] := "TWO ";
units[4] := "THREE ";
units[5] := "FOUR ";
units[6] := "FIVE ";
units[7] := "SIX ";
units[8] := "SEVEN ";
units[9] := "EIGHT ";
units[10] := "NINE ";
units[11] := "TEN ";
units[12] := "ELEVEN ";
units[13] := "TWELVE ";
units[14] := "THIRTEEN ";
units[15] := "FOURTEEN ";
units[16] := "FIFTEEN ";
units[17] := "SIXTEEN ";
units[18] := "SEVENTEEN ";
units[19] := "EIGHTEEN ";
units[20] := "NINETEEN ";

tens[1] := " ";
tens[2] := "TEN ";
tens[3] := "TWENTY ";
tens[4] := "THIRTY ";
tens[5] := "FORTY ";
tens[6] := "FIFTY ";
tens[7] := "SIXTY ";
tens[8] := "SEVENTY ";
tens[9] := "EIGHTY ";
tens[10] := "NINETY ";
```

```
END ChqLib.
```

Figure 11.3: The Cheque Conversion Module

The CheqLib Module, when called, sets up the text words array, then excutes the NumToWords procedure, this checks for values less then or equal to one (1) dollar and if so proceses that value here and returns to the caller, otherwise the whole dollars value is used in the first call to IntoWords.

IntoWords keeps calling itself recursively, breaking up the whole dollars into Millions, Hundreds of Thousands, Thousands, Hundreds, Tens, Units until the last single digit dollar value is obtained whereby the program starts its return 'cycle' passing back the Text Word for the current single digit.

In the following table, read down the left side columns then back up the right columns to get the *feel* of recursion.

Read Down			Read Up	
Processed By	Input Dollars	Cents	Final Text Words	
NumToWords	1026.34	34	ONE THOUSAND AND TWENTY SIX DOLLARS AND 34 CENTS	
	Whole Dollars	Remainder	Returned Text Words	
↓ IntoWords	1026			↑
↓ IntoWords	1000	26 (dollars)	ONE THOUSAND AND	↑
↓ IntoWords	20	6 (dollars)	TWENTY	↑
↓ IntoWords	6	nothing	SIX	↑

Figure 11.4: Example recursion processing

This type of problem could also, and usually is, be done in an iterative manner, however the use of recursion proved to be the best method. I originally started to develop the program iteratively and it soon became quite difficult to keep track of when to insert the 'AND' word, whereas in the recursive code that decision just *fell into place* ! ,probably reflecting the very essence of recursive technique.

Benefits :

1. recursion leads to a more natural expression of the problem, no need to set up local variables to hold temporary values which can obscure the intent of the code.
2. the code automatically processes the largest values entered, there is no need to keep track of where the code is in the processing cycle, ie: are we processing , hundreds or thousands ? etc.
3. the code automatically continues to call it self until the dollar units are is reached, the code does not know how many digits there is in the number to be converted.
4. the returned text words sentence is dynamically formed as the recursion returns from each call, no need to have special string formatting code to create the sentence.

Disadvantages:

1. more complex to debug initially
2. not everybody understands the recursion principle so maintance may be a problem

Chapter 12

Debugging Oberon Code

Unfortunately programmers are all prone to making errors ¹, and therefore we all need access to techniques of being able to prove that our programs actually produce the results desired.

To this end numerous *debugging* techniques have been developed and these vary considerably depending upon the programming language being used.

Typical debug methods are (not an exhaustive list) :

1. 'trace' code a programmer inserts in the program that prints small bits of information about the current 'state' (values) in selected variables or procedures entered/left. All languages offer this useful technique, however some languages make such a mess of the program output that debugging the output is almost as difficult as debugging the program!. Blackbox Oberon offer the special view - *Log* accessed via module StdLog, that allows the programmer to easily output trace information without disturbing form or report output from the running program. A very useful feature.
2. 'debug windows' that allow the program to execute under control of the programmer who can observe that actual line of code (in text form) and modify variables as the program executes, most modern languages support this style of debugger. Oberon does not, as explained in the Blackbox Oberon documentation, because the very nature of object oriented languages is the internal complexity of the interactions between objects, modules, etc, as the program executes, that following the code is a very difficult exercise.²
3. 'defensive error trapping' debugging via special code that is executed just before the actual procedure code is executed is a technique developed especially for OOPL's and championed in the Eiffel language and also offered in Blackbox Oberon .

¹that means YOU & ME !

²I'd have to agree with this observation, while debugging Delphi code, when written like Pascal and not OOP, the debugger is very useful, however when debugging Delphi OOP code the numerous 'other' procedures that enter the code cycle when stepping thru code becomes a nightmare, not impossible just very difficult.

12.1 Trace Code

The following small module has many lines of debugging code imbedded in the source code, they are commented out for normal processing. However, when activated, this code will produce a small trace script on the StdLog view for inspection every time the procedure is called.

```

MODULE QuickquoteUtils;

IMPORT Dates, Strings, StdLog, SqlDB;

PROCEDURE SqlDate*(OUT ndate :ARRAY OF CHAR);
(* get system date dd/mm/yyyy and return yyyy-mm-dd for MYSQL records *)
VAR
  date : Dates.Date;
  sdate : ARRAY 11 OF CHAR;
  syear : ARRAY 5 OF CHAR;
  smonth : ARRAY 3 OF CHAR;
  sday : ARRAY 3 OF CHAR;

  strt      : INTEGER; (* current FIND start position *)
  fnd       : INTEGER;  (* current position of found '/' in a sdate *)
BEGIN
  Dates.GetDate(date );                                (* get todays date as integer dd mm yyyy *)

  Dates.DateToString(date, Dates.short, sdate);        (* convert to string form dd/mm/yyyy*)
  (** StdLog.String(sdate); (* trace system date returned*) *)

  strt := 0;                                           (* start on first character *)
  Strings.Find(sdate, '/', strt , fnd);                (* look for first / *)
  (** StdLog.Ln; StdLog.Int(strt); StdLog.String(" "); StdLog.Int(fnd); StdLog.Ln; *)
  Strings.Extract(sdate, strt, (fnd - strt), sday);

  strt := fnd+1;                                       (* start 1 char after first / *)
  Strings.Find(sdate, '/', strt, fnd);                 (* look for second / *)
  (** StdLog.Int(strt); StdLog.String(" "); StdLog.Int(fnd); StdLog.Int(fnd - strt); StdLog.Ln; *)
  Strings.Extract(sdate, strt, (fnd - strt) , smonth);

  Strings.Extract(sdate, fnd+1, 4, syear);              (* get yyyy *)
  ndate := syear$ + '-' + smonth$ + '-' + sday$;       (* form yyyy-mm-dd *)
  (** StdLog.String(" " + ndate$); StdLog.Ln; (* trace converted date *) *)
  END SqlDate;
END QuickquoteUtils.

```

Figure 12.1: Debugging using Trace Code

In example the module is converting the standard operating systems time (Windows dd/mm/yyyy in this case) into the yyyy-mm-dd form required for MySQL database records.³ The string extraction process needs to extract and collect the various parts of the date :day , month, year, into sepearate string variables for re-packing into the required format for passing back to the caller.

The code simply scans the text version of the systems date from left to right picking out the parts of the text for dd, mm, yyyy. It does this by using the '/' character as field delimiters to the parts of the date.

The tracing code, if activated, will display the progress of this scanning on the Log file thus :

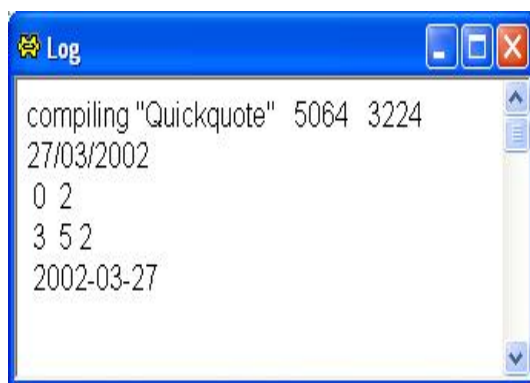


Figure 12.2: Poor Trace Log of Date Conversion

Now, while the above information is useful and correct it suffers from one very important fault. **It does not identify from where it came, nor what the information is !!.**

Better output would be :



Figure 12.3: Better Trace Log of Date Conversion

This output is still quite simple, but it tells us which procedure created this trace, and, the variable names and values, so we can relate this information back to the source code of the procedure being debugged

³this module is NOT the best method of creating the required date format!

Remember that a trace of a variables value, without information about its source in a program of many hundreds of procedures will not be very informative to you as you debug, let alone any programmer that has the misfortune to debug YOUR program in future !

The code below illustrates a better method of manipulating the systems date into the format required.

```

MODULE QuickquoteUtils;

IMPORT Dates, Strings, StdLog, SqlDB;
PROCEDURE SqlDateQuick*(OUT ndate :ARRAY OF CHAR);
(* get system date dd/mm/yyyy and return yyyy-mm-dd for MYSQL records *)
VAR
  date : Dates.Date;
  syear : ARRAY 5 OF CHAR;
  smonth : ARRAY 3 OF CHAR;
  sday : ARRAY 3 OF CHAR;

BEGIN
  Dates.GetDate(date ); (* get todays date as integer dd mm yyyy *)
  Strings.IntToStringForm(date.day, Strings.decimal, 2, '0', FALSE, sday );
  Strings.IntToStringForm(date.month, Strings.decimal, 2, '0', FALSE, smonth );
  Strings.IntToStringForm(date.year, Strings.decimal, 4, '0', FALSE, syear );

  ndate := syear$ + '-' + smonth$ + '-' + sday$; (* form yyyy-mm-dd *)
  END SqlDateQuick;

(* ===== *)

END QuickquoteUtils.

```

Figure 12.4: A better method of system date conversion

Why is it better?. Well,

- (a) its a lot less code !, and simpler to read and understand.
- (b) it uses the `Dates.Date` variable datatype as the direct source of the current date, which can be assumed to be correct⁴
- (c) it uses the standard Oberon systems functions for all the manipulation of the date data and we should be able to rely upon those functions to work correctly
- (d) no pointers to places in a string are used, unlike the first example, page 162 , this reduces the possibiity of programer error.
- (e) no code is required to calculate the length of the extracted string (s), as required when using the `Strings.Extract` procedure calls.

⁴if we can't rely upon Blackbox Oberon to produce correct results from standard functions we should not be using Blackbox Oberon at all

12.2 ASSERT

At first sight the ASSERT seems to be nothing more than a different way of using IF statements to ensure that procedure input parameters values are within a specified range, while this is true, the ASSERT offers more than that in that they are syntacally easier to construct, allow direct connection to a user defined error message and are more easily identified as something special to the reader of the code.