ViewEdit User's Guide

preliminary draft 1

• WARNING • This is the first alpha release of ViewEdit. It is untested and not yet beta quality. It should be used with extreme caution: it may crash your system or even erase files from your hard disk. Save your work often and don't use ViewEdit when a system crash would cause you to lose data.

Preface

Welcome to the ViewEdit User's Guide, and to the power of ViewEdit. ViewEdit is a MacApp utility program that allows you to create view hierarchies in a what-you-see-is-what-you-get editing environment, rather than in a compiled resource file.

ViewEdit gives you as much view-editing power as Rez but provides Commando-like dialogs for entering values into each 'view' resource field. This relieves you of having to remember which fields and values are associated with which 'view' types.

ViewEdit also allows you to draw, resize, and move your views using the standard Macintosh interface. It even creates and rearranges your view hierarchies as you go!

Before you read this document, you should understand these concepts:

- **View hierarchies.** These are introduced in the "Architecture" section of Chapter 4 of the *MacApp 2.x Manual (Interim Version).*
- **View classes.** These are described in the MacApp® 2.0B5 Display Architecture Release Notes.
- **View resources.** These are explained in the "Creating View Templates" recipe in Chapter 7, "The CookBook", in the *MacApp 2.x Manual (Interim Version).*

This guide is divided into two parts: a step-by-step tutorial and a command reference. ViewEdit is simple enough to use that you may find you won't need to refer to this manual frequently. However, you should read through this manual at least once, for there are many shortcuts and features hidden in ViewEdit, as well as a few eccentricities.

A first look at ViewEdit

This section shows you how to use ViewEdit to edit the view resources in the DemoDialogs sample application. Before you begin this tutorial, you should build the DemoDialogs sample. If you are new to MacApp and the MPW environment, Chapter 6, of the *MacApp 2.x Manual (Interim Version)*, "How to Install and Use MacApp", will show you how to build the sample applications.

After building DemoDialogs, open the ViewEdit application by double-clicking on its icon.

The resource file window

The first thing that you will see after starting ViewEdit is the empty "Untitled-1" window. Close this window and choose the Open command from the File menu. Then select the DemoDialogs application from the Standard File dialog.

The window that appears is reminiscent of the resource file window in ResEdit. There is an icon for each type of resource in DemoDialogs. The icon for 'view' resources is always in the upper-left hand corner (and if you have a color screen, you will see it is also the only colored icon).

Here is the resource file window from DemoDialogs:

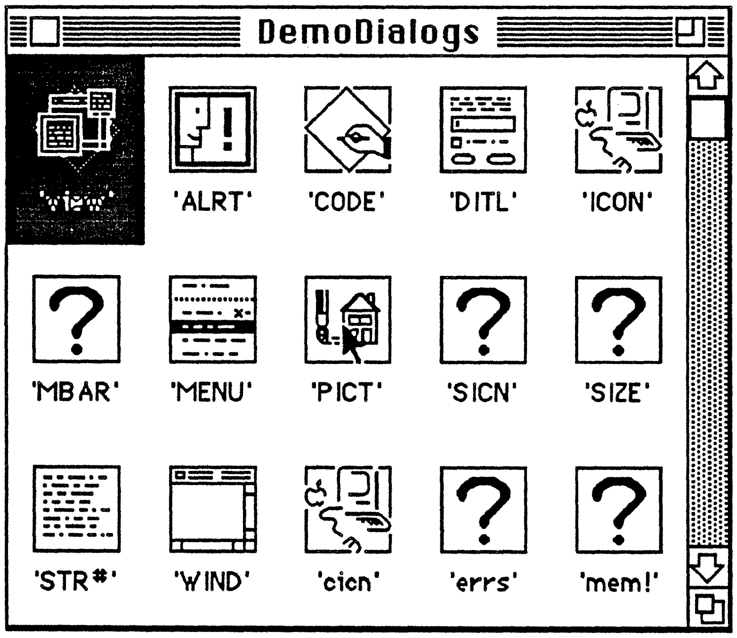


Fig 1.1 DemoDialogs resource file window

Because there is a view icon present, you can tell DemoDialogs already contains view resources. If no view resources existed, you could create an initial view resource at this point by choosing the Create Resource command from the Edit menu.

The resource type window

If you double-click on any of the resource type icons in the resource file window, you will open a window that contains a complete list of resources of that type. Alternatively, you can select the resource type with the cursor keys and press the Return or Enter key.

For example, double-clicking on the 'ICON' resource icon will open this window:

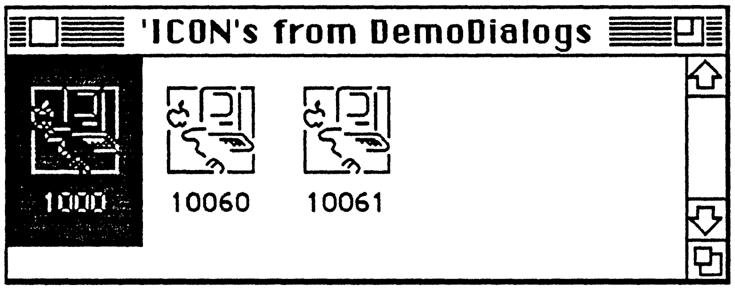


Fig 1.2 'ICON' resource type window

ViewEdit will display the ID's of a variety of resources, but the only resources it allows you to examine and edit are views. If you try to "open" any of these 'ICON' resources, as you might in a complete resource editor, nothing will happen.

On the other hand, when you open the 'view' resource type window, which looks like this:

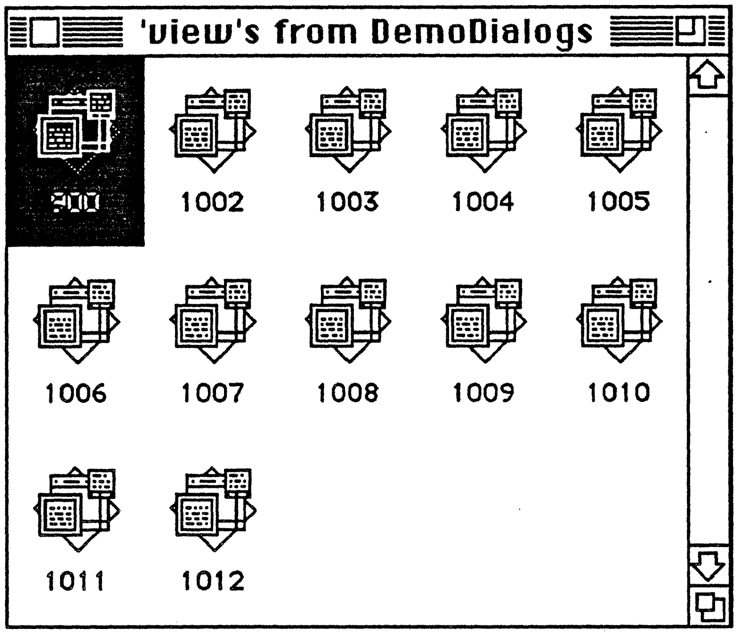


Fig 1.3 'view' resource type window

you have the ability to open and edit each of these resources. At this point you may also want to create a new 'view' resource, by choosing the Create Resource command from the Edit menu.

Double-clicking on any of the 'view' resource icons will result in opening a **view-editing window.** (Again, you can alternatively select a 'view' resource icon with the cursor keys, and then press the Return or Enter key.)

The view-editing window

Simply put, the view-editing window allows you to edit your view resources. Let's look at a sample view-editing window. Double-click on the icon representing 'view' resource ID 1008, which will open this view-editing window:

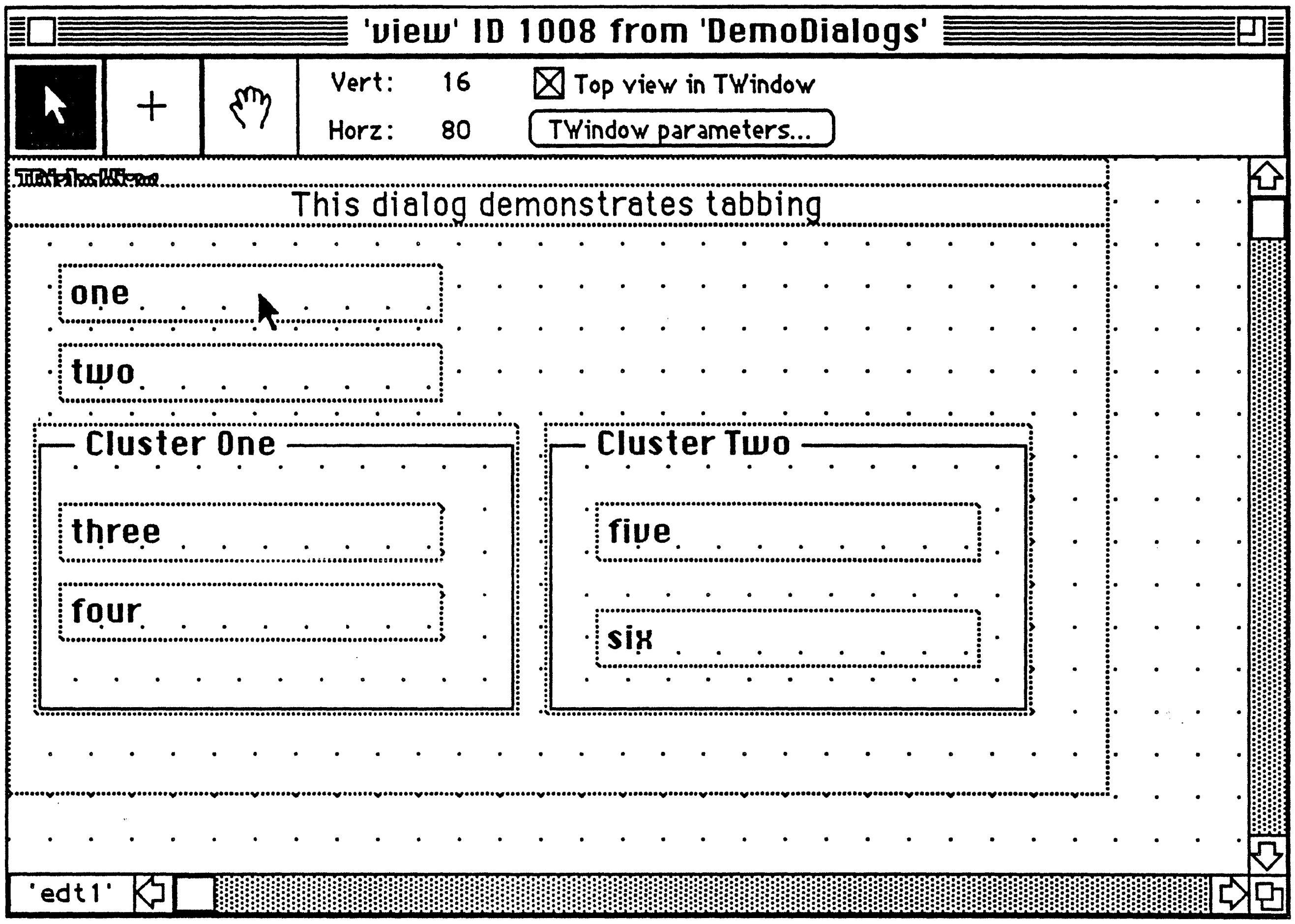


Fig 1.4 'View' ID 1008 view-editing window

To refresh your memory, this view resource corresponds to the "Tabbing Test" dialog in DemoDialogs, which looks like this:

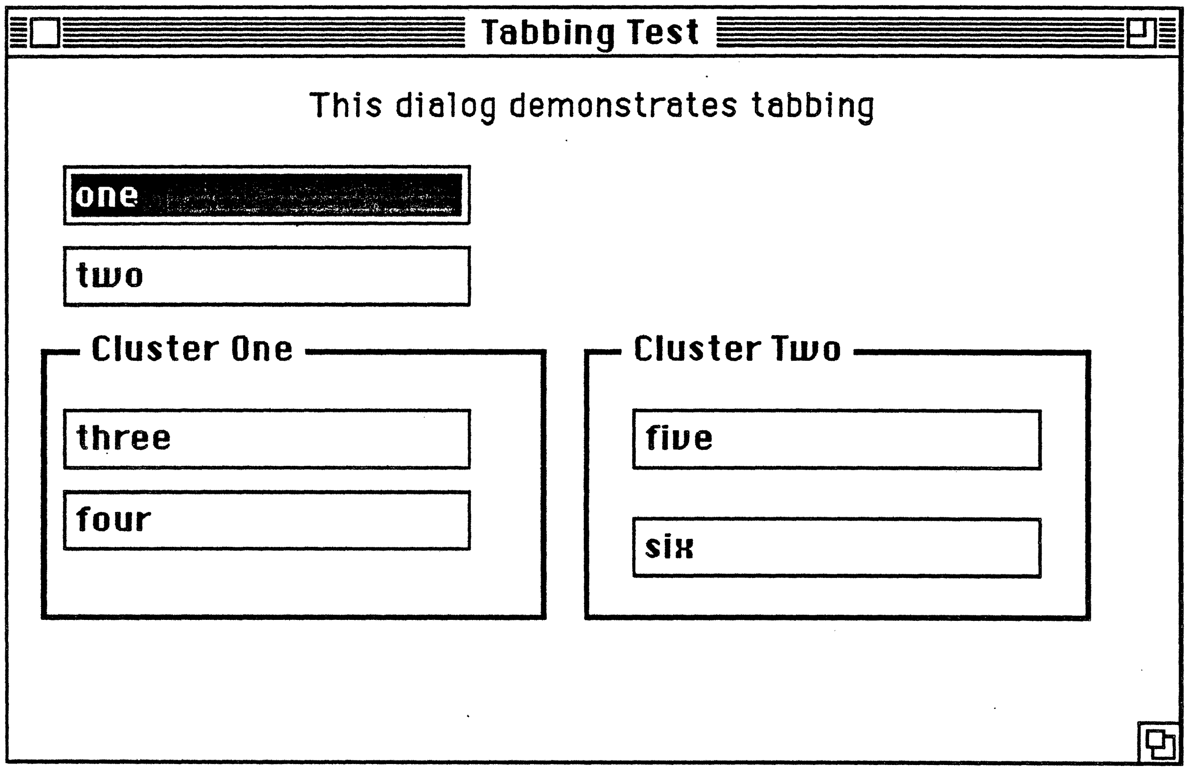


Fig 1.5 "Tabbing Test" dialog from DemoDialogs

As you can see, the view-editing window closely resembles the actual dialog from the application. However there are a few important differences.

First of all, the view-editing window has a palette attached to it. This palette consists of three tools, a vertical and horizontal location indicator, a check box, and a button. The elements of this palette, called the control palette, are described in the following sections.

Under the control palette is the actual representation of the view resource. Each view in the view resource hierarchy is represented here. Each visible view is surrounded by a dotted line, and certain view types have other visual clues. For example, the static and editable text views in this view-editing window are filled with their initial text. Similarly, the cluster views each are drawn with their label and border. This allows the view-editing window to duplicate the actual dialog as closely as possible. However some views, like dialog views, have no visual representation in the final application. Views of this sort are labeled by their type name in their upper left hand corner. In the current view-editing window, you can see the label "TDialogView" (which is partially hidden) in the upper left hand corner of the entire view-editing window. As you see more and different types of views, you'll see exactly how much of a visual clue is associated with each type. You might be surprised how faithful some of these visual cues are.

You might be wondering why the "TDialogView" label is partially hidden. This is due to the implementation of the view hierarchy. In the view-editing window, superviews are placed under their subviews. In this example, the TDialogView view is the superview of the static text view that contains the text "This dialog demonstrates tabbing." Therefore, everything associated with the TDialogView view (of which the "TDialogView" label is the most conspicuous part) is drawn below the uneditable text view. The same holds true for the first two editable text views and the two clusters. Each cluster is the superview of two more editable text views; so once again, the subviews are drawn on top of their superview.

Another aspect of the view-editing window you might notice is that it is covered with dots. These dots represent the current **grid** of the window. This grid, like the grid in MacDraw, constrains the drawing and moving of views.

As you move the pointer around the view-editing window, several items are updated. In the bottom-left-hand corner a message box which displays the identifier of the view the pointer is currently over. In Figure 1.4, the identifier is 'edt1'. Also as you move the pointer around, you will see its current coordinates appear in the Vert and Horz fields in the control palette. The coordinate values displayed in these fields are always relative to the local coordinate system of whichever view the pointer is currently over. For example, in Figure 1.4, the pointer is positioned over the editable text view containing the text "one." The vertical and horizontal coordinates shown are 16 and 80. Since the local coordinate system of this text box starts at (0, 0) in its upper left hand corner, this means that the pointer is currently 16 pixels down and 80 pixels over from the upper left hand corner of this box.

• NOTE • Actually, the coordinates shown are "snapped" to the current grid. In other words, they are rounded to the nearest value allowed by the current grid resolution.

Another important point to note is that the grid is relative to the origin of every view. If some subview is not currently aligned to its superview's grid, then its grid will be slightly shifted from its superview's grid. This is the case with every subview in Figure 1.4.

The selecting tool

The first of the three tools in the control palette is the selecting tool, which is the standard "pointer" cursor. It is the default tool when you open a view-editing window. You can choose this tool either by clicking on its icon in the control palette or by choosing the Select Views command in the Mode menu. With this tool you can select, resize, and open a view.

Selecting a view, as you might imagine, requires a single click anywhere in that particular view. Of course, if the view is covered by other views, you will have to find some uncovered piece to be able to select it—otherwise you'd have no place to click.

You can execute editing commands on the selected view, including Cut, Copy, and Clear from the Edit menu. You can also Delete by pressing the Delete key.

A selected view looks like this:

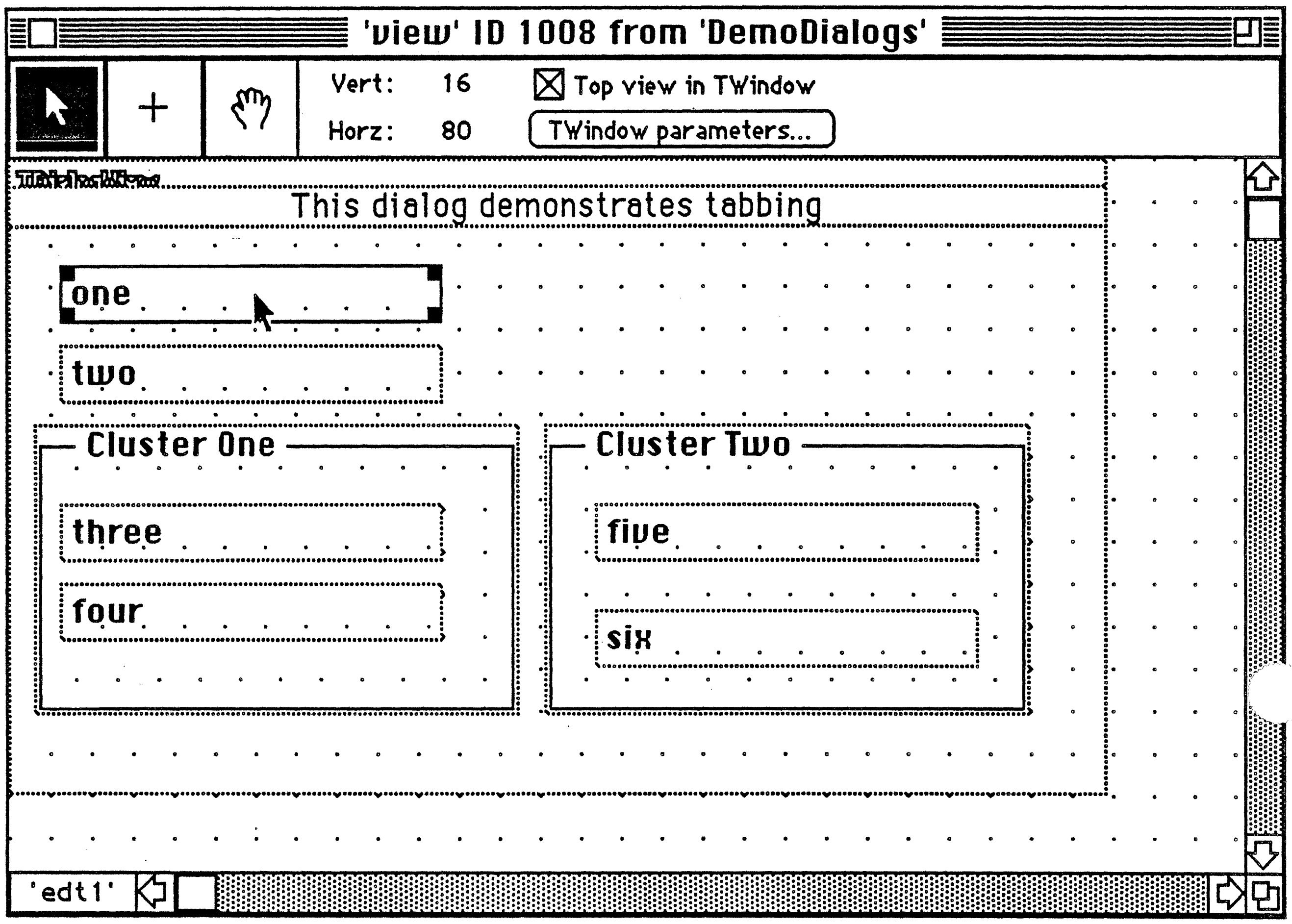


Fig 1.6 A selected editable text view

Notice that four **handles** have appeared—one in each corner of the view. These handles allow you to resize the view, by clicking and dragging. For example, if you drag the lower-right handle to the right, you can resize the view accordingly:

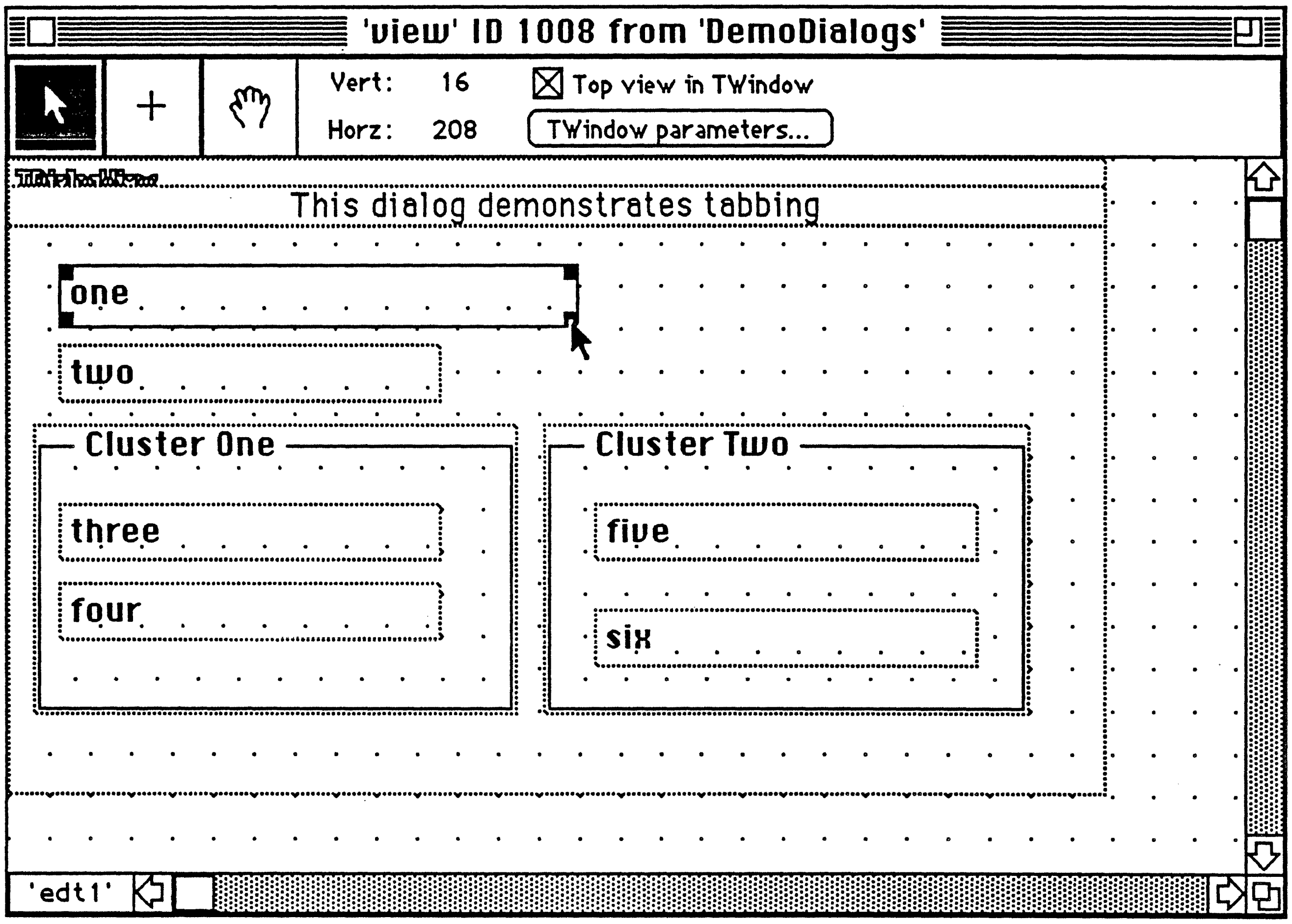


Fig 1.7 Resizing a view

If you're performing each step of this tutorial in ViewEdit, you probably noticed the effect of the grid. Although the editable text view was originally not aligned with its superview's grid, once you grabbed that handle and started to move it, it immediately snapped to the nearest grid point (in the superview's grid). Unless you recalibrate or turn off the grid, that corner will stay aligned to grid points for each subsequent move or resizing.

Notice only that corner handle is now completely aligned to the superview's grid. To align the others, you need to select and move them. For example, you can now align the upper right handle by grabbing and dragging it:

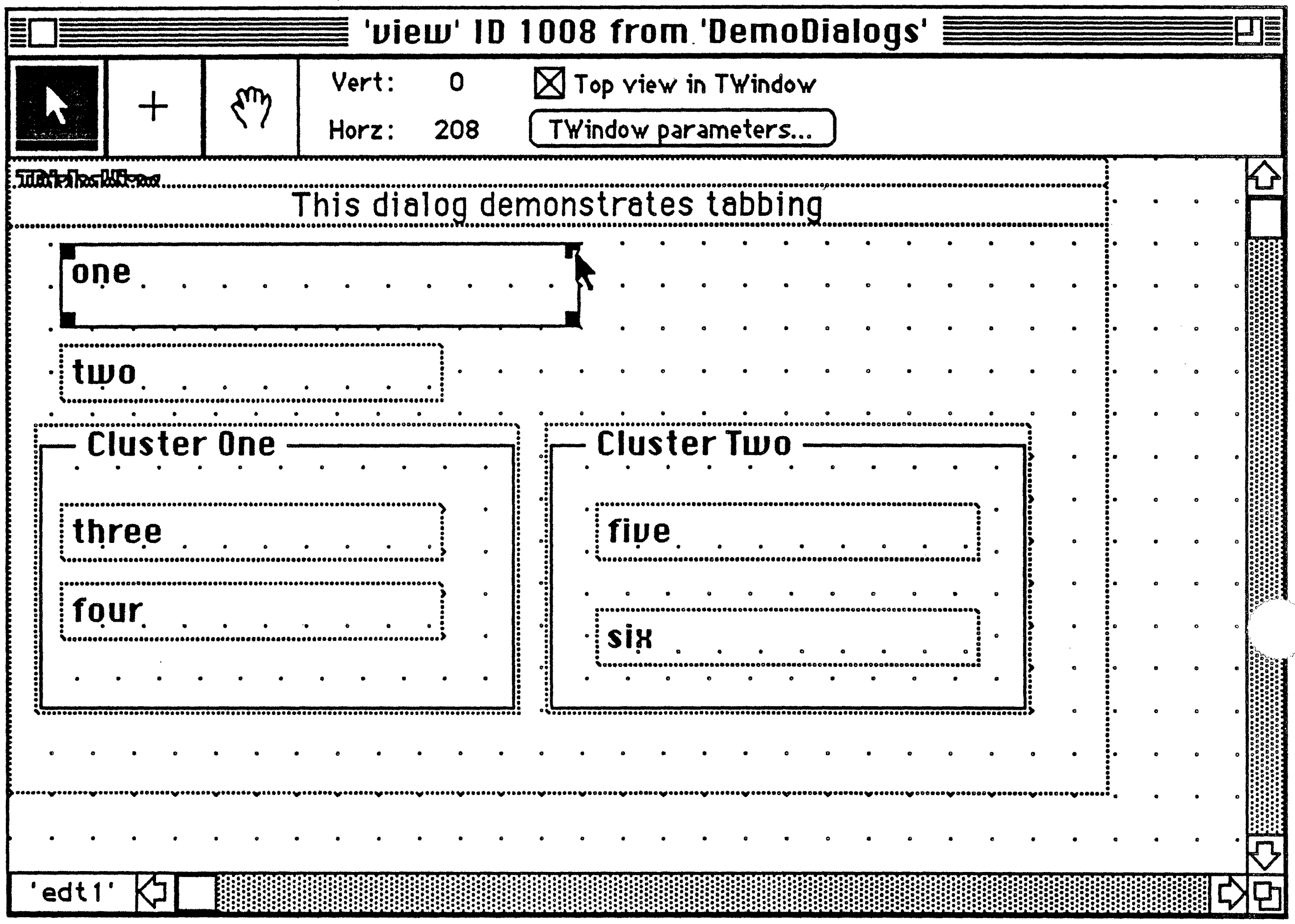


Fig 1.8 More view resizing

Of course, if you only wish to align a view to its superview, you don't have to move each handle. You can simply choose the Snap to Grid command in the Arrange menu.

At this point, you may not like the new position of the editable text view. You can always revert to the last saved version of the entire file (which would be acceptable, since this is the only change so far), or you can resize the view back to its original size. To do this, you need to manipulate the grid.

There are a number of ways that you can manipulate the grid. You can make the grid invisible by selecting the Show Grid command in the Arrange menu (which is a toggle command) but this won't turn the grid off. To do that, you must choose the Grid Values command from the Arrange menu, unhighlight the Use Horizontal Grid and the Use

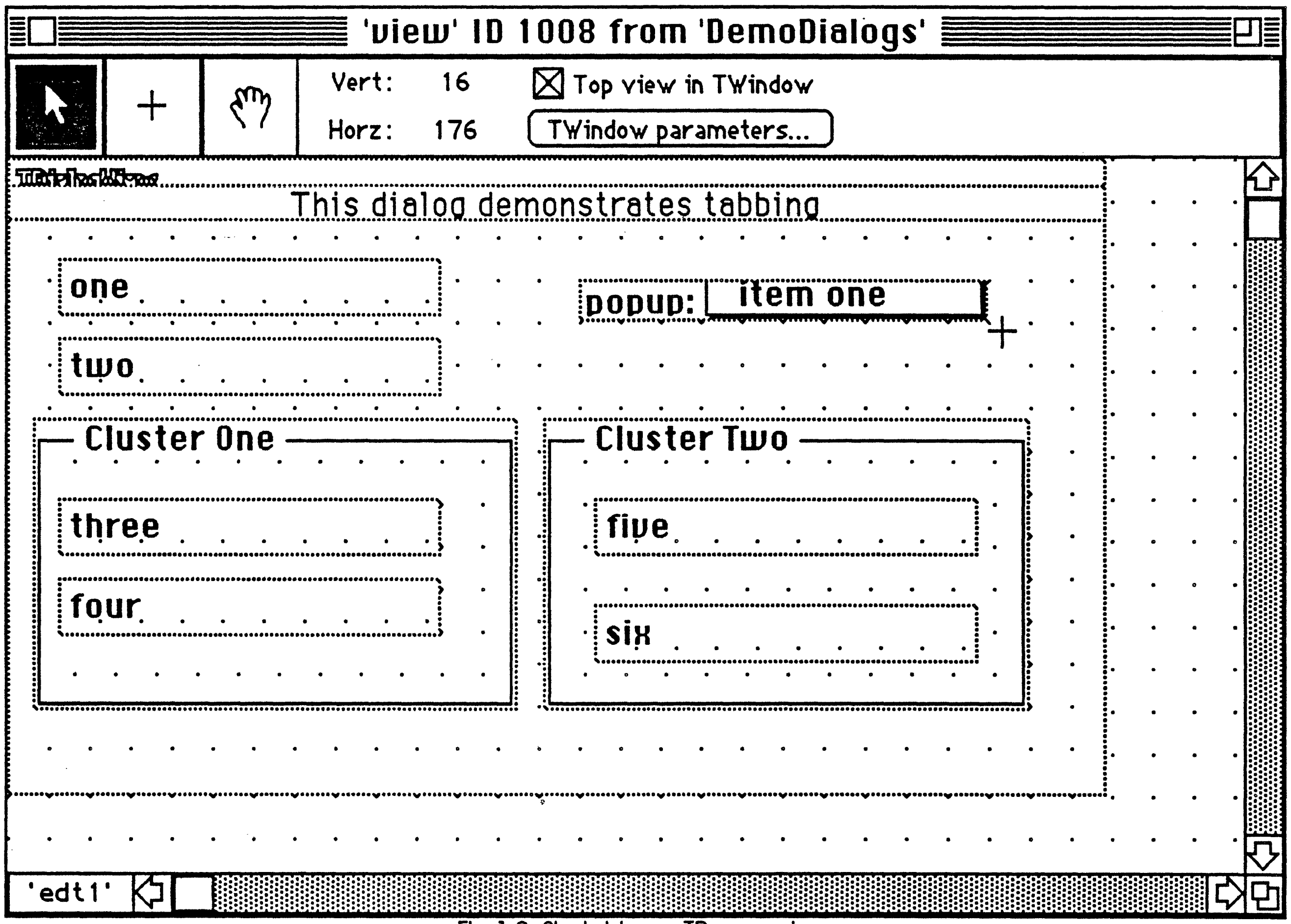
Vertical Grid check boxes in the Grid Values dialog box. Notice you can also change the grid resolution in this dialog box. (This simultaneously changes the grid in every view of the view editing window.)

When a view is selected, you can automatically resize it to align to the current grid of its superview by choosing the Snap to Grid command in the Arrange menu. You can also rearrange the order of subviews in a superview. (Remember that the order of subviews affects how they appear on the screen. A subview "in front" of another subview will cover the other subview if their boundary rectangles intersect.) The Send Back and Bring Forward commands in the Arrange menu allow you to reposition the selected view.

The sketching tool

The sketching tool (a crosshair pointer) allows you to create new objects by **sketching**—clicking and dragging to define the boundary rectangle of the new view, much as you would draw a rectangle in MacPaint. The view that is created is added to the view hierarchy as a subview of whichever view the initial click occurred in. The type of view that is created is whichever view type is currently selected in the Views menu.

For example, if you select the TPopup view type from the Views menu and sketch the following rectangle:


'view' ID 1008 from 'DemoDialogs'

Vert: 16

Horz: 176

[✓ Top view in TWindow]

'edt1'

Fig 1.9 Sketching a TPopup view

A TPopup view is created as a subview of the TDialogView view. Notice that this sketching, as you might expect, is constrained to the current grid.

There are three ways to select the sketching tool. Like the selecting tool, you can select the crosshair icon from the control palette or you can choose the Draw Views command from the Mode menu. As a third alternative you can hold down the command key. This will change the selecting tool into the sketching tool for as long as you hold the command key down. (If you release the command key while actually doing a sketch, the sketching tool will remain until you release the mouse.)

• WARNING • A final important fact about sketching a new view: the new view is not automatically selected. In other words, if the editable text view containing "one" was selected before you sketched the TPopup view, that text view will remain selected even after you've created the new view. This means that if you press the delete key immediately after sketching a new view, thinking that you will delete it, you may actually delete an entirely different view—the last one selected. Fortunately, the Undo command will bring back the lost view.

The moving tool

The moving tool (represented by the "grabbing hand" icon) allows you to grab a view and move it to another location, as well as to another place in the view hierarchy.

As with the sketching tool there are three ways to select the moving tool: selecting the grabbing hand icon from the control palette, choosing the Drag Views command from the Mode menu, and holding down the Option key. This la choice is a shortcut which temporarily changes the selecting tool into the moving tool.

Moving a view has interesting repercussions on your view hierarchy. If you move a view out of the boundaries of its superview, the following dialog box appears:

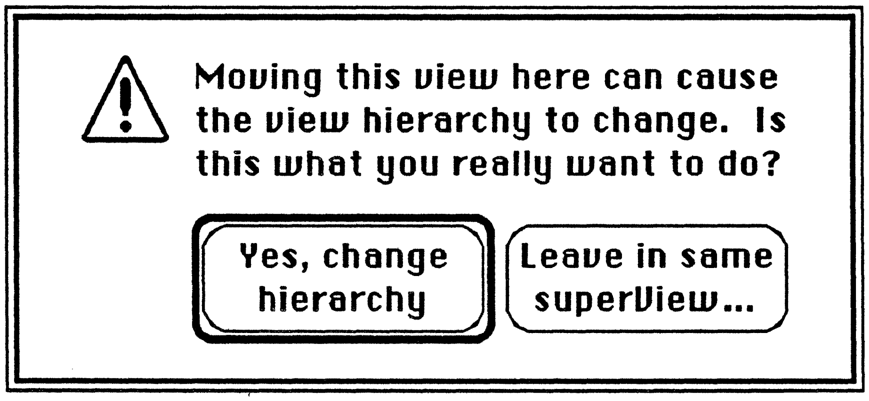

Moving this view here can cause the view hierarchy to change. Is this what you really want to do?

Fig 1.10 Change of hierarchy dialog box

This dialog box is straightforward: you can have the view remain a subview of the same superview as before, despite the fact that you've moved it, or you can have the view become a subview of whichever view its new location implies.

When you move a view, it is the position of its upper-left corner that decides which view is the "proper" superview. In other words, if the upper-left corner of the view moves out of the boundary rectangle of its superview, and into another, then the dialog box in Figure 1.10 appears, and if you specify that you did intend to change superviews, then it is the location of this upper-left corner that decides which is the new superview.

To avoid this dialog box altogether, you can hold down the Control key (on keyboards that have a Control key) while you are moving the view. This forces the view to change its place in the hierarchy.

As with the sketching tool, the moving tool does not select the item that is being moved—you must select with the selecting tool only. Be careful not to assume that the moved view is now selected and try to delete it by pressing the Delete key!

The view description dialog

You've seen how to select a view by clicking on it with the selecting tool. If you double-click on a view, the view "opens" into a **view description dialog**. (Alternatively, you can press return or enter while the view is highlighted.)

A view description dialog, like a Commando dialog, contains fields, pop-up menus, and help information to allow you to control every aspect of the different view types. For example, if you double-click on the selected TDialogView, like this:

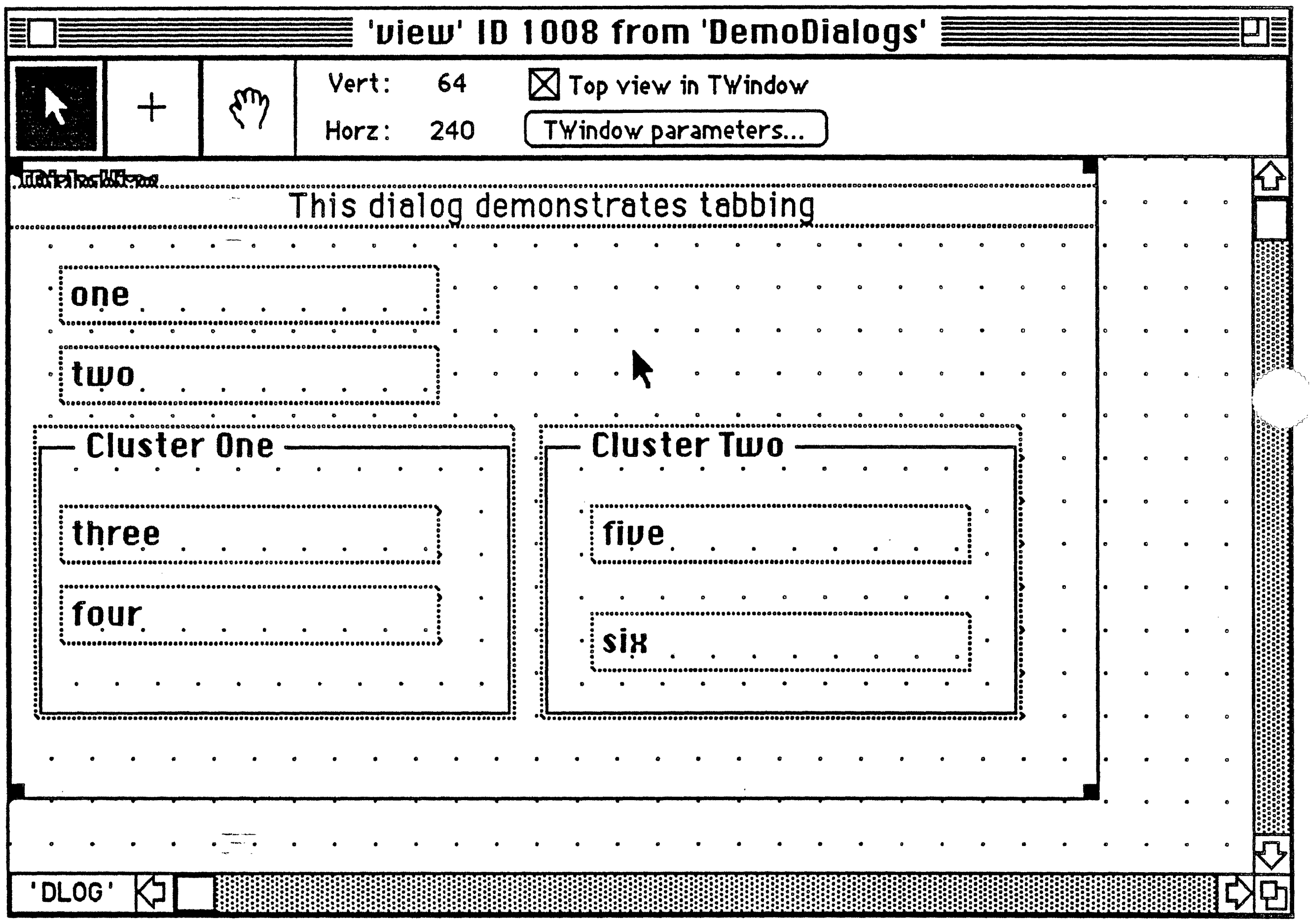
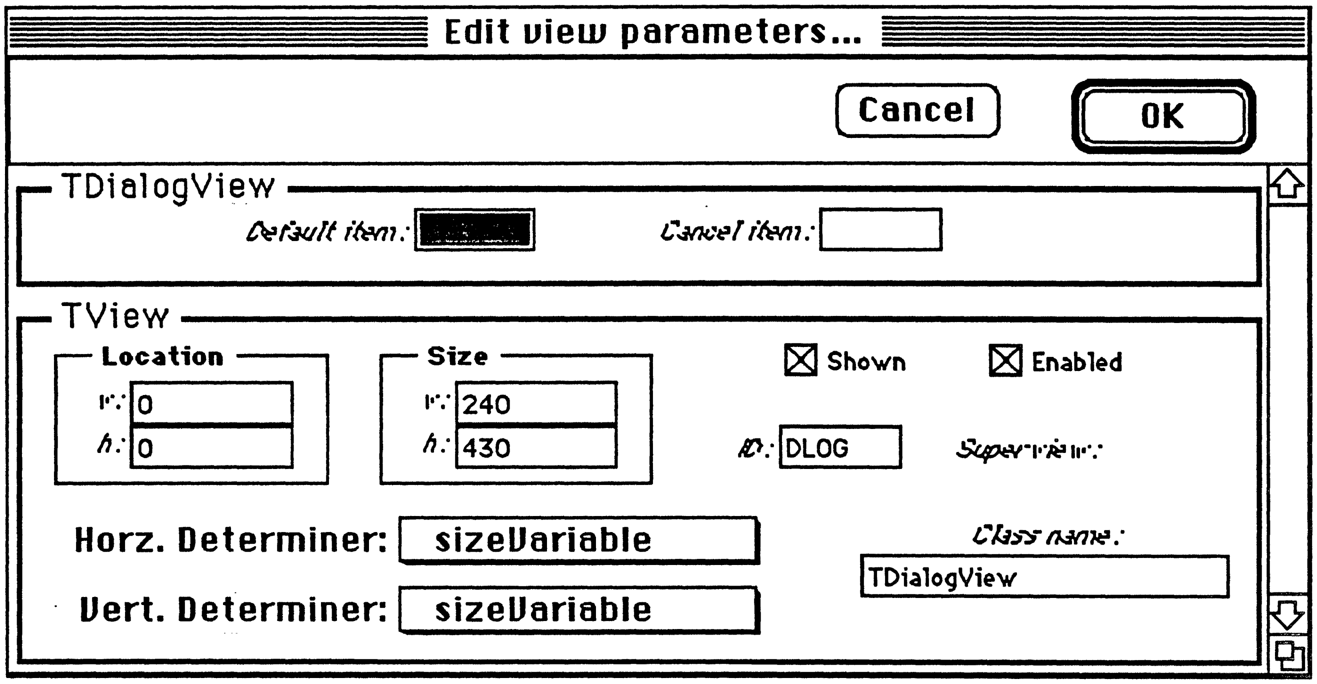


Fig 1.11 Opening the TDialogView view

you will open the following view description dialog:



TDialogView

Default item: [redacted] Cancel item:

TView

Location
v: 0
h: 0

Size
v: 240
h: 430

☒ Shown ☒ Enabled

ID: DLOG Superview:

Horz. Determiner: sizeVariable

Vert. Determiner: sizeVariable

Class name: TDialogView

Fig 1.12 A TDialogView view description dialog

This view description dialog consists of two parts, or **view description boxes.** The upper view description box allows you to enter values associated with the fields of a TDialogView. The lower view description box allows you to enter inherited fields—in this case, those belonging to the TView class.

Each view description dialog follows this pattern—each has a number of view description boxes. The uppermost box represents the specific class of the view that was opened. Each successive box represents the ancestor class of the previous box, until the class TView is reached.

Let's take another example. If you double-click on the static text box, like this:

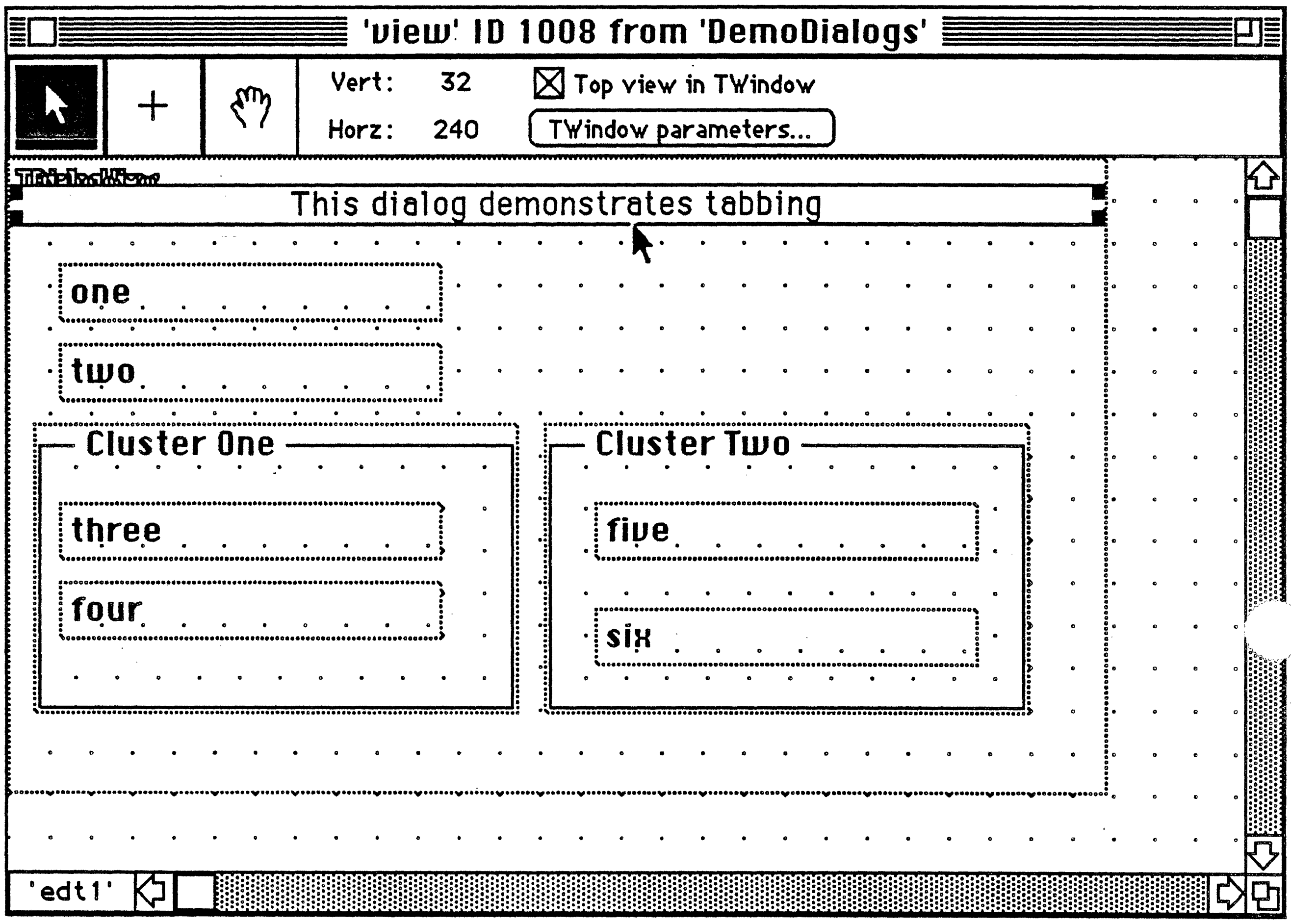
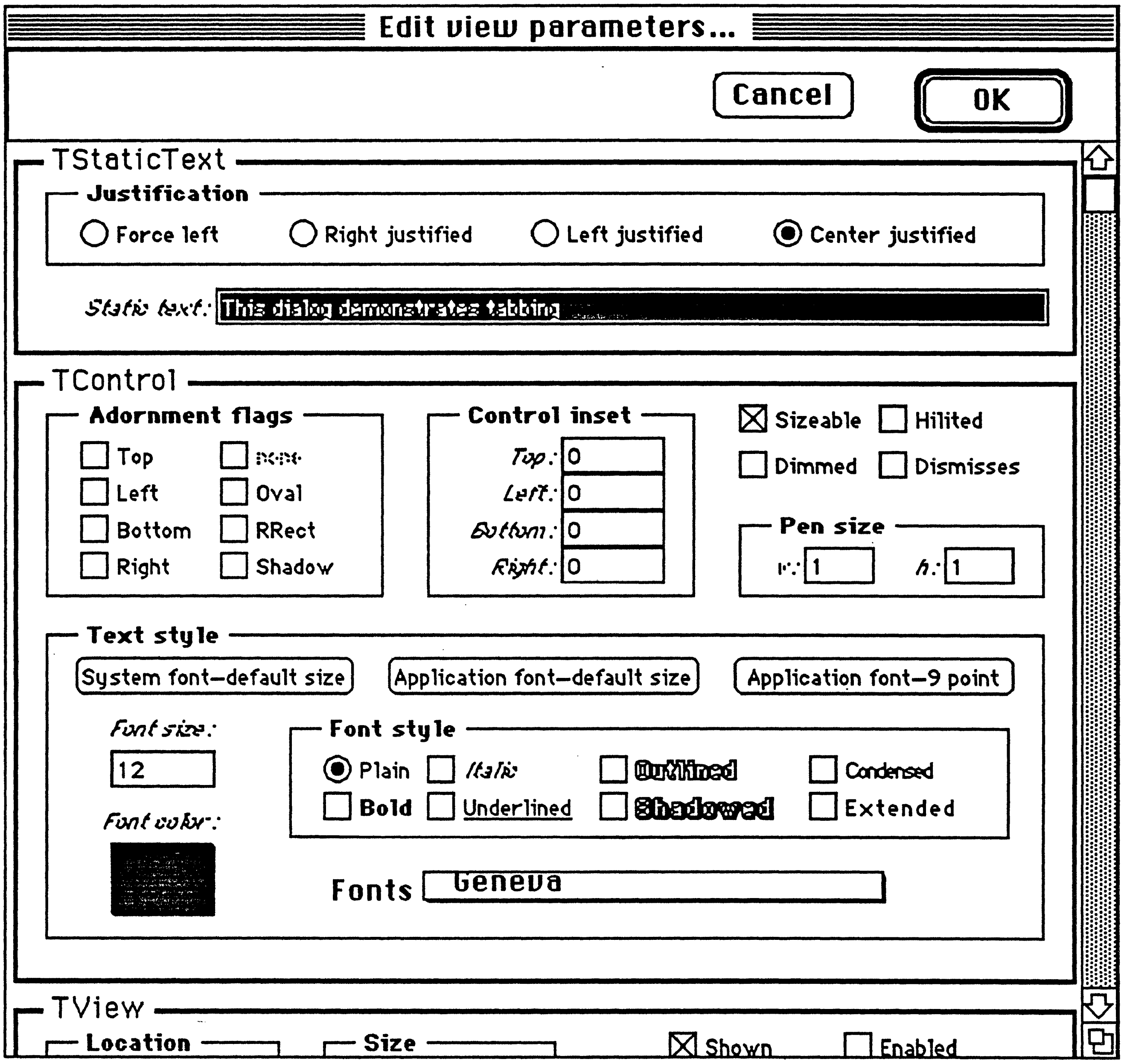


Fig 1.13 Opening a TStaticText view

Then you will open the following view description dialog:



TStaticText

Justification: Force left, Right justified, Left justified, [✓ Center justified]

Static text: This dialog demonstrates tabbing

TControl

Adornment flags: Top, Left, Bottom, Right, [illegible], Oval, RRect, Shadow

Control inset: Top: 0 Left: 0 Bottom: 0 Right: 0

[✓ Sizeable] Hilited Dimmed Dismisses

Pen size: v: 1 h: 1

Text style

Font size: 12

Font color: ■

Font style: [✓ Plain] Italic Outlined Condensed Bold Underlined Shadowed Extended

Fonts Geneva

TView

Location Size [✓ Shown] Enabled

Fig 1.14 A TStaticText view description dialog

As before, you can see a number of view description boxes—one for each class in the view class hierarchy between TStaticText and TView.

As a final example, open the view description dialog for a view of the TEditText class:

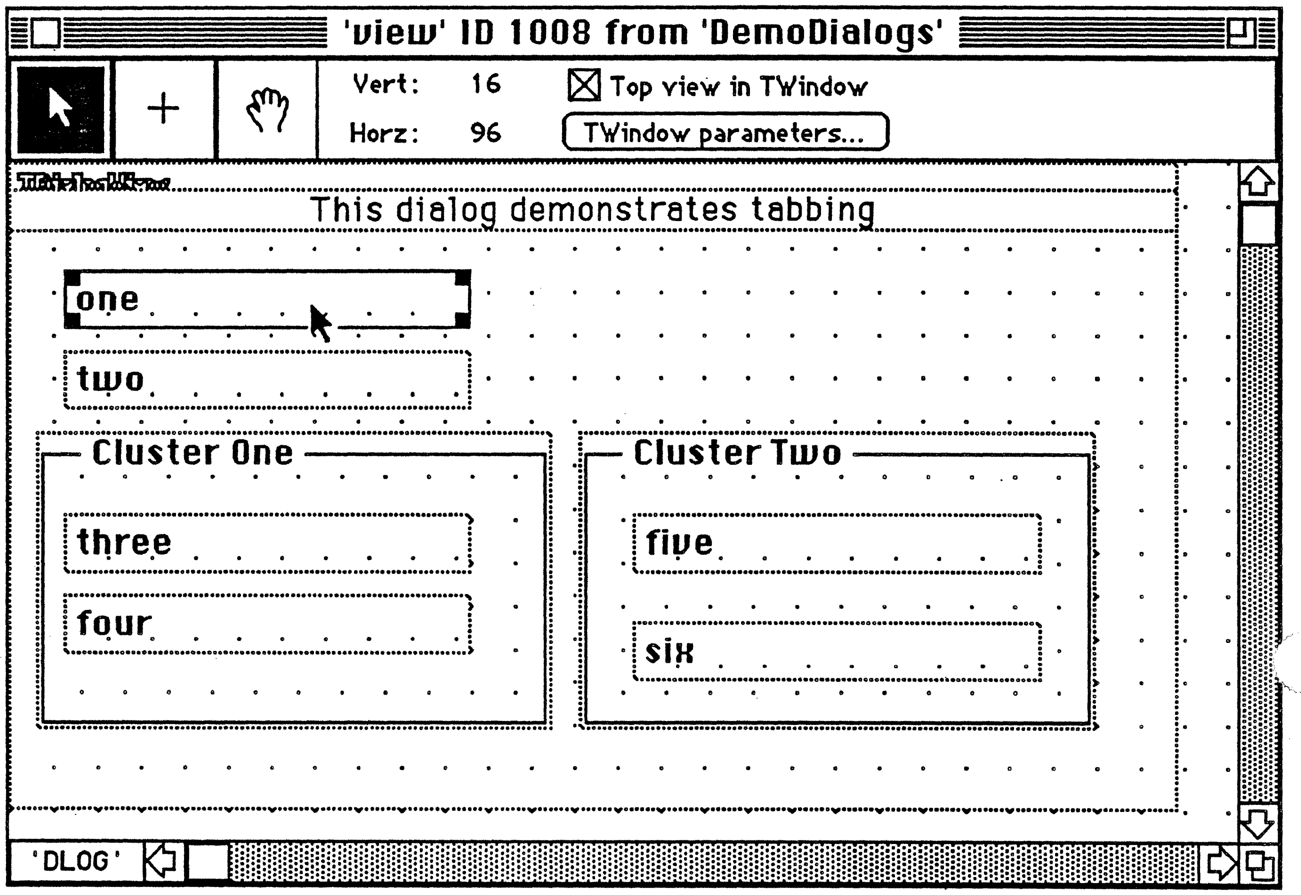
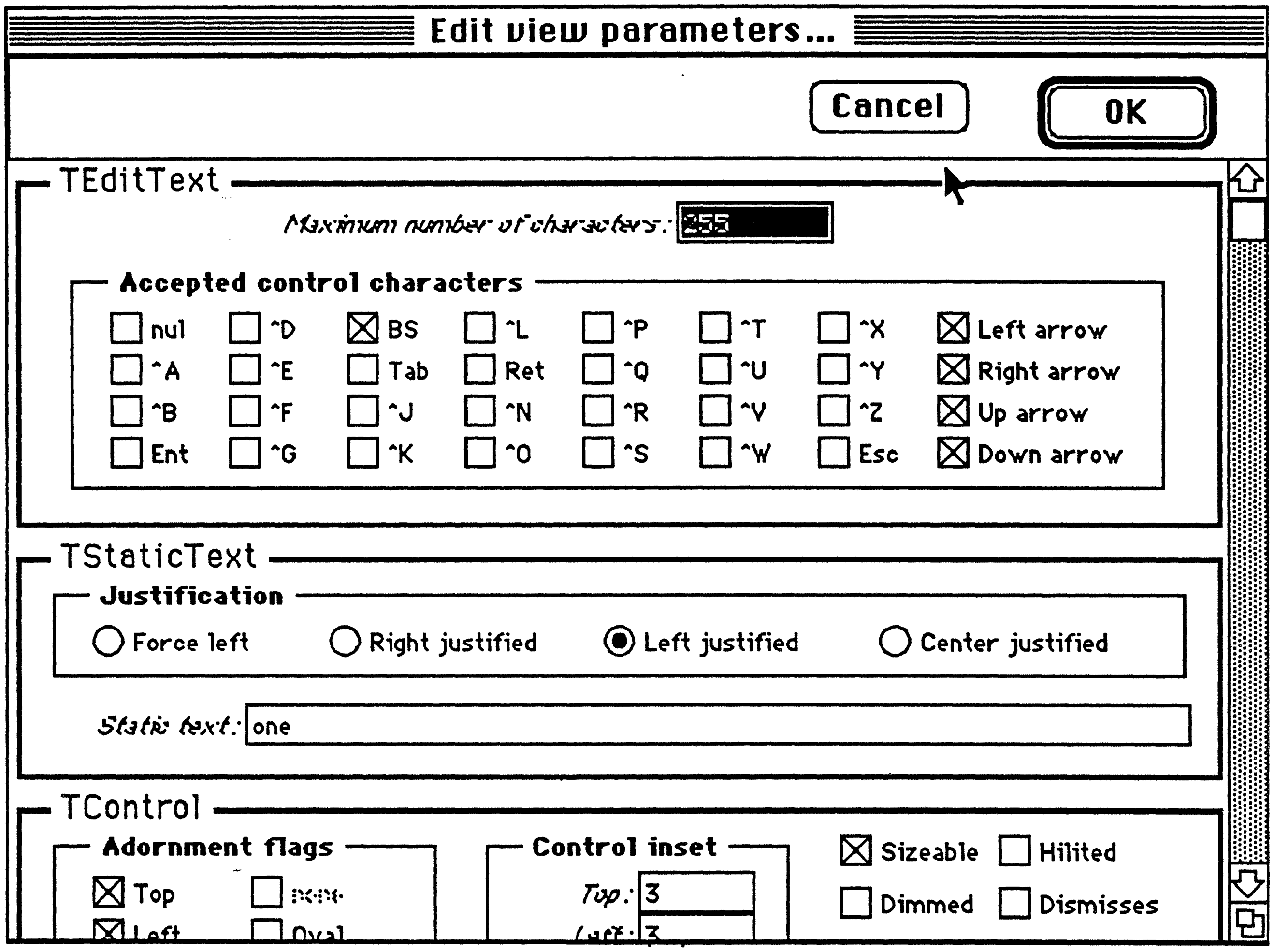


Fig 1.15 Opening a TEditText view

This view description dialog should look like this:



Edit view parameters...

Cancel OK

TEditText

Maximum number of characters: 255

Accepted control characters

nul	^D	[✓ BS]	^L	^P	^T	^X	[✓ Left arrow]
^A	^E	Tab	Ret	^Q	^U	^Y	[✓ Right arrow]
^B	^F	^J	^N	^R	^V	^Z	[✓ Up arrow]
Ent	^G	^K	^O	^S	^W	Esc	[✓ Down arrow]

TStaticText

Justification

Force left Right justified [✓ Left justified] Center justified

Static text: one

TControl

Adornment flags

[✓ Top] [illegible]

[✓ Left] Oval

Control inset

Top: 3

Left: 3

[✓ Sizeable] Hilited

Dimmed Dismisses

Fig 1.16 A TStaticText view description dialog

Here you can see an interesting change—the scroll bar actually becomes necessary for a dialog!

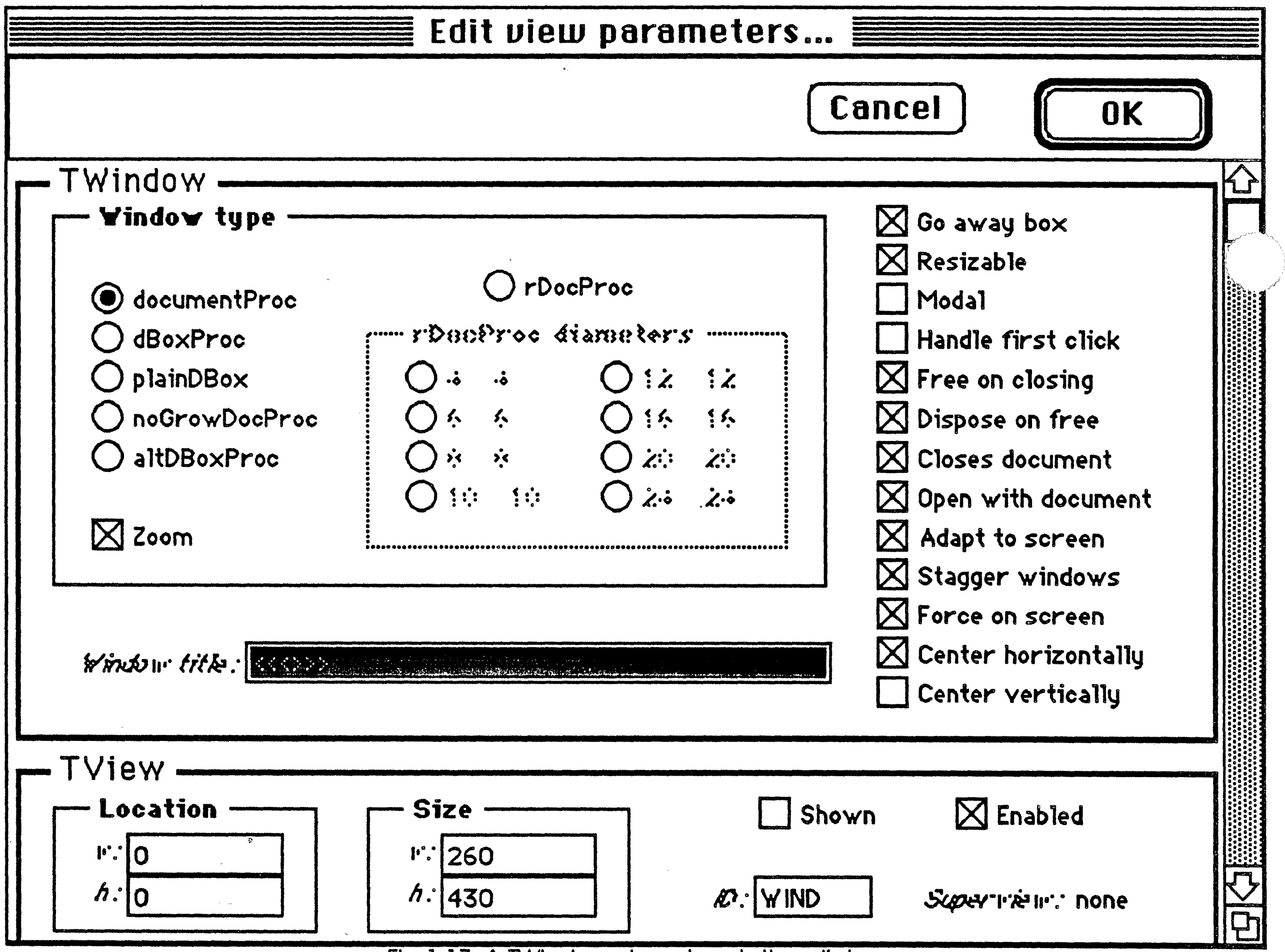
The different view description boxes allow you to input all of the information that a view resource file does. For further description of the different fields and what they mean, see the "Creating View Templates" recipe in Chapter 7, "The CookBook", in the *MacApp 2.x Manual (Interim Version)*, and the MacApp Release Notes. For an example of each type of view description box, see the "View Description Boxes" section at the end of this guide.

The window description dialog

So far, you've seen a way to create and edit every type of view object *except windows.* Window view objects are a special case because they must always be at the top of the view instance hierarchy.

If you want the views displayed in your view-editing window to be subviews of an actual window object, then click on the Top View in TWindow checkbox in the control palette. This tells ViewEdit that the view hierarchy you have created is to be made a subview of a TWindow view object. The topmost views in the view-editing window will be placed in the TWindow view in the same position that they are currently placed in the view-editing window.

To edit the fields of other view objects, you double-click in their boundary rectangle to open their view description dialog. Similarly, to edit the fields of window view objects, you click on the TWindow parameters button in the control palette. This opens the view description dialog for the window superview. For the 'view' ID 1008 from previous examples, this dialog looks like this:



TWindow

Window type

● documentProc
○ dBoxProc
○ plainDBox
○ noGrowDocProc
○ altDBoxProc

○ rDocProc

rDocProc diameters

○ 4 4
○ 6 6
○ 8 8
○ 10 10
○ 12 12
○ 16 16
○ 20 20
○ 24 24

☒ Zoom

Window title: <<<>>>

☒ Go away box
☒ Resizable
☐ Modal
☐ Handle first click
☒ Free on closing
☒ Dispose on free
☒ Closes document
☒ Open with document
☒ Adapt to screen
☒ Stagger windows
☒ Force on screen
☒ Center horizontally
☐ Center vertically

TView

Location
v: 0
h: 0

Size
v: 260
h: 430

☐ Shown
☒ Enabled

ID: WIND
Superview: none

Fig 1.17 A TWindow view description dialog

In all other ways, this view description dialog is the same as other view description dialogs.

• WARNING • Currently, changes made in view description dialogs ***are not undoable***. While the view description dialog is open, the Undo command in the Edit menu will not work—don't try it. When you close the view description dialog, the Undo command does not undo changes made in the dialog, and may be dangerous. Don't use it until you've done some new undoable action!

A few more view examples

So far, you've really only seen a few of the many different types of views. You should probably use ViewEdit to experiment with some of the other views in DemoDialogs and other sample applications. Be sure to make a copy of any important file that you open with ViewEdit!

In your experimenting, you may run across a few issues that haven't been mentioned yet. For example, if a view in a view-editing window is any of the resource-driven views (pop-ups, icons, patterns, pictures), and a resource of the appropriate type and number exists in the resource file, ViewEdit will put a "working" version of that view in the view-editing window. If a corresponding resource does not exist, ViewEdit puts its own version of that type of view in the view-editing window. This feature allows your view-editing window to be faithful to the actual view in your application.

Another hint is to remember that ViewEdit displays what is in actual 'view' resources—sometimes these may look different than actual views you see in an application. For example, certain views are missing, or incorrectly sized. This discrepancy occurs when an application modifies its view resources before displaying them.

A good example of this is TSScrollbars. Remember that TScroller views programmatically create their corresponding TSScrollbar so you won't see TSScrollbar views in view-editing windows. You will, however, see a *space* for the TSScrollbar view, as in the case of 'view' resource ID 900 from DemoDialogs:

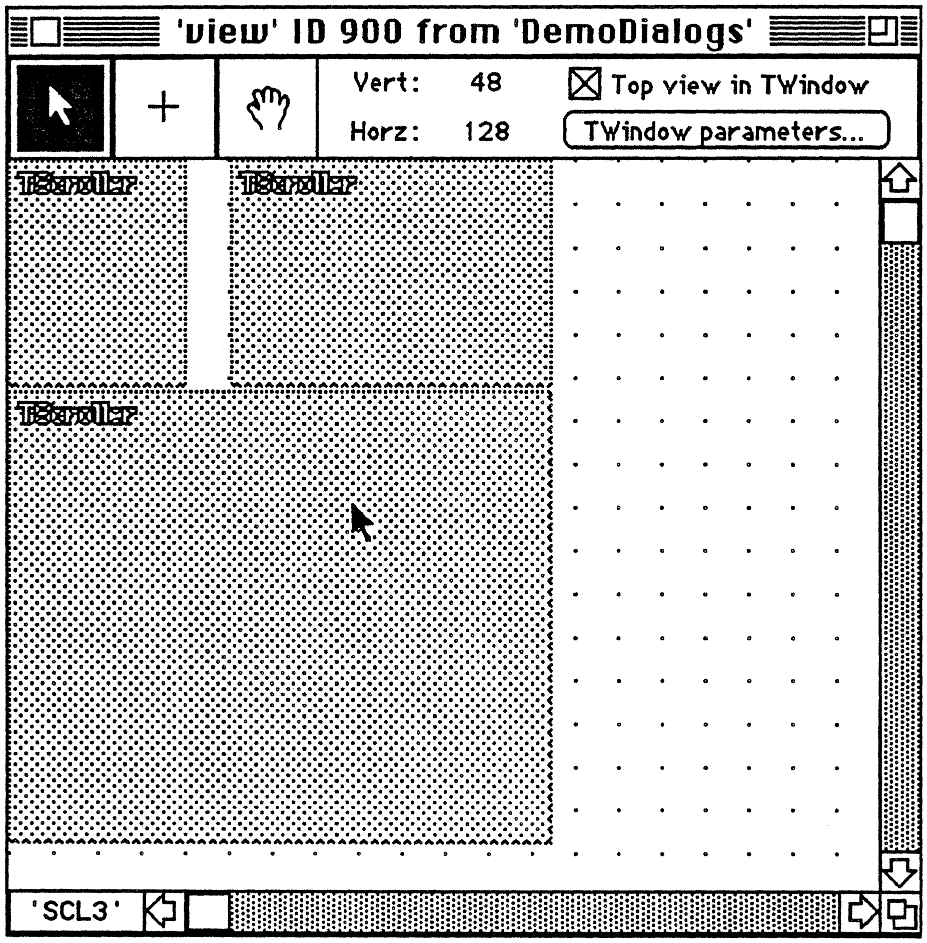


Fig 1.18 TScroller views

This view-editing window represents the views in an Inspector window, which looks like this:

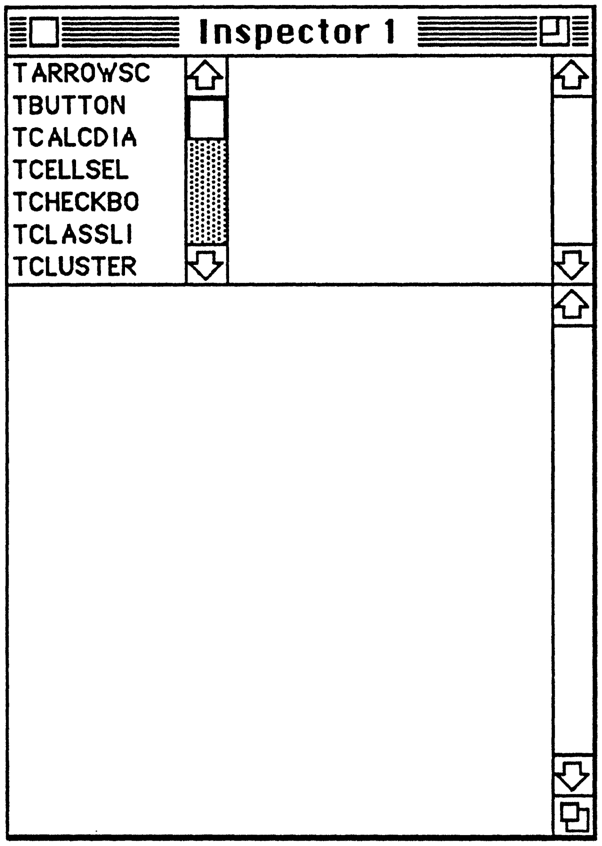


Fig 1.19 An Inspector window

As you can see, space has been left for the TSScrollbar ŏbjects, but they are not themselves represented in the view-editing window.

• NOTE • Since TScrollers will create up to two TSScrollbar objects for you, the TSScrollbar class is not represented in the Views menu. This means you cannot create TSScrollbar views from within ViewEdit. Use DeRez and Rez to add extra TSScrollbar views to your view hierarchy.

The file commands

When you have finished editing with a view-editing window, you can close that window. If you have made any changes, the 'view' icon for that window will have its identifier highlighted. This is a marker to help you remember which resources have been edited. Of course, none of your changes are saved until you choose the Save command from the File menu.

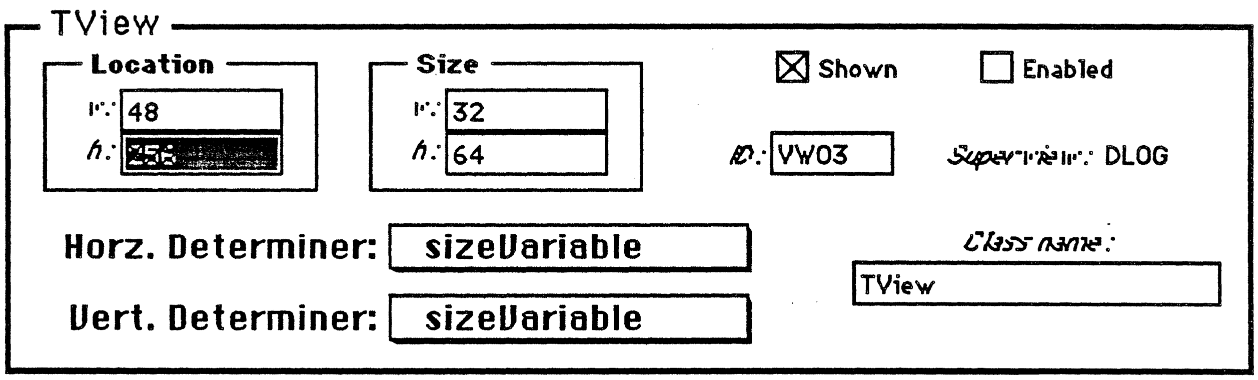
You can also mark items for deletion by selecting them, and then choosing the Delete Resource command from the Edit menu. Again, the resource is not actually deleted until you save the file.

Remember that ViewEdit saves its view hierarchies as resources in a resource file. You can access these through DeRez and Rez, but it should be possible for you to only use ViewEdit, given that you have the patience to use it carefully through this early release.

Good luck and happy view editing!

View Description Boxes

This section contains an example of each of the different types of view description boxes, in the order that they appear in the Views menu. Refer to the "Creating View Templates" recipe in Chapter 7, "The CookBook", in the *MacApp 2.x Manual (Interim Version)*, and the MacApp 2.0 Display Architecture Release Notes for a description of the different fields and possible values.



TView

Location
v: 48
h: 256

Size
v: 32
h: 64

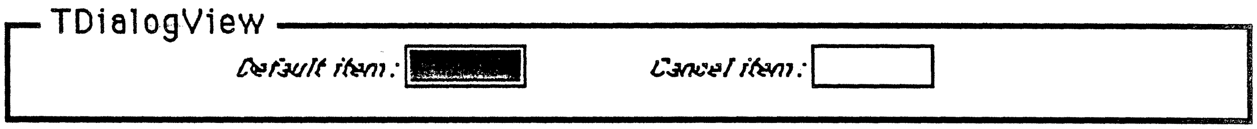
☒ Shown ☐ Enabled

ID: VW03 Superview: DLOG

Horz. Determiner: sizeVariable

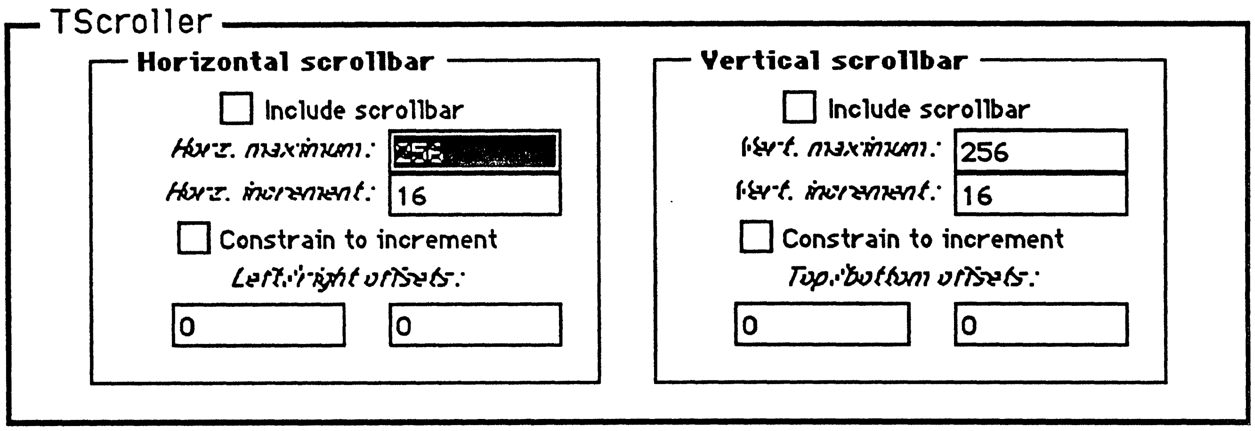
Vert. Determiner: sizeVariable

Class name: TView



TDialogView

Default item: Cancel item:



TScroller

Horizontal scrollbar

☐ Include scrollbar

Horz. maximum: 256

Horz. increment: 16

☐ Constrain to increment

Left/right offsets: 0 0

Vertical scrollbar

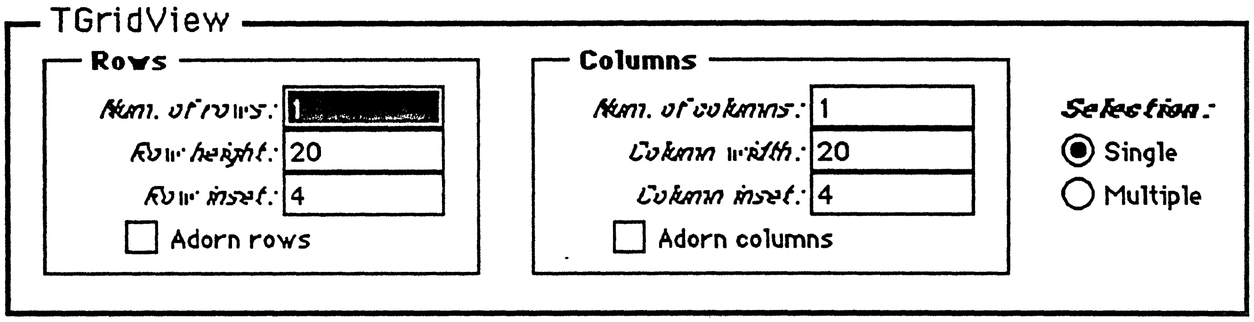
☐ Include scrollbar

Vert. maximum: 256

Vert. increment: 16

☐ Constrain to increment

Top/bottom offsets: 0 0



TGridView

Rows

Num. of rows: 1

Row height: 20

Row inset: 4

☐ Adorn rows

Columns

Num. of columns: 1

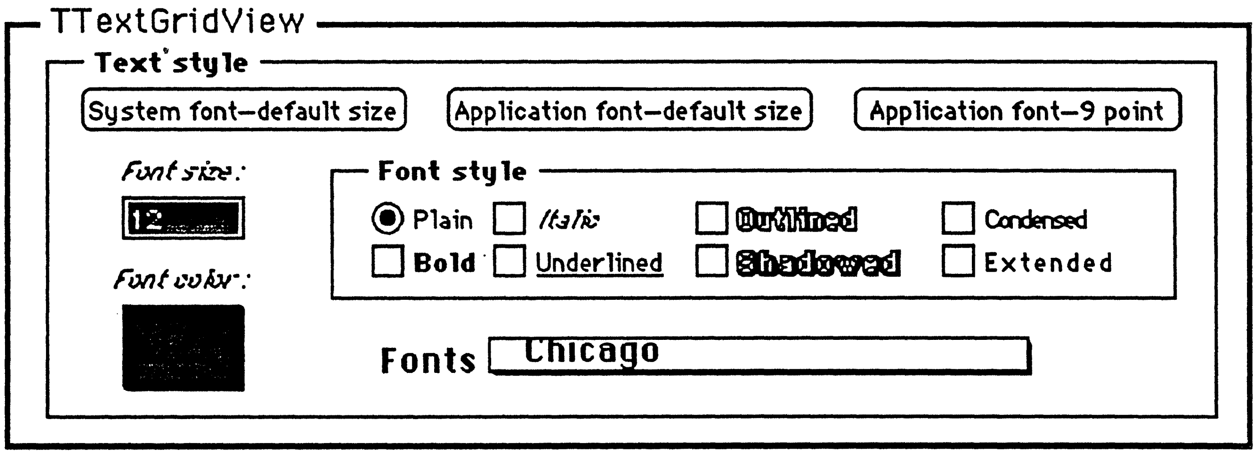
Column width: 20

Column inset: 4

☐ Adorn columns

Selection:

[✓ Single] Multiple



TTextGridView

Text style

System font–default size | Application font–default size | Application font–9 point

Font size: 12

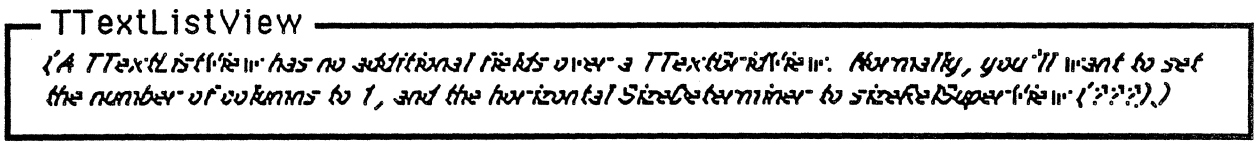
Font color: ■

Font style

[✓ Plain] ☐ *Italic* ☐ Outlined ☐ Condensed

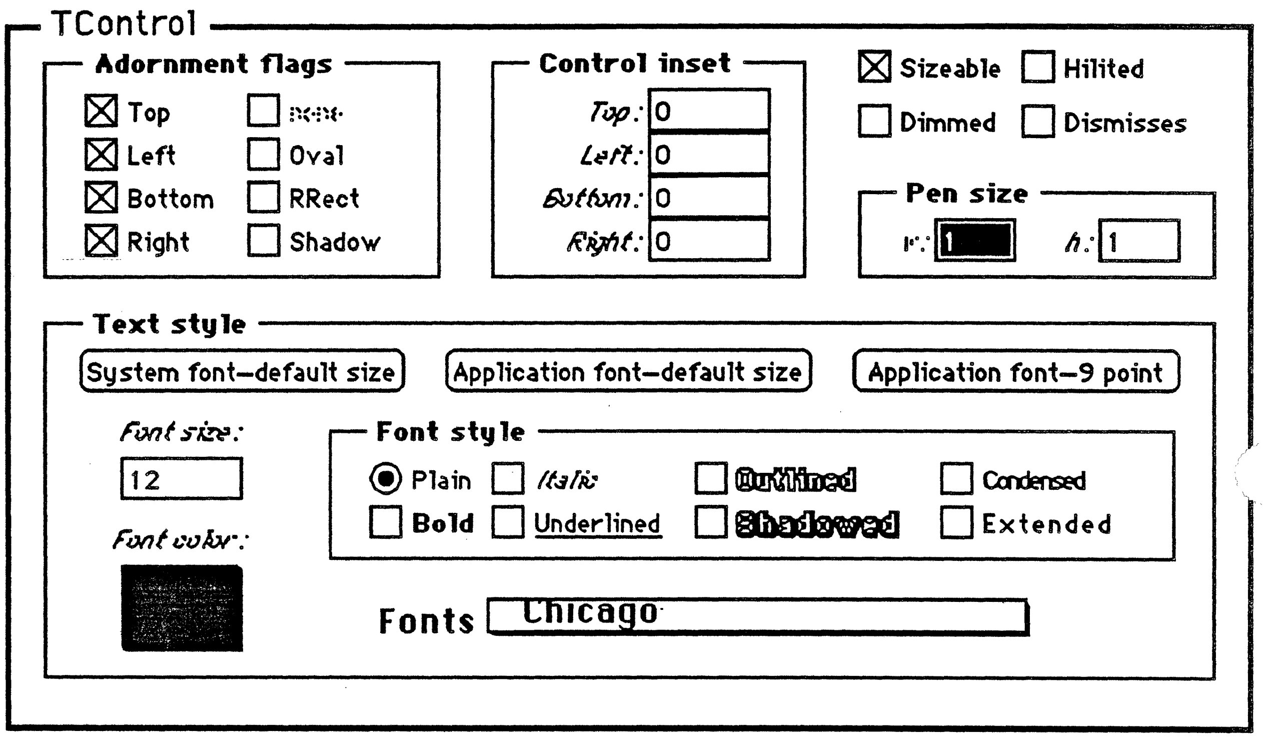
☐ **Bold** ☐ Underlined ☐ Shadowed ☐ Extended

Fonts Chicago



TTextListView

(A TTextListView has no additional fields over a TTextGridView. Normally, you'll want to set the number of columns to 1, and the horizontal SizeDeterminer to sizeRelSuperView (???).)



TControl

Adornment flags

- [✓ Top] [ ] [illegible]
- [✓ Left] [ ] Oval
- [✓ Bottom] [ ] RRect
- [✓ Right] [ ] Shadow

Control inset

Top:	0
Left:	0
Bottom:	0
Right:	0

[✓ Sizeable] [ ] Hilited
[ ] Dimmed [ ] Dismisses

Pen size

v: 1 *h:* 1

Text style

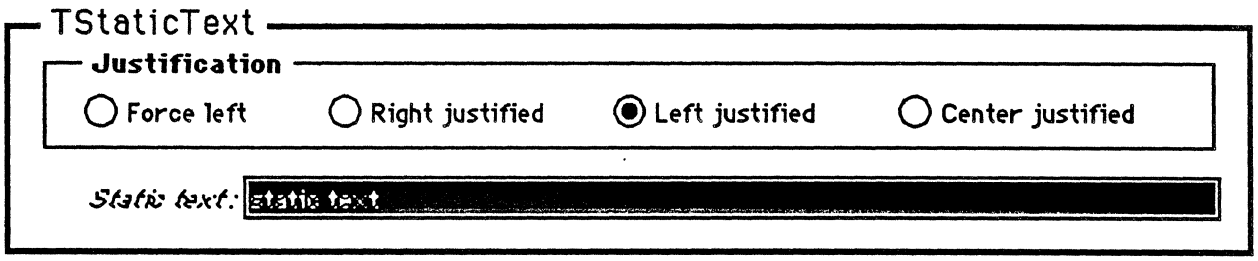
Font size: 12

Font color: ■

Font style

[● Plain] [ ] *Italic* [ ] Outlined [ ] Condensed
[ ] **Bold** [ ] Underlined [ ] Shadowed [ ] Extended

Fonts Chicago

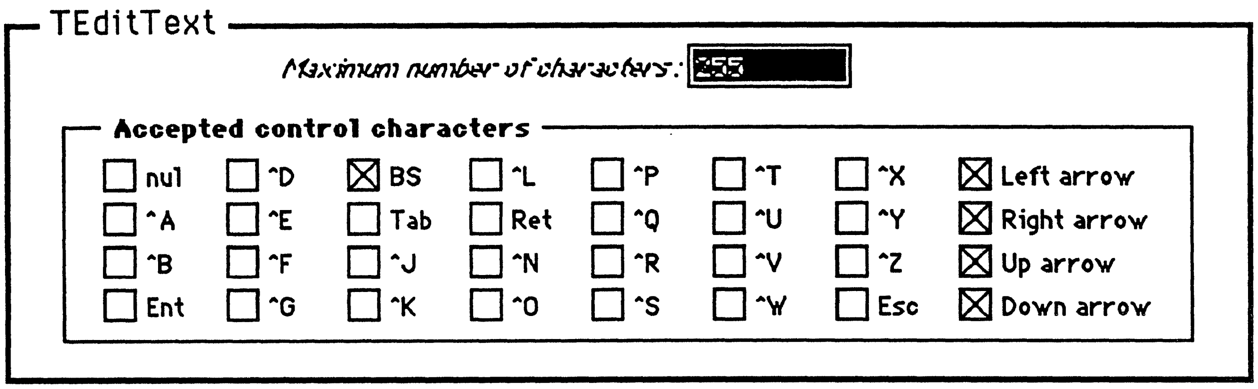


TStaticText

Justification

[ ] Force left [ ] Right justified [● Left justified] [ ] Center justified

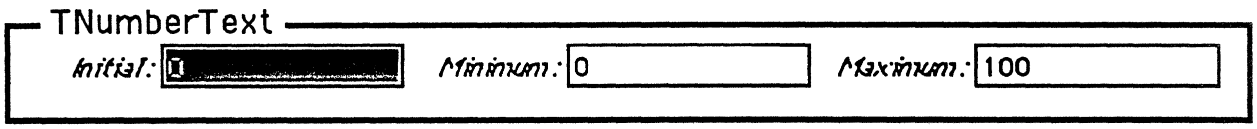
Static text: static text


TEditText

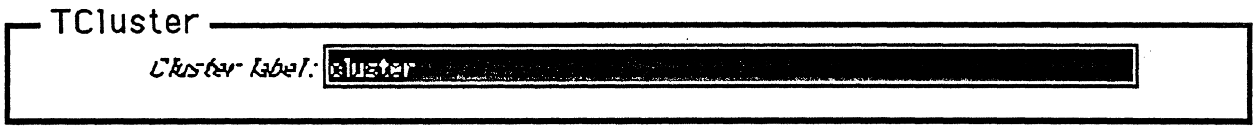
Maximum number of characters: 255

Accepted control characters

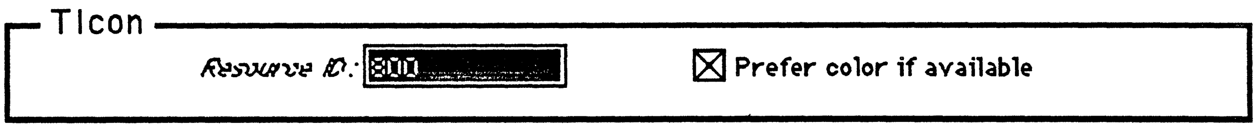
☐ nul	☐ ^D	☒ BS	☐ ^L	☐ ^P	☐ ^T	☐ ^X	☒ Left arrow
☐ ^A	☐ ^E	☐ Tab	☐ Ret	☐ ^Q	☐ ^U	☐ ^Y	☒ Right arrow
☐ ^B	☐ ^F	☐ ^J	☐ ^N	☐ ^R	☐ ^V	☐ ^Z	☒ Up arrow
☐ Ent	☐ ^G	☐ ^K	☐ ^O	☐ ^S	☐ ^W	☐ Esc	☒ Down arrow


TNumberText

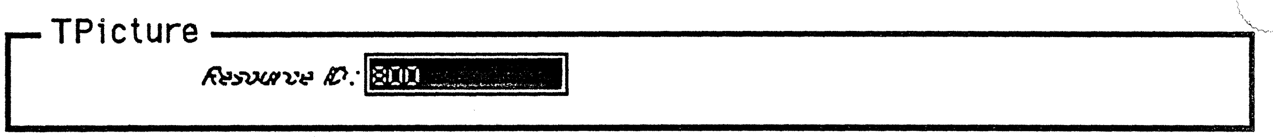
Initial: 0 Minimum: 0 Maximum: 100

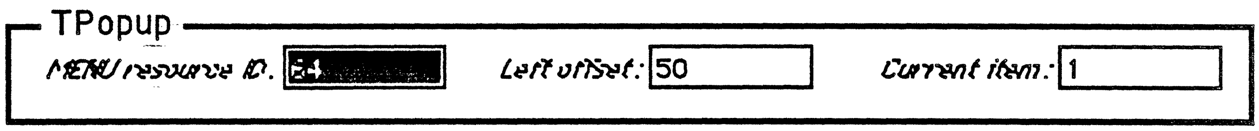

TCluster

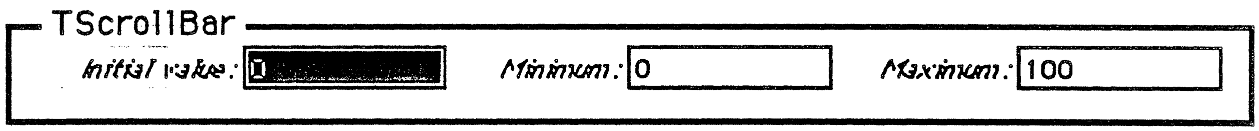
Cluster label: cluster

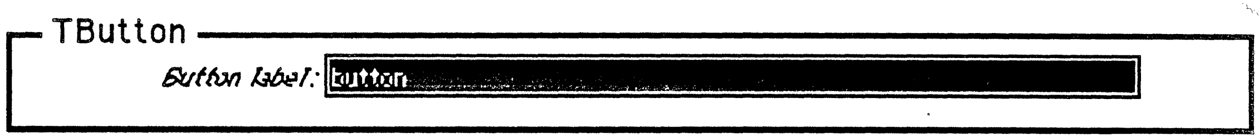

TIcon

Resource ID: 800 ☒ Prefer color if available


TPicture
Resource ID: 800


TPopup
MENU resource ID: 24 Left offset: 50 Current item: 1


TScrollBar
Initial value: 0 Minimum: 0 Maximum: 100


TButton
Button label: button

MacApp 2.0 Memory Management

Any of the following situations can cause an application to stop with a System Error alert, without any chance for that application to gain control:

- Not enough memory to load a code segment.
- Not enough memory to load a PACK resource.
- Not enough memory to save the bits under a menu which is pulled down.
- Not enough memory to load a defproc (WDEF, MDEF, CDEF, LDEF).
- Not enough memory for Standard File to create its file list.
- Add your personal favorite here.

Not all commercial Macintosh applications deal correctly with these issues, in large part because doing so involves some rather complicated code. MacApp contains a memory management mechanism which is designed to help keep a MacApp application from getting into these critical memory situations. It does require some work on your part, but makes it much, much easier to produce a robust application than if you did your own memory management from scratch.

MacApp's scheme works by dividing available heap space into permanent memory and temporary memory (also known as code reserve). Permanent memory is the space occupied by data which your application allocates: objects and any subsidiary data structures you may create. Temporary memory is reserved for your code segments plus any resources and/or memory needed by the Macintosh Toolbox for a short period of time.

MacApp makes sure that your application can't crash for one of the above reasons by always reserving enough space for temporary memory requests to be satisfied. You tell MacApp how much memory needs to be reserved, and it does the rest. It only needs to know whether a given memory request is permanent or temporary. Objects created via New (as well as TObject.ShallowClone and all other MacApp methods which allocate memory) are automatically taken from permanent memory. You can ask for a new handle from permanent memory by calling NewPermHandle instead of NewHandle; for other kinds of requests you set the "permanent" flag by calling PermAllocation(TRUE), make the request, then set the flag back (PermAllocation returns the previous value of the flag as its result). Any other requests (such as those made by the ROM) default to temporary memory.

Note that it is dangerous to have the permanent flag TRUE for any length of time, as any toolbox call which allocates memory or any procedure call which could load a segment will then operate just as it would with no memory management mechanism. For example, calling a procedure in another segment which isn't loaded when the flag is TRUE and there is no permanent memory available will cause a segment loader bomb, even if plenty of temporary memory is available. When debugging is on, MacApp checks for the flag being TRUE in the main event loop or when a segment is loaded and drops into the debugger if it is.

If you need to call a routine which allocates both permanent and temporary memory (such as TextEdit, which allocates permanent data structures and which also loads temporary resources such as fonts), you can do so with the permanent flag set FALSE, and then call CheckReserve afterwards. CheckReserve is a BOOLEAN function which returns TRUE if there is still enough temporary memory. If CheckReserve returns FALSE, you should undo the memory allocation. For convenience, there is also a procedure named FailNoReserve, which calls Failure(memFullErr, 0) if CheckReserve returns FALSE.

Figuring out how much temporary memory to reserve takes some thought. You could specify the sum of the sizes of all your code segments and all the other kinds of memory the Toolbox uses, but that would be wasteful because not all of these items are in memory at the same time. In general, you need to know the sum of the sizes of these items at the point in your application where the largest number of them are in use simultaneously. Some of the Toolbox items mentioned above usually need not be considered in this calculation. For example, the saved bits underneath a menu are only allocated when MacApp calls MenuSelect, which is only done in the main event loop, at which time all the non-resident code segments are unloaded. This is also the case for MDEFs. These will seldom be larger than the code segments which you can load.

There are some situations you do have to watch out for. For example, putting up a standard Open or Save dialog can use up quite a bit of memory: PACK 3 (Standard File), PACK 2 (Disk Initialization), plus the list of files in the current folder which Standard File creates. Also, although the Font Manager will not fail if there is insufficient memory to load a desired font, it will substitute a less suitable font, and your program's screen appearance will degrade. Thus, you should include enough memory to load the largest font you will use in your temporary memory figure. Remember, it will only be used when your Draw method is called, or when you perform text measurement.

Printing is another situation where you can run low on memory. Strictly speaking, it's OK to run out of memory while printing: an alert will come up saying the document could not be printed because there was not enough memory. However, users can find it frustrating if they create a document which is too large to print. If you want to make sure that any document a user can create can be printed, you should factor in the memory taken up by the Print Manager while printing — along with any of your code segments which may be present — when calculating the amount of memory to reserve for temporary allocations.

It is usually easier to split your temporary memory size up into several pieces which you calculate independently. For your code, you may want to use the MacApp 'seg!' resources, which let you list the code segments you want considered (see below). That way, you can just determine which segments are loaded at the time of maximum temporary memory use, and let MacApp figure out their size at execution time. For Toolbox use, you can use some fixed constants. For variable resources like fonts, you can actually alter the temporary reserve size while your application is running.

Depending on how careful you want to be about your application's memory use, you can just pick a comfortably large number (which wastes memory), or you can watch your program in action using the MacApp debugger and MacsBug and figure out the smallest safe number (which is a fair amount of work but gives the best use of available memory).

High Water Mark

The MacApp debugger helps you figure out which set of resources takes the most room by keeping a high water mark for loaded resources. Under the heap & stack command (H), the I subcommand now gives the maximum amount of memory used by loaded code segments, PACKs, defprocs, and so forth, and the R subcommand resets this number to zero. Under the toggle flags command (X), the R flag reports whenever a new high water mark is reached, and if the B (memory management break) flag is set, MacApp will enter the debugger. In calculating this number, MacApp only considers resources on its resource lists (see below).

Note that this won't take into account memory allocated by, say, Standard File or the Print Manager. You should always check the amount of memory used in these situations and any other situation where the Toolbox can allocate large amounts of memory. The best way to do that is to break before and during such a situation, and use the M subcommand of the H command to see how much permanent memory is available (the number labelled "(permanent) FreeMem"). The difference (call it "extra") will be the sum of the sizes of objects you've allocated and those that the toolbox has allocated but which MacApp doesn't track specifically (see "MacApp Resource Lists," below). Also note the set of segments loaded. The sum of "extra" and the "locked resources" number displayed in the debugger, minus any permanent memory you allocated, is the actual amount of temporary memory in use. You can use "extra" to reserve more temporary memory with a mem! resource (see below).

Remember, though, that you are only observing the memory in use at one point in time; memory usage can be greater for a brief period of time, and you won't necessarily catch it in the debugger. Sometimes a trial and error approach is necessary to determine the exact amount of memory being used.

Currently, the most space intensive print driver is the LaserWriter driver. For version 3.1 of the LaserWriter driver, we have empirically determined that "extra" is about 40K. In release 4.0 of the LaserWriter driver, this amount varies, and can be as high as 56K. Moreover, this may well increase in future releases. Fortunately, recent releases of the LaserWriter driver recover gracefully from out of memory conditions, giving the appropriate error message. If you do not need to insure that it is always possible to print, you can eliminate this from your permanent memory reserve.

There is a sporadic bug in LaserWriter driver 3.1 which can cause heap space to be permanently lost. This only occurs when a bitmap font is downloaded to the LaserWriter. Bitmap fonts are only downloaded when font substitution is off in Page Setup (font substitution is always off if you set FractEnable to TRUE or turn off the driver's line layout algorithm) and the user selects Geneva, New York, or Monaco, or if the user selects any other font which is not available in PostScript form (such as Athens or Mobile). The driver finds the largest available size of that font, makes it unpurgeable, then downloads it. Occasionally the driver will not make the font purgeable again, and it remains in memory until the application quits. Since the font is the largest size the driver could find, it takes a significant amount of space (8K for Geneva 24). This bug was fixed in release 3.3 of the LaserWriter driver.

The Low Space Reserve

MacApp keeps a special handle around which it will dispose of in order to satisfy a permanent memory request. This handle is called the low space reserve (it's also sometimes called the permanent memory reserve). You can test if your application is running low on memory by calling the BOOLEAN function MemSpaceIsLow. MacApp makes this test periodically and will call the method TApplication.SpaceIsLow, which you can override to take any action you want. The default version will periodically put up an alert advising the user that memory is low.

Another important issue which the reserve handle helps with is making sure that users don't lose data. Because of the way the Macintosh Memory Manager operates, it makes no guarantee that a particular set of objects which was once allocated in a heap of a given size can again be allocated in a heap of the same size (although it will come pretty close). As a result, you should not allow user documents to grow until they fill the entire heap, since it is possible that you would not be able to read them back in again. Also, if space becomes so scarce that there is no room to create a new command object, users won't be able to decrease their document size, even with a command like Clear.

You can use the low space reserve to handle these problems. If MemSpaceIsLow returns TRUE, don't enable any commands that increase document size, such as Paste, drawing, typing, etc. Always enable commands which allow the user to decrease document size (such as Clear, backspace, etc.). The DrawShapes sample program illustrates this technique. Commands such as Cut and Copy require more thought. On the one hand, they can increase memory use, getting you into the situation where command objects can't be created. On the other hand, if a user wants to decrease a document's size, having Cut and Copy available prevents having to simply throw data away. What you do here depends on your application. For example, UTEView allows Cut when space is low, but not Copy.

If you have a command which allocates additional memory, you should check for space being low at the end of your DoIt method, since otherwise it's possible for the reserve to be in place at the start of the command and completely gone by the end. You should treat running out of low space reserve the same as running out of memory. You can call FailSpaceIsLow, which calls Failure with an error of memFullErr if MemSpaceIsLow returns TRUE. Your command's failure handler should back out any changes it made. Only commands which decrease or don't change memory use should be allowed to eat into the low space reserve.

MacApp itself calls FailSpaceIsLow in several places, including the end of TApplication.OpenNew and TApplication.OpenOld to make sure that a new document doesn't decrease available free space too much. For TApplication.OpenOld, however, MacApp temporarily halves the low space reserve so that existing documents have a little "breathing room" to deal with the nondeterministic behavior of the Memory Manager.

The seg! and mem! Resource Types

MacApp initially sets the size of the temporary memory reserve by looking at all resources of type seg! and type mem!. The sizes of all code segments whose names are listed in any seg! resource are added up as part of the temporary memory reserve. Note that the segment names are the those generated *after* segment mapping. Each mem! resource has three long integer quantities: an amount to add to the temporary memory (code) reserve, an amount to add to the low space reserve, and an amount to add to the size of the stack. MacApp calculates the size of the temporary memory reserve by adding up the sizes of all segments in all seg! resources and the first number from all mem! resources. It calculates the low space reserve by adding up the second number from all mem! resources, and it calculates the size of the stack by adding up the third number from all mem! resources.

This approach allows a great deal of flexibility. For example, the Debug.r file adds the debugging segments to the list of segments, so that if debugging is turned on there will be room to load them. MacApp defines an initial set of segments, reserves 8K of stack space, reserves 4K of low space reserve, and reserves 4K extra of temporary memory. You can easily add to (or even subtract from, except for the stack) MacApp's values by supplying your own mem! and seg! resources. Remember, however, that these resources only govern the initial value of these numbers. If you wish to change any of them while your program is running (except the stack size, which can't be changed), you will have to call the routine SetMemReserve (see the MacApp source code for details).

Remember to factor in temporary memory taken up by things other than resources (such as the memory taken by the Print Manager while printing). For example, the MacApp debugger may tell you that the largest set of resources loaded at one time occurs when opening a document and totals, say, 130K. When printing, your resources may only total 110K. However, the LaserWriter driver uses another 40K of temporary memory, so your maximum temporary memory use is 150K, while printing. In that case you would list the segments you use during printing in your seg! resource, and use a mem! resource to reserve an additional 40K for the Print Manager.

Commit Methods Must Not Fail

Note that if your command object overrides Commit, it is vital that it not cause a Failure. MacApp must call the Commit method of the most recent command (if it has not been undone) in order to save the document or quit the application; if Commit fails, your user will be really stuck, unable even to exit the application. The MacApp command architecture assumes that by the time Commit is called, the command was successful. There are three ways of dealing with this problem.

The first and best way is to set up your data structures in such a way that your Commit method doesn't need to allocate any memory (or increase net memory use). Of course, this isn't always possible. The second way is to preallocate the memory which your command needs to Commit in your DoIt method. Then, when Commit is called, free this memory and proceed. The third way is to allocate the memory you need to Commit from the temporary memory pool instead of the permanent memory pool. You should only do this if your Commit method's memory use has an upper bound, and if the memory will be disposed of by the end of the Commit method. If you do this, you must make sure that your temporary memory reserve is big enough to accommodate this memory use (you can use a mem! resource to do this).

For an example of how subtle these considerations can be, look at the Paste command in the DrawShapes sample application, whose Commit method can actually fail under certain circumstances. The Commit method moves the shapes which have been pasted from the list of "virtual" shapes associated with the command onto the actual list of shapes for the document. It does this by repeatedly deleting a shape from the virtual list, then inserting it into the actual list. Although this seems safe because it does not cause net memory usage to increase, it can still fail, because it may not be possible to grow the actual list's handle by four bytes even though another handle has just shrunk by four bytes and the heap may be completely unfragmented. This is just a consequence of the way the Macintosh memory manager works.

Two possible solutions to this problem: grow the actual list in the DoIt method so that the memory is already allocated, or perform the transfer of shapes with the permanent allocation flag off (currently this is not possible without overriding TList). Note that this latter option is safe because net memory utilization is not increasing; by allowing the Commit method to briefly eat into temporary space, we are just giving the memory manager a little more "breathing room" in which to rearrange the heap.

Although your user won't get stuck if your UndoIt or RedoIt method fails, he or she will probably get upset. The feeling of safety and confidence which users get from having undo available will vanish the first time they try to use it and it fails. The message they get is "Undo doesn't work." Be nice to your users, and make sure that your UndoIt and RedoIt methods can't fail either. A less desirable alternative is to detect in your IMyCommand or DoIt method that Undo won't work and put up an alert that the command will not be undoable. Do this before your command makes any changes, and give the user a chance to cancel the command. If the user says OK, remember to set fCanUndo to FALSE in your command so Undo will be disabled. If the user says Cancel, you can cause your command to fail without putting up an error message by calling Failure(0, 0). MacApp uses this technique itself to handle Cancel choices in dialogs.

One more hint for commands: try not to allocate any memory in your IYourCommand method. There is nothing actually wrong with this, but since the previous command has not yet been committed and freed, and the Undo Clipboard might still be around, it's more likely to fail. If you allocate your memory in your DoIt method, more space will be available.

MacApp Resource Lists

MacApp keeps a list (actually several) of the resources which it considers "temporary:" that is, they are considered as not taking up permanent memory. It constructs this list at application startup time, and uses it whenever it calculates how much temporary memory is in use and how much more to reserve. These resources are the ones that the MacApp debugger describes in the H command. MacApp will also purge these resources when trying to satisfy a temporary memory request unless they are locked.

By default, this list contains all resources of type CODE, PACK, LDEF, CDEF, WDEF, and MDEF (except those which come from ROM, or reside in the System heap). Normally, you don't need to worry about this list at all. It's OK for temporary resources to not be on the list, although they will be purged frequently when space is low, and they won't be figured into the resource statistics in the debugger. If you have such resources (fonts, for example), and if you want to reserve some memory for them, you may want to consider adding them to the list. Look at the UMemory unit to see how the lists are managed.

If you do put fonts on the list, you should move them high (with MoveHHi) and lock them when you do so to prevent MacApp from purging them; the font manager expects that fonts marked nonpurgeable won't be purged.

Segmenting Your Application

In order for the memory management mechanism to work properly, your application must be segmented properly. The more code which is unnecessarily dragged into memory, the larger your temporary memory reserve must be, and the less space is available for your user's data in any given memory configuration. MacApp defines the following code segments for your application:

ARes	Resident application code. Anything that gets called frequently in the main event loop (such as your DoSetupMenus methods), drawing, or typing.
ADebug	Your debugging code, i.e. any procedures or methods only present when debugging is on.
AFields	All of your Fields methods should go here.
AInit	Code which you only use at application start-up time (IYourApplication).
ATerminate	Code which you only use at application shut-down time.
ASelCommand	Code used to select the next command (DoMenuCommand, DoMouseCommand, DoKeyCommand, and IYourCommand methods).
ADoCommand	Code used for executing commands (all other TYourCommand methods except IYourCommand).
AClipboard	Clipboard code that is not part of a command (MakeViewForAlienClipboard, GivePasteData, WriteToDeskScrap, but NOT TYourPasteCommand).
AOpen	Code used when opening (DoMakeViews, DoMakeWindows, IDocument, DoMakeDocument, IDocument, IView).
AClose	Code used when closing (TYourDocument.Free, FreeData).
AReadFile	Code used when reading or reverting (DoRead, ShowReverted, DoInitialState).
AWriteFile	Code used when writing (DoWrite, DoNeedDiskSpace, SavedOn).
AFile	Code used when reading or writing (typically you won't have anything in this segment).
ANonRes	Catch-all non-resident segment, for infrequently used methods (e.g. ResizeWindow).

The default segment mapping established by MacApp combines these segments (except for ARes) with the corresponding ones for MacApp. You can override some of MacApp's mappings by specifying your own mappings in your make file, and you can override MacApp's mappings entirely by placing a definition for the Make variable SegmentMappings in your make file (Make will issue a warning, but your definition will override MacApp's). Look at MacApp.make1 for guidance.

Even using the suggested segment mappings, you may overflow the 32K bytes limit on the size of a segment. You can override MacApp's segment mappings in your application's .make file by providing your own -sn mappings; the Linker gives your segment mappings priority over the ones MacApp specifies.

You may want to make some application code segments resident (i.e. never unloaded). This is the case with ARes, for example. To do this you can use the 'res!' resource (see below) or call SetResidentSegment (generally after calling InitToolbox, InitPrinting, etc. but before doing New(gYourApplication)) in order to make ARes resident. Many of the sample programs, including DrawShapes and DemoText, use the 'res!' technique.

Generally, there is a trade-off involved in segmentation. A few large segments make your program run faster, but increase its memory requirements. More, smaller segments decrease memory requirements (and are a necessity to run in a small Switcher partition), but make your program slower to start up. This latter problem can be alleviated by using the JumpStart utility included in the Macintosh Development Utilities product, available from APDA.

If you're really desperate to decrease segment sizes, you can change MacApp's segmentation by editing the source code and increasing the number of segments. This is not recommended. Be extremely careful about which segments are resident if you do this.

The res! Resource Type

The 'res!' resource type is used to identify code segments that are to be made resident (i.e. never unloaded). It's format is similar to the 'seg!' resources in that each 'res!' resource consists of a list of segment names. When MacApp initializes the application it makes resident any segment listed in any 'res!' resource found. 'res!' resources are already included for MacApp's resident segments. You can add a 'res!' resource to your application's resource file to list resident segments you've defined. Note that as with 'seg!' resources the segment names in the 'res!' resources are the names *after* segment mapping.