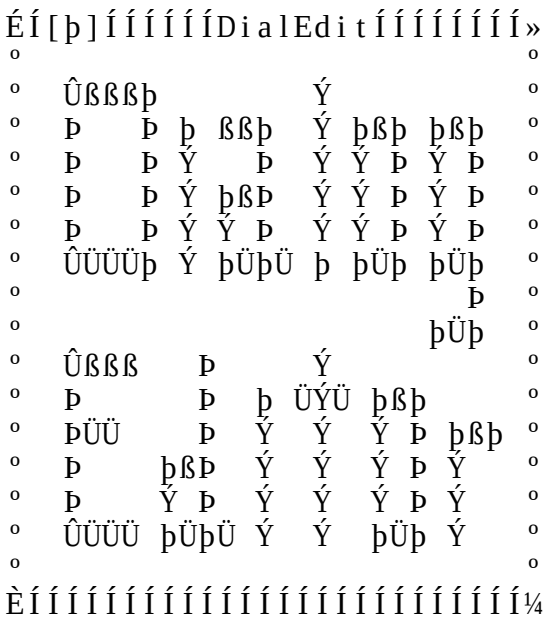
DialEdit
Dialog Editor Manual



A Visual Editor for TurboVision™ Dialogs

Table of Contents / Overview

The Capabilities section briefly describes the main capabilities of the Dialog Editor.

The General Operations section begins by defining basic concepts (terminology subsection). The remaining subsections describe the general processes of designing a dialog. The General Operations section provides details of generic operations that apply to all dialog components. (Thus, these details do not need to be repeated in the Object Dialogs section, which describes operations for individual dialog components.)

The Main Screen / Main Commands section provides an overview of the main commands, available via the Menu Bar (pull-down menus), the Status Line, or the keyboard.

The File Commands section describes the commands available from the File menu.

The Dialog Commands section describes the commands available from the Dialog menu.

The General Editing Features section describes features that are common to most or all of the edit dialogs described later.

The Dialog Dialogs and Component Dialog sections describe the dialogs used to create and edit dialogs and dialog components. The order of presentation of the dialog components is from the simplest to the most complex. In general, for most objects there is a 'new object' command that executes a 'new object' dialog to create and initially define the object. Double-clicking the object executes a similar 'change object' dialog to modify it. These dialogs handle all of the object features except position and size, which are handled by dragging operations described in the General Operation section.

The Notes section has some advice on use of TurboVision objects.

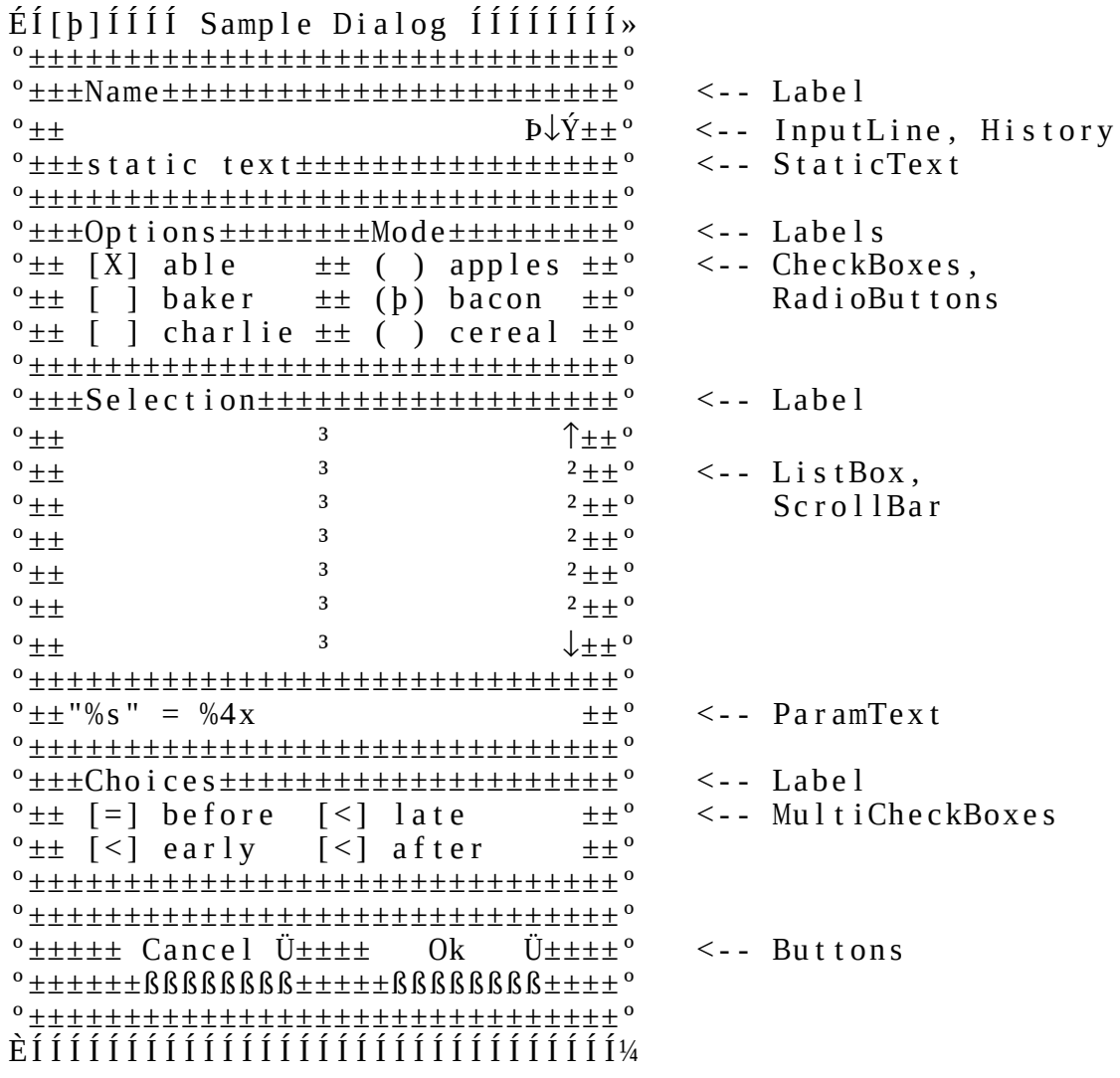
1. Capabilities ==========

The Dialog Editor provides a means to create and edit TurboVision dialogs and similar windows. The layout and visual appearance of these are easily changed by moving and re-sizing the component parts on the screen. The dialogs (or windows) can be saved in a file and retrieved later, and Pascal code can be generated to initialize the dialogs and to describe the associated data records.

The Dialog Editor can be installed in the Tools menu of the Borland Pascal IDE, where the generated Pascal code can be copied into a program and customized or integrated as needed. Or the Dialog Editor can be used alone to generate Pascal source files. [Investigate use with the Tools menu of Virtual Pascal.]

Dialog components may include any of these objects: StaticText, ParamText, Button, InputLine (with optional standard Validator and optional History -- only the History icon is visible), RadioButtons, CheckBoxes, MultiCheckBoxes, and ListBox. Many of these can

have an associated Label object, and the ListBox has an associated vertical ScrollBar object. See sample dialog above.



String data appearing in edit dialogs are saved in a history buffer and history file, allowing previous used data to be retrieved and reused. The default data in edit dialogs can be redefined (also saved in the history file). These features, with variations, can be used to copy and paste object definition data.

2. General Operation

2.1 Terminology

The dialog currently being designed and shown on the screen is called the 'trial dialog', and its components are called 'trial objects', such as trial buttons, trial input lines, etc. Although it looks like a real dialog, it is not executed -- it is just displayed and edited. Only one trial dialog is shown on the screen at a time, but any dialog can be saved in a 'dialog collection' file.

Pascal code can be generated from a trial dialog, and the generated code inserted into your program. The dialog created by the generated code, and its component objects, are called 'real' objects. The 'real' component objects are standard TurboVision objects.

The trial objects shown by the Dialog Editor look like the real objects, with minor exceptions. For example, the trial dialog has a resize corner; the real one does not. Also, the trial objects behave differently than the real objects. For example, trial Buttons can be moved and re-sized, but can't be pressed. The Dialog Editor actually displays a descendant of Tbutton called TTrialButton, and uses this to generate Pascal code for a TButton. (When we capitalize an object name it refers to a standard TurboVision object.)

We will call trial Labels, History icons, and Scrollbars 'secondary' objects, because they are always 'attached' to, or associated with, one of the other types of dialog components, which we will call 'primary' objects. (See the sample dialog above for examples.)

There are other secondary objects that are never visible in a trial dialog: four standard types of Validator objects, and the HistoryWindow and HistoryViewer objects. These are optionally attached to, or associated with, InputLine objects. In a real dialog, the Validator objects are also invisible, but show a visible MessageBox if an attempt is made to shift the focus from an InputLine with invalid data. The HistoryWindow and HistoryViewer objects are associated with History icons attached to InputLines. (See Borland documentation for further details.)

For each primary object and any associated secondary objects, and also for the trial dialog window, there is an 'edit' dialog with two variations to create and edit the primary object and to add or remove secondary objects, as applicable. In all, there are four means of editing an object:

(1) a 'new object' dialog to create/define the object(s),
(2) dragging operations to move/resize the object(s),
(3) a 'change object' dialog to modify the object.
(4) testing the trial dialog, which changes the displayed data values. Source code for a data record with these values can be generated.

The 'new object' and 'change object' variations of the edit dialog for each primary object are very similar. For example, the 'Change InputLine' dialog is just like the 'New InputLine' dialog, except for two additional buttons.

In summary, 'edit' dialogs are used to create and edit a 'trial' dialog, which looks like the 'real' dialog that you want to include in your program. Source code is generated to insert into your program, and to edit further, if needed. When your program runs, the 'real' dialog appears and executes.

In case you need to redo this development cycle, you should save the trial dialog in a dialog collection file. This is a resource file of trial dialogs that can be fetched, edited, and translated to source code again. It should not be confused with a resource file of real dialogs, which is an alternative representation of the object initialization code of a program.

2.2 Creating Objects

To begin a Dialog design, a new trial Dialog is created by the 'New Dialog' command, which executes a 'New Dialog' dialog (detailed later). Unlike a normal dialog, the trial Dialog has a resize corner, and can be resized. But the Pascal code generated for the corresponding 'real' dialog is normal (not resizable), and its Bounds will correspond to the current position and size of the trial dialog.

For each primary object to be inserted into the trial Dialog, there is a 'new object' command which executes a 'new object' dialog, such as 'New Button', 'New InputLine', etc. The 'new object' dialogs allow specification of visible features (such as the text on a Button) and invisible features (such as the name of the command that the Button will generate).

2.3 Moving / Resizing Objects

A 'new object' dialog does not specify the size or position of the object. The new object is initially inserted with a default size at the bottom right of the trial Dialog. The new object can then be dragged to the desired position (with the left mouse button) and dragged to the desired size (with the right mouse button). For either type of dragging, it is only necessary to place the mouse initially any place on the object. Every object has a minimum size, and cannot grow bigger than the space inside the frame of the trial dialog. If the trial Dialog is not big enough to accommodate the new component, or too big, its resize corner (bottom right) can be dragged to change its size.

Secondary objects generally move automatically, as follows. An example is a Label associated with an InputLine. When the primary object (the InputLine) is moved, the secondary object (the Label) will automatically move (when dragging stops) the same amount, maintaining its position relative to the primary object. But when the secondary object is moved, the primary object does not move, thus re-defining the relative positioning. Labels follow the top left corner of their primary object. The vertical Scrollbar of a ListBox follows either the left or right side of the ListBox (the nearest side), and follows both the vertical position and vertical size of the ListBox. History icons cannot be dragged separately, but follow the associated InputLine.

2.4 Changing / Deleting Objects

Double-clicking on any trial object executes a 'change object' dialog, which is similar to the 'new object' dialog, thus allowing any feature to be changed. The 'change object' dialogs also provide for deletion of dialog components and for changing the 'Z-order' of the components. Details are given in the "Change Object Dialogs" section.

The Dialog | Test function can be used to change the data shown by the dialog. Although this data is saved and retrieved as part of the trial dialog, it is not part of the real dialog definition, but rather defines the data record transferred to and from the dialog (by the SetData and GetData methods) when it is executed.

2.5 Saving / Retrieving / Coding Dialogs ----------

The trial dialog can be saved in a 'dialog collection' file. This is actually a resource file, but cannot be used as a resource file for your program, because the objects in it are 'trial' objects, not 'real' objects, with differences as noted above. But the dialog collection file is useful for retrieving previous dialog designs. Pascal code can be generated for the current dialog, which includes an Init constructor for creating the dialog, and a declaration of a record type associated with the GetData and SetData methods of the dialog.

Given that the dialog was named GadgetDialog, the generated code would include the following (the 'const' part is optional):

```
type
    PGadgetDialog = ^TGadgetDialog;
    TGadgetDialog = object(TDialog)
        constructor Init;
    end;

    PGadgetData = ^TGadgetData;
    TGadgetData = record
        {definition of data field names and types}
    end;

const
    GadgetDefaults: TGadgetData = (
        {definition of default data values}
    );
```

The easiest way to use the dialog is to use the ExecuteDialog function of the App unit. A typical use of the dialog would be:

```
procedure ExecuteGadgetDialog;
var
    GadgetData: TGadgetData;
begin
    with GadgetData do begin
        { set values of GadgetData fields }
        if Application^.ExecuteDialog(
            New(PGadgetDialog, Init, @GadgetData
        ) <> cmCancel
        then begin
            { act on new values of GadgetData fields }
        end;
    end;
end;
```

The GadgetData variable might need to be global. If the constant GadgetDefaults is defined, then GadgetData can be easily initialized by the statement:

```
GadgetData:= GadgetDefaults;
```

If the dialog uses more buttons than the usual Ok and Cancel buttons, the 'then' clause of the 'if' statement might need to include a 'case' statement to provide actions for each button (except Cancel). (The standard commands cmOk, cmCancel, cmYes, and cmNo, which are normally associated with TButton objects, terminate execution of a TDialog, and are returned by the ExecuteDialog function. If other commands are used, generally the descendant dialog must override the HandleEvent method.)

2.6 Using History Icons (Þ↓Ý) ----------

Most of the input lines of the dialogs for editing objects and changing options have history icons. (A history icon is a small down-arrow at the right end of an input line.) Clicking on a history icon opens a small history window that lists previous string values of the associated input line and other input lines with a similar function. (See Borland documentation for more details.) The history windows can be used to restore old string values, or to copy-and-paste string data from one input line to another.

The history data is read from the default history file when the program starts, and written to the history file when the program exits. The 'Options' section below describes how to specify the history file.

3. Main Screen / Main Commands ==========

3.1 Menu Bar ----------

The Dialog Editor has a menu bar at the top of the screen with these items on the left side:

```
±±Dialog_Editor±±File±±Dialog±±New±±±±±±±±±±±±±±±±±
```

Each of the menu items on the menu bar can be activated by clicking or dragging with the mouse, or by typing the highlighted letter with the ALT key down. Menu items can also be selected with the F10 key and arrow keys. Any menu item can be cancelled with the ESC key or by clicking at another position. Menu items shown in gray are not currently selectable.

Once a menu is 'pulled down', items can be selected and activated by clicking with the mouse, or by typing the highlighted letter, or selecting with the arrow keys and activating with the Enter key.

At the right end of the menu bar is a Swap control, an Edit Mode control ('Default' or 'Paste'), and a clock display, such as:

```
±±±±±±±±±±±±±±±±±±±Swap±±Default±±12:20:39±
```

The Swap control provides quick switching between two dialogs. Operation of the Swap control is explained in the section "Special Controls".

The Edit Mode display indicates whether dialogs for creating new objects will be initialized with default data or with data from a paste buffer. Operation of the Edit Mode control is explained in the section "Special Controls" and in the section "New Object Dialogs".

The clock display is in hour:minutes:seconds format.

Activating the 'Dialog Editor' menu item shows a copyright notice with the date of the current version of the Dialog Editor:

```
±±Dialog_Editor±±±File±±Dialog±±New±±±±±±±±±±±±±±±±±±±±
 ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
 ³ DialEdit 7-15-04 (C) J.M.Clark ³
 ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

Activating the 'File' menu item pulls down the following menu:

```
±±Dialog_Editor±±±File±±Dialog±±New±±±±±±±±±±±±±±±±±±±±
                     ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
                     ³ Output options..  ³
                     ³ Select files..    ³
                     ³ Change dir..      ³
                     ³ DOS shell         ³
                     ³ Exit              ³
                     ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

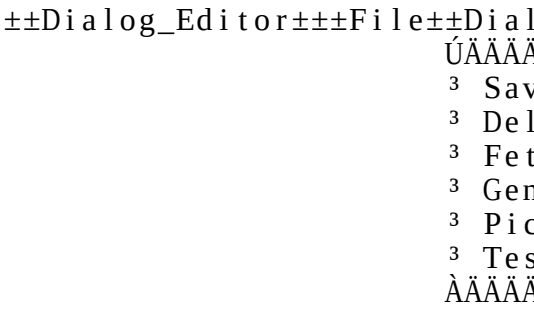
In this and other menus, an elipsis (..) indicates that action is not immediate, but that a dialog is executed next, allowing for choices and possible cancellation before committing to action.

Each of these items is explained in detail in the section 'File Commands' below. Briefly, these commands do the following:

Output options..	setup and output pictures and/or code.
Select files..	select files used by the Dialog Editor.
Change dir..	change the DOS default (current) directory.
DOS shell	go temporarily to DOS.
Exit	quit the Dialog Editor.

The Output command can combine the functions of the Gen Code and Picture commands (next), and can process a list of selected dialogs at one command.

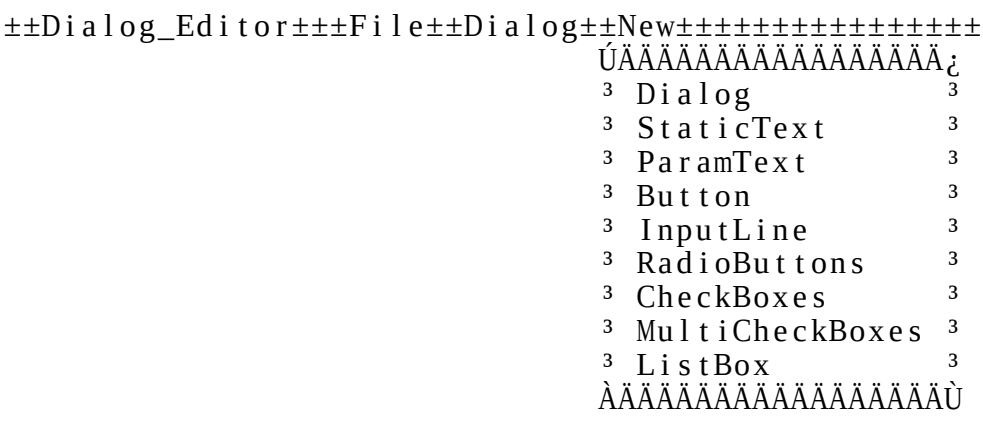
Activating the 'Dialog' menu item pulls down the following menu:



Each of these items is explained in detail in the section 'Dialog Commands' below. Briefly, each of these commands operates on one 'current' command as follows:

Save	save current dialog in the dialog collection file.
Delete	delete current dialog from dialog collection file.
Fetch..	fetch a dialog from the dialog collection file.
Gen code	generate Pascal code for the current dialog.
Picture	output a picture of the current dialog.
Test	test current dialog, and change data values.

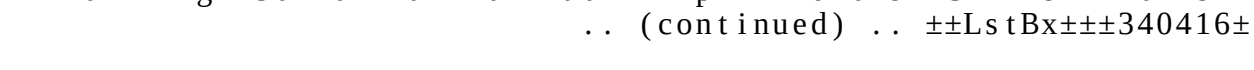
Activating the 'New' menu item pulls down the following menu:



Activating one of these items executes a dialog box to create the named TurboVision object. These dialogs are explained in the section 'Object Dialogs' below. For each object, there is a 'New Object' dialog to create the object, and a similar 'Change Object' dialog to edit the object.

3.2 Status Line

The Dialog Editor has a 'status' line at the bottom of the screen with these items, explained next:



At the end of the status line is a display of the number of bytes of available memory, such as shown above.

The Exit item is equivalent to the Exit command in the File menu. The other items are equivalent to the commands in the New menu, as follows:

```
Dilg     Dialog
StTxt    StaticText
ParTxt   ParamText
Butn     Button
InpLn    InputLine
RdBtns   RadioButtons
ChkBxs   CheckBoxes
MltBxs   MultiCheckBoxes
LstBx    ListBox
```

Each of the items on the status line can be activated by clicking with the mouse or by typing the highlighted letter with the ALT key down.

4. File Commands ==========

Here, we provide more details of the commands available from the File menu.

4.1 Output Options ----------

The Output Options command of the File menu activates this dialog:

```
ÉÍ[þ]ÍÍÍÍ Output Options ÍÍÍÍÍÍÍÍ»
º±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±º
º±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±º
º±± Output includes.. ±±±±±±±±±±±±±º
º±± [ ] Pictures    [X] TV Code ±±º
º±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±º
º±± Show.. ±±±±±±± Code Format±±±º
º±± [ ] Resize   ±± ( ) Bare    ±±º
º±± [ ] Shadow   ±± (•) Unit    ±±º
º±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±º
º±± Pattern of.. ±± Indent.. ±±±±±º
º±± [±] Backgrnd ±± [T] spaces  ±±º
º±± [²] Shadow   ±±±(T = tab)±±±±±º
º±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±º
º±± [ ] Save  Cancel Ü  Pick.. Ü±±º
º±±±±±±±±±±  ßßßßßßßß  ßßßßßßßß±±º
º±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±±º
ÈÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍÍ¼
```

This dialog sets up various options before picking out (with another dialog) which dialogs in the collection file will be translated to picture form and/or TurboVision source code and sent to the Code Output (file or standard output).

Selecting Pictures will generate text pictures of the selected dialogs, such as those used in this documentation. Selecting TV Code will generate Pascal code for each selected dialog declaring an object type for the dialog, a record type for the associated data, and providing an Init constructor. At least one of these must be checked, or there will be nothing to do.

Generated Pascal code will include a definition of default data values for each dialog for which the 'Gen. Defaults' option was selected by the 'New Dialog' or 'Change Dialog' dialogs. (See the 'Creating a Trial Dialog' and 'Editing the Trial Dialog' sections.)

If both Pictures and TV Code are checked, the pictures are enclosed in (* *) comment delimiters. The sequence will be pictures for all selected dialogs, declarations for all selected dialogs, and Init constructors for all selected dialogs.

The Show.. and Pattern of.. options specify how Pictures are to be generated, and the Code Format and Indent.. options specify how TV Code is to be generated.

If the Unit Code Format is selected, additional syntax is included as needed to make the output file a compilable unit.

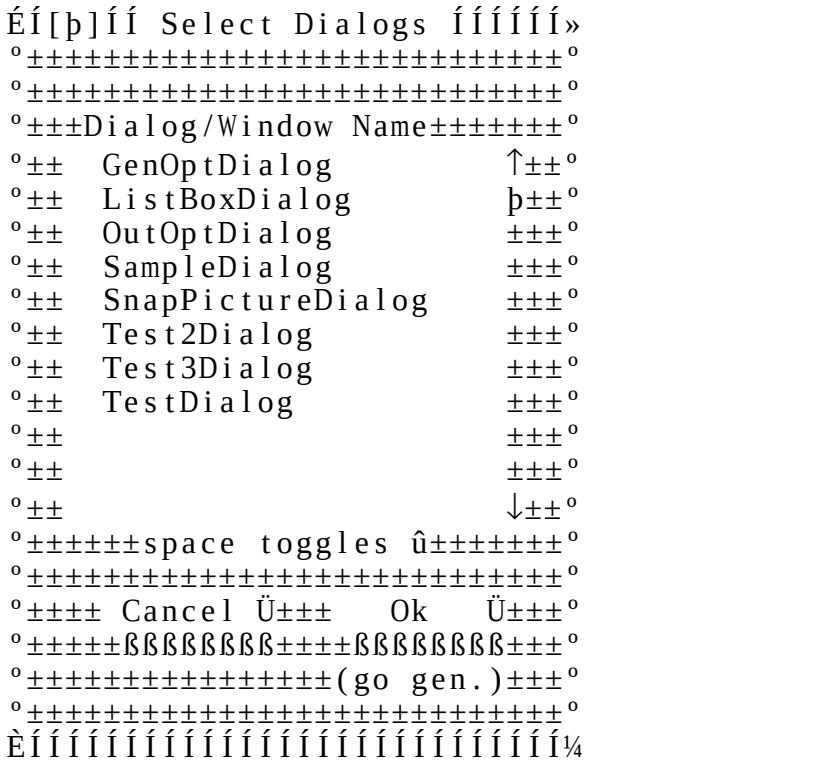
The Indent item is a MultiCheckBox with values represented by the symbols 1234T, which specify how the source code and/or pictures will be indented. A digit specifies that number of spaces for each level of indentation, and a T indicates that a tab character will be used for each level. (Pictures are indented one level.) Use the space bar or mouse clicks to change the selection.

The Show checkboxes indicate whether the Dialog (or Window) frames will include a resize corner, and whether a shadow will be included in each picture.

The 'Pattern of' MultiCheckBoxes select shade characters for the background and for the shadow (if any) of the Dialog (or Window).

Checking the Save checkbox causes all of dialog selections to be saved in the History file as new default values (unless the dialog is cancelled). This does not change the default value of the Save checkbox however.

Pressing the Pick button activates the next dialog:



If you only wanted to set Picture and TV Code options (and perhaps also change the defaults of these) without producing any output now, you can cancel this dialog or click Ok without selecting any dialogs.

The ListBox displays a list of dialogs that are stored in the current dialog collection file. (The captions 'space toggles û' and 'go gen' are reminders that this dialog operates differently than a similar dialog titled 'Fetch Dialog', which selects only one dialog.) Use the up and down arrow keys or the mouse to move the highlight, then press the space bar to add or remove a check mark (û) to the dialog name.

After check-marking as many dialog names as you like, clicking Ok or pressing the Enter key will cause the selected dialogs to be fetched from the collection file and processed according to the Output Options dialog, generating picture and/or code output.

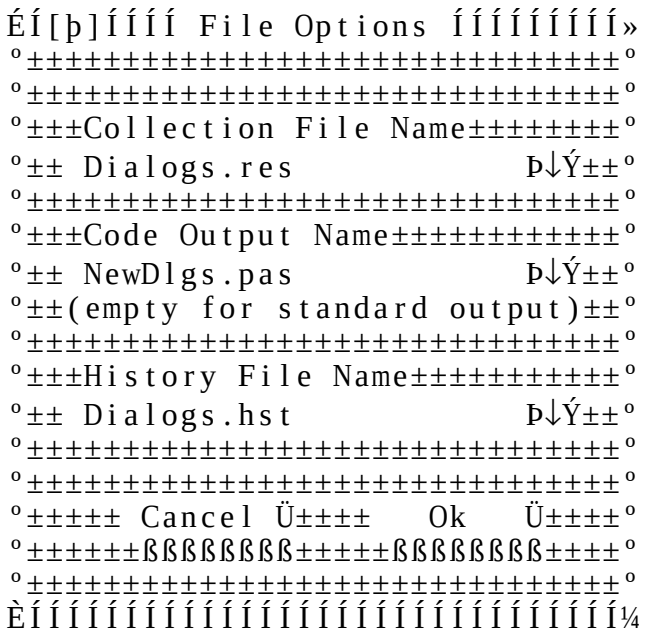
If pictures are included, the dialogs will be briefly put on the screen. But if there is already a trial dialog on-screen with edit changes that were not saved, a MessageBox will appear asking if you want to save it. If it has no unsaved changes, it will be removed from the screen without asking.

Multiple output operations will append data to the output file, because the output file is not closed until the output filename is changed or the Dialog Editor exits. This doesn't make sense if you have selected the Unit Code Format option. If you want to output multiple unit files you must change the output name between output operations.

The output file is opened when first used, and any existing file will be overwritten at this time.

4.2 Select Files

This command executes a dialog like this (defaults shown):



The InputLine labelled 'Collection File Name' specifies the filename of the dialog collection file. The 'Code Output Name' InputLine specifies the filename of the file to receive pictures and/or Pascal code generated for trial dialogs, but an empty string indicates that the standard output is to be used. The 'History File Name' InputLine specifies the name of the file used to save history data (see the 'Using History Icons' section above). The file names follow DOS conventions and may include drive and path specifications.

For the Collection and Code file names, if the file name is changed, the old file will be closed if open. The new file is not opened until first used.

If the History file name is changed, the current history data will be written to the old file, then history data read from the new file if it already exists. The history file is also read when the program starts, and written when the program exits.

If the Dialog Editor is installed in the Tools menu of the IDE, use the $SAVE ALL $CAP EDIT macros in the IDE setup, and use the default standard output option of the Dialog Editor. The standard output will be directed to a Transfer Output window. From this window, the generated code can be copied and pasted into a program source file, or the Transfer Output window can be Saved As a file.

These options can also be set with parameters on the DOS command line. The general syntax is:

```
dialedit [ /cCollFile ] [ /oCodeFile ] [ /hHistFile ]
```

Optionally, the filenames can be separated from the option codes by spaces. The default options can also be set 'permanently' (by modifying the dialedit.exe file) with the CONFIG program. The syntax is similar:

```
config dialedit [ /cCollFile ] [ /oCodeFile ] [ /hHistFile ]
```

4.3 Change Dir

This command executes the ChDirDialog (change directory) dialog from the StdDlg unit, which displays a directory tree for the current drive, and allows you to select any drive/directory as 'current'. (See TurboVision documentation.)

4.4 DOS Shell

This command executes a copy of the DOS command interpreter, with limited memory. Type 'exit' at the DOS prompt to return to the Dialog Editor.

4.5 Exit

This command returns control to the operating system from which the Dialog Editor was started.

5. Dialog Commands

Here, we provide more details of the commands available from the Dialogs menu.

5.1 Save

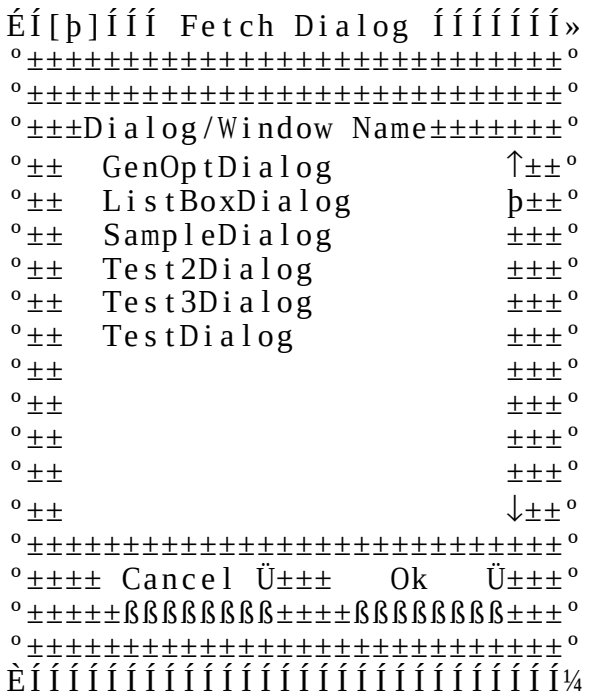
The dialog collection file is the file named by the File / Select command described above. If such a file does not exist, a new file will be created. The name assigned to identify the trial dialog to be saved in the file is the concatenation of the BaseName and TypeName fields as assigned to the trial dialog by the 'New Dialog' or 'Change Dialog' dialog. If a dialog specified by this name already exists in the file, it will be replaced. If needed, use the Fetch Dialog command to review existing dialog names, and use the 'Change Dialog' dialog to change the name of a trial dialog before saving it.

5.2 Delete

This command deletes the current trial dialog from the dialog collection file, after confirming the action with a MessageBox naming the dialog. To delete any other dialog, first use the Fetch Dialog command to select the dialog. (This allows you to see the dialog that you are deleting.)

5.3 Fetch

This command executes a dialog box like this, showing the names of dialogs (if any) in the current dialog collection file. This command does nothing if there is no dialog collection file.



Select a dialog name by clicking it with the mouse (or using arrow keys) and then click the Ok button (or use Enter key) to fetch the selected dialog. The fetched dialog is displayed as the current trial dialog, and can be edited, or code generated for it. If there is a trial dialog on the screen before the Fetch Dialog command is executed, which was modified but not saved, a message box will appear asking if you want to save it. Else, the fetched dialog will replace the current trial dialog, because the Dialog Editor will not show more than one trial dialog at a time.

Click the Cancel button (or use ESC key) if you just wanted to see the dialog names.

5.4 Gen Code

This command uses the options set by the File | Output Options command, and the code output file (or standard output) as specified by the File | Select Files command.

Pascal code is generated (but no picture) for the trial dialog currently shown on the screen (if any), as described for the File | Output Options command.

5.5 Picture

This command uses the options set by the File | Output Options command, and the code output file (or standard output) as specified by the File | Select Files command.

A picture is generated (but no Pascal code) for the trial dialog currently shown on the screen (if any), as described for the File | Output Options command.

5.6 Test

This command creates a real dialog from the trial dialog, which you can test. Initial data values are copied from the trial dialog. To copy the data values from the real dialog back to the trial dialog, you must press an 'Ok' button, which means that the dialog must have an 'Ok' Button. (Actually, it's the cmOk command that matters, not the Button text.) Pressing buttons that generate cmCancel, cmYes, or cmNo commands will also terminate the dialog, but do not save the data.

Data transferred to the trial dialog in this manner can be saved in the dialog collection file, and can be used to generate code for a default data record.

The real dialog temporarily created by this command slightly differs from the real dialog created by the generated code, as follows. (1) Validators are not activated. (2) ListBoxes do not have a list. (3) ParamText objects have nil, null, or zero data values. For (2) and (3), you will need to connect the generated code to data sources that you specify.

6. Generating Code and Picture Output

DialEdit supports two styles of generating Pascal code and dialog pictures. The two styles don't mix well -- you should use either one or the other.

6.1 One-at-a-Time Generation

One style is to generate dialog code or pictures one at a time; these are appended to the selected output file, which will be closed when you exit DialEdit. From the menu bar, the Dialog | Gen Code and Dialog | Picture commands support this style. The contents of the output file will probably not be in the proper sequence or complete, and will need editing to produce a compilable Pascal file.

6.2 All-at-Once Generation

The other style generates all the Pascal code needed for one or more dialogs in one ready-to-compile Pascal unit file. First, you must save all of the dialogs that you need for your program, using the Dialog } Save command. If you have defined default data for the dialogs, and want these to be in the generated code, check the 'Gen. Defaults' option for each dialog. Then --

> Use the File | Select Files command, if needed, to define the output file.
>
> Use the File | Output Options command to set the output options, checking the Save checkbox if you want to save these options as defaults. Then click the 'Pick.." button. In the next dialog, use the space bar and arrow keys to put a check mark on the saved dialogs for which you want to generate code, then click Ok.

This opens, rewrites, and closes the output file. If you selected 'Unit' as the Code Format, the output file will be a complete Pascal unit file for the selected dialogs. You will only need to edit this file if you want extra features not supported by DialEdit.

7. General Editing Features ==========

7.1 Special Controls ----------
(and an overview of copy-and-paste operations)

The Swap control provides quick switching between the last two trial dialogs that have been saved or fetched. Clicking on the 'Swap' word on the menu bar activates this function if two such dialogs exist (two dialog names are remembered). Deleting a dialog will disable this feature until another dialog is fetched or a new one is created and saved.

Activating the Swap function automatically saves the current trial dialog (if any) and fetches the previous trial dialog. Repeating the Swap function switches between the same two dialogs (until another dialog is fetched or a new one is created and saved). This is convenient when copying and pasting objects from one dialog to another.

The Edit Mode display indicates whether dialogs for creating new objects will be initialized with default data or with data from a paste buffer. Clicking on the Edit Mode display toggles the mode. The mode is also set by buttons appearing in an edit dialog. The initial mode is Default.

As an example, suppose we want to copy a CheckBoxes object from an existing 'Old' dialog to a 'New' dialog. This will provide an overview of copy-and-paste operations, as well as illustrate the use of the Swap and Edit Mode controls.

First we create dialog 'New', then fetch dialog 'Old'. But before we actually get dialog 'Old', we are prompted "Save current dialog?" and we answer "Yes". The editor then saves dialog 'New' and fetches dialog 'Old'. We double-click the CheckBoxes object in dialog 'New', which brings up a 'Change CheckBoxes' dialog. We click the 'Copy' button in this dialog to copy the CheckBoxes specifications into a Paste buffer, and the Edit Mode is automatically changed to Paste. We click on the Swap control, and we switch back to dialog 'New'. We click on 'ChkBxs' at the bottom of the screen, and get a 'New CheckBoxes' dialog which shows the specifications from the Paste buffer. (We can change the CheckBoxes specifications at this point if we like.) We click the Ok button, and the imported CheckBoxes appear in dialog 'New'. We then drag the CheckBoxes to the desired location, and perhaps also resize it. (If the CheckBoxes object had an attached Label object, the Label and its relative position would also be copied automatically.)

Swapping again, we can go back to dialog 'New' to copy more objects. But since the Dialog Editor reserves a separate space in the Paste buffer for each type of object, we can copy-and-paste one of each type of object from one dialog to another in each Swap cycle. If we needed to copy most of the dialog, it would be easier to start by copying the entire dialog -- by changing the name of the dialog and saving it with the new name.

When done copying and pasting, we generally would want to click the Edit Mode control, toggling it to 'Default' mode, to be prepared to create objects. The 'New Object' dialogs will now start with default data instead of data from the Paste buffer.

7.2 'New Object' Dialogs

When execution of a 'new object' dialog begins, the dialog data are initialized either from a 'Default' buffer or from a 'Paste' buffer, depending on the current state of the 'edit buffer mode'. We will explain all these new terms next.

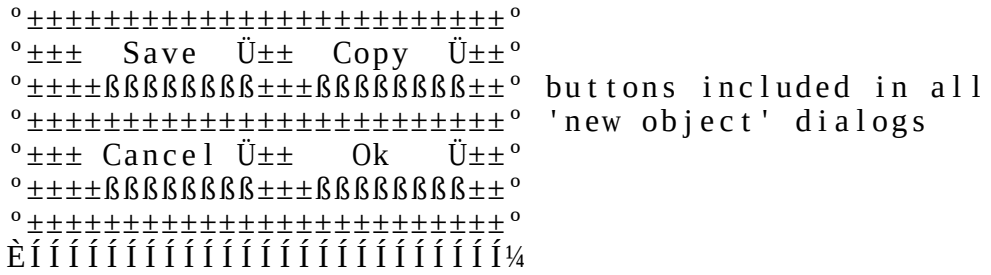
The 'dialog data' are the values shown in the dialog box, which specify features of the new object.

The Default buffer defines default data for each type of object that can be created and edited. When the Dialog Editor starts, this buffer is loaded from the History file, if it exists; otherwise, it is set to built-in default values (shown in the sections describing each edit dialog). When the Dialog Editor exits, the Default buffer is stored in the History file.

The Paste buffer holds temporary data for each type of object that can be created and edited. When the Dialog Editor starts, this Paste buffer is copied from the Default buffer after the Default buffer is initialized. The Paste buffer is not saved when the Dialog Editor exits. The Paste buffer is used to copy object specifications (except for size and position) from one dialog to another, or within a dialog.

The 'edit buffer mode' specifies which buffer will be used to initialize a 'new object' dialog. This mode is indicated by the word 'Default' or 'Paste' shown on the menu bar. Clicking on the displayed word toggles the mode. When the Dialog Editor starts, the mode is Default.

All 'new object' dialogs provide these four buttons:



The Save button ends the dialog, saving the dialog data in the Default buffer, and setting the edit buffer mode to 'Default'. No object is created.

The Copy button ends the dialog, saving the dialog data in the Paste buffer, and setting the edit buffer mode to 'Paste'. No object is created.

The Cancel button ends the dialog, with no effect (as though the dialog had not executed).

The Ok button ends the dialog, creating a new object as specified by the current dialog data. If secondary objects are specified, these are also created as specified.

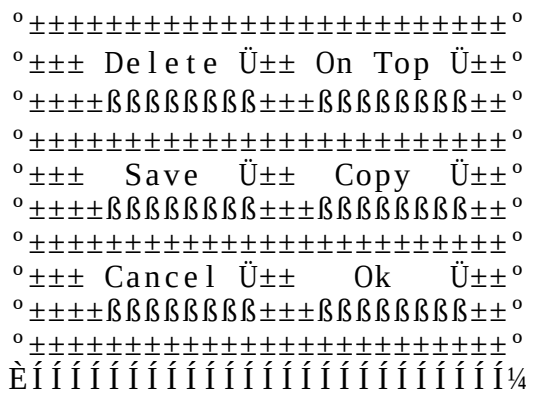
If the new object is a dialog, the trial dialog is positioned near the top left of the screen. This is generally convenient, because it can grow downward and to the right as needed, and because the 'new object' and 'change object' dialogs are placed near the bottom right of the screen, where they generally will not obscure the trial dialog. But all dialogs can be dragged to new positions if desired.

If the 'Center Dialog' option was chosen, the generated code will generate a centered dialog regardless of the position of the trial dialog. Otherwise, the trial dialog can be left in the top left position while it is edited, then dragged to its 'correct' position before executing the Dialog | Gen code or Dialog | Save command.

If the new object is a component of a dialog, the primary object and any associated secondary objects are inserted near the bottom right corner of the trial dialog. You will generally need to drag the new object to the desired location and adjust its size (as described in the next section).

7.3 'Change Object' Dialogs

Except for the 'Change Dialog' dialog, a 'change object' dialog has two additional buttons as shown below. For changing general features (not components) of the trial dialog, see the section 'Editing the Trial Dialog' below.



Additional buttons for all 'change object' dialogs (except 'Change Dialog')

The Delete button ends the dialog, and deletes the object from the trial dialog.

The On Top button does the function of the Ok button (described below), and also moves the object to the top of the Z-order, equivalent to being the last object to be inserted into the trial dialog. Associated secondary object(s) are also moved on top. The Z-order determines the order of the data fields of the data record associated with the dialog. Z-order can be examined by temporarily sliding one object toward another, and observing which one is in front (on top). Z-order can also be examined by executing the Dialog | Test command; the tab key shifts the focus from one dialog component to another in Z-order.

The Save button ends the dialog, saving the dialog data in the Default buffer, and setting the edit buffer mode to 'Default'. The object is not changed.

The Copy button ends the dialog, saving the dialog data in the Paste buffer, and setting the edit buffer mode to 'Paste'. The object is not changed.

The Cancel button ends the dialog, with no effect (as though the dialog had not executed).

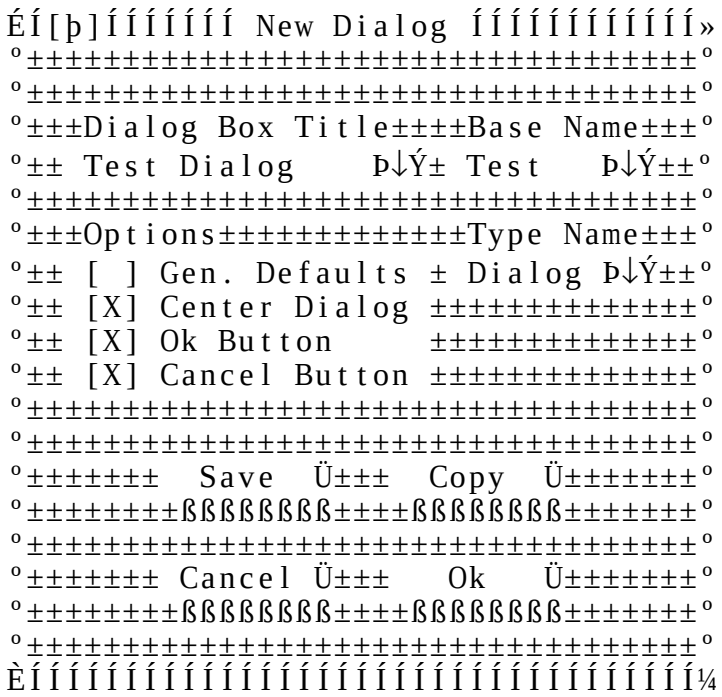
The Ok button ends the dialog, changing the object as specified by the current dialog data. Secondary objects may be created, changed, or deleted, depending on how the current dialog data compares to the current state of the trial object.

8. Dialog Dialogs

These dialogs create and change a trial dialog.

8.1 Creating a Trial Dialog

The New Dialog command executes this dialog (defaults shown):



If a trial dialog with unsaved changes is on the screen when the New Dialog command is executed, then before the 'New Dialog' dialog appears, you will be prompted with a message box asking if you want to save the current dialog. Click the Yes or No button.

The Dialog Box Title is the title to be shown centered in the top edge of the dialog window frame.

The Base Name and Type Name determine the names used in the generated code. For the default values of 'Test' and 'Dialog', the generated code includes:

```
{ TestDialog }

type
    PTestDialog = ^TTestDialog;
    TTestDialog = object(TDialog)
        constructor Init;
    end;

    PTestData = ^TTestData;
    TTestData = record
        {definition of data field names and types}
    end;
```

If these values were changed to 'Report' and 'Window', the generated code would include:

```
{ ReportWindow }

type
    PReportWindow = ^TReportWindow;
    TReportWindow = object(TWindow)
        constructor Init;
    end;

    PReportData = ^TReportData;
    TReportData = record
        {definition of data field names and types}
    end;
```

The record definition will include data field names and types corresponding to component objects of the dialog, in Z order.

Notice that Type Name specifies the name of the ancestor of the 'trial dialog' object (excluding the 'T' prefix). Although this documentation and the Dialog Editor program uses the name 'Dialog' throughout, here the type can be changed to Window or some other related object type, such as a descendant of TDialog. Although the Dialog Editor is oriented to design of dialogs, it can also to useful for the design of similar objects, even if the generated code must be modified.

The Options CheckBoxes make these selections:

[X] Gen. Defaults - causes the generated code to include code such as:

```
const
    ReportDefaults: TReportData = (
        {definition of default data values}
    );
```

The default data values are taken from the values displayed in the trial dialog, which can be changed by the Test Dialog command. This includes a string value for each InputLine, a hexa-

decimal value for each CheckBoxes and MultiCheckBoxes, and a decimal value for each RadioButtons.

But ListBox and ParamText components are handled differently, because these generally make indirect reference to data. Generally, you will need to modify the generated code to complete these data references. For ListBoxes, the List pointer is set to nil, and the Selection value is set to zero. For ParamText, %s fields are set to nil, %x fields to $0, %c fields to $20 (space character), and %d fields to 0.

[X] Center Dialog - includes this statement in the Init constructor of the dialog (even though the trial dialog is not shown centered):

```
Options:= Options or ofCentered;
```

[X] Ok Button - includes an Ok button in the dialog, which will generate this code:

```
Insert(New(PButton, Init(R, '~O~k', cmOk, bfDefault)));
```

[X] Cancel Button - includes a Cancel button in the dialog, which will generate this code:

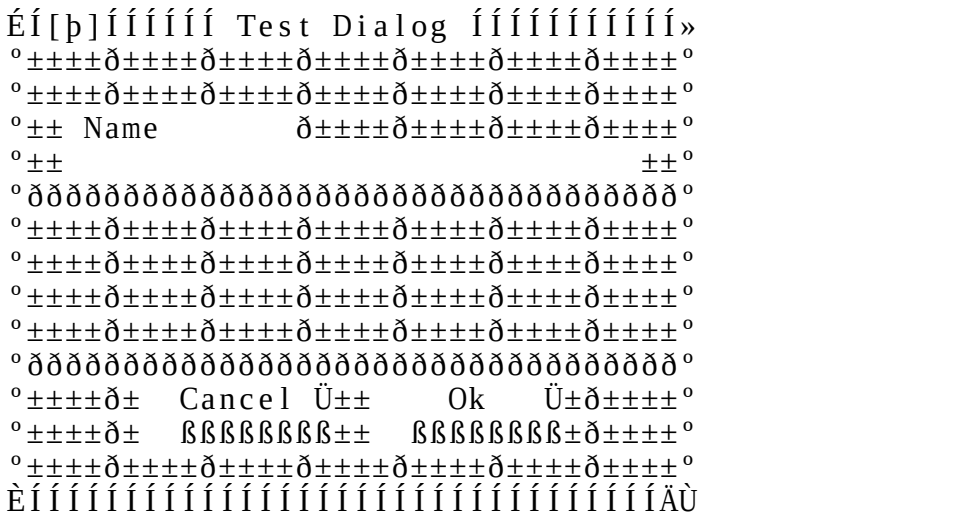
```
Insert(New(PButton, Init(R, 'C~a~ncel', cmCancel,
    bfNormal)));
```

8.2 Editing the Trial Dialog

The trial dialog can be moved by pointing to the top edge of the dialog with the mouse and dragging. The trial dialog can be resized by pointing to the resize (bottom right) corner of the dialog with the mouse and dragging.

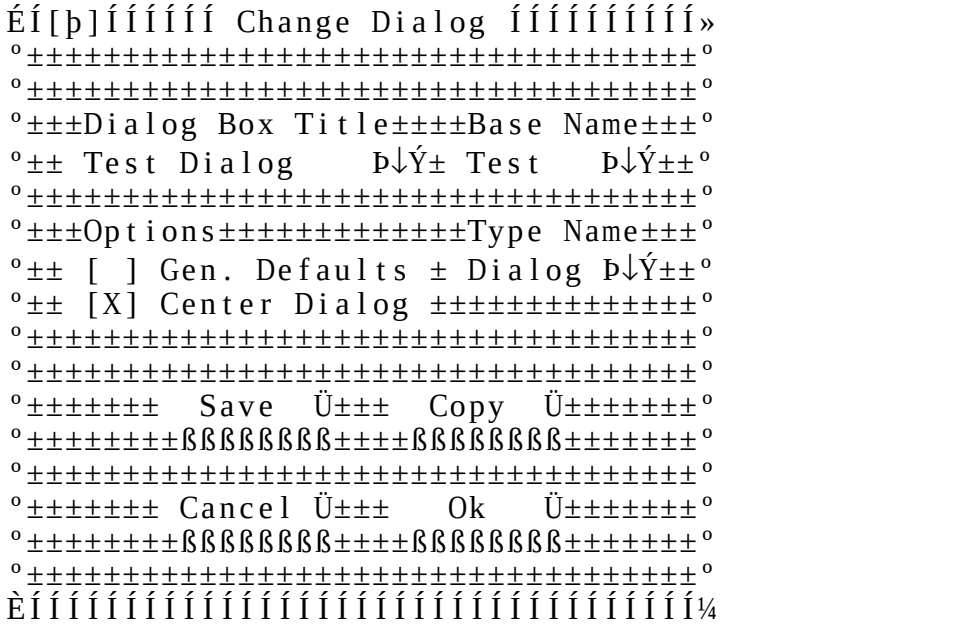
Clicking on the close icon in the dialog frame (or pressing the ESC key) closes the dialog; you will be prompted with a message box asking if you want to save the dialog if it has unsaved changes.

The trial dialog has an optional grid background that can help to check the spacing, position, and size of component objects. (See example below.) Double-clicking on the background toggles between the grid style and the normal appearance.



Also, some objects, such as labels, can include space that looks exactly like a normal background, and thus is not evident except when the grid background is shown. Such space can be important: for example, the selectable area of a Label can extend beyond the Label text. (See 'Name' Label in above example.) The grid background is not included in the 'real' dialog created by the generated code.

To change other features of the trial dialog, double-click either side or the bottom of the dialog frame (not the top edge or the resize icon). This action executes this dialog:



This functions like the 'New Dialog' dialog, except that the button options (not needed here) are omitted.

(For other operations on the current trial dialog, see the Dialog commands Save, Gen Code, Picture, and Test.)

9. Component Dialogs

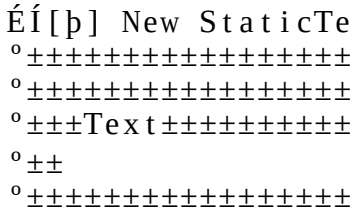
Each component object described next is created by a 'New XXX' command, where 'XXX' is the name of the object type, which executes a 'New XXX' dialog with buttons as described in the section 'Creating Objects' above.

Each component object can be moved / resized as described in the section 'Moving / Resizing Objects' above.

Double-clicking the component object executes a 'Change XXX' dialog like the 'New XXX' dialog, but with two more buttons as described in the section 'Changing / Deleting Objects' above. (Also, the word 'New' in the dialog title becomes 'Change'.)

Because the buttons at the bottom of the edit dialogs are the same in appearance and operation for all component objects, the following sections will show and explain only the top part of each dialog. Default data is shown in each dialog picture.

9.1 StaticText Dialog

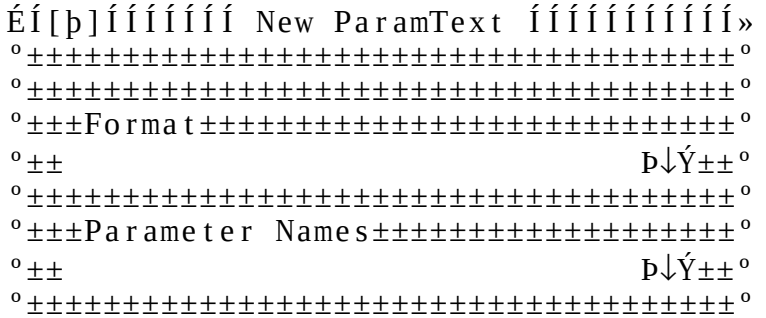


The New StaticText dialog generates a StaticText object with the given Text, but only if the Text string is not empty. Sample generated code:

```
R.Assign(22, 21, 45, 22);
Insert(New(PStaticText, Init(R, 'computed')));
```

("var R: Trect;" is also declared.)

9.2 ParamText Dialog



The New ParamText dialog generates a ParamText object only if the Format and Parameter Names strings are not empty. The Format item is the text string used to initialize the ParamText object, and used by the ParamText object to format data (using the FormatStr procedure) when the dialog is executed. (See Borland documentation for more details.) The Parameter Names item is a list of names, separated by commas, to be used as field names for the associated data record. (Spaces are permitted after the commas.) The number of names determines the number of data parameters; this number is used for the last ParamText.Init parameter.

When coding the data record, it is assumed that the order of the names corresponds to the order of the '%' codes in the Format string (the '%nnn#' code, if present, is not interpreted). In the data record, the following types are used according to the '%' codes, as follows:

```
code     type
----     ----------
%s       : ^string;
%c       : longint {lo char};
%d       : longint;
%x       : longint;
```

For example, if the Format string is:

```
st=%s de=%d ch=%c hx=%x
```

and the Parameter Names string is:

```
str, dec, chr, hex
```

then the generated code for the data record will include:

```
str: ^string;
dec: longint;
chr: longint {lo char};
hex: longint;
```

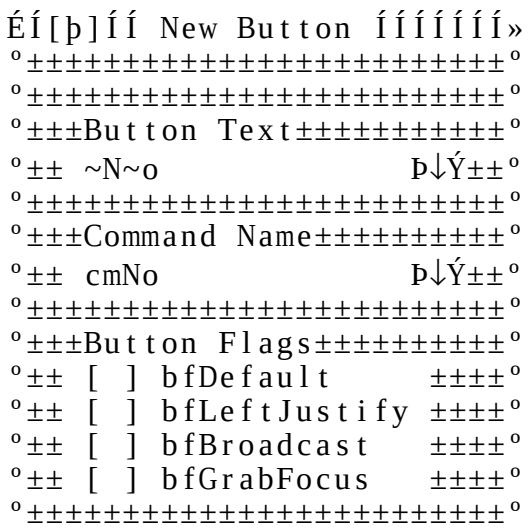
and the generated code for the dialog will include:

```
Insert(New(PParamText, Init(R, 'st=%s de=%d ch=%c hx=%x', 4)));
```

The comment "lo char" is a reminder that for %c data, only the low byte of the longint is used, cast as type 'char'.

The **trial** ParamText does not execute FormatStr to translate data, (there is no data) but instead simply displays the format string 'as is'. Be sure to resize the trial ParamText as needed. (You will need to enable the grid background to see the allocated length of the ParamText object.)

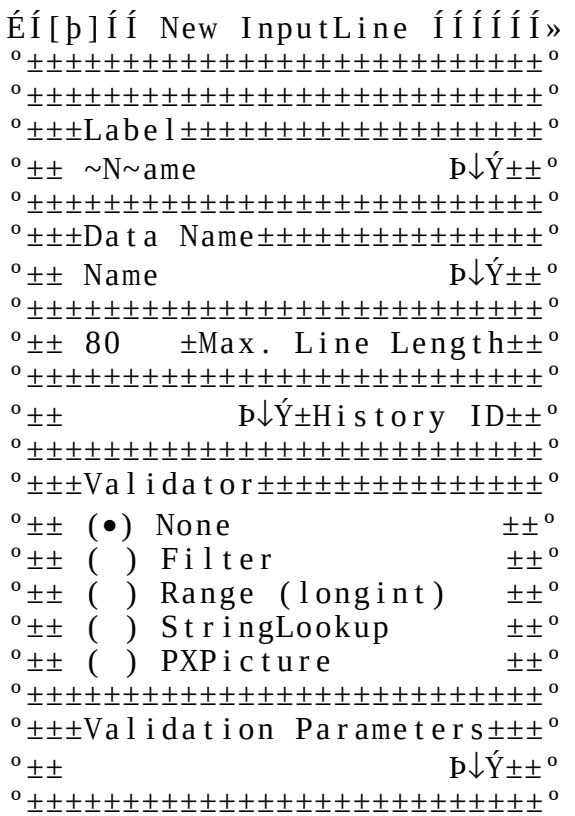
9.3 Button Dialog



The Button Text item specifies the text on the button, with the '~' characters used to indicate highlighted characters (shortcut keys). The Command Name item specifies the name of the command that the button will activate. The Button Flags CheckBoxes indicate optional bfXXXX values that are to be OR-ed into the Flags field of the Button. If none are checked, the value is bfNormal. Sample generated code:

```
R.Assign(36, 31, 46, 33);
Insert(New(PButton, Init(R, '~O~k', cmOK, bfDefault)));
```

9.4 InputLine Dialog



The Label item specifies an associated Label object, unless an empty string is specified.

The Data Name item is the name of the associated data field in the data record type associated with the dialog.

The Max. Line Length is the maximum length of the string that the InputLine object edits. (With the default values, the associated data field is "Name: string[80]".)

The History ID item specifies an associated History icon, with the given ID value, at the immediate right of the InputLine. The History ID item is actually a string that may be a constant identifier (that your program will define) or a number in the range of a word. But an entry of " or '0' or a string beginning with a space character specifies no History icon. The trial History icon is not movable or resizable, but automatically follows the movement and resizing of the InputLine.

The Validator RadioButtons can be used to select an optional Validator object to be linked to the InputLine. For choices other than 'None', the generated code initializes one of the following descendants of TInputLine, defined in the TvInput unit, which incorporates corresponding standard validators from the Validate unit, as follows:

TInputLine descendant	TValidator descendant
TFilterInputLine	TFilterValidator
TRangeInputLine	TRangeValidator
TStringLookupInputLine	TStringLookupValidator
TPXPictureInputLine	TPXPictureValidator

If any of these are used, the code should include TvInput in its 'uses' clause. (Of course, you can use your own custom validators; select a similar validator here, then modify the generated code later.)

The Validation Parameters string is used to specify the parameters used to initialize the chosen validator. When a validator is chosen, a line of help text is displayed below the Validation Parameters input line, suggesting the form of the parameters, as follows:

Validator choice	help text for parameters
None	(no parameters needed)
Filter	ValidChars: TCharSet
Range	Min, Max: longint
StringLookup	S: PStringCollection
TPXPicture	Pic:String; Fill:Boolean

The Validator Parameters string should be the parameters as suggested by the help text, as they should appear in the Pascal code for initializing the validator. For more details, see the documentation for the standard validators.

The RangeValidator includes a Transfer method that converts the string to a longint, which is the type of the field in the dialog's data record. The other validators leave the data in the form of a string.

An associated Label is added or deleted by making the label text non-empty or empty. An associated History icon is added or deleted by making the History ID a valid or invalid constant as described above. The length of the InputLine is automatically adjusted when a History icon is added or deleted.

An example of generated code is:

```
R.Assign(3, 27, 17, 28);
IL:= New(PRangeInputLine, Init(R, 10, 0, MaxLongint));
Insert(IL);
R.Assign(3, 25, 21, 26);
Insert(New(PLabel, Init(R, 'A~d~der Stages per', IL)));
R.Assign(17, 27, 20, 28);
Insert(New(PHistory, Init(R, IL, hiPipe)));
```

(Declarations of the variables R and IL, and the constant hiPipe are also generated. Since PRangeInputLine is defined in the TvInput unit, "uses TvInput" is also declared.)

9.5 RadioButtons Dialog ----------

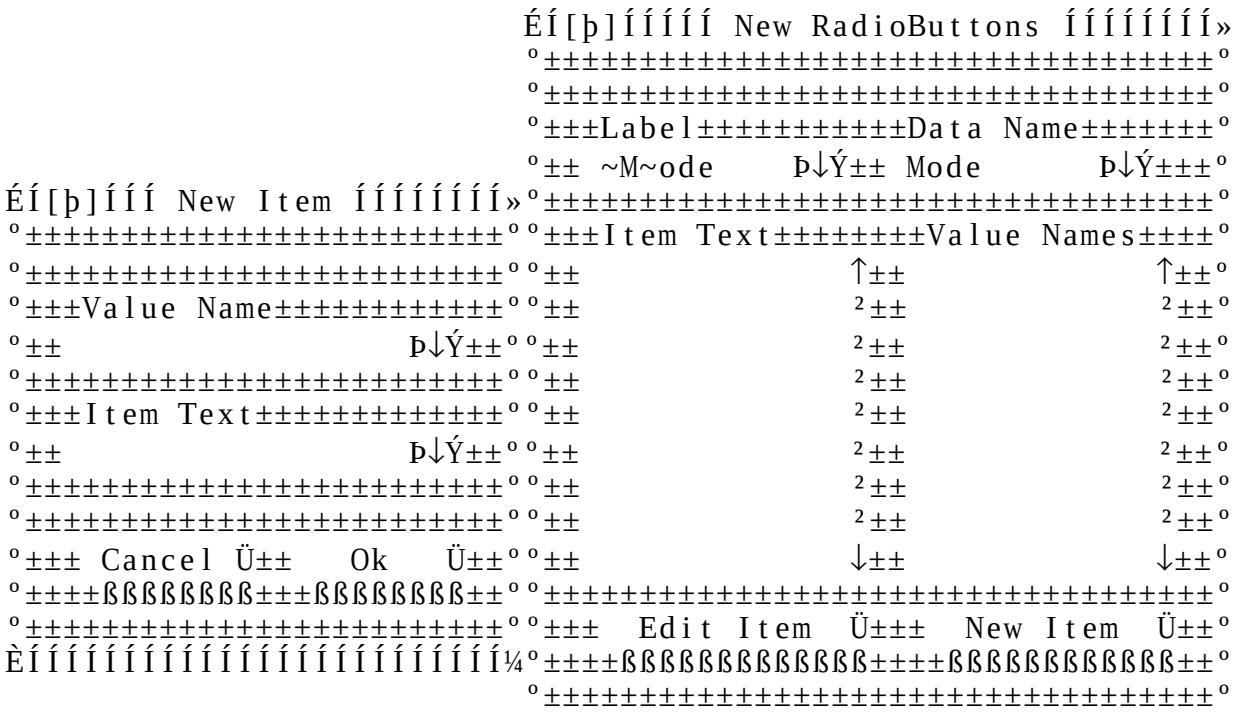
An example of generated RadioButtons code:

```
const
     ipCorrelator = 0;  {Pattern values}
     ipSummer =     1;
     ipMultiplier = 2;
     ipSquarer =    3;
     ipArbitrary =  4;
...

     R.Assign(3, 5, 19, 10);
     RB:= New(PRadioButtons, Init(R,
          NewSItem('~C~orrelator',
          NewSItem('~S~ummer',
          NewSItem('~M~ultiplier',
          NewSItem('S~q~uarer',
          NewSItem('~A~rbitrary',
          nil)))))
     ));  Insert(RB);
     R.Assign(3, 4, 18, 5);
     Insert(New(PLabel, Init(R, '~I~nput Pattern', RB)));
```

The Label and Data Name items are the same as for InputLine.

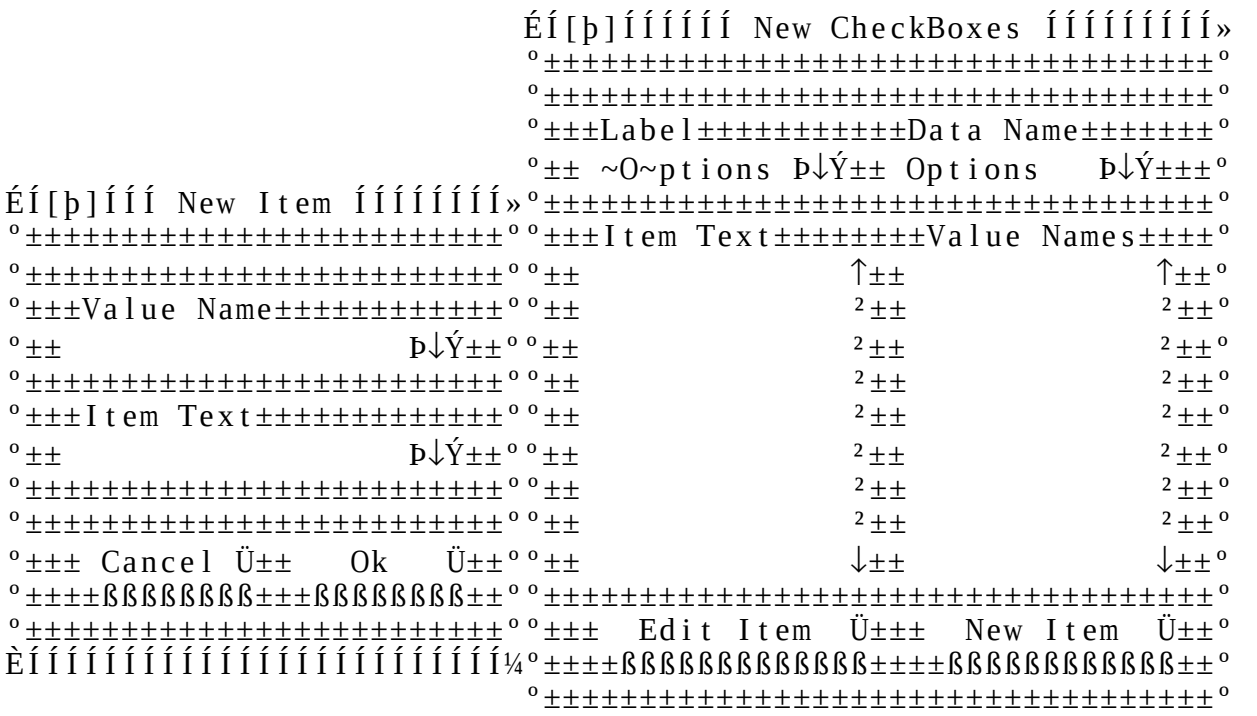
The Item Text ListBox is the place to enter the names to be shown for the CheckBoxes (such as '~C~orrelator' in the above example). The Value Names ListBox is the place to enter the names of the selection values (such as 'ipCorrelator' in the above example). Click the 'New Item' button, or press the Enter key (the 'New Item' button is the default) to enter a new item. This executes the 'New Item' dialog shown next to the 'New RadioButtons' dialog, below. Enter names in the 'Item Text' and 'Value Name' input lines, and click 'Ok' (or press Enter).



New 'Item Text' and 'Value Name' strings are entered at the selected (highlighted) position in the 'Item Text' and 'Value Names' ListBoxes. To change an item name, select it (click on it, or use arrow keys to move the selection) in the 'Item Text' ListBox, then click the 'Edit Item' button (or press space key). This executes a 'Change Item' dialog that looks like and works like the 'New Item' dialog. To delete an item, change it to an empty string. (Items that begin with a space character are also deleted.)

The edit dialogs for RadioButtons, CheckBoxes, and MultiCheckBoxes (the Cluster descendants) each have 'Item Text' and 'Value Names' ListBoxes as described above. The Save button in these dialogs copies the 'Item Text' and 'Value Name' data into a Paste buffer area shared by all three Cluster types, but the other edit dialog data is copied to separate Paste buffer areas for each Cluster type.

9.6 CheckBoxes Dialog



The Label and Data Name items are the same as for InputLine.

The 'Item Text' and 'Value Names' ListBoxes and the 'Edit Item' and 'New Item' buttons are used as described for RadioButtons.

The data field transferred to and from a CheckBoxes object is a word. This works OK if the number of check boxes is 16 or less, or if the Use32 unit is used to declare a word to be a longint. If the number of check boxes exceeds 16, the generated code will include this warning comment after the word declaration:

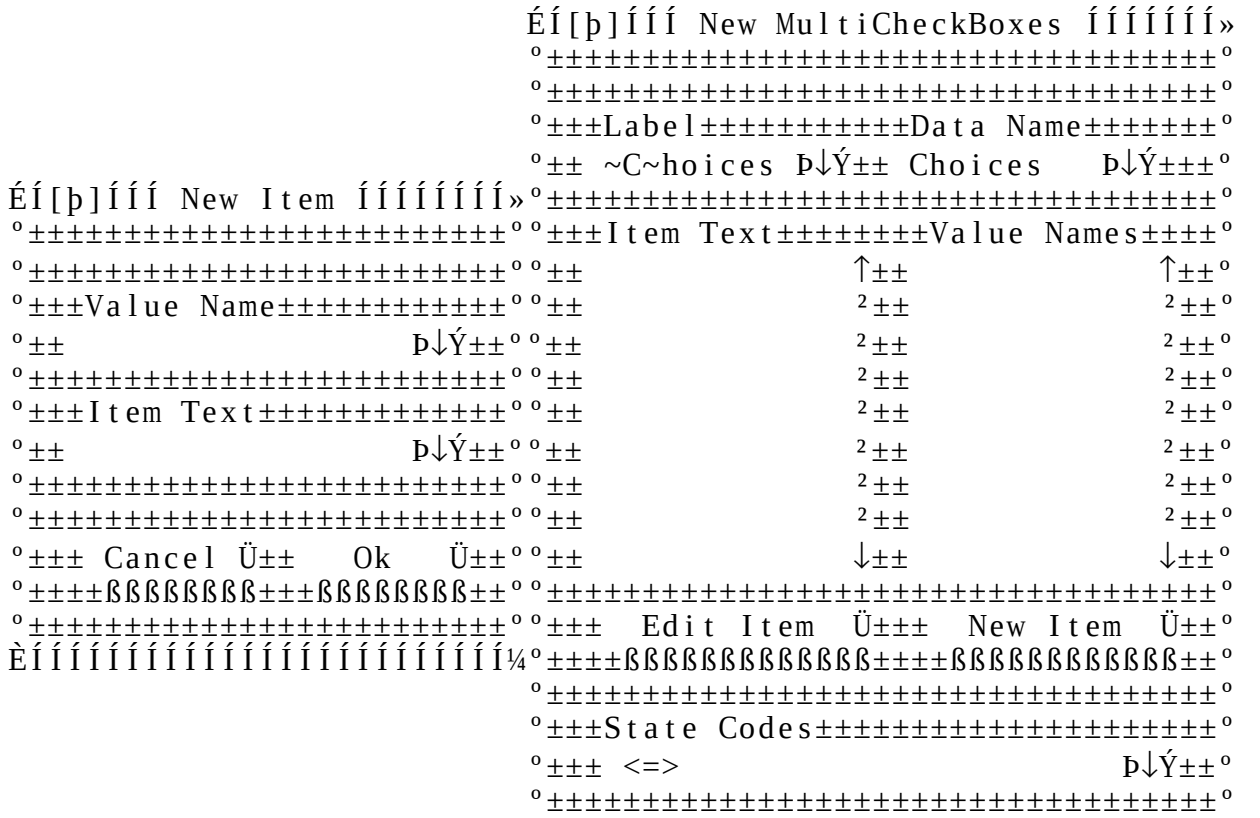
```
{+ WARNING: Need Use32 +}
```

9.7 MultiCheckBoxes Dialog

The Label and Data Name items are the same as for InputLine.

The 'Item Text' and 'Value Names' ListBoxes and the 'Edit Item' and 'New Item' buttons are used as described for RadioButtons.

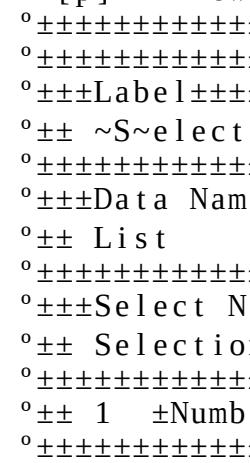
The State Codes input line provides a string that lists the characters that represent the states of each MultiCheckBox. The corresponding values are encoded as 0,1,2.. in the order listed, but when the real MultiCheckBox is used, the space key (or clicking an item) changes the state of the selected item in the reverse order.



Two other parameters used to initialize the MultiCheckBoxes are derived from the States string: the number of states (Range) is the length of the States string; and the number of bits used to represent each state (item value) is derived from the number of states. If the total number of bits needed to encode all item values exceeds 32 (the number of bits in a longint), a warning comment is inserted in the generated code such as:

```
{+ WARNING: 40 bits needed! +}
```

9.8 ListBox Dialog ----------



The Label item is the same as for InputLine.

A ListBox has two data fields in the data record associated with the dialog: a pointer to a StringCollection, and an integer index indicating the currently selected string in the collection. The Data Name item specifies the name of the PStringCollection field, and the Select Name item specifies the name of the selection index field.

The Number of Columns item specifies the number of columns of the ListBox.

10. Notes ==========

These notes are about use of TurboVision objects.

10.1 Using a ListBox: ----------

An item can be selected by double-clicking it with the mouse, or moving the highlight with the cursor keys and pressing space. When an item is selected, the ListBox executes:

```
Message(Owner, evBroadcast, cmListItemSelected, @Self);
```

But to use this feature, you need to override the HandleEvent method of the Dialog and provide code to respond to the broadcast command as desired.

10.2 Names for TCollection Methods that operate on items ------

Borland's choices of these names are inconsistant and confusing. The worst is the overriding of TObject.Free for a different purpose. The following translation is offered, where the 'obvious' names match the ordinary use of the names free, delete, dispose, load, and store as used elsewhere in the TurboVision library.

obvious name	TurboVision name
FreeItem	Free
DeleteItem	Delete
DisposeItem	FreeItem
LoadItem	GetItem
StoreItem	PutItem

==========