The FormsVBT Reference Manual

Version 3.4

Marc H. Brown and James R. Meehan

April 26, 1996

Contents

1. Introduction

FormsVBT is a system for building graphical user interfaces (GUIs). It consists of a language for describing an application's user interface, a stand-alone application for constructing the user interface, and a runtime library for communicating between an application's code and its user interface.

A user interface in FormsVBT is a hierarchical arrangement of *components*. Components include passive visual elements, basic interactors, modifiers that add interactive behavior to other components, and layout operators that take groups of low-level components and organize them geometrically. In the FormsVBT language, the arrangement is written as a symbolic expression (S-expression). The outermost expression is the *form* or top-level component, and subexpressions are either properties that modify a component or other, subordinate components.

The FormsVBT interface builder, `formsedit`, provides a *text editor* and a *result view* of the user interface, as shown in Fig. 1. The text editor displays the S-expression underlying the user interface, while the result view shows the user interface as it will look at runtime, with proper reaction to mouse and keyboard activity, as well as proper sizing and stretching. Of course, the result view cannot reveal exactly how an application's user interface will look and behave, since there is no application code running, but it's usually pretty close. The result view is updated as the user edits in the text view. Interacting in the result view does not update the text view or change the underlying S-expression.

The runtime library provides the communication between an application and its user interface. There are procedures to convert an S-expression into a window object, procedures to register *event-handlers* that will be invoked in response to user actions, procedures to retrieve and modify the values of the components, procedures to change the appearance (and even the hierarchy) of the components, and so on.

Each component in FormsVBT is implemented by a window class (i.e., a VBT) provided by VBTkit or Trestle. Most of the things that you'd want to do with a component can be done via FormsVBT. However, there may be occasions when you would like direct access to the underlying VBT. FormsVBT provides such access. (You'll probably find it helpful to have a copy of the reference manuals for VBTkit [2] and Trestle [5], as well as the Trestle Tutorial [6].)

The FormsVBT system is implemented in Modula-3[3, 7]. It is based on an earlier system implemented in Modula-2+[1]. That version was unique among user interface development environments for its multi-view editor, and noteworthy for its extensibility and simplicity. In this implementation, the editor is not multi-view; there is no graphical, direct-manipulation editor integrated with the text view. Also, this implementation is not extensible by clients.

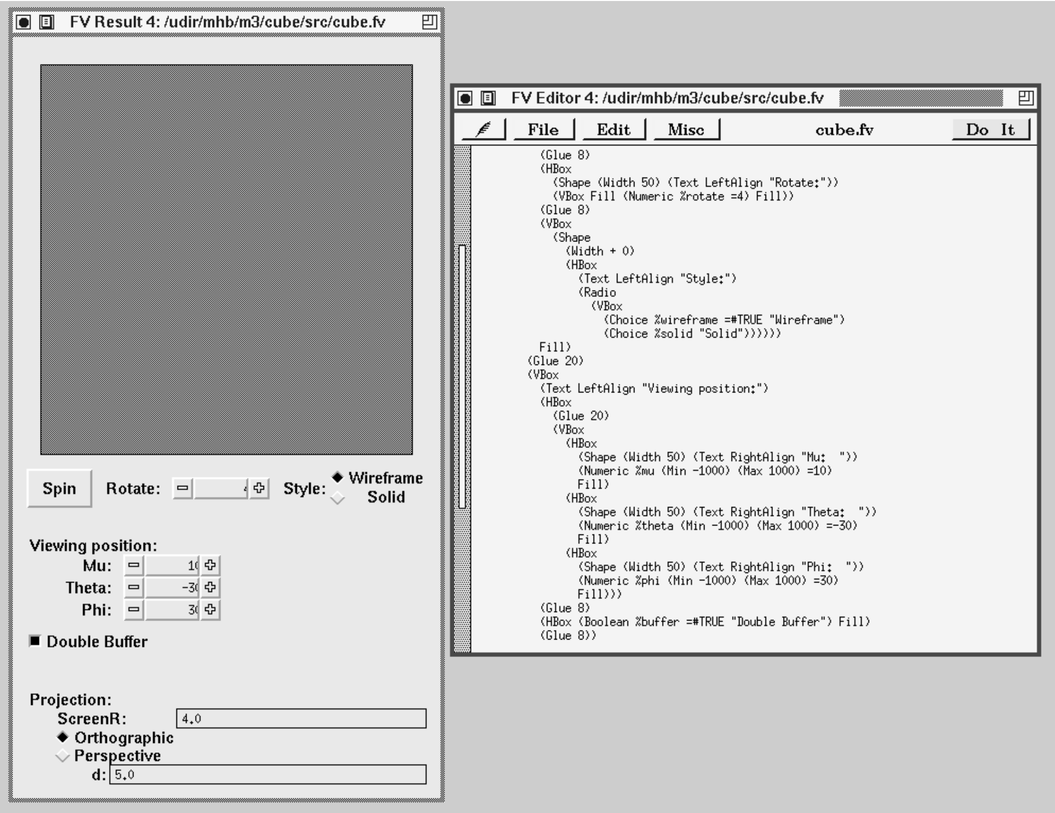


Figure 1.1: *The FormsVBT interface builder,* `formsedit`*, in action. The Text View is on the right and the Result View is on the left.*

Indeed, one of the primary themes shaping the development of the system has been to deliver a "95% solution." By that, we mean that 95% of what clients do should be trivial to do; of the remaining 5%, 95% of that should be pretty easy to do; and the remaining things should be possible and no harder to do than without FormsVBT.

2. Tutorial

To use FormsVBT, you need a copy of SRC Modula-3 (Version 3.3 or later) and an X server for your system. If you have these, you may want to compile and run the example programs as you read this chapter.

2.1 Getting Started

The first example program is in the file `Hello.m3`:

```
MODULE Hello EXPORTS Main;
IMPORT FormsVBT, Trestle;
VAR fv := FormsVBT.NewFromFile("Hello.fv"); BEGIN
  Trestle.Install(fv);
  Trestle.AwaitDelete(fv)
END Hello.
```

The program builds a *form* (sometimes called a "dialog box" or a "user interface") whose description is contained in a file named `Hello.fv`. It installs the form in a top-level window, and then waits until that window is deleted by the user. The window installed by the program is shown in the left half of Fig. 2.1.

The file `Hello.fv` contains the following S-expression:

```
(VBox
  (Text "Hello FormsVBT!")
  (Bar)
  (HBox (Text "Left") (Bar) (Text "Right")))
```

The top-level component is a `VBox`. A `VBox` takes an arbitrary number of "children" (sub-components) and arranges them vertically from top to bottom. This `VBox` has 3 children: `Text`, `Bar`, and `HBox`. A `Text` displays a text string, a `Bar` draws a line orthogonal to the orientation of its parent, and an `HBox` arranges its children horizontally, from left to right. The `HBox` has 3 children, two `Text`s and one `Bar`.

The standard way that you compile and link your programs is to use `m3build`. The `m3makefile` for the "Hello FormsVBT!" application is as follows:

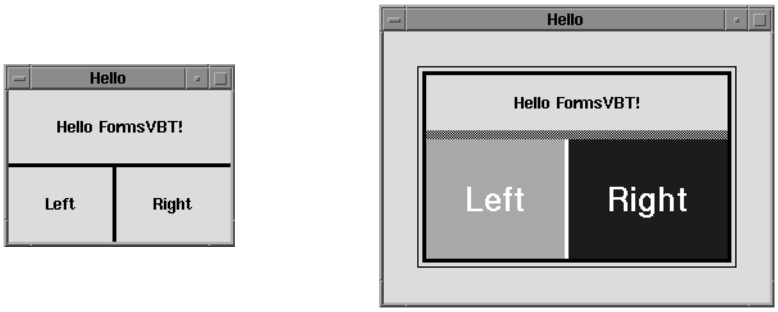


Figure 2.1: *The "Hello FormsVBT!" example program. The initial version is on the left; the second version on the right.*

```
import         (formsvbt)
implementation (Hello)
program        (Hello)
```

Then you can compile and link the `Hello` program by typing the shell-command `m3build -S` in the directory containing the source code.

Actually, most Modula-3 programmers follow the convention of storing all of the source files for an application in a directory called `src`. The `m3build` command, when run from `src`'s parent directory, stores all of its derived files (including the executable) in a subdirectory whose name depends on the platform on which you are running. For example, on DECstations, the derived directory is `DS`; on an Alpha running OSF, the directory is `AOSF`. When you follow this directory structure, you should invoke `m3build` without any arguments.

Here's a slightly fancier version of the interface (shown in the right half of Fig. 2.1):

```
(Rim (Pen 20)
  (Border (Pen 1)
    (Rim (Pen 2)
      (Border (Pen 2)
        (VTile
          (Text "Hello FormsVBT!")
          (HBox
            (LabelFont (PointSize 240))
            (Color "White")
            (Text (BgColor "Pink") "Left")
            (Bar)
            (Text (BgColor "VividBlue") "Right")))))))
```

The top-level component is a `Rim` whose `Pen` property has a value of 20. A `Rim` must contain exactly one child (a `Border` in this case), and it surrounds its child with some background space. Here, the `Rim` provides 20 points of background space between each edge of the window manager's window frame and the rest of the interface. A `Border` is just like a `Rim`, but draws with the foreground color instead of the background color. We replaced the VBox with a VTile, and deleted its Bar child. A VTile is like a VBox, but it also automatically inserts a dividing bar between its children; by dragging the dividing bar, the user can control the division of space among the children. In this example, the HBox has been given two properties, Color and LabelFont. These control the foreground color and font used by the HBox and all of its descendants. Similarly, the BgColor property changes the background color used.

The fancy version of "Hello FormsVBT!" is in the file HelloFancy.fv. To run the application using this file, either modify the application to use HelloFancy.fv or rename the file HelloFancy.fv to be Hello.fv. Alternatively, you might find it enjoyable to run the FormsVBT interactive UI builder, formsedit. Just type the shell-command

```
formsedit HelloFancy.fv
```

Exercise 1: Write the FormsVBT S-expression for $\mathcal{T}^4$, a Trestle Tiling Monster of Order 4. (See the *Trestle Tutorial*, Fig. 2 on page 5.)

2.2 Resources

A resource is constant data needed by an application program at runtime; often it is "loaded" at startup time. Almost all FormsVBT programs have resources, such as the .fv (pronounced "dot ef vee") files that specify the user interface. Other typical resources specific to an application include bitmaps, cursors, and help-texts.

When an application is built, its resources can be "bundled" with the executable image. The primary benefit of this feature is that applications are self-contained with respect to the resources they need. Thus, you can copy an executable to a remote site and you won't need to copy the resource files and install them in the same place as they were when the application was built. Also, your application will be insulated against changes in library resources.

The easiest way to do this is to name the resources and the bundle in the m3makefile, as in this example:

```
import         (formsvbt)
resource       (Hello.fv)
bundle         (HelloBundle)
implementation (Hello)
program        (Hello)
```

The second line declares that there is a resource named Hello.fv. The third line has the effect of collecting all the named resources (only one in this case) and creating an interface called HelloBundle that provides access to them. The program would then be modified to look like this:

```
MODULE Hello EXPORTS Main;
IMPORT FormsVBT, HelloBundle, Rsrc, Trestle;
VAR
  path := Rsrc.BuildPath(HelloBundle.Get());
  fv   := NEW (FormsVBT.T).initFromRsrc ("Hello.fv", path);
BEGIN
  Trestle.Install(fv);
  Trestle.AwaitDelete(fv)
END Hello.
```

The call to `HelloBundle.Get` returns a bundle that is used to create a resource-path, which is then searched by the `initFromRsrc` method.

But what if you *want* the application to use new resource files? For example, you might have changed some details of the `.fv` file that don't require any changes to the application code. Do you have to rebuild the entire application?

Fortunately, the answer is no. However, you do need to tell FormsVBT that you want it to look for those resources in the file system *before* it looks for them among the resources that were bundled into the application. You do this by changing the resource-path so that it includes one or more directories before the bundle.

The convention is to use environment variables whose names are spelled by combining the program's name with the string `"PATH"`. This variable should be set to a list of directory-names, each separated by a colon. So, if you want to run the `Hello` program using the `Hello.fv` file that's in Smith's home directory instead of the one that's bundled with the application, you would type something like this shell command:

```
setenv HelloPATH /user/smith
```

In the program, you would construct a resource-path that included this directory by adding the name `HelloPATH`, prefixed with a dollar sign:

```
MODULE Hello EXPORTS Main;
IMPORT FormsVBT, HelloBundle, Rsrc, Trestle;
VAR
  path := Rsrc.BuildPath("$HelloPATH", HelloBundle.Get());
  fv   := NEW (FormsVBT.T).initFromRsrc ("Hello.fv", path);
BEGIN
  Trestle.Install(fv);
  Trestle.AwaitDelete(fv)
END Hello.
```

2.3 The FormsVBT Language

Syntactically, there are three types of components in FormsVBT: leaves, filters, and splits. A *leaf* has no children; a *filter* has exactly one child; and a *split* has any number of children.

The FormsVBT *leaf* components include passive objects like texts and pixmaps, as well as interactive objects like scrollbars and type-in fields.

A *filter* modifies its child's looks or behavior in some way. We've seen how a `Border` draws a border around its child. Another common filter is `Boolean`. It adds a check box to the left of its child and makes the box and the child sensitive to mouse clicks. It's important to realize that the child may be any arbitrarily complex arrangement of components, although a `Text` component is the most common.

The purpose of most *splits* is to divide the display area among its component-children (sub-components). In addition to the horizontal and vertical splits that we've seen, FormsVBT provides a temporal split (`TSplit`) to display exactly one child at any given time, and a z-axis split (`ZSplit`) to display children as overlapping subwindows.

Components are written as lists containing the component's type, followed by some number of properties, followed by some number of sub-components. Properties are written as lists containing a keyword and a value. For example, in the S-expression:

```
(HBox
  (LabelFont (PointSize 240))
  (Color "White")
  (Text "Left")
  (Bar)
  (Text "Right")
```

the parent-component's type is `HBox`. This component has two properties; the first property has the keyword `LabelFont` and the value `(PointSize 240)`; the second has the keyword `Color` and the value `"White"`. It has three sub-components: `(Text "Left")`, `(Bar)`, and `(Text "Right")`.

The value of each property is type-checked when the description is parsed. The possible types include strings, integers, and real numbers, as well as more complicated types like color and font specifications.

So far, we have seen two kinds of properties. *Class* properties, like `Pen`, are defined in conjunction with specific components, and are allowed only on components of that class. *Inherited* properties, like `Color` and `LabelFont`, may be specified for any component, though they are not relevant to all component types. The inherited properties have the feature that a value specified for one component becomes the default value for all descendants of that component. Thus an inherited property applies not to one component, but to an entire subtree.

FormsVBT supports a third type of property, *universal* properties. A universal property can be specified on any component, and its value applies only to that component.

Exercise 2: In `HelloFancy.fv`, wrap a `Scale` component around the top-level `Rim`. The `Scale` has two class properties: `HScale` and `VScale`. What happens when the values of both of these properties are set to 1.75? What happens when you nest `Scale` filters?

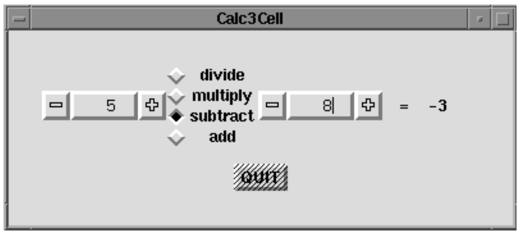


Figure 2.2: *The three-cell calculator application.*

2.4 The Three-Cell Calculator Application

A more interesting application is a three-cell calculator.[1] The user can enter two numbers and an arithmetic operation to perform on the two numbers. The result is computed and displayed whenever the user selects a new arithmetic operation or types a new number. Fig. 2.2 shows the application in action.

The user interface is described by the following FormsVBT expression:

```
(Shape (Width 300 + 100 - 50) (Height + 25)
  (Rim (Pen 20)
    (VBox
      (HBox
        (VBox Fill (Numeric %num1 =5) Fill)
        (Radio %functions =add
          (VBox
            (Choice %div "divide")
            (Choice %mul "multiply")
            (Choice %sub "subtract")
            (Choice %add "add")))
        (VBox Fill (Numeric %num2 =2) Fill)
        (Text "=")
        (Text %result LeftAlign ""))
      (Glue 10)
      (HBox Fill (Guard (Button %exit "QUIT")) Fill))))
```

The tokens that start with percent signs are names assigned to components. For example, the `Text` component where the application stores the result of each computation is named `result`. An application can access only named components at runtime.

This form contains the following components that we have not seen before:

- A `Shape` is used to give a component explicit size constraints, typically as a function of its child's size. Here, the `Shape` declares that its acceptable width is between 250 and 400 points,

[1] Readers may wish to compare the FormsVBT implementation of this example with that of SUIT [8].

and its preferred width is 300 points. The preferred and minimum height of the `Shape` are the same as those of its child; its maximum height is 25 points more than the maximum of its child.

- `Fill` and `Glue` are used as children of an `HBox` or `VBox`. `Fill` displays as background space that will "stretch" as needed in the orientation of its parent. It is used here to keep the `Numeric` component centered vertically between the top of the string `divide` and the bottom of the string `add`. `Glue` displays as background space that doesn't stretch. In this form, it provides 10 points of space. Judicious use of `Fill` and `Glue` will facilitate creating very regular and pleasing-looking user interfaces.

- `Numeric` components are numerical widgets; in our example, the one on the left has an initial value of 5, and the one on the right has an initial value of 2. A user can change the displayed number by clicking on the plus or minus buttons, or by typing in the type-in field located between the buttons. To start typing, the user needs to move the *keyboard focus* to the type-in region by clicking there with the mouse. Also, nothing about the typing is reported to the application until a carriage return is typed. At that point, the application is notified that the `Numeric` has a new value, but not what the user did to enter this new value. (If the application really wants to find this out, it can inquire whether the user clicked on the plus or minus buttons, or entered a new number in the type-in region.)

- The `Radio` unites all of the `Choice` components which are among its descendants into *radio buttons*. A `Choice` adds a diamond to the left of its child and causes the diamond of the selected choice to appear dark and recessed. The application is notified whenever the user changes the selected item. Note that a `Radio` does not impose any particular geometric arrangement on the layout of its radio buttons.

- A `Button` adds the "look-and-feel" of a button to its child. The application is notified when the user clicks on a button. Again, keep in mind that the contents of a button can be any arbitrary arrangement; here, it's a simple text string.

- The `Guard` component requires that you click on it *twice in a row* before the program"quits". The first click removes the "guard" (shown visibly by the diagonal lines) and allows subsequent mouse activity to be reported to its descendant, a `Button` in this case. The second click causes the `Button` to be invoked, and the guard to be reinstated. A `Guard` is usually put around a `Button`, but it may also be put around any component. For example, if you wanted to protect the "divide" radio button, you'd simply wrap a `Guard` component around the first `Choice` expression.

In FormsVBT, as in most GUI toolkits, an application is structured as an initialization routine, which runs in one thread, and a collection of event-handling procedures, which run in other threads. When an application is run, it initializes dialogs and then transfers control to the toolkit. The main thread waits until the toolkit returns control, which it does when all the dialogs have been deleted.

Here is the complete application for the three-cell calculator (see the file `Calc3Cell.m3`):

```
MODULE Calc3Cell EXPORTS Main;
IMPORT Fmt, FormsVBT, Text, Trestle, VBT;
PROCEDURE NewForm (): FormsVBT.T =
  VAR
    fv  := FormsVBT.NewFromFile ("Calc3Cell.fv");
    qcl := NEW (FormsVBT.Closure, apply := Quit);
    ccl := NEW (FormsVBT.Closure, apply := Compute);
  BEGIN
    FormsVBT.Attach (fv, "exit", qcl);
    FormsVBT.Attach (fv, "num1", ccl);
    FormsVBT.Attach (fv, "num2", ccl);
    FormsVBT.Attach (fv, "functions", ccl);
    RETURN fv
  END NewForm;
PROCEDURE Quit (cl  : FormsVBT.Closure;
                fv  : FormsVBT.T;
                name: TEXT;
                time: VBT.TimeStamp) =
  BEGIN
    Trestle.Delete (fv)
  END Quit;
PROCEDURE Compute (cl  : FormsVBT.Closure;
                   fv  : FormsVBT.T;
                   name: TEXT;
                   time: VBT.TimeStamp) =
  VAR
    answer: REAL;
    first       := FLOAT (FormsVBT.GetInteger (fv, "num1"));
    second      := FLOAT (FormsVBT.GetInteger (fv, "num2"));
    fn          := FormsVBT.GetChoice (fv, "functions");
  BEGIN
    IF Text.Equal (fn, "add") THEN
      answer := first + second
    ELSIF Text.Equal (fn, "sub") THEN
      answer := first - second
    ELSIF Text.Equal (fn, "mul") THEN
      answer := first * second
    ELSIF Text.Equal (fn, "div") THEN
      answer := first / second
    END;
    FormsVBT.PutText (fv, "result", Fmt.Real (answer))
  END Compute;
BEGIN
  VAR fv := NewForm(); BEGIN
    Trestle.Install(fv);
    Trestle.AwaitDelete(fv)
  END
END Calc3Cell.
```

The parameters to an event-handler (e.g., `Quit` and `Compute` in the `Calc3Cell` program) identify the dialog (`fv`) in which the event happened and the name of the interactor causing the event.

The event-handler's first parameter, named `cl` in this example, is a `FormsVBT.Closure` that is specified when the event-handler is attached. Its `apply` method *is* the event-handler. The standard way of passing additional information to the event-handler is to create a subtype of `FormsVBT.Closure`, with new fields, and possibly new methods, for handling the new information. The `time` parameter is a timestamp associated with the user event that caused the event-handler to be invoked. The timestamp is needed for certain operations, like acquiring the keyboard focus.

We say that a component "generates an event" when the user does something in a component that causes the event-handler to be invoked. The semantics of what causes an event to be generated is specific to each component.

The Three-Cell Calculator application creates a form and passes it to Trestle, the window manager, which "installs" it, just as the "Hello FormsVBT!" application did. Here, as part of building a form from the S-expression in file `Calc3Cell.fv`, we also attach event-handlers to the components to which the application will respond. The `Quit` event-handler, which is attached to the component named `exit` (the button labeled "QUIT"), deletes the window from Trestle. The `Compute` event-handler, which is attached to both of the `Numeric` components as well as the radio buttons, retrieves the values stored in both `Numeric` components, determines which arithmetic function the user selected, performs the operation, and then displays the result.

Exercise 3: Add your favorite operator to the application and to the user interface. (If you're undecided about which operator is your favorite, try GCD.)

2.5 Improving Readability

The Three-cell Calculator S-expression illustrates a number of common abbreviations that help make the FormsVBT language more readable.

A percent sign is an abbreviation for the `Name` property. That is, the FormsVBT parser reads `%`*xyz* exactly as if it were `(Name` *xyz*`)`.

An equals sign is an abbreviation for the property called `Value`. That is, the FormsVBT parser reads =*xyz* exactly as if it were `(Value` *xyz*`)`. By convention, any component whose value can be changed interactively by a user has a `Value` property.

Components that display some type of object, like a string or a pixmap, specify the object using a property called `Main`. For example, to display a pixmap from a file named `Trumpet`, you'd say `(Pixmap (Main "Trumpet"))`. However, the `Main` property can be abbreviated by omitting the keyword `Main` and the associated parentheses, e.g., `(Pixmap "Trumpet")`.

A `Text` component that has no properties other than `Main` can be further abbreviated simply by giving a string. For example, `(Text (Main "QUIT"))` can be reduced to `(Text "QUIT")` and then to `"QUIT"`. Other examples of this are the children of the four `Choice` components in the last program. If you want to specify any properties on a `Text` component (such as a name, font, color, or alignment), you can abbreviate `Main`, but you still need to write `(Text ...)`.

Boolean properties have a value of either `TRUE` or `FALSE`. The default value of *all* Boolean properties is `FALSE`. Mentioning the name of a Boolean property is an abbreviation for specifying a true value. For example, in the Three-Cell Calculator, the token `LeftAlign` is an abbreviation for `(LeftAlign TRUE)`.

Finally, leaf components without any properties can be written without parentheses, e.g., `Fill`.

The following chart summarizes these abbreviations:

```
(Text "t")        "t"
(Name n)          %n
(Value v)         =v
(Main m)          m
(boolprop TRUE)   boolprop
(proplessleaf)    proplessleaf
```

Exercise 4: The following interface contains a textual label, a type-in field, and a button:

The interface is 250x75 points, and it uses `Button`, `Frame`, `Pixmap`, `Rim`, `Shape`, `Text`, and `TypeIn` components, in addition to some `HBoxes`, `VBoxes`, `Glues`, and `Fills`. Appendix A describes the class-specific properties for each component. Write a concise FormsVBT expression for this form.

FormsVBT provides two additional ways to make S-expressions more readable. First, an S-expression can be split across multiple files (resources). To insert a file named `HelpDialog.fv`, just include the expression

```
(Insert "HelpDialog.fv")
```

wherever you want the file to be inserted. The `Insert` expression can appear anywhere in an S-expression; logically, it is replaced by the contents of the named file before the S-expression is parsed. The second way to make the form more readable is by using macros. Syntactically, a macro is an inherited property with the name `Macro`. For details on macros, see Section 3.5.

2.6 Separating the UI from the Application

One of the ways that user interface toolkits like FormsVBT simplify the construction of interactive, graphical applications is by forcing a separation of the interaction-specific parts from the application-specific parts. This allows the *interface designer* to concentrate on the design of the interface and the *application programmer* on the implementation of the application-specific code.

In FormsVBT, the only UI components known to the application are those that are given names. The application is insensitive to the layout of components and to the existence of all unnamed components. There is even some insulation between the application and the UI for named components: one component may be replaced with another whose behavior with respect to the application is the same.

For instance, in the Three-Cell Calculator interface from Section 2.4, we could replace the `Text`-component named `result` with any other component that can store text, such as a `Typescript`. A `Typescript` would capture a history of all values computed by the application. (We would also need to delete the `LeftAlign` and `Main` properties to make this change.)

We could also replace the radio buttons with items in a pulldown menu by replacing this expression

```
(Radio %functions
  (VBox
    (Choice %div "divide")
    (Choice %mul "multiply")
    (Choice %sub "subtract")
    (Choice %add "add")))
```

with the following expression:

```
(Menu "?" (Radio %functions =add
  (HBox
    (VBox
      (Choice MenuStyle %div "divide")
      (Choice MenuStyle %mul "multiply"))
    (VBox
      (Choice MenuStyle %sub "subtract"))
      (Choice MenuStyle %add "add")))))
```

See Fig. 2.3.

The first child of a `Menu` is the "anchor" of a pulldown menu; click on it to get a menu displayed. The second child of a `Menu` is an arbitrary S-expression, displayed when the user clicks on the anchor. In the example above, the first child is a `Text` component displaying a question mark. The second child contains four radio buttons, arranged in a 2-by-2 matrix. The `MenuStyle` property causes a radio button to react when the mouse rolls into it rather than on a mouse click.

The contents of this menu emphasizes an earlier point about composition. A `Menu` does not impose any structure on the contents of the menu. One merely composes a `Menu` out of 2 children: a

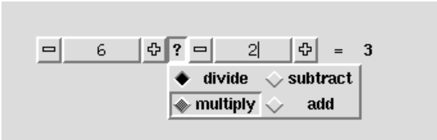


Figure 2.3: *A modified UI for the three-cell calculator application. The cursor (not visible in the figure) is over the string "multiply."*

child that is the anchor button, and a child that appears when the anchor button is activated. A "traditional" pulldown menu is a `VBox` whose children are `MButtons`.

Exercise 5: Change the program so that a symbol for the current operator is displayed instead of the question mark. Hint: Assign name to the quoted question mark, by using the expanded format `(Text %op "?")`, and call `FormsVBT.PutText` to change what is displayed in a `Text` component.

2.7 Subwindows

The three-cell calculator will crash if we try to divide by 0. Let's change the application to pop up a dialog box warning the user if there is an attempt to divide by 0. We need to modify the `Compute` event-handler by adding a test for a divisor equal to zero just before the division:

```
...
ELSIF Text.Equal (fn, "div") THEN
  IF second = 0.0 THEN
    FormsVBT.PopUp (fv, "errorWindow");
    RETURN
  END;
  answer := first / second
END;
...
```

The call to `FormsVBT.PopUp` will cause the named dialog to appear.

It is easy to add a dialog named `errorWindow` to the calculator's S-expression that was given in Section 2.4. The S-expression becomes the following:

```
(ZSplit
  (ZBackground (Shape ...))
  (ZChassis %errorWindow
    (Title "Error Message")
```

```
    (Rim (Pen 20)
      (Text %errorText "Can't divide by zero."))))
```

A `ZSplit` takes an arbitrary number of children and displays them as overlapping windows. The first child is the background; it is always visible. The visibility and location of the other children are under program control. The `ZChassis` wraps a "banner" around its child; the banner is responsive to mouse activity for the common window controls of closing, moving, and resizing. A call to `FormsVBT.PopUp` will cause a specified child of a `ZSplit` to appear. By default, a `ZChassis` is not initially visible.

Another common use of subwindows is to allow a user to specify additional information for a command. For example, the "Save As..." button found in many applications pops up a dialog box, which is a subwindow, with a way to enter the name of a file. A button like "About Bazinga..." pops up a subwindow containing information about the application called Bazinga.

In situations like these, it's a burden on the programmer to write an event-handler that simply calls `FormsVBT.PopUp`. To simplify this common case, FormsVBT provides a `PopButton`. This component is just like a `Button`, but before its event-handler is called, it causes a designated subwindow to appear. In practice, applications often don't need to attach any event-handler to a `PopButton`.

For grins, we'll now change the original three-cell calculator user interface so that the radio buttons are in a subwindow that is completely controlled by the user. Clicking on the `"?"` menu will cause the subwindow to appear. The window can be closed and repositioned without any application code. We need make two small changes to the original S-expression given in Section 2.4 to add subwindows. First, replace the radio buttons by a button that causes a subwindow to pop-up. That is, change

```
(Radio %functions ...)
```

to

```
(PopButton (For fnWindow) "?")
```

Second, move the `Radio` expression into a subwindow, by enclosing it in a `ZChassis`, and wrapping a `ZSpit` around the root. Now, `Calc3Cell.fv` looks like this:

```
(ZSplit
  (ZBackground (Shape ...))
  (ZChassis %errorWindow ...)
  (ZChassis %fnWindow
    (Radio %functions ...)))
```

Exercise 6: Add an "About Three-Cell Calculator..." button. It should pop-up a subwindow with appropriate information. If you want to put the button inside of a pull-down menu, use `PopMButton`.

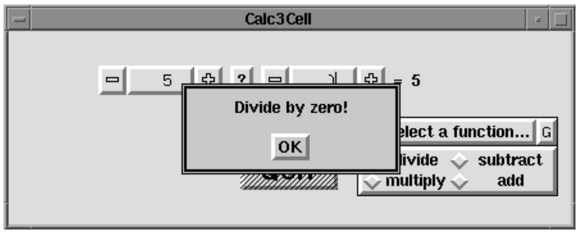


Figure 2.4: *The three-cell calculator application with a modal dialog.*

2.8 Modal Dialogs

When a subwindow appears, the rest of the form and all other subwindows remain active. In the case of the operator-subwindow in Section 2.7 (i.e., the `ZChassis` named `fnWindow`), this behavior was desirable. However, this behavior may not be desirable for the error-message subwindow. That is, some application writers would like to force the user to explicitly close the error message subwindow before continuing to interact in the application. In the UI jargon, this is called a *modal dialog*.

A simple way to do this is to bring up the error subwindow as before, but also to “deactivate” the background—make it unresponsive to user actions—while the subwindow is displayed. When the dialog is finished, we “re-activate” the background. A FormsVBT component called `Filter` is used to set the *reactivity* of its child to be *active* (the default case), *passive* (mouse and keyboard events are not sent), *dormant* (like passive, but it also grays out the child and changes the cursor), or *vanished* (like passive, but also draws over the child in the background color, thereby making it invisible).

Changing the modeless subwindow in the calculator so that it is modal requires only a trivial change. First, add a `Filter` just inside the `ZBackground`. Name this component `zbg`. Second, in the application, add

```
FormsVBT.MakePassive(form, "zbg")
```

after the call to `FormsVBT.PopUp`. Finally, you need to register an event-handler for the `ZChassis` named `errorWindow`. The event handler will be invoked when the subwindow is closed; it contains the following line:

```
FormsVBT.MakeActive(form, "zbg")
```

You might also wish to eliminate the banner on the subwindow. To do so, change the `ZChassis` to be a `ZChild`, and add a `CloseButton` somewhere in the subwindow. The `CloseButton` button will cause the subwindow in which it is contained to be taken down. Fig. 2.4 shows the modified application.

Exercise 7: Make the error window in the three-cell calculator modal in the manner suggested in this section: In the `.fv` file, add a `Filter` inside the `ZBackground`, change the error subwindow from a `ZChassis` to a `ZChild`, and add a `CloseButton` to the error window. In the `.m3` file, change the application code so that the background is made passive when the error window appears, and re-activated after error window disappears.

Exercise 8: When the error dialog appears while the subwindow containing operators is visible, the operators are not deactivated, although the main form is deactivated. Change the form so that *everything* except the error subwindow is made passive. Don't modify the application! (Hint: Use two `ZSplits`, one the background child of the other.)

2.9 A File Viewer

It's easy to hook up the FormsVBT text-editing widget to an application to make a *bona fide* text editor. The file-viewer application, shown in Fig. 2.5, contains a type-in area on the top for entering the name of a file and a fully functional text editor that occupies the bulk of the window. The text editor is in read-only mode.

The S-expression for the application, in the file `Viewer.fv`, is quite simple:

```
(Rim (Pen 10) (Font (WeightName "Bold"))
  (VBox
    (HBox
      (Frame Lowered (TypeIn %fileName))
      (Glue 10)
      (Button %exit "QUIT"))
    (Glue 10)
    (Shape (Height 200 + inf) (Width 300 + inf)
      (Frame Lowered (TextEdit ReadOnly %editor)))))
```

The application is structured as in the three-cell calculator application in Section 2.4. A `NewForm` procedure converts the S-expression into a runtime object and registers event-handlers. Only one event-handler is needed here; `ReadFile` is attached to the type-in field `fileName`. It is invoked whenever you type a carriage return in the type-in field. The code is straightforward:

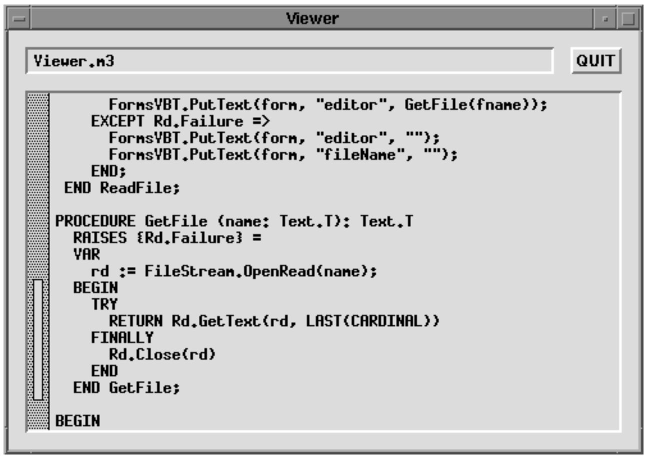


Figure 2.5: *A simple file viewer application.*

```
PROCEDURE ReadFile (cl   : FormsVBT.Closure;
                    fv   : FormsVBT.T;
                    name: TEXT;
                    time: VBT.TimeStamp) =
  VAR fname := FormsVBT.GetText (fv, "fileName");
  BEGIN
    TRY
      FormsVBT.PutText (fv, "editor", GetFile (fname));
    EXCEPT
      Rd.Failure =>
        FormsVBT.PutText (fv, "editor", "");
        FormsVBT.PutText (fv, "fileName", "");
    END;
  END ReadFile;
```

The event-handler first retrieves the string you typed into the type-in field named `fileName`. It then calls an internal procedure `GetFile` to retrieve the contents of a file by that name, and finally stores the contents into the text-editor widget. If an error is encountered while trying to retrieve the contents of the file, `ReadFile` catches the exception that is raised and just erases the contents of the type-in field and the text editor. The application is shown in Fig. 2.5.

Exercise 9: Add a Reset button to the left of the Quit button. Clicking on this button should clear the contents of the type-in field. For extra credit, interpret a double click to also clear the contents of the editor. To detect a double-click, you will need to examine the `VBT.MouseRec` that is available

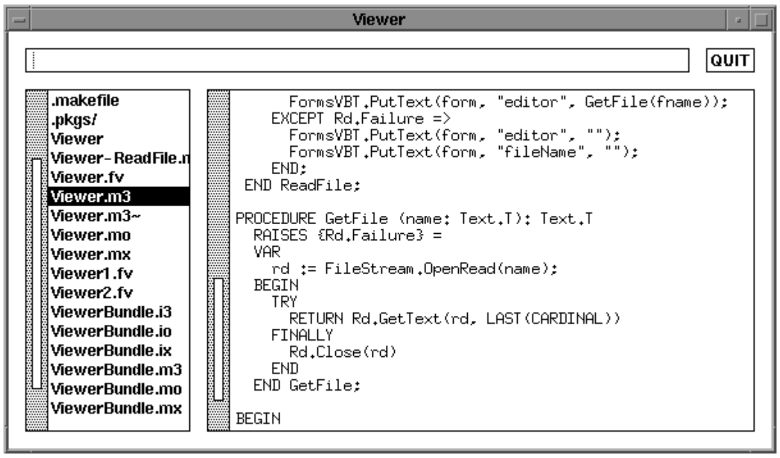


Figure 2.6: *The file viewer application again, but now, file names can be specified in the type-in field at the top or using the file browser at the left.*

---

from `FormsVBT.GetTheEvent` to the Reset button's event-handler.

Exercise 10: Add a pop-up to signal when the file could not be opened, rather than clearing the type-in field.

If you substitute `FileBrowser` for `TypeIn`, you'll be able to traverse the file system by double-clicking on directories (those items ending with a slash) in a browser. The file browser generates an event when you double-click on a file. Note that the application does not need be changed at all!

While it's nice to be able to traverse the hierarchy by mousing around in the file browser, there are times when it is more desirable simply to type in a pathname. No problem. We'll just add a type-in field to the S-expression. Here's the new S-expression (see Fig. 2.6):

```
(Rim (Pen 10) (ShadowSize -1)
     (BgColor "White") (LightShadow "Black")
     (DarkShadow "Black")
  (VBox
    (HBox
      (Frame Lowered (TypeIn %fileNameString))
      (Glue 10)
      (Button %exit "QUIT"))
    (Glue 10)
    (HBox
      (Shape (Width 100)
        (Frame Lowered (FileBrowser %fileName)))
      (Glue 10)
      (Shape (Height 200 + inf) (Width 300 + inf)
        (Frame Lowered (TextEdit ReadOnly %editor))))))
```

(A negative value for the inherited property `ShadowSize` is a convention for telling FormsVBT to give feedback using a flat, 2-d style rather than a Motif-like, 3-d style.)

We also need to change the application slightly to register the `ReadFile` event-handler with the type-in field (i.e., `fileNameString`) *as well as* with the file browser (i.e., `fileName`). In addition, procedure `ReadFile` itself needs to be changed trivially to initialize `fname` from the interactor that caused the event-handler to be invoked:

```
VAR fname := FormsVBT.GetText (fv, name);
```

So far so good, but there's no tie between the file browser and the type-in field.

Exercise 11: Implement event-handler methods for the file browser (`fileName`) and the type-in field (`fileNameString`) to keep them synchronized. That is, typing a path into the type-in field should cause the browser to change the directory it is displaying. The directory displayed by the file browser is set by calling `FormsVBT.PutText`, passing in the name of the directory to be displayed. What happens if you specify a directory that doesn't exist?

If you didn't do the last exercise, now is your last chance …

It turns out that FormsVBT already provides a component that ties a type-in field to a file browser. The component is called a `Helper`, and it has a class-specific property called `For` that names the file browser to which it is tied. So, if you change the expression

```
(TypeIn %fileNameString)
```

to

```
(Helper (For fileName))
```

and replace the initialization of variable `fname` in the original program as described above, then the type-in field and the file browser will stay synchronized, without any application-code intervention.

Exercise 12: What happens when you replace "`Helper`" by "`DirMenu`"? What happens when you tie a file browser to both a `Helper` and `DirMenu`?

---

3. The FormsVBT Language

The FormsVBT language provides a mechanism for textually describing a user interface. The language is not a general-purpose programming language. It has no variables, control structure, or run-time computation. Rather, it's a declarative textual description of the hierarchical arrangement of *components* that make up the user interface, written as an *S-expression*. Fig. 3.1 shows a simple user interface and its S-expression.

3.1 Basic Syntax

Each component is written as an S-expression that begins with type of the component, such as `Border`, `Button`, or `VBox`. Following the type are subexpressions that describe either*properties* or *children* (sub-components). Properties provide additional information that control the appearance or behavior of the component. The S-expression in Fig. 3.1 contains properties `Pen`, `Pattern`, `Width`, `Height`, `BgColor`, and `Color`. Property-expressions are distinguished from child-expressions by their names. There is one important rule to remember:

In any S-expression, all property expressions must appear before any sub-components.

Properties are discussed in detail in Section 3.3.

3.2 Components

The components of FormsVBT can be categorized in two ways: by the number of children that they take, or by their function. In the first categorization, we have *leaves*, *filters*, and *splits*:

- A *leaf* has no children.
- A *filter* has exactly one child.
- A *split* contains any number of children.

In Fig. 3.1, `Text` and `Pixmap` are leaf components; `Border`, `Button`, and `Shape` are filters; and `VBox` is the only component in the form that is a split.

The second way to categorize the components is by their function:


```
(Border (Pen 20) (Pattern "NWDiagonal")
  (Button %cornet
    (VBox
      (Shape (Width 50) (Height 40)
        (Pixmap "Trumpet"))
      (Text (BgColor "Black") (Color "White") "Horn"))))
```

Figure 3.1: *A very simple user interface and its description in the FormsVBT language. The user interface consists of a button that is surrounded by a border. The button itself displays a pixmap of a trumpet above the word "Horn". At runtime, the application will register an event-handler for the component named "cornet"; the event-handler will be invoked when the user clicks on the button.*

- *Passive visuals* establish appearance and spacing. Examples include `Text`, `Glue`, and `Border`.
- *Basic interactors* contain an editable value that both the user and the application can read and modify. Examples include `TypeIn`, `Numeric`, and `FileBrowser`.
- *Interactive modifiers* add behavior to a child. The prototypical example is a `Button`. Most interactive modifiers also add some type of visual feedback. For example, a `Button` adds a "shadow" around its child to make the child appear 3-dimensional. Some modifiers, like a `Boolean`, also involve an editable value.
- *Helpers* are components that control other components in some way. For example, a `PopButton` is a button that causes a subwindow (`ZChassis` or `ZChild`) to appear. A `DirMenu` is a pull-down menu that is used with a `FileBrowser`. The items in the menu are the ancestors of the directory currently being displayed.
- *Groupers* declare that all their descendants belong to one group for some particular purpose. A `Radio` unites a group of radio buttons, which are `Choice` components.
- *Geometers* take an arbitrary number of children and lay them out in some way. Examples include `VBox`, `HPackSplit`, and `TSplit`. An `HPackSplit` formats its children like words in an unjustified paragraph. A `TSplit` is a temporal split—at any time, exactly one of its children is visible.

3.3 Properties

Properties provide information that modifies the appearance or behavior of a component. A property subexpression has the following format:

(*keyword value*)

The keyword implies the expected *type* of the value; values are type-checked when the description is parsed by FormsVBT.[1] Nearly all properties have default values and can be omitted.

For example, the expression

```
(Boolean (CheckMark TRUE) (MenuStyle TRUE) "Gravity")
```

defines a Boolean interactor with two properties. The `CheckMark` property says to use a check mark rather than a check box for visual feedback, and the `MenuStyle` property says that the Boolean should be responsive to a mouse rolling into it, rather than responding only to a mouse click. You would use `MenuStyle` when the Boolean is an element of a menu.

A *value* must have one of the following types:

Text A quoted string.

```
(Border (Pattern "NWDiagonals") ...)
```

Cardinal A positive integer.

```
(TSplit (Value 4) ...)
```

Integer An integer.

```
(Numeric (Min -100) (Max 100))
```

Real A real number. A whole number does not need a decimal point, and a number between -1 and 1 does not need a leading zero.

```
(Border (Pen 4.25) ...)
```

Boolean The token `TRUE` or `FALSE`.

```
(Boolean (MenuStyle TRUE) ...)
```

CardinalList A list of positive integers.

```
(MultiBrowser (Value 1 5 3 19))
```

TextList A list of quoted strings.

[1] If you are developing a form using the `formsedit` interface-builder, then type- and syntax-errors will be reported and highlighted each time that you issue the "Do It" command. If an application gives the FormsVBT runtime system a syntactically incorrect S-expression to parse, FormsVBT will raise an exception to signal the syntax error.

```
(FileBrowser (Suffixes "i3" "m3"))
```

Symbol A name. For example, the `For` and `Name` properties are of this type:

```
(PageButton (For letters) "Next")
(Button (Name no) "Cancel")
```

Names are either identifiers (a letter followed by any number of letters, digits, or underscores), or non-empty sequences of characters from the set

```
! # $ % & * + - . / : < = > ? @ [ ] ^ _ { } ~
```

or a sequence of characters and escape sequences surrounded by vertical bars (e.g., `|Sue's button|`). The escape sequences are

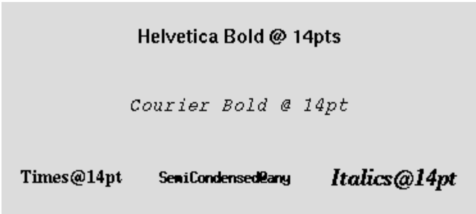
```
\n \t \r \f \\ \|
```

and `\` followed by three octal digits.

Font The name of a font conforming to the specifications in "X Logical Font Descriptions" [9]. The font can be specified in two ways: A quoted string in the form that `xlsfonts` prints and accepts as a pattern (we call this the *string format*), or a list of parenthesized keyword pairs for the parts of a font (we call this the *list format*). Consider the following example:

```
(VBox
  (Text (LabelFont "helvetica_bold14")
      "Helvetica Bold @ 14pts")
  (Text (LabelFont "-*-courier-medium-*-140-*")
      "Courier @ 14pt")
  (HBox (LabelFont (Family "Times")
                   (PointSize 140))
    (Text "Times@14pt")
    (Text (LabelFont Reset
                     (Family "*")
                     (Width "semicondensed"))
       "SemiCondensed@any")
    (Text (LabelFont (PointSize 180)
                     (Slant "i"))
        "Italics@14pt"))))
```

The font specification for the top two children of the VBox use the string format. The other font specifications are in the list format. Here's what the example looks like:



In the list format, the keywords for the parts of a font are

```
Foundry  Family     WeightName  Slant
Width    PointSize  HRes        VRes
Spacing  AvgWidth   Registry    Encoding
```

`PointSize`, `HRes`, `VRes`, and `AvgWidth` take cardinal values or the string `"*"`. All the others take strings.

Unspecified parts of a font take on the value of the nearest ancestor component for which the part was specified using the list format. However, the keyword `Reset` causes all unspecified parts of a font to take on the default values assigned by FormsVBT.

Color The description of a color either as a triplet of real numbers between 0.0 and 1.0 representing RGB or HSV values, or as a string. The following example shows both formats:

```
(Pixmap
  (BgColor .5 0.23 1.0)
  (Color "VeryPaleRed") "MailBox")
```

The triplet may be preceded by one of the symbols `RGB` or `HSV`. The default is `RGB`. The symbol `HSV` represents hue-saturation-value. Example:

```
(BgColor HSV 0.1 0.45 0.222)
```

Appendix B.1 describes the conventions used for naming colors.

Sx An S-expression in the FormsVBT language. For example, the `Title` property of a `ZChassis` (a frame for a subwindow) has this type:

```
(ZSplit
  (ZBackground ...)
  (ZChassis
    (Title
      (HBox
        Fill
        "Window # "
        (Border (Text %wid =""))
        Fill))
    ...))
```

Enumeration A set of mutually exclusive tokens. FormsVBT supports the following enumerations:

Type	*Keywords*
Alignment	`Center, LeftAlign, RightAlign`
Axis	`Horizontal, Vertical`
FeedbackStyle	`CheckBox, CheckMark, Inverting`
Reactivity	`Active, Passive, Dormant, Vanish`
ScrollStyle	`HorOnly, VerOnly, NoScroll, AlaViewport, Auto`
ShadowStyle	`Flat, Raised, Lowered, Ridged, Chiseled`

Think of each enumeration as a collection of Boolean properties, at most one of which may be specified as `TRUE`. If no choices are specified, then it's as if the default choice was given. Here are some examples:

```
(Viewport (VerOnly TRUE) (Horizontal TRUE) ...)
(Filter (Dormant TRUE) (Vanish FALSE)
    (Button ...))
(Frame (Raised TRUE)
    (Rim (Frame (Lowered TRUE) ...)))
```

Size The description of the dimensions of a component along some axis. It has the syntax

```
[size] [+ stretch] [- shrink]
```

where size, stretch, and shrink are specified as points in real numbers. Stretch and shrink, if both specified, may be in either order. Spaces are required around the plus and minus signs. The keyword `Inf` is used to indicate a very large value for stretch. See Section 3.6.1 for more details.

At The location of a subwindow component relative to its parent. There are two ways to specify this location: you can say where the center or a particular corner should be positioned; or you can specify where the four edges should appear. To position a subwindow by its center or corner, you write

```
(At h v [Center | NW | NE | SE | SW] [Scaled | Absolute])
```

If you don't specify the center or a corner, the default is `Center`. If you don't specify whether `h` and `v` are scaled or absolute, the default for this form is `Scaled`, which means that `h` and `v` indicate the proportionate placement in the horizontal and vertical directions of the center or corner; in this case, `h` and `v` must be numbers in the range 0–1. Otherwise, in the absolute case, `h` and `v` represent the horizontal and vertical distance, in points, between the subwindow's center or corner, and the parent window's northwest corner.

To position a subwindow by its edges, you write

```
(At west east north south [Absolute | Scaled])
```

The default in this case is `Absolute`, not `Scaled`; it indicates the distance in points between the subwindow's edges and the parent's west and north edges. Note that this is the only case in which you can specify the subwindow's exact size.

`Scaled` indicates the proportion of the parent window's width and height that mark the subwindow's boundaries. For example,

```
(At .10 .10 .90 .90 Scaled)
```

is effectively a "10% Rim" around the subwindow, while

```
(At .50 .50 1 1 Scaled)
```

places the subwindow in the parent window's southeast quadrant. See Section 3.7 for more details.

3.3.1 Varieties of Properties

Properties come in three varieties: class-specific, inherited, and universal.

Class-Specific Properties

Class-specific properties are defined in conjunction with a specific component class, and are allowed only on components of that class. It is fine for several components to use the same specific property.

There are two very common class-specific properties: `Main` and `Value`.

- `Main`

  Many passive leaf components exist to display some object; such an object is specified by a property called `Main`, whose type varies. If this property exists, it is usually required. Its value is usually specified by a shorthand: it simply follows the component keyword, without the word `Main` or parentheses. Here are two examples, in the abbreviated format:

  ```
  (Texture "Gray")
  (Pixmap "OpenRightArrow")
  ```

- `Value`

  The `Value` property specifies the initial state of some user-modifiable value. The type of this property depends upon the value type of the interactor. No class has both a `Main` property and a `Value` property. Here are two examples, in the abbreviated format:

  ```
  (Numeric =5)
  (TextBrowser (From "choices.txt") =(2 5 1))
  ```

Other specific properties (and there are many) are described with their component classes in Appendix A.

Universal Properties

Universal properties are applicable to components of all classes, and have a system-defined meaning. There is currently only one such property: `Name`.

- `Name` (type: **Symbol**; default: none)

  The name of a component, for access by the application. The type of the `Name` property is a **Symbol** and it has no default value. A form may not contain duplicate names; not all components need to have a `Name` property. The property may be abbreviated: `(Name goButton)` can be written as `%goButton`.

Inherited Properties

Inherited properties, like universal properties, may be specified for any component, though they are not relevant to all classes. But they have the special feature that a value specified for one component becomes the default value for all descendants of that component. Thus an inheritable property specification applies not to one VBT, but to an entire subtree. The inherited properties are: `Font` and `LabelFont`; `Color` and `BgColor`; `LightShadow`, `DarkShadow`, and `ShadowSize`. In essence, the inherited properties determine the overall "look and feel" of the user interface.

- `Font` (type: **Font**; default: see below)

  The font for components that display selectable text, such as `TextEdit` and the type-in part of a `Numeric`. The type of the `Font` property is **Font**, and the default value would be written in list format as follows:

  ```
  (Font
    (Foundry "*")
    (Family "fixed")
    (WeightName "medium")
    (Slant "r")
    (Width "normal")
    (PointSize 120)
    (HRes "*")
    (VRes "*")
    (Spacing "*")
    (AvgWidth "*")
    (Registry "iso8859")
    (Encoding "1"))
  ```

  Essentially, it's a 12-point, fixed-width font that can be scaled (using the `Scale` component).

- `LabelFont` (type: **Font**; default: see below)

  The font for components that display non-selectable text, such as `Text`, and various browsers such as `MultiBrowser` and `FileBrowser`. The type of the `LabelFont` property is **Font**, and the default value would be written in list format as follows:

  ```
  (LabelFont
    (Foundry "*")
    (Family "helvetica")
    (WeightName "bold")
    (Slant "r")
    (Width "*")
  ```

```
    (PointSize 120)
    (HRes "*")
    (VRes "*")
    (Spacing "*")
    (AvgWidth "*")
    (Registry "iso8859")
    (Encoding "1"))
```

Essentially, it's a 12-point, boldface helvetica font that can be scaled (using the `Scale` component).

- `Color` (type: **Color**; default: `0 0 0`)

  The foreground color; used for displaying text, bars, borders, the "on" pixels of pixmaps and textures, and so on. The default foreground color is black.

- `BgColor` (type: **Color**; default: `.8 .8 .8`)

  The background color; used for displaying text background, glue, and the "off" pixels of textures. The default background color is a light gray.

- `LightShadow` (type: **Color**; default: `1 1 1`)

  The color used for the "light shadow" in implementing a Motif-like 3-d look. The default light shadow is white.

- `DarkShadow` (type: **Color**; default: `.333 .333 .333`).

  The color used for the "dark shadow" in implementing a Motif-like 3-d look. The default dark shadow is a dark gray.

- `ShadowSize` (type: **Real**; default: `1.5`).

  The absolute value of this property is the size of the "shadow" in implementing a Motif-like 3-d looks. The default value is 1.5 points.

Look and Feel

In order to have an effective Motif-like look and feel, you need to change the `LightShadow` and `DarkShadow` whenever you change the `BgColor`. Shiz Kobara [4] provides an excellent set of guidelines for choosing harmonious color triples.

On a grayscale monitor, objects are displayed using the intensity of their color.

On a monochrome monitor, FormsVBT does not support a Motif-like look and feel. Rather, the user interface appears "Macintosh-like." For example, feedback on buttons is given by inverting the image of an object rather than raising and lowering the object; a Radio button uses bitmaps showing a filled or empty circle rather than a 3-d diamond that is either raised or recessed. Behind the scenes, two things are happening. First, on monochrome displays, `BgColor` displays as background and the other colors display in foreground. Second, the FormsVBT interactors are implemented in such a way as to give feedback using a 2-d style when displaying on a monochrome dispay.

Actually, the FormsVBT interactors use the Motif-like style only when they are on a non-monochrome display *and* the `ShadowSize` property is positive. Therefore, you can force a non 3-d look for a color or gray-scale monitor by setting the `ShadowSize` to be 0. You should probably also change the `BgColor` to be white in this case. Alternatively, you may find it convenient to set `ShadowSize` to be a negative number and the shadow colors to be black, as follows:

```
(BgColor "White")
(LightShadow "Black")
(DarkShadow "Black")
(ShadowSize -1.5)
```

This setup will cause the shadows on various objects, like buttons, to appear as black borders.

3.4 Syntactic Shortcuts

This section describes various shortcuts that make FormsVBT descriptions more readable.

1. The `Main` property may be given simply by giving its value, without the keyword or parentheses. This is allowed only for leaf components.

   `(Texture (Main "LightGray"))` ≡ `(Texture "LightGray")`

2. The `Value` property may be abbreviated by an equal sign, without parentheses, *and with no intervening space*.

   `(Numeric (Value 27))` ≡ `(Numeric =27)`

   Exception: If it's a **TextList** or **CardinalList**, then parentheses are needed.

   `(MultiBrowser (Value 4 9 2))` ≡ `(MultiBrowser =(4 9 2))`

3. The `Name` property may be abbreviated by a percent sign.

   `(Button (Name xyz) ...)` ≡ `(Button %xyz ...)`

4. Any Boolean-valued property may be set true simply by giving its name, without parentheses. By convention, the default value of all Boolean properties is false.

`(TypeIn (Scrollable TRUE))` ≡ `(TypeIn Scrollable)`

An element of an enumeration is a Boolean.

`(Viewport (Auto TRUE) ...)` ≡ `(Viewport Auto ...)`

5. A component of class `Text` may be given simply as a quoted string, provided that no properties other than the string are specified.

   `(Text "Hello")` ≡ `"Hello"`

6. Any leaf component from the following list

   `Bar Chisel Fill Glue Ridge`

   may be given by its keyword, without parentheses.

   `(HBox "A Heading" (Fill))` ≡ `(HBox "A Heading" Fill)`

3.5 Macros

The FormsVBT language supports macros. A macro is a procedure that returns an S-expression, called the *expansion*, that replaces the macro call; that is, the expansion is itself a FormsVBT expression. The parameters passed to the macro are not evaluated by the call, although they may be evaluated in the body of the macro. A macro definition can appear anywhere a component or property can appear.

A macro-definition has the following syntax:

`(Macro` *name [*`BOA`*]* `(`*formal* `...` *formal*`)` *expression*`)`

A formal parameter is either a name or a list of the form `(`*name default*`)` where *default* is any S-expression, the default value for the parameter.

A macro uses either positional binding or keyword binding, but not both. If the definition includes the keyword `BOA` ("By Order of Arguments"), then the macro uses positional binding, and the macro-call must have the form

`(`*name actual* `...` *actual*`)`

The actuals are bound to the formals in left-to-right order.

If the definition does not include the keyword `BOA`, then the macro uses keyword binding, and the macro-call must have the form (*name* (*formal actual*) ... (*formal actual*))

The actuals are bound to the formal parameters with corresponding names.

The number of actual parameters may not exceed the number of formal parameters. If there are fewer actuals than formals, then all the remaining formals must have default values.

The body of the macro-definition is an expression that is evaluated (expanded) when the macro is called. Typically, the body is a quoted or backquoted S-expression. As in Common Lisp macros, quoted S-expressions are constants; they expand into themselves. Backquoted expressions are *templates*; all of the subexpressions are treated as constants except for expressions preceded by a comma or a comma-atsign combination. In the expression `` `(A ,x B) ``, the value of `x` is substituted as the second element of the list; the expanded list will always have length 3. In the expression `` `(A ,@x B) ``, the value of `x` must be a list, and the elements of that list are "spliced in" between `A` and `B`; the expanded list will have length 2 (if the value of `x` is the empty list) or more.

For example, here is a macro that puts a 2-point border around its argument, after surrounding the argument by 16 points of background space on all four sides:

```
(Macro Boxed (x)
  `(Border (Pen 2) (Rim (Pen 16) ,x)))
```

The call `(Boxed (x (Text (BgColor "Red") "Warning")))` expands to

```
(Border (Pen 2)
  (Rim (Pen 16)
    (Text (BgColor "Red") "Warning"))))
```

If the definition of `Boxed` had included the keyword `BOA` then the expression could have been written as

```
(Boxed (Text (BgColor "Red") "Warning"))
```

Thus, for all practical purposes, we've effectively added a new filter-component called `Boxed` to the FormsVBT language.

Here is an example showing the use of default values:

```
(Macro Ht BOA (v (n 16)) `(Shape (Height ,n) ,v))
```

With this definition, the call `(Ht (Button "Go!") 20)` expands into

```
(Shape (Height 20) (Button "Go!"))
```

The call `(Ht (Button "Stop"))` uses the default value of `n` and expands into

```
(Shape (Height 16) (Button "Stop"))
```

An example using comma-atsign:

```
(Macro V (items)
  `(VBox (Color "Red") Fill ,@items Fill))
```

Given this definition, the call `(V (items ("abc" "def" "hij")))` expands into

```
(VBox (Color "Red") Fill "abc" "def" "ghi" Fill)
```

Macros must be defined before they are called. The effect of using a macro to redefine an existing name (e.g., `VBox`) is undefined.

It is permitted for a macro to expand into another `Macro`-expression, or into an expression *containing* another `Macro`-expression. Nested backquotes are permitted; they follow Common Lisp evaluation-semantics.

The expressions that are permitted in the body of a macro are not restricted to quoted and back-quoted expressions. As we have already seen, an expression may be the name of a formal parameter; the value of such an expression is the value of the corresponding actual parameter. Other expressions that are permitted include the following:

- `(Cat` *x y z …* `)`
  There must be at least two arguments, and all of them must have type `TEXT`. The result has type `TEXT`. Example: `(Cat "Gate-" x "-button")`
- `(Empty` *x*`)`
  The argument must be a `TEXT`; the result has type `BOOLEAN`.
- `(Equal` *x y*`)`
  The arguments may have any type; the result has type `BOOLEAN`.
- `(Length` *x*`)`
  The argument must be a `TEXT` or a list; the result has type `INTEGER`. (FormsVBT does not support a separate type for cardinals.)
- `(Sub` *s start count*`)`
  The argument *s* must be a `TEXT`; *start* and *count* must be non-negative integers. The result is a `TEXT`.
- `(SymbolName` *x*`)`
  The argument must be a symbol; the result is a `TEXT`.
- `(Intern` *x*`)`
  The argument must be a `TEXT`; the result is a symbol.
- `(Cons` *x y*`)`
  The first argument may have any type. (All S-expressions are REFs.) The second argument must be a list. The result is a list.

- `(List` $x\, y\, z \ldots$`)`
  The arguments may have any type; the result is a list.

- `(List*` $x\, y \ldots z$`)`
  There must be at least two arguments. The last argument must be a list; the others may have any type, and they are "consed" onto the front of the last argument.
  Example: `(List* 1 2 3 '(a b))` ≡ `(1 2 3 a b)`

- `(Append` $x\, y\, z \ldots$`)`
  All the arguments must be lists; the result is a list.

- `(Nth` $x\, n$`)`
  The first argument must be a list; the second argument must be an integer in the range `[0 .. RefList.Length(`x`) - 1]`. The result is the nth element of the list.

- `(NthTail` $x\, n$`)`
  The first argument must be a list; the second argument must be an integer in the range `[0 .. RefList.Length(`x`) - 1]`. The result is the nth tail of the list.

- `(IF` *pred* $x\, y$`)`
  The value of the first argument must be a `BOOLEAN`. If the value is `TRUE`, then x is evaluated, and its value is the value of this expression. Otherwise, y is evaluated, and its value is the value of this expression. I.e., this is `IF` as in Lisp, not as in Modula-3.

- `(AND` $x\, y\, z \ldots$`)`
  All the arguments must be of type `BOOLEAN`, as is the result. The arguments are evaluated from left to right. If any argument evaluates to `FALSE`, the value of this expression is `FALSE`, and the remaining arguments are not evaluated. If all the arguments evaluate to `TRUE`, or if there are no arguments, then the value of this expression is `TRUE`.

- `(OR` $x\, y\, z \ldots$`)`
  All the arguments must be of type `BOOLEAN`, as is the result. The arguments are evaluated from left to right. If any argument evaluates to `TRUE`, the value of this expression is `TRUE`, and the remaining arguments are not evaluated. If all the arguments evaluate to `FALSE`, or if there are no arguments, then the value of this expression is `FALSE`.

- `(NOT` x`)`
  The argument must be a `BOOLEAN`, as is the result.

- `(=` $x\, y\, z \ldots$`)`
  There must be at least two arguments. If x is a number (integer or real), then all the other arguments must be numbers of the same type as x, and the result is `TRUE` if they are all equal, and `FALSE` otherwise. If x is not a number, then the result is `TRUE` if all the arguments are the same REF.

- `(<` *x y z ...*`)`
  `(<=` *x y z ...*`)`
  `(>` *x y z ...*`)`
  `(>=` *x y z ...*`)`
  There must be at least two arguments, and they must all be numbers of the same type as *x*. The result is of type `BOOLEAN`.
  `(<` *x y z ...*`)` ≡ `(AND (<` *x y*`) (<` *y z*`) ...)`
  Likewise for the other operations.

- `NIL`
  This is a constant.

Macros provide one kind of extensibility to the FormsVBT language. Another kind of extensibility is provided by the `realize` method for a `FormsVBT.T` object. The `realize` method allows the programmer to define subtypes of the `VBT` classes that FormsVBT uses, such as the `FVTypes.FVButton`. However, it is not currently possible for the client to extend the language with any other `VBT` classes, such as `TranslateVBT.T` or a client-defined subtype of `VBT.Leaf`. See Section 4.7 for details.

3.6 Layout

Every component has a *natural size* in the horizontal and vertical axes; these are its *width* and *height*. It may also have *shrinkability* and *stretchability* in each axis, to allow it to adapt in a visually pleasing way as the window is resized. The *minimum* size of a child in each axis is its natural size minus its shrinkability, and the *maximum* size in each axis is its natural size plus its stretchability. The *size range* of a component in each axis is the interval between its minimum and its maximum.

The size ranges in each axis are computed for a top-level window by a bottom-up process. Each split computes its ranges as a function of the size ranges of its children; the function used depends on the type of the split.

FormsVBT uses TEX's "boxes-and-glue" layout model. At the center of the layout strategy are two split classes, `HBox` and `VBox`. These organize the layout of their children along the horizontal and vertical axes, respectively. To keep the discussion simple, we will explain the algorithm for `HBox`.

An `HBox` reports its size as follows. An `HBox`'s natural width is the sum of the natural widths of its children; its width shrinkability is the sum of the width shrinkabilities of its children (but no more than its natural size), and its width stretchability is the sum of the width stretchabilities of its children. An `HBox`'s height range is the intersection of the height ranges of its children (if the intersection is empty, the children's maximum heights are increased until the intersection is non-empty). The `HBox`'s natural height is the maximum of the natural heights of its children, projected into the range.

Ultimately, the shape of each top-level window is controlled by the user through a window manager. The window manager allows the user to shrink and grow a top-level window in each axis. However, the window manager will not let user grow a top-level window beyond its maximum size bounds, or shrink a top-level window below its minumum size bounds, in each axis. When a top-level window's size is changed, the new size-information is propagated down through the top-level window's tree of subwindows. How each split component communicates this information to its children depends on the type of the component.

When an HBox is given some screen real estate to allocate among its children, here's what happens. In the vertical dimension, it gives each child the same vertical height it has been given; that's easy. In the horizontal dimension, things are more interesting. The HBox computes the sum of the natural widths of the children; this is the natural width of the HBox. Ideally, the HBox would give each child its natural width and that's all there is to it. If this is not possible, then either the HBox has extra space it must divide among the children, or the HBox must take away space from its children.

In the first case, the HBox allocates its extra space in proportion to the children's stretchabilities. For example, if the first child has twice as much stretchability as the middle child, and three times as much as the third and last child, then the extra space is divided $6/11$, $3/11$, and $2/11$ to the children, from left to right. In the second case, the HBox takes away space from the children in proportion to the amount that each child can shrink.

If the sum of the minimum sizes of the children is greater than the size of the HBox, then the HBox is said to be *overfull*. In this case the children are considered in order and given their minimum sizes, as long as there is room. The first child that doesn't fit is given all the space that's left, and the remaining children are given size zero.

If the sum of the maximum sizes of the children is less than the size of the parent, the split is said to be *underfull*. This produces a state in which the children are stretched larger than their maximum sizes, but in proportion to their relative stretchabilities.

3.6.1 How Sizes are Specified

Most of the time, sizes are not given explicitly; natural sizes are allowed to take effect. Leaf components have an inherent natural size that is usually data-dependent. For example, the size of a Text is the size of the rectangle needed to display it in the appropriate font. In each axis, it has no shrinkability but "infinite" stretchability. A vertical Scroller has a fixed width (a natural size with no stretch and no shrink). Its natural vertical size is quite small (enough to show a "thumb"), it has no vertical shrinkability, and it has infinite vertical stretchability. In practice, a vertical scrollbar is an element of an HBox, so it takes on the size of the other elements of the HBox.

A filter component derives its size information from its child. A Border component, for example, takes the size of its child, but adds twice the border's thickness in both dimensions. A Guard takes on precisely the size of its child. Appendix A describes how each component computes its shape.

A property of type **Size** is used to describe the size of a component along one dimension. It has the syntax

```
[size] [+ stretch] [- shrink]
```

where size, stretch, and shrink are specified as points in real numbers. Stretch and shrink, if both specified, may be in either order. Spaces are required around the plus and minus signs. The keyword Inf is used to indicate a very large value for stretch; it may also be spelled inf or INF.

A natural size may be overridden, completely or in part, by specifying the `Width` and `Height` properties on a `Shape` filter. For the sake of simplicity, let's consider just the `Width` property. There are eight situations to consider: when size, stretch, and shrink are all missing; when just size is given; when just stretch and shrink appear; and so on.

See Figure 3.6.1 for details.

A few common paradigms merit mention. First, to remove whatever inherent stretchability a component has, use

```
(Shape (Width + 0) ...)
```

Second, to make a component stretchy regardless of its inherent stretchiness, use

```
(Shape (Width + Inf) ...)
```

And third, to set a component to a particular size, e.g, 100, use:

```
(Shape (Width 100) ...)
```

3.6.2 Precedence of Size Constraints

The various constraints on the size of an object sometimes come into conflict. They take precedence as follows:

1. Downward-propagating constraints: window size forced by Trestle, vertical size forced by an `HBox`, and so on.

2. Explicit size information, given by a `Shape` filter.

3. Upward-propagating natural size information: inherent size of leaves, filters taking size from their children, an `HBox` taking width from the sum of its children's widths, on so on.

3.7 Subwindows

In addition to organizing child components by grouping them horizontally or vertically, FormsVBT allows child components to overlap. The split that does this organization is called a `ZSplit`.

There are two very different ways to use a `ZSplit`.

1. If you don't like arranging elements in horizontal and vertical boxes, you could place each element at a specific location. Many UI Builders follow this model; it has its advantages and disadvantages. We rarely use this style at SRC.

2. You can use `ZSplits` as a container for overlapping, often transient *subwindows* that are not installed as top-level windows.

Without loss of generality, we'll talk just about the second style of use.

A `ZSplit` is written like this:

all missing	<q-p, q, q+r> A no-op; reports the child's size
size	<size, size, size> Constrains child's natural size to size, with no stretch or shrink
- shrink	<q-shrink, q, q+r> Forces child's shrink to be shrink; doesn't change child's natural size or stretchability
+ stretch	<q-p, q, q+stretch> Forces child's stretch to be stretch; doesn't change child's natural size or shrinkability
- shrink + stretch	<q-shrink, q, q+stretch> Changes child's shrink to be shrink and its stretch to be stretch; doesn't change child's natural size
size - shrink	<size-shrink, size, size> Changes child's size to be size with no stretchability and with shrink shrinkability
size + stretch	<size, size, size+stretch> Changes child's size to be size with no shrinkability and with stretch stretchability
size - shrink + stretch	<size-shrink, size, size+stretch> Changes child's size to be size with shrink shrinkability and with stretch stretchability

This table describes what Shape reports, as a function of its child's size. The notation $< q-p, q, q+r >$ refers to the child's size: the natural size is q; it has p shrinkability, so it can shrink to a minimum of $q-p$, and it can stretch to a maximum of $q+r$.

```
(ZSplit
       The first child:
  (ZBackground ...)
       Any number of other z-children:
  (ZChild ...)
  (ZChassis ...)
  ...
  )
```

The first child is called the *background*. It is displayed below all other children.

The other children are ordered from bottom to top in the z-dimension. A non-background child of a `ZSplit` should be a `ZChild` or a `ZChassis`. It has an `Open` property to say if it should be initially visible ("mapped") or invisible ("unmapped").

It also has an `At` property to control where it should appear when it is made visible. The syntax of the `At` property is described below.

A `PopButton` is a button that causes a named subwindow to appear. A `PopMButton` is a version of `PopButton` that is appropriate for inclusion in a menu.

A `CloseButton` is a button that causes a named subwindow to disappear.

A `ZGrow` is a button that is used to change the size of a subwindow. A `ZMove`-component is used to change a subwindow's position by dragging.

A `ZChassis` is just a `ZChild` that has a standard configuration, including a frame whose banner includes a `CloseButton`, a title inside a `ZMove`, and a `ZGrow`.

The "At" Property

The location of a subwindow (denoted by a `ZChild` and `ZChassis` component) is specified using a property named `At`. The `At` property is a list that can take one of two forms: the "corner" form (two numbers, an optional corner, and an optional coordinate type); or the "edges" form (four numbers and an optional coordinate type). The coordinate types are either `Absolute`, which means that the coordinates represent the distance in points from the background window's northwest corner, or `Scaled`, which means that the coordinates represent a fraction (a number in the range 0–1) of the background window's width or height. Here are the two forms of the `At` property: `(At` *h v* `[Center | NW | SW | NE | SE] [Scaled | Absolute] )` `(At` *west east north south* `[Absolute | Scaled] )`

If the list contains two numbers, then these coordinates specify the center of the subwindow. If the list contains two numbers and a corner, then these coordinates specify the position of that corner of the subwindow. In either case the default coordinate type is `Scaled`.

For example, `(At 0.5 0.5)` means that the center of the subwindow should be placed halfway across and halfway down the background window, i.e., that it should be centered in the background window. (If there is no `At` property, this is the default.)

`(At 0.2 0.3 NW)` means that the northwest corner of the subwinbdow should be place 20% of the way across the background window, and 30% down. This can also be written as `(At 0.2 0.3 NW Scaled)`. Scaled coordinates must be written as numbers (integers or reals) in the range 0–1.

`(At 100 237.5 Absolute)` means that the center of the subwindow should be placed 100 points east and 237.5 points south of the background window's northwest corner.

`(At 100 237.5 SE Absolute)` means that the subwindow's southeast corner should be placed at that position.

Alternatively, you may specify the *edges* of the subwindow by using a list with four numbers, representing the west, east, north, and south edges, in that order. The numbers may be followed by a coordinate type; the default coordinate type in the 4-number form is `Absolute`, not `Scaled` as it is in the 2-number form.

If the coordinate type is `Absolute`, then the coordinates represent the distance in points from the background window's northwest corner; this is the only form in which you can specify the subwindow's actual size. For example, `(At 20 120 60 300)` means that the subwindow's width should be 100 points wide ($120 - 20$) and 240 points tall ($300 - 60$), and that its northwest corner should be 20 points east and 60 points south of the background window's northwest corner.

`(At .10 .90 .30 1 Scaled)` means that the subwindow occupies the middle 80% horizontally ($.90 - .10$) and the bottom 70% ($1 - .30$) of the background window.

`(At .5 1 .5 1 Scaled)` would place the subwindow in the southeast quadrant of the background window.

Here are some additional examples:

```
(ZChassis %A (At .2 .3 NW) ...)
(ZChassis %B (At 130 200 SE) ...)
(ZChassis %C (At .1 .6 .2 1) ...)
(ZChassis %D (At 20 120 60 300)
```

If the `ZSplit` containing the subwindow is 200 points wide and 300 points high, here is what each specification means:

- Subwindow `A` has its northwest corner at (40, 90). That is, its northwest corner is 40 points to the right of the `ZSplit`'s left edge and 90 points below the top.
- Subwindow `B` has its southeast corner at (130, 200).
- Subwindows `C` and `D` have the same positions: The northwest corner is at (20, 60) and the southeast corner is at (120, 300).

The actual location of a subwindow has two additional restrictions. First, FormsVBT will ensure that the size it gives a subwindow will be within the subwindow's acceptable dimensions: the northwest corner stays fixed, and the southwest corner is adjusted. Second, FormsVBT will not pop up a subwindow with its northwest corner north or west of the visible portion of its parent; it will move the subwindow away from the specified position in order to bring it into view.

3.8 Catalog of Components

This section provides a brief description of current FormsVBT components. Appendix A describes the details of each component.

Visual components

These leaf and filter components have no interactive behavior; they are used to provide appearance and positioning control.

`Border`	displays space, in the foreground color, around its child
`Rim`	displays space, in the background color, around its child
`Pixmap`	displays a pixmap, centered
`Text`	displays a single-line text
`Texture`	displays a textured rectangle
`Bar`	a line in the foreground color; child of HBox or VBox
`Glue`	a piece of background filler; child of HBox or VBox
`Fill`	an infinitely stretchable background filler; child of HBox or VBox
`Frame`	draws a 3-d border around its child
`Chisel`	like Bar, but line appears 3-d, chiseled into background
`Ridge`	like Bar, but line appears 3-d, raised above background
`Scale`	enlarges or shrinks child
`Shape`	constrains shape of child

The following filters respond to mouse activity. They do not report events to the application program.

`Viewport`	adds scrollbars around a child for panning
`Filter`	controls reactivity and visibility of child

Basic Interactors

These are leaf components that have a user-modifiable value. They should always be named so that the application will have access to the value.

`Numeric`	an editable integer
`Browser`	a group of lines of text, of which one may be selected
`MultiBrowser`	a `Browser` in which multiple lines may be selected
`Generic`	a placeholder to be taken over by the application
`Scroller`	a vertical or horizontal scrollbar

Text Editing Interactors

These leaf components provide extensive text editing facilities.

`TypeIn`	an editable-text region, typically single-line
`TextEdit`	a scrollable text-editing area
`Typescript`	a `TextEdit` with a reader and writer

File Browser Interactors

A `FileBrowser` displays the names of the files in a directory, initially the current working directory. The user can traverse the file system by double-clicking on elements in the browser. There are two related leaf interactors that facilitate traversal.

`FileBrowser`	the list of filenames
`Helper`	an area for typing filenames
`DirMenu`	a pulldown menu containing the names of parent directories

Basic Buttons

These filters take a child and add some interactive behavior to it.

`Button`	generates an event when clicked
`Guard`	click once to remove and expose underlying component
`TrillButton`	generates an event while mouse is down

Boolean and Radio Buttons

A `Boolean` and `Choice` are types of buttons that also maintain some state. A `Radio` is a "grouper": it takes a child and changes neither its appearance nor its behavior. Rather, it specifies that it and all its descendants are members of one "group" for some particular purpose.

`Boolean`	toggles on/off when clicked
`Choice`	a radio button; selects itself when clicked
`Radio`	defines a group of Choice components

Drag and Drop

These buttons provide a way to implement "drag-n-drop" and to get semantic feedback.

`Source`	a button that is dragged
`Target`	the thing into which a `Source` is dropped

Menus

`Menu`	a pull-down menu; pulls down when anchor is clicked
`MButton`	a pull-down menu element; generates an event on up-click

Other buttons that can be put into a menu are `Boolean`, `Choice`, and `PopMButton`.

Horizontal and Vertical Splits

Splits take an arbitrary number of children and lay them out in some fashion. These splits implement the TEX-like "boxes-and-glue" layout model.

`HBox`	horizontal layout
`VBox`	vertical layout
`HTile`	an `HBox` with user-adjustable divider bars between children
`VTile`	`VBox` with user-adjustable divider bars between children
`HPackSplit`	arranges children like words in a paragraph
`VPackSplit`	arranges children like paragraphs in a multi-column newspaper

The layout algorithm for `HTile` and `VTile` is slightly different than for `HBox` and `VBox` when the size of one of its children changes. In the case of the tiles, the algorithm tries to keep existing children with their same relative sizes, which might have been adjusted by the user from their initial assignments. The `HBox` and `VBox` always re-assign sizes, independent of the current sizes of the children.

Subwindows

A `ZSplit` organizes its children as overlapping subwindows. The following components allow the user to control the appearance, location, and size of subwindows.

`CloseButton`	closes a subwindow
`PopButton`	pops up a subwindow
`PopMButton`	a menu item that pops up a subwindow
`ZBackground`	needed around the background child
`ZChild`	needed around non-background children
`ZGrow`	a button for resizing a subwindow
`ZMove`	a button for repositioning a subwindow
`ZChassis`	a handy combination of `ZChild`, `CloseButton`, `ZMove`, and `ZGrow`

Temporal Windows

`TSplit`	a temporal window that organizes its children so that exactly one child is visible at any given time.
`LinkButton`	displays a specific child in a `TSplit`.
`PageButton`	switches children displayed in a `TSplit`.

A common use of a `TSplit` is to make an arbitrary component appear or disappear under user control. The component whose visibility is to be toggled is put into a `TSplit` with one sibling: a component with no size. A `LinkButton` to the component will cause it to appear, and another `LinkButton` to the sibling will effectively cause the component to disappear.

4. Programming with FormsVBT

All ordinary client access to the FormsVBT system is handled by the `FormsVBT` interface. The other interfaces exported by the FormsVBT package are given in Appendix B.

4.1 The FormsVBT Interface

FormsVBT is a system for building graphical user interfaces. FormsVBT provides a special-purpose language for describing user interfaces, an interface-builder that allows editing of such descriptions, and a runtime library for applications to make use of the user interfaces.

The locking level for any procedure in this interface that may alter an installed `VBT` is `LL.sup = VBT.mu`. (See the *Trestle Reference Manual* for a complete description of locking levels [6].) Most applications don't need to worry about `VBT.mu` because their event-handlers don't fork any threads that call FormsVBT.

```
INTERFACE FormsVBT;

IMPORT AnyEvent, Color, Filter, Rd, Rsrc, Sx, Thread, VBT,
       Wr, ZSplit;

EXCEPTION
  Error (TEXT);
  Unimplemented;
  Mismatch;
```

4.2 Creation, allocation, and initialization

An object `fv` of type `FormsVBT.T` (or simply, a *form*) is created by parsing an S-expression. These expressions are usually stored in files with the suffix `.fv`. One way of creating a form is to call the procedure `NewFromFile`, or the method `fv.initFromFile`, with the name of such a file; the expression is parsed, and if there are no errors, a new `VBT` is created and stored in the form, which is returned.

It is also possible for a program to generate a description "on the fly" and then use it to create a form. The methods `fv.init`, `fv.initFromRd`, and `fv.initFromSx` support these options. Forms can also be stored in resources (`fv.initFromRsrc`) and in URLs (`fv.initFromURL`).

```
TYPE
  T <: Public;
  <* SUBTYPE T <: MultiFilter.T *> (* ... *)
  Public = Filter.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (description : TEXT;
                   raw         : BOOLEAN   := FALSE;
                   path        : Rsrc.Path := NIL    ): T
                   RAISES {Error};

             initFromFile (filename : TEXT;
                           raw      : BOOLEAN   := FALSE;
                           path     : Rsrc.Path := NIL    ): T
                 RAISES {Error, Rd.Failure, Thread.Alerted};

             initFromRd (rd   : Rd.T;
                         raw  : BOOLEAN   := FALSE;
                         path : Rsrc.Path := NIL    ): T
                 RAISES {Error, Rd.Failure, Thread.Alerted};

             initFromSx (sx   : Sx.T;
                         raw  : BOOLEAN   := FALSE;
                         path : Rsrc.Path := NIL    ): T
                 RAISES {Error};

             initFromRsrc (name : TEXT;
                           path : Rsrc.Path;
                           raw  : BOOLEAN    := FALSE): T
                 RAISES {Error, Rd.Failure,
                         Rsrc.NotFound, Thread.Alerted};

             initFromURL(baseURL  : TEXT;
                         raw      : BOOLEAN := FALSE): T
                 RAISES {Error, Rd.Failure, Thread.Alerted};

             realize (type, name: TEXT): VBT.T RAISES {Error};

             <* LL.sup = VBT.mu *>
             snapshot (wr: Wr.T) RAISES {Error};
             restore  (rd: Rd.T) RAISES {Mismatch, Error};

           END;
```

The call `fv.init(description, raw, path)` initializes `fv` as a form and returns `fv`. It creates a `VBT`, *v*, from `description`, which must contain a single, valid S-expression. The methods `initFromFile`, `initFromRd`, `initFromSx`, `initFromRsrc`, and `initFromURL` provide analogous support for files, readers, S-expressions, and named resources.

The `raw` parameter is used to control that actual internal structure `fv`. Regardless of the value of `raw, fv` is a multi-filter and `MultiFilter.Child(fv)` will always return *v*. Internally, `fv` is a filter; if `raw` is `TRUE`, then the filter's child is *v*. Otherwise, `fv` is "cooked", which means there are several filters inserted between *v* and `fv`, so that the filter's child has the following structure:

```
(ZSplit
  (Filter
    (HighlightVBT
      (Filter v))))
```

The filter above *v* supports the common case of making an entire form passive without requiring an explicit `Filter` interactor in the description. It also functions to restore the keyboard focus to whichever of the form's descendant-`VBT`s had most recently acquired the keyboard focus. The `ZSplit` supports menus and other pop-up operations, even if there is no `ZSplit` explicitly mentioned in the description. To get the `ZSplit` that is inserted, use `GetZSplit`. Clients should not traverse a cooked form directly. We reserve the right to change the filters that are inserted.

The `path` parameter is used for looking up all resources that are mentioned in the form: the name of a `Pixmap` or `Image`; a file for `Insert`; the `ItemFromFile` property on `Browser` and `MultiBrowser`; and the `From` property on `Text`, `TypeIn`, and `TextEdit`. (The `initFromURL` looks up resources as URLs, relative to `baseURL`.)

Briefly, the description of a form is an S-expression whose first element is the name of a component (e.g., `HBox`), and whose other elements are either properties (e.g., `Color`), or other components, typically describing the `VBT`-children of the outer component.

The `VBT`-tree is created during a depth-first traversal of the S-expression. On the way down, each `VBT` is *allocated*, typically with a call to `NEW(...)`. Then the subexpressions, if any, are traversed. On the way back up, each `VBT` is *initialized*, typically with a call to `v.init(...)`. The result is returned to the caller, where it is typically an argument to the parent's `init` method.

In other words, allocation occurs top-down, and initialization occurs bottom-up. (For more details on allocation, see Section 4.7.)

For each subexpression, the parser produces a `VBT` whose type is defined in the `FVTypes` interface, and whose name corresponds to the first element of the subexpression. For example, from the S-expression `(HBox ...)`, the parser creates an object of type `FVTypes.FVHBox`.

```
PROCEDURE NewFromFile (filename: TEXT;
                       raw                 := FALSE;
                       path    : Rsrc.Path := NIL    ): T
  RAISES {Error, Rd.Failure, Thread.Alerted};
```

Create a new form from the description in the file. `Rd.EndOfFile` *is signalled as* `Error`.
Equivalent to `NEW(T).initFromFile (name, raw, path)`

```
PROCEDURE GetZSplit (fv: T): ZSplit.T RAISES {Error};
```

Return the `ZSplit` *that "cooked" mode inserts. An exception is raised if* `fv` *was created with* `raw = TRUE`.

4.3 Events and Symbols

4.3.1 Attaching event-handlers

Most interactive components in the user interface generate events. (See the Appendix A for a description of all components.) To register an event-handler for such a component, the component must be named, and the client must call `Attach` or `AttachProc`, giving the name of the component and a procedure to be called when an event occurs in that component.

```
PROCEDURE Attach (fv: T; name: TEXT; cl: Closure) RAISES {Error};
```

Attach an event-handler ("callback") to the component of `fv` *whose name is given by* `name`. *If there is no such component, or if that component does not generate events (e.g.,* `Text`*), then* `Error` *will be raised. If* `cl` *is* `NIL`, *then any existing event-handler for that component is removed. Otherwise, when an event occurs in the named component, the implementation calls*

```
          cl.apply(fv, name, time)
```

```
TYPE
  Closure = OBJECT
            METHODS
              apply (fv: T; name: TEXT; time: VBT.TimeStamp);
            END;
```

```
PROCEDURE AttachProc (fv        : T;
                      name      : TEXT;
                      p         : Proc;
                      eventData: REFANY := NIL) RAISES {Error};
```

This is an alternate, somewhat simpler way to attach an event-handler. When an event occurs in the named component, the implementation calls

```
       p(fv, name, eventData, time)
```

```
TYPE
  Proc = PROCEDURE (fv        : T;
                    name      : TEXT;
                    eventData: REFANY;
                    time      : VBT.TimeStamp);
```

These event-handlers do not provide any other details, such as what key or mouse button was pressed, or whether it was a double-click. If such information is needed, call `GetTheEvent` to retrieve it.

```
PROCEDURE AttachEditOps (fv         : T;
```

```
                              editorName: TEXT;
                              cut, copy, paste, clear,
                              selectAll, undo, redo,
                              findFirst, findNext, findPrev: TEXT := NIL)
  RAISES {Error};                     <* LL.sup = VBT.mu *>
```

Create and attach event-handlers for common editing operations.

`editorName` must be the name of a text-editing component: `TextEdit`, `TypeIn`, `Numeric`, or `Typescript`. If `cut` is not `NIL`, then it must be the name of a component (typically a menu-button), and `AttachEditOps` will create an event-handler for it that will invoke the **Cut** operation on the text-editing component. Similarly, if `copy` is not `NIL`, then it should name a component for which `AttachEditOps` will create an event-handler that invokes the **Copy** operation on the text-editing component. Likewise for `paste`, `clear`, and so on.

4.3.2 Access to the current event

```
PROCEDURE GetTheEvent     (fv: T): AnyEvent.T    RAISES {Error};
PROCEDURE GetTheEventTime (fv: T): VBT.TimeStamp RAISES {Error};
```

Retrieve the details of the event that is currently in progress. These routines may be called only during the dynamic extent of an event-handler attached to some component via `Attach` *or* `AttachProc`.

```
PROCEDURE MakeEvent (fv: T; name: TEXT; time: VBT.TimeStamp)
  RAISES {Error};
```

`MakeEvent` *invokes the event-handler for the component of* `fv` *whose name is* `name`. *A component has an event-handler if attached via* `Attach` *or* `AttachProc`, *or if the component is a PopButton, PopMButton, PageButton, PageMButton, LinkButton, or LinkMButton.*

`MakeEvent` is useful when one part of a large program wishes to communicate with another part, by pretending that the named event occurred. For example, a client might want typing a particular control-character in a text-editing component to have the same effect as selecting a menu-item such as "Quit." `MakeEvent` provides a way to link the two events to the same handler.

```
VAR MakeEventMiscCodeType: VBT.MiscCodeType; (* CONST *)
```

The exact type of the result of `GetTheEvent` *depends on the user action that caused the event to be generated, a key, a mouse-click, etc. If the event was actually caused by a call to* `MakeEvent`, *the type of the result will be* `AnyEvent.Misc`, *and the value of its* `type` *field will be* `MakeEventMiscCodeType`.

4.3.3 Symbol management

```
PROCEDURE AddSymbol (fv: T; name: TEXT) RAISES {Error};
```

Add a "virtual" component to `fv` *with the given* `name`. *The form will behave as if there was a component called* `name` *(i.e., the call* `GetVBT(fv, name)` *will return a valid* `VBT`*).*

This procedure is most useful as a means to communicate between distant parts of a large program. One part of the program would use `AddSymbol` *to create a new symbol; another part would call* `MakeEvent` *to invoke an event-handler for the symbol.*

`Error` *is raised if* `name` *is already defined in* `fv`.

```
PROCEDURE AddUniqueSymbol (fv: T): TEXT;
```

Just like `AddSymbol`, *but finds a name that has not been used yet. The name is returned.*

4.4 Reading and Changing State

In response to an event or other occurrence, a program may want to read or change the state of various interactors in the form. This is handled by the various Get and Put procedures. Get procedures take the form and the name of the interactor, and return its value. Put procedures take the form, the name of the interactor, and the new value to be set.

There are several Get procedures and several Put procedures, for convenient handling of various Modula-3 types. These should be used as appropriate to the type of the interactor: `GetText` for a `TypeIn`, `GetInteger` for a `Numeric`, `GetBoolean` for a `Boolean` or `Choice`, etc. However, some conversions are supported: `PutInteger` to a `TypeIn` will convert the integer into text; `GetInteger` will likewise attempt to convert the text of the `TypeIn` to an integer (and return 0 in case of failure). All Get and Put procedures, however, will raise `Error` if applied to a component that does not have a value.

4.4.1 Access to the `Main` and `Value` properties

```
PROCEDURE GetText (fv: T; name: TEXT): TEXT
  RAISES {Error, Unimplemented};
```

This is implemented for `Browser`, `FileBrowser`, `Numeric`, `Text`, `Typescript`, *and the text-interactors:* `TextEdit`, `TypeIn`, *and* `TextArea`.

```
PROCEDURE PutText (fv: T; name: TEXT; t: TEXT; append := FALSE)
  RAISES {Error, Unimplemented};
```

This is implemented for `Browser`, `FileBrowser`, `Pixmap`, `Text`, `Typescript`, *and the text-interactors:* `TextEdit`, `TypeIn`, *and* `TextArea`. *For* `Text` *and the text-interactors, if* `append` *is true, then* `t` *is added to the end of the current text, rather than replacing it.*

```
PROCEDURE GetInteger (fv: T; name: TEXT): INTEGER
  RAISES {Error, Unimplemented};
PROCEDURE PutInteger (fv: T; name: TEXT; n: INTEGER)
  RAISES {Error, Unimplemented};
```

These are implemented for `Browser`, `Numeric`, `Scroller`, *and* `TSplit`. `PutInteger` *only is implemented for* `Audio`.

If you use `PutInteger` to select the nth child of a `TSplit` and that child has a text-editing component that has the `FirstFocus` property, then the text-editor will acquire the keyboard focus, and if it's a `TypeIn`, its text will be selected in replace-mode.

```
PROCEDURE GetBoolean (fv: T; name: TEXT): BOOLEAN
  RAISES {Error, Unimplemented};
PROCEDURE PutBoolean (fv: T; name: TEXT; val: BOOLEAN)
  RAISES {Error, Unimplemented};
```

These are implemented for `Boolean` *and* `Choice`.

4.4.2 Access to arbitrary properties

FormsVBT provides access to properties other than `Main` and `Value`. The intention is to provide access to all the inherited and class properties. For example, the `Scroller` component has an integer-valued property named `Min`, so it should be possible to call

```
GetIntegerProperty(fv, name, "Min")
```

to retrieve that value, or

```
PutIntegerProperty(fv, name, "Min", 6)
```

to change the value to 6.

WARNING: The current implementation provides access only to the inherited properties, and even that access is limited.

Note also that changing the value of a property in a component will not affect its subcomponents.

```
PROCEDURE GetTextProperty (fv: T; name, propertyName: TEXT): TEXT
  RAISES {Error, Unimplemented};
```

This is implemented for the `Font` *and* `LabelFont` *properties for all components, as well as the* `Items` *and* `Select` *properties of* `Browser`*s, and the* `ActiveTarget` *property of* `Source`*s.*

```
PROCEDURE PutTextProperty (fv: T; name, propertyName: TEXT; value: TEXT)
  RAISES {Error, Unimplemented};
```

This is implemented for the `Color`, `BgColor`, `Font`, *and* `LabelFont` *properties for all components, as well as the* `Items` *and* `Select` *properties of* `Browser`*s.*

```
PROCEDURE GetIntegerProperty (fv: T; name, propertyName: TEXT):
  INTEGER RAISES {Error, Unimplemented};
```

```
PROCEDURE PutIntegerProperty (fv                : T;
                              name, propertyName: TEXT;
                              value             : INTEGER)
  RAISES {Error, Unimplemented};
```

This is implemented for the `Min` *and* `Max` *properties of* `Numerics`*; the* `Min`, `Max`, `Step`, *and* `Thumb` *properties of* `Scrollers`*; and the* `Quality`, `ImageWidth`, `ImageHeight` *and* `MSecs` *properties of* `Videos`*, and the* `NorthEdge`, `SouthEdge`, `EastEdge` *and* `WestEdge` *properties of all VBTs.*

```
PROCEDURE GetRealProperty (fv: T; name, propertyName: TEXT): REAL
  RAISES {Error, Unimplemented};
PROCEDURE PutRealProperty (fv                : T;
                           name, propertyName: TEXT;
                           value             : REAL  )
  RAISES {Error, Unimplemented};
```

This is implemented for the `HScale` *and* `VScale` *properties of* `Scales`*.*

```
PROCEDURE GetColorProperty (fv: T; name, property: TEXT): Color.T
  RAISES {Error, Unimplemented};
```

Return the color used by the named component. `property` *must be one of* `Color`, `BgColor`, `LightShadow`, *or* `DarkShadow`*.*

```
PROCEDURE PutColorProperty (         fv            : T;
                                     name, property: TEXT;
                            READONLY color         : Color.T)
  RAISES {Error, Unimplemented};
```

Set the color used by the named component. `property` *must be* `Color` *or* `BgColor`*.*

```
PROCEDURE GetBooleanProperty (fv: T; name, propertyName: TEXT):
  BOOLEAN RAISES {Error, Unimplemented};
PROCEDURE PutBooleanProperty (fv                : T;
                              name, propertyName: TEXT;
                              value             : BOOLEAN)
  RAISES {Error, Unimplemented};
 (* This is implemented for the "ReadOnly" property of
   "TextEdit"s, and the shadow styles of "Frame"s.  The
   "PutBooleanProperty" is implemented for the "Synchronous",
   "Paused" and "FixedSize" properties of "Video", and for the
   "Mute" and "MuteWhenUnmapped" properties of "Audio" *)
```

4.4.3 Access to the underlying `VBT`s

```
PROCEDURE GetVBT (fv: T; name: TEXT): VBT.T RAISES {Error};
```

Return the `VBT` *corresponding to a named interactor in* `fv`. `Error` *is raised if there is no such* `VBT`.

```
PROCEDURE GetName (vbt: VBT.T): TEXT RAISES {Error};
```

If `vbt` *is the* `VBT` *corresponding to a named interactor in some form, returns the name given to that interactor. Otherwise, raisses* `Error`.

4.4.4 Radios and Choices

```
PROCEDURE GetChoice (fv: T; radioName: TEXT): TEXT
  RAISES {Error, Unimplemented};
PROCEDURE PutChoice (fv: T; radioName, choiceName: TEXT)
  RAISES {Error, Unimplemented};
```

Get/Put the name of the selected `Choice` *in a radio-button group. If there is no selection,* `GetChoice` *returns* `NIL`. *If* `choiceName` *is* `NIL`, *the radio-group will have no selection.*

```
PROCEDURE MakeSelected (fv: T; choiceName: TEXT) RAISES {Error};
PROCEDURE IsSelected (fv: T; choiceName: TEXT): BOOLEAN
  RAISES {Error};
```

Set/test a `Choice`*-button without referring to its group.*

4.4.5 Generic interactors

```
PROCEDURE GetGeneric (fv: T; genericName: TEXT): VBT.T
  RAISES {Error};
```

Retrieve the `VBT` *used by the named* `Generic` *interactor.*

```
PROCEDURE PutGeneric (fv: T; genericName: TEXT; vbt: VBT.T)
  RAISES {Error};
```

Replace the named `Generic` *interactor with* `vbt`, *which may be* `NIL`. *When* `NIL` *is specified, a default (and initial)* `VBT` *is used: a* `TextureVBT` *with 0 size and 0 stretch in each dimension.*

4.4.6 Special controls for Filters

The `(Filter ...)` expression in FormsVBT supports a feature called *reactivity*. This has one of four states: Active, Passive, Dormant, or Vanished. The state can be specified in the description and changed by the application at runtime. The default state is Active. In the Passive state, the component and its descendants, if any, are unresponsive to mouse clicks. The Dormant state is like Passive, but the component and descendants are "grayed out." Dormant is often to be preferred over Passive, because it provide additional feedback to the user. In the Vanished state, the component becomes unreactive and disappears entirely.

A cursor is specified when the state is set, and the name is interpretted by the Trestle implementation. An empty string (the default value) indicates that you don't care about the cursor shape.

Standard X screentypes support the cursors named in *X Window System* by Scheifler et. al. [**?**] Appendix B. Therefore, for example, `XC_arrow` returns a cursor that behaves like the X arrow cursor on X screentypes, and like the default cursor on screentypes that have no cursor named `XC_arrow`.

```
PROCEDURE MakeActive  (fv: T; name: TEXT; cursor:= "") RAISES {Error};
PROCEDURE MakePassive (fv: T; name: TEXT; cursor:= "") RAISES {Error};
PROCEDURE MakeDormant (fv: T; name: TEXT; cursor:= "") RAISES {Error};
PROCEDURE MakeVanish  (fv: T; name: TEXT; cursor:= "") RAISES {Error};
```

Find the nearest ancestor of the named component that is of type `FVFilter`*, and set its state and cursor as indicated. The exception is raised if no such ancestor can be found.*

```
PROCEDURE IsActive   (fv: T; name: TEXT): BOOLEAN RAISES {Error};
PROCEDURE IsPassive  (fv: T; name: TEXT): BOOLEAN RAISES {Error};
PROCEDURE IsDormant  (fv: T; name: TEXT): BOOLEAN RAISES {Error};
PROCEDURE IsVanished (fv: T; name: TEXT): BOOLEAN RAISES {Error};
```

Find the nearest ancestor of the named component that is of type `FVFilter`*, and test its state as indicated. The exception is raised if no such ancestor can be found.*

4.4.7 Access to Subwindows

```
PROCEDURE PopUp (fv        : T;
                 name      : TEXT;
                 forcePlace: BOOLEAN          := FALSE;
                 time      : VBT.TimeStamp := 0    )
  RAISES {Error};
```

Pop up the named subwindow.

Assuming that `name` is the name of an element of `fv` that can be popped up, pop it up. That is, the named element must be a non-background child of a `ZSplit`, or some descendant thereof. In the latter case, the ancestor that is a direct child of the `ZSplit` will be the thing popped up. Call this ancestor *zchild*. `PopUp` is equivalent to activating

`(PopButton (For` *zchild*`) ...)`

If the target *zchild* is already open or has been opened before and has been moved by the user (to a location that is now visible), it will normally be left where the user left it. The `forcePlace` option will force it instead to be returned to its canonical place.

If the subwindow contains a text-editing component that has the `FirstFocus` property, then that component will acquire the keyboard focus, and if it's a `TypeinVBT.T`, its text will be selected in replace-mode.

```
PROCEDURE PopDown (fv: T; name: TEXT) RAISES {Error};
```

The inverse of `PopUp`*: make the named element (or suitable ancestor) invisible. This is implemented using* `ZSplit`*'s unmapping. (Unfortunately, this doesn't cause the keyboard focus to be lost.) The exception is raised if* `name` *is not the name of an element of* `fv`*.*

4.4.8 Special controls for text-interactors

```
PROCEDURE TakeFocus (fv       : T;
                     name     : TEXT;
                     eventTime: VBT.TimeStamp;
                     select                      := FALSE)
  RAISES {Error};
```

Give the keyboard focus to a specified interactor. An exception is raised if the interactor is not of a suitable class to take it; however, no exception is raised if the keyboard focus cannot be taken because of a timeout, i.e., an invalid `eventTime`*. If* `select` *is* `TRUE` *and the focus was taken, then select the entire contents of the interactor's* `TextPort` *as a primary selection in replace-mode.*

4.5 Saving and restoring state

FormsVBT allows clients to save and restore the entire state of a form.

A *snapshot* is an S-expression that captures the state of components in a form. The call `fv.snapshot(wr)` writes a snapshot of `fv` to the writer `wr`, and the call `fv.restore(rd)` reads a snapshot from the reader `rd` and restores the components of `fv` to the state in the snapshot.

A snapshot produced by the default method contains only those named components that have a modifiable value. More precisely, a component is part of a snapshot if (1) it has a name and (2) the call to `GetText`, `GetInteger`, `GetReal`, `GetBoolean`, or `GetChoice` does not raise an exception. If you want to include a component into the snapshot that has state but does not respond to `GetText`, `GetInteger`, etc., then you need to override the defaults methods.

The `snapshot` method raises the `Error` exception if there is a problem writing the snapshot to the writer.

The `restore` method raises the `Error` exception if there is a syntax error in the S-expression or if there is any type of problem with the reader.

When restoring, the snapshot need not precisely match the set of interactors in the form. If the snapshot lacks values for some fields that the form contains, those fields will be left alone. If the snapshot has values for some fields that the form does not contain, the `restore` method should raise `Mismatch`, but only after restoring all the values that do match. If the snapshot has a value for a field that the form contains, but the types do not agree, this is a show-stopping error; the `restore` method should raise `Error`. Catching `Mismatch` is useful when you want to continue to tolerate snapshots from old versions of a form.

The default `snapshot` and `restore` methods write S-expressions in the following format:

```
((name1 value1)
 (name2 value2) ...)
```

4.6 Dynamic Alteration of Forms

FormsVBT provides facilities for modifying a form while a program is running. For example, one might want to add or delete items in a menu.

The procedure `Insert` parses a description of new form in the context of an existing form, and `Delete` removes a component and all of its descendents.

The procedure `InsertVBT` is used for forms within forms, where the subforms are independent from the forms containing them to avoid name clashes. You need to use `DeleteVBT` to delete a subform inserted this way.

Any resizing that may be appropriate after the modifications to the form is performed automatically. For the common case of modifying menus, this is not an issue because the menu is (almost certainly) not visible at the time the alteration takes place.

```
PROCEDURE Insert (fv          : T;
                  parent      : TEXT;
                  description: TEXT;
                  n           : CARDINAL := LAST(CARDINAL)): VBT.T
  RAISES {Error};
<* LL.sup = VBT.mu *>
```

`Insert` parses a description in the context of an existing form, that is, in `fv`'s namespace, so that names already defined in `fv` are visible while the description is being parsed, and with the state (color, resource-path, etc.) that was in effect for `parent`.

Once the new `VBT` is created, it is inserted into the named component, which must be a Split, as the `n`th child. It is also returned.

```
PROCEDURE InsertFromFile (fv       : T;
                          parent   : TEXT;
                          filename: TEXT;
                          n        : CARDINAL := LAST(CARDINAL)): VBT.T
  RAISES {Error, Rd.Failure, Thread.Alerted};
  <* LL.sup = VBT.mu *>

PROCEDURE InsertFromRsrc (fv     : T;
                          parent: TEXT;
                          name   : TEXT;
                          path   : Rsrc.Path;
                          n      : CARDINAL    := LAST(CARDINAL)): VBT.T
  RAISES {Error, Rd.Failure, Rsrc.NotFound, Thread.Alerted};
<* LL.sup = VBT.mu *>
```

`InsertFromFile` and `InsertFromRsrc` read a description from a file or named resource, and then call `Insert`.

```
PROCEDURE Delete (fv    : T;
```

```
                  parent: TEXT;
                  n     : CARDINAL;
                  count : CARDINAL   := 1) RAISES {Error};
  <* LL.sup = VBT.mu *>
```

Delete the children whose indices are in the range `[n .. (n + count - 1)]` *from the named component, which must be a Split. The names of the* `n` *components, as well as the names of all of the desendants of those components, are removed from* `fv`*'s namespace.*

```
PROCEDURE InsertVBT (fv    : T;
                     name  : TEXT;
                     child : VBT.T;
                     n     : CARDINAL := LAST (CARDINAL))
  RAISES {Error};
  <* LL.sup = VBT.mu *>
```

Insert `child` *as the* `n`*th child of the named component, which must be a Split. The names of components in* `child` *are not added to* `fv`*'s namespace. Thus,* `InsertVBT` *is typically used for "forms within forms."*

```
PROCEDURE DeleteVBT (fv    : T;
                     name  : TEXT;
                     n : CARDINAL;
                     count    : CARDINAL := 1)
  RAISES {Error};
  <* LL.sup = VBT.mu *>
```

Like `Delete`*, this procedure deletes the children whose indices are in the range* `[n .. (n + count - 1)]` *from the named component, which must be a Split. Unlinke* `Delete`*, the names of the* `n` *components, as well as the names of all of the desendants of those components, are not removed from* `fv`*'s namespace. Thus,* `DeleteVBT` *is typically only used with children that were inserted using* `InsertVBT`.

```
END FormsVBT.
```

4.7 Subclasses of components

As the subexpressions describing the form `fv` are being parsed, the `VBT`-components are created (allocated) by calling

`fv.realize(`*type*`, `*name*`)`

where *type* is the name of the first element of the subexpression, and *name* is the `Name` property specified in the subexpression, or the empty string if no such property was specified. For example, if the description contains the expression

```
(Menu %mainMenu ...)
```

then the FormsVBT parser will call

```
fv.realize("Menu", "mainMenu")
```

to create the `VBT`.

By overriding the `realize` method of `fv`, the client can create subtypes for any or all of the components. For each kind of form, there is a corresponding type in the `FVTypes` interface. For example, the result of parsing `(Menu ...)` is an object that is a subtype of `FVTypes.FVMenu`. The `realize` method must allocate and return a `VBT` that is a subtype of the corresponding type in `FVTypes`.

For example, suppose you wanted the form to keep a count of the number of menus it contains, and for each menu to store its own index.

```
TYPE
  MyForm = FormsVBT.T OBJECT
             count: CARDINAL := 0
           OVERRIDES
             realize := Realize
           END;
  MyMenu = FVTypes.FVMenu OBJECT
               index: CARDINAL
           END;

PROCEDURE Realize (fv: MyForm; type, name: TEXT):
  VBT.T RAISES {FormsVBT.Error} =
  BEGIN
    IF Text.Equal (type, "Menu") THEN
      WITH m = NEW (MyMenu, index := fv.count) DO
        INC (fv.count);
        RETURN m
      END
    ELSE                     (* use the default *)
      RETURN FormsVBT.T.realize (fv, type, name)
    END
  END Realize;
```

Note that the `realize` method does not *initialize* the `VBT` that it allocates. Actually, it may initialize any *private* fields, such as the `index` field in this example, but the `VBT`'s `init` method should not be called inside the call to `fv.realize`, since it will be called later during a "bottom-up" initialization phase. Of course, the client may also override the `init` method to control what happens in that phase.

A more complicated case arises with text-editing components. Textports are *contained* in three forms: `TextEdit`, `Typescript`, and `Numeric`. In a `TextEdit` components, the textport is in an exported field, `TextEditVBT.T.tp`. If the `realize` method allocates a `TextPort.T`, even a private subtype of `TextPort.T`, it should not call the textport's `init` method, since FormsVBT will do that in the initialization phase, passing some of the current state information (such as background color and the width of the "turn margin") to the textport's `init` method. The same applies to `Typescript` components, since `TypescriptVBT.T` is a subtype of `TextEditVBT.T`. Similarly, the textport in a `Numeric` component is in an exported field, `NumericVBT.T.typein`; again, it may be allocated but not initialized in the `realize` method.

If you wish to redefine the interpretation of keystrokes, you do so by overriding the `filter` method of the textports. The following code illustrates how to do this.

```
TYPE
  MyForm = FormsVBT.T OBJECT
             OVERRIDES realize := Realize END;

PROCEDURE Realize (fv: MyForm; type, name: TEXT): VBT.T
  RAISES {FormsVBT.Error} =
  BEGIN
    IF Text.Equal (type, "TextEdit") THEN
      RETURN
        NEW (FVTypes.FVTextEdit,
             tp := NEW (TextPort.T, filter := MyFilter))
    ELSIF Text.Equal (type, "Numeric") THEN
      RETURN NEW (FVTypes.FVNumeric,
                  typein := NEW (NumericVBT.Typein,
                                 filter := MyFilter))
    ELSIF Text.Equal (type, "Typescript") THEN
      RETURN NEW (FVTypes.FVTypescript,
                  tp := NEW (TypescriptVBT.Port,
                             filter := MyFilter))
    ELSIF Text.Equal (type, "TypeIn") THEN
      RETURN NEW (FVTypes.FVTypein, filter := MyFilter)
    ELSE                         (* use the default *)
      RETURN FormsVBT.T.realize (fv, type, name)
    END
  END Realize;
```

The `realize` method can also be used to integrate any VBT, including leafs and filters into a form. The components `Any`, `AnyFilter`, and `AnySplit` are defined to be `VBT.Leaf`, `VBT.Filter`, and `VBT.Split` respectively.

5. FormsEdit

FormsEdit is a stand-alone application that allows you to develop a FormsVBT user interface: the layout, the colors, the fonts, the text, the buttons, the pop-up windows, the shadows, … everything except the application code that does whatever it is you're building a user interface to.

The shell-command `formsedit` takes an optional argument, the name of an `.fv` file you wish to edit. It also takes optional X11 parameters for specifying the initial display and geometry; the manpage has the details.

The following sections describe the user interface.

5.1 Getting started

When you start the program, you'll see two windows: an editor and a result view (see Figure 5.1). The editor has a simple FormsVBT S-expression in it:

```
(Rim (Pen 10)
  (Text (Name ignoreMe) "This space available for a small fee"))
```

The result view is simply a window that contains the text, surrounded by 10 points of whitespace, or in this case, "greyspace", since the default background color is light grey.

5.2 The menubar

The menubar has four menus and a button.

5.2.1 The quill-pen menu

This menu has four items:

About FormsEdit ... shows the copyright notice and other information.

Help pops up a window with an online help-file, containing a list of all the components and their properties.

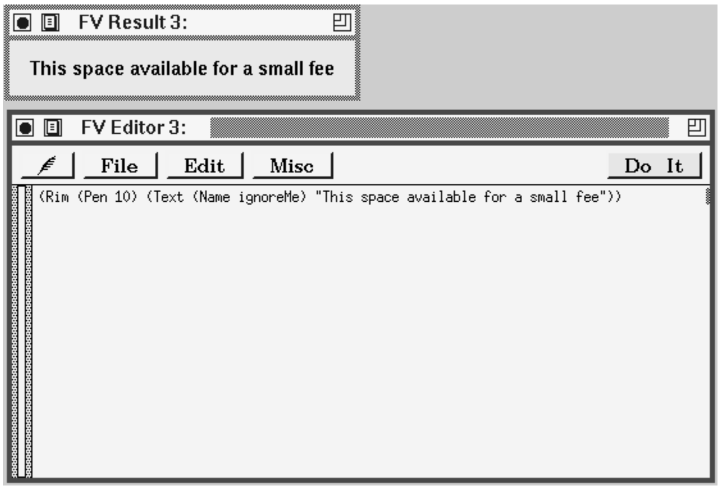


Figure 5.1: *The initial text-editor window shown by* `formsedit`.

Editing Model includes a choice of keybindings and selection controls; they also determine the keyboard equivalents that appear in the menu items. The four choices, Ivy, Emacs, Mac, and Xterm, are documented in the *VBTkit Reference Manual*[2].

Quit This terminates FormsEdit.

5.2.2 The File menu

This menu (see Figure 5.2) contains a standard list of items:

New creates another pair of windows, using the same, simple S-expression.

Open..., which is selected in the figure, brings up a file browser.

Close closes the window. If you click **Close** or **Quit** while there are unsaved changes, you will be asked whether you want to save them.

Save and **Save As...** are self-explanatory.

Revert to Saved re-reads the expression from the disk file.

PP setup... brings up a window with a `Numeric` component that lets you establish the width that the pretty-printer should use; typing Return causes the S-expression to be pretty-printed at the new width. The user-interface descriptions in `.fv` files tend to grow fairly quickly. If you can afford the screen real-estate, you might try reshaping the editor window to be as wide as possible, setting the pretty-printer width to 150, and typing Return.

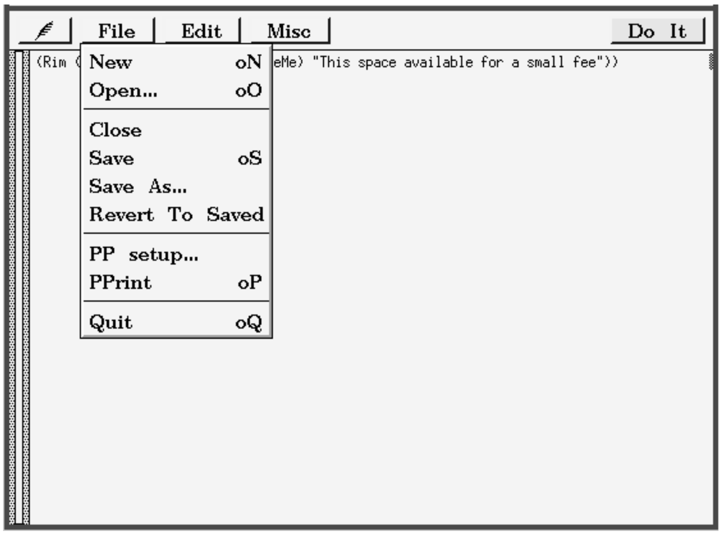


Figure 5.2: *The File menu.*

PPrint invokes the pretty-printer, and rewrites the S-expression in the window.

5.2.3 The Edit menu

The **Edit** menu has buttons for **Undo** and **Redo**; buttons for the standard editing commands **Cut**, **Copy**, **Paste**, **Clear**, and **Select All**; a **Find...** button that brings up a dialog box for specifying the string you wish to search for; and buttons for **Find Next** and **Find Prev**, which look for the current Source selection.

5.2.4 The Misc menu

The **Misc** menu contains an item for examining the named components in the form you're editing—it shows their names, types, and shapes; an item for producing a snapshot (see Section 4.5); and a button to bring up the error-message window, which normally disappears 5 seconds after it displays a message. This menu also contains items that allow you to move the editor and result windows from one screen to another.

5.2.5 The "Do It" button

The space to the right of the **Misc** button contains the name of the file being edited (if the window is showing a file). When there are unsaved changes to the window, a "note" icon also appears here.

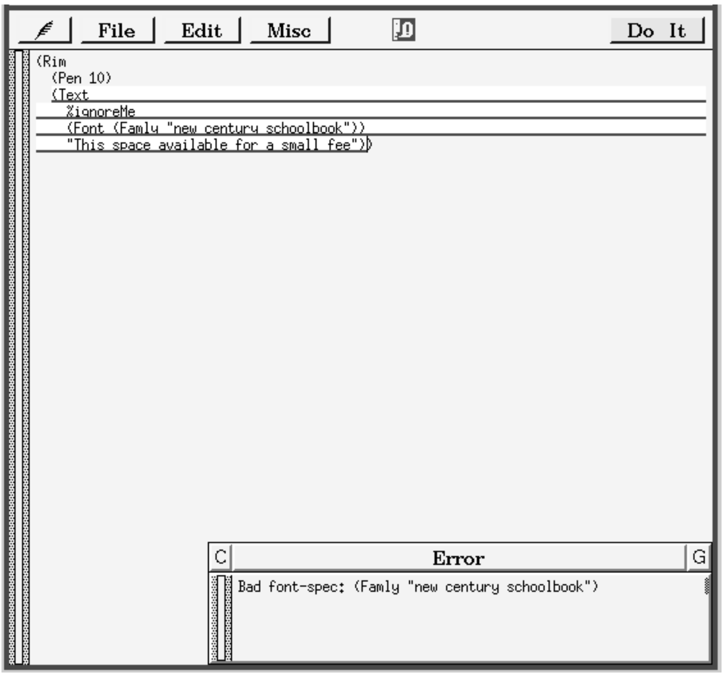


Figure 5.3: *The error window.*

After you've edited the file describing your user interface, you'll want to see what it looks like. Click the **Do It** button. On some keyboards, there's a (big) key labeled **Do**; you can press that instead of clicking the button. The key labeled **Enter** may also be used.

Every time you click the **Do It** button (or press the **Do** key or the **Enter** key), FormsVBT parses the entire S-expression, and updates the result view accordingly. The result-view window will change its shape, if necessary, to give the form its preferred shape.

5.3 Errors

What if there's an error in your form? The parser will detect it and highlight the nearest enclosing S-expression. An error window will pop up, explaining what the error was. If you click the **OK** button in the error window, the highlighting will disappear.

For example, suppose you wanted to change the font in the sample S-expression, and instead of writing `Family`, you misspelled it as `Famly`. When you hit the **Do It** button, the error window pops up; see Figure 5.3. The `Text` subexpression is highlighted, and the error message says: `Bad font-spec: (Famly "new century schoolbook")`.

When an error has been detected, the result view is not changed. If you open a window onto a file that contains an erroneous S-expression, its result window will be in some undefined state.

If we correct the misspelling and click **Do It**, the error window will disappear.

A. Full Description of Components

This appendix contains a complete description of each FormsVBT component. Each description contains the following sections:

- A banner containing the component's name and syntactic classification (`Leaf`, `Filter`, or `Split`). A box around the name, e.g., $\boxed{\text{Button}}$ indicates that the component generates an *event*; `FormsVBT.Attach` and `FormsVBT.AttachProc` can be used to attach an *event-handler* to such a component.
- A short description of the component.
- The class-specific properties, if any, for the component.
- A description of the component's interactive behavior, if it generates an event.
- The component's shape information.
- Additional notes (optional).

The first line of each class-specific property contains the property name, the access information, the type, the default value, and a description of the property. Example:

`Value` GP (Boolean, `FALSE`)

The property name, `Value` in this example, is the symbol that would appear in the S-expression. The access information is indicated by the small letters G and P. G means that the value of this property can be retrieved at runtime ("Get"); P means that the value can be set at runtime ("Put").

If the name of the property is `Value` or `Main`, then you can retrieve the value of the property by calling `GetText`, `GetInteger`, or `GetBoolean`, depending on the type of the property, and you can set the value by calling `PutText`, `PutInteger`, or `PutBoolean`.

For properties other than `Value` or `Main`, you can retrieve the value of the property by calling `GetTextProperty` or `GetIntegerProperty`,and you can set the value by calling `PutTextProperty` or `PutIntegerProperty`. These procedures take the name of the property as an argument.

The descriptions in this appendix do not include the components' inheritable properties (see section 3.3, page 39). All (well-behaved) components use these properties for displaying information. Some of these can be accessed at runtime, as the following list indicates:

`Font`	GP	(Font, default on page 39) The font used for Browser, DirMenu, FileBrowser, Helper, MultiBrowser, Numeric, TextEdit, Typein, and Typescript.
`LabelFont`	GP	(Font, default on page 39) The font used for Text and the Title of a ZChassis.
`Color`	P	(Color, `"Black"`) The foreground color.
`BgColor`	P	(Color, `0.8 0.8 0.8`) The background color.
`LightShadow`		(Color, `"White"`)
`DarkShadow`		(Color, `0.333 0.333 0.333`)
`ShadowSize`		(Real, `1.5`)

Some components define enumerations that specify mutually exclusive choices for a single, unnamed property. Syntactically, these are all Boolean properties, but unlike other Boolean properties, which always have a `FALSE` default value, one of these will default to `TRUE`. Furthermore, specifying any of them as `TRUE` has the effect of declaring all the others to be `FALSE`.

For example, the shadow-style of a Frame may be raised, flat, lowered, ridged, or chiseled. The default is `Raised`, but if you write

```
(Frame Lowered ...)
```

or, equivalently,

```
(Frame (Lowered TRUE) ...)
```

then that has the effect of declaring `Raised`, `Flat`, etc., to be false.

Bar Leaf

Displays a horizontal or vertical line, using the foreground color, with the specified size and stretchability in the principal direction of its parent. `Bar` is exactly like `Glue` except that it uses the foreground color (`Color`) instead of the background color (`BgColor`).

`Main`	(Size, `1 + 0 - 0`) The size and stretchability in the principal direction of its parent.

Shape The principal direction is explicitly specified; the other direction has zero preferred and minimum size and is infinitely stretchable, thereby taking on the parent's shape.

Notes Must be a child of a horizontal or vertical split (`HBox`, `HVTile`, `VBox`, or `VTile`).

See Also Ridge and Chisel

Boolean

Filter

A Boolean, on-off interactor.

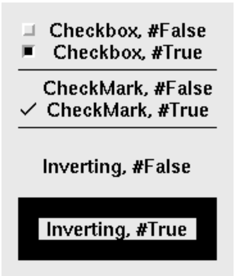
`Value` GP	(Boolean, `FALSE`) The current state.
`MenuStyle`	(Boolean, `FALSE`) When set, the interactor should be a child of a `Menu`, in which case it will react on the upclick. Otherwise, it reacts on the downclick.
	Feedback choices (mutually exclusive):
`CheckBox`	(Boolean, `TRUE`) Give feedback with a "check-box" icon.
`CheckMark`	(Boolean, `FALSE`) Give feedback with a "check-mark" icon.
`Inverting`	(Boolean, `FALSE`) Give feedback by displaying a border around the child `VBT`.

Behavior If `CheckBox` (the default) is set, FormsVBT adds a three-dimensional check-box icon to the left of its child. To indicate a false value, the check-box is raised and empty; to indicate a true value, the check-box is lowered and filled-in. Any click on the check-box or on the child toggles state and generates an event on the upclick. `CheckMark` causes a different set of bitmaps to be used to indicate state. `Inverting` causes no bitmaps to be used. Actually, "inverting" is a (historical) misnomer: on a non-monochrome display, a three-dimensional shadow is put around the child, and the shadow is raised (when false) and lowered (when true).

Shape When `Inverting` is false, the shape of this interactor is the shape of its child, plus 16 pixels wider on the west side. When `Inverting` is true, the shape of the interactor is the shape of the child plus the shadow.

Notes The `CheckMark` property, in conjunction with `MenuStyle`, can be used to implement the Macintosh-style "checks" on menu items. However, because the check-mark is put to the left of its child, menu elements will look misaligned if some elements are `MButtons` and others are `Booleans` with check-marks.

See Also Choice and Radio


Checkbox, #False
Checkbox, #True
CheckMark, #False
CheckMark, #True
Inverting, #False
Inverting, #True

Border

Filter

Displays a border around its child.

`Pen`	(Real, 1.0) The thickness of the border.
`Pattern`	(Text) The name of a pixmap-resource used for the border's texture, which defaults to `Pixmap.Solid`.

Shape The shape of its child, plus twice the value of `Pen` in each dimension.

See Also Rim and Frame

Browser — Leaf

A browser on a collection of text strings.

Contents (`Items` takes precedence):

`Items` GP — (TextList)
The contents of the browser.
Example: `(Items "red" "blue")`. When this property is accessed at runtime, the TextList is passed a single text string, with `\n` used to separate entries. `PutTextProperty` replaces the entire contents of the browser and sets the selection to `NIL`. `GetTextProperty` returns the elements in the browser, from top to bottom.

`From` — (Text)
If present, names a resource from which the initial browser contents will be taken.

Choices (`Value` takes precedence):

`Value` GP — (Integer, `-1`)
The position of the selected item. `0` means the first item; `-1` means no item is selected.

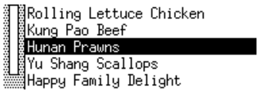
`Select` GP — (Text)
The text of the selected item. `PutTextProperty` selects the first matching item if there is one; otherwise it selects nothing. `GetTextProperty` returns the text of the selected item, or `NIL` is there is no selection.

`Quick` — (Boolean, `FALSE`)
If true, every selection action is reported as an event. Otherwise, only double-click actions are reported.

Behavior Displays items vertically, with a scrollbar at the left. Clicking selects the item under the mouse. Double-clicking on an item generates an event on the second up-click. If `Quick` is true, single-clicking on an item generates an event on the up-click.

Shape At minimum, large enough to hold its scrollbar plus the single string `"XXXX"` in the font being used. Infinitely stretchable in both dimensions.

See Also MultiBrowser


Rolling Lettuce Chicken
Kung Pao Beef
Hunan Prawns
Yu Shang Scallops
Happy Family Delight

Button

Filter

A button. Surrounds its child with a raised shadow, and generates an event when clicked.

Behavior On a down-click, the shadow becomes recessed; restores the raised shadow on an up-click or chord-abort, and generates an event if the mouse is still within the button on the up-click.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

See Also MButton

Chisel Leaf

Displays a chiseled, three-dimensional horizontal or vertical line with the specified size in the principal direction (horizontal or vertical) of its parent.

`Main`	(Real, `1.5`) The size in the principal direction.

Shape The principal direction is explicitly specified; the other direction has zero preferred and minimum size and is infinitely stretchable, thereby taking on the parent's shape.

Notes Must be a child of a horizontal or vertical split (`HBox`, `HVTile`, `VBox`, or `VTile`).

See Also Bar and Ridge

Choice — Filter

A choice button is one of a group of "radio buttons." There must be a Radio component somewhere in its ancestry. Choice components must be named.

`Value` GP	(Boolean, `FALSE`) Whether currently selected.
`MenuStyle`	(Boolean, `FALSE`) When set, the interactor should be a child of a Menu, in which case it will react on the upclick. Otherwise, it reacts on the downclick.
	Feedback choices (mutually exclusive):
`CheckBox`	(Boolean, `TRUE`) Give feedback with a "check-box" icon.
`CheckMark`	(Boolean, `FALSE`) Give feedback with a "check-mark" icon.
`Inverting`	(Boolean, `FALSE`) Give feedback by drawing a border around the child `VBT`.

Behavior If `CheckBox` is `TRUE`, Choice adds a three-dimensional diamond to the left of its child. The diamond is raised and empty for false, lowered and filled-in for true. Any click on the diamond or on the child selects this "button," unselects any other member of the group that might have been selected, and generates an event. MenuStyle causes different reaction to the mouse clicks, as described above.

Shape The shape of its child, plus 16 pixels wider on the west side when not Inverting. When Inverting, the shape of the child plus the border.

See Also Boolean and Radio

CloseButton Filter

A button that closes a subwindow when clicked. Its target may be specified using the For property; otherwise, it is the nearest subwindow ancestor of the CloseButton itself.

`For` (Symbol)
If given, this names the target. The named component must be either an overlapping (non-background) child of a `ZSplit`, or a descendant of something that is. In the latter case, the actual target will be the `ZSplit` child, not the named descendant.

Behavior Like a Button, but before generating an event, closes its target (if the target is not already closed).

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

DirMenu Leaf

A “directory menu” is a button connected to a FileBrowser. The button displays the name of the last (nearest) component of the FileBrowser’s current directory.

`For`	(Symbol) The name of a FileBrowser. This property is required.

Behavior The directory menu shows the FileBrowser’s parent-directories, one parent per line, down to first component of the path, which is typically the root directory. Selecting an item in this menu makes that parent-directory be the FileBrowser’s current directory.

Shape The shape of a TextVBT containing the name of the parent, plus the shadow.

Notes A DirMenu never generates events in its own name. It can be accessed in its own name, but this is not recommended.

In a typical ”Open File...” dialog, the DirMenu is above the FileBrowser, and the Helper is below. See Figure A

See Also FileBrowser and Helper

FileBrowser — Leaf

A FileBrowser is used for examining directories and selecting files.

`Value` GP — (Text, `"."`)
On retrieval, the value is the full pathname of the selected file, or `NIL` if no file is selected. When specified, the value may be absolute or relative (to the currently displayed directory), and may name a file or a directory. If a directory is specified, that directory is displayed but no file is selected, hence the new retrievable value is `NIL`. An initial specification, if relative, is relative to the current working directory of the application. Thus, the default initial value of "." displays the current working directory.

`Suffixes` — (TextList)
If this property is specified, the file-browser will show only those files whose suffixes are in this list. The strings should not include the period; to include files that have no suffix, use the empty string.

Example: `(Suffixes "i3" "m3" "" "fv")`

(Directories are always shown.)

`ReadOnly` — (Boolean, `FALSE`)
If true, the browser will only accept selection or naming of files that already exist. Otherwise, user may name a new file by typing in the helper (see Helper).

CONTINUED...

Behavior The FileBrowser displays a list of the files in the current directory, in alphabetic order. The user can *select* a file by clicking on its name, and *activate* it by double-clicking. Auto-scrolling works as for Browser. In the default setup, only activating a file generates an event. Activating a directory makes that the current directory, and changes the display to show it.

If a helper is present (see Helper), it displays the pathname of the current directory whenever the current directory is changed. The user may also type a filename in the helper, and press Return to activate it.

There are three "states" of selection in a FileBrowser with a Helper. After a new directory has just been set, there is no selection; the value is `NIL`. When the user clicks on an item in the browser, the browser has the selection, shown by a highlighted item; when the user types any character in the helper, the helper has the selection; the highlight vanishes from the browser. Thus it is possible to give a name for a file that does not yet exist, provided that ReadOnly is false. A relative pathname typed in the Helper is relative to the current directory.

We recommend using `(LabelFont "fixed")` for a filebrowser.

Shape At minimum, large enough to hold its scrollbar plus the single string `"XXXX"` in the font being used, plus the shadow. Infinitely stretchable in both dimensions.

Notes A Helper never generates events in its own name. It can be accessed in its own name, but this is not recommended.

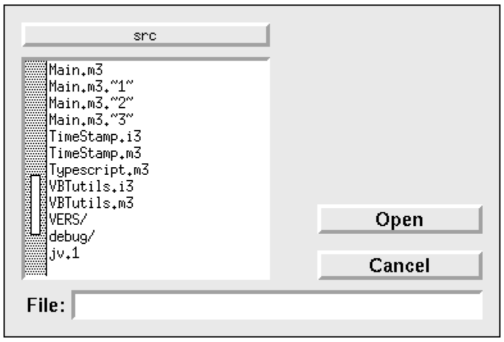
OS-related errors can occur: nonexistent directory in path, current directory became inaccessible, protection violation, etc. The default reaction to these errors is just to refuse to activate anything. In such a state, `GetText` will return `NIL`. The user can get out of this state by typing an absolute pathname of a directory that is known to exist. The underlying `FileBrowserVBT` interface has a mechanism (the `error` method) whereby the client can be notified of such errors, to report them to the user appropriately.

See Also Helper and DirMenu

CONTINUED...

It is common (and recommended) practice to combine a FileBrowser, a DirMenu, and a Helper, with activation and cancellation buttons, in an arrangment like the following, which could be used in an “Open...” dialog.

```
(ZChassis
  (VBox
    (HBox
      (Shape (Width 150) (Height 150)
        (VBox (LabelFont "fixed")
          (DirMenu (For fb))
          (Glue 6)
          (Frame Lowered (BgColor "White")
             (FileBrowser %fb ReadOnly))))
      Fill
      (Shape (Width 100)
       (VBox Fill
          (Button %open "Open")
          (Glue 10)
          (CloseButton "Cancel"))))
    (HBox
      (Shape (Width 30) (Height 16) "File:")
      (Frame Lowered (BgColor "White")
        (Helper (For fb) (Font "fixed"))))))
```



Fill Leaf

This is used for spacing other objects. Fill uses the background color, and it is essentially a shorthand for `(Glue 0 + Inf)`.

Shape Both dimensions have zero preferred and minimum size and are infinitely stretchable. Thus, the non-principal direction takes on the same shape as its parent.

Notes Must be a child of a horizontal or vertical split (`HBox`, `HVTile`, `VBox`, or `VTile`).

See Also Glue

Filter

Filter

Overlays its child with reactivity.

`Cursor`	(Text) Names the cursor that will be displayed when the mouse is over the child. The default is cursor is defined by the Trestle implementation.
	Reactivity choices (mutually exclusive):
`Active`	(Boolean, `TRUE`) When true, mouse and keyboard events are relayed to child. This is the normal case.
`Passive`	(Boolean, `FALSE`) When true, doesn't allow mouse or keyboard events to go to the child; in addition, the cursor is changed to `Cursor.NotReady`, a watch-face.
`Dormant`	(Boolean, `FALSE`) When true, doesn't send mouse or keyboard events to the child; it also draws a grey screen over the child.
`Vanish`	(Boolean, `FALSE`) When true, doesn't send mouse or keyboard events to the child; in addition, it draws over the child in the background color thereby making it invisible.

Shape The shape of its child.

Notes Of the four state properties, exactly one can be in effect at any instant. If more than one is specified, they are considered in the order `Vanish`, `Dormant`, `Passive`, and `Active` to find the first one that is true. If all are false (including `Active`, which defaults to true), an error is raised.

To test the reactivity of a Filter, you can call one of the following procedures in the FormsVBT interface: `IsActive`, `IsPassive`, `IsDormant`, or `IsVanished`. To change the reactivity or the cursor, call `MakeActive`, `MakePassive`, `MakeDormant`, or `MakeVanish`.

FormsVBT provides a mechanism for accessing the nearest Filter component above a named interactor. Thus, Filter interactors are typically left unnamed, and some named descendant is used to reference the Filter from the application program.

As mentioned, the default is cursor is defined by the Trestle implementation. Standard X screentypes support the cursors named in *X Window System* by Scheifler et. al. [9] Appendix B. Therefore, for example, `XC_arrow` returns a cursor that behaves like the X arrow cursor on X screentypes, and like the default cursor on screentypes that have no cursor named `XC_arrow`. *)

Frame — Filter

Displays a three-dimensional border around its child.

	Shadow-style choices (mutually exclusive):
`Raised` GP	(Boolean, `TRUE`)
`Flat` GP	(Boolean, `FALSE`)
`Lowered` GP	(Boolean, `FALSE`)
`Ridged` GP	(Boolean, `FALSE`)
`Chiseled` GP	(Boolean, `FALSE`)

Shape The shape of its child, plus twice the value of ShadowSize in each dimension.

See Also Border and Rim

Generic Leaf

A placeholder, intended to be taken over by the application. Should always be given a name, so the application can access it. Often has some application-defined interactive behavior. Until taken over, this has the shape and appearance of

```
(Shape (Width 0 + 0) (Height 0 + 0) "")
```

Notes To take over a Generic, use `PutGeneric`; to retrieve the VBT, use `GetGeneric`. A Generic is implemented as a Filter.T, whose child is the VBT specified using `PutGeneric`. Whenever `PutGeneric` is invoked, the size of the new VBT is propagated appropriately.

Generic should be used only when there is no comparable interactor provided by FormsVBT, or when the VBT will change dynamically. If you want to use a *subtype* of an interactor, you should override the `realize` method of the FormsVBT object; see `FormsVBT.i3`.

Glue — Leaf

A piece of filler for spacing other objects. Glue displays using the background color, BgColor. (To use the foreground color, use a Bar component.) Unlike Fill, Glue has specified size and no stretchability in the principal direction (horizontal or vertical) of its parent.

`Main`	(Size, `1 + 0 - 0`) The size and stretchability in the principal direction of its parent.

Shape The principal direction is explicitly specified; the other direction has zero preferred and minimum size and is infinitely stretchable, thereby taking on the parent's shape.

Notes Must be a child of a horizontal or vertical split (`HBox`, `HVTile`, `VBox`, or `VTile`).

See Also Fill

Guard

Filter

A filter that covers its child with a striped shadow. On the first click, the guard is removed, exposing and activating the child. If the mouse moves out of the guarded area, the guard returns.

Behavior On down-click, the shadow becomes recessed; restores the raised shadow on up-click or chord-abort, and generates an event if the mouse is still within the button on the up-click.

Shape The shape of its child plus the shadow.

Notes A guard is often used around a component whose action has has potentially serious side-effects; e.g., `(Button "Delete")`, `(Boolean "Override")`.

Help Split

A “helper bubble.” The first child, the “anchor,” is displayed as if Help were not present. The second child, the “bubble,” is popped up when the mouse is over the anchor for a “sufficiently long time” (the amount of time is implementation-specific), and the bubble remains displayed as long as the mouse is in over the anchor.

Shape The shape of the anchor child.

Notes The bubble appears in the southwest corner of the anchor. It would probably be better were it displayed closer to where the mouse first entered the anchor.

The following macro creates a helper bubble that contains some text on a yellow background.

```
(Macro TextBubble BOA (child text)
  `(Help ,child
    (Border (Pen 1) (Color "Black") (BgColor "LightYellow")
      (Rim (Pen 4)
        (Text ,text)))))
```

Helper

Leaf

A type-in field connected to a FileBrowser. A Helper is used for typing filenames, either to select a new file or to switch to another directory.

`For`	(Symbol) The name of a FileBrowser. This property is required.
`FirstFocus`	(Boolean, `FALSE`) If this Helper is in a subwindow or TSplit-child, then when that component appears, the keyboard focus will go to this Helper, and its typein field will be selected in replace-mode. See the note about the TypeIn's FirstFocus property.
`TabTo`	(Symbol) If given, this is the name of the component to which the keyboard focus wil be transferred when the user types Tab.
`ExpandOnDemand`	(Boolean, `FALSE`) If true, the text area will grow and shrink vertically, as required, to contain the entire text.

Behavior The helper displays the pathname of the FileBrowser's current directory. The user can also type a name in the Helper; typing Return will then activate that file in the FileBrowser.

In a non-ReadOnly FileBrowser, a Helper is necessary in order to specify a file that does not yet exist (e.g., that the application should create), as in a "Save As ..." dialog.

We recommend using `(Font "fixed")` in the helper.

Shape The width is zero with infinite stretchability, and the height is as high as one line in the current font.

Notes A Helper never generates events in its own name. It can be accessed in its own name, but this is not recommended.

See Also DirMenu and FileBrowser

HBox Split

Organizes its children horizontally, in order from left to right. If it is wider than the sum of its children's widths, the excess is distributed equally among all stretchable children, as far as they will stretch. If all stretchability is exceeded, the excess will be given to the last child. If it is narrower than the sum of its children's widths, it clips on the right, perhaps making some children entirely invisible. All children have height equal to the height of the HBox. Section 3.6 explains the layout model in detail.

Shape The width is the sum of its children's widths; the height is the maximum of the children's heights, but is stretchable only if all of them are.

See Also VBox

HPackSplit — Split

Organizes its children like words of text in a ragged-right paragraph. The children in any given row have their north boundaries aligned, and all children that are first in their row have their west borders equal to the west border of the parent. A child is horizontally clipped only if its requested horizontal size exceeds the parent's horizontal size; in this case the child will be alone in its row.

`HGap`	(Real, `2.0`) This many points separate each child horizontally.
`VGap`	(Real, `2.0`) This many points separate each row of children.
`Background`	(Text, `"White"`) Inter-children (`HGap`), inter-row (`VGap`), and end-of-line spaces are displayed in this texture.

Shape The shape is unconstrained in the principal axis and fixed in the other axis.

See Also VPackSplit

HTile Split

Organizes its children horizontally, in order from left to right, with an adjusting bar between each child. The adjusting bar allows the user to move the boundary between the children, subject to the size range allowed by each child.

Shape Same as HBox.

See Also VTile

Insert — Leaf

Insert is not a component at all. It is a syntactic form for specifying the name of another resource whose contents are to be included at this point in the S-expression. (It is the only such form in the language.)

`Main`	(Text) The text argument is the name of a resource containing one or more S-expressions.

Behavior Insert is convenient for breaking a large FormsVBT description into several files. Typically, such files contain ZChild, ZChassis, and Macro forms.

LinkButton Filter

A button that provides "random access" to a child of a TSplit. Switches a TSplit to display the specified child.

`For`	(Symbol) The target, which must be specified. The named component must be either a TSplit child, or a descendant of something that is. In the latter case the TSplit child is the true target.

Behavior Like a Button, but before generating an event, switches a TSplit to display a specific child.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

Notes The LinkButton need not be a descendant of the TSplit it controls. You may specify more than one LinkButton for the same TSplit.

See Also LinkMButton, PageButton, and TSplit

LinkMButton Filter

This is the "menu-style" equivalent of LinkButton; it provides "random access" to a child of a TSplit.

`For` (Symbol)
The target, which must be specified. The named component must be either a TSplit child, or a descendant of something that is. In the latter case the TSplit child is the true target.

Behavior Like an MButton, but before generating an event, it switches a TSplit to display a specific child.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

See Also LinkButton, PageMButton, and TSplit

MBUTTON

Filter

A “menu button” is usually a member of a Menu. It can be used anywhere, but its behavior will seem weird in almost any other context.

Behavior Its child is surrounded by a flat shadow that is recessed whenever the mouse rolls into it. Restores the flat shadow on up-click or chord-abort, and generates an event if the mouse is still within the button on the up-click.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

See Also Button

Menu

Split

A pull-down menu. The first child, the “anchor”, looks just like a button, and when it is clicked, it pops up the second child. The second child can be any component, and is surrounded by a raised shadow. All other children are ignored. A menu requires a ZSplit somewhere in its ancestry; FormsVBT provides one of these by default.

`NotInTrestle` (Boolean, `FALSE`)
If true, the menu is installed into a local ZSplit, rather than directly into Trestle, when popped up. Menus installed in Trestle may exceed the size of the containing window.

Behavior Pops up the second child when the first child is clicked, and keeps it there until the mouse button is released. Typically, the the second child contains a VBox of menu buttons (MButton, PopMButton, PageMButton, LinkMButton).

An event is generated just after the anchor button is activated, and before the second child of Menu is popped up. This is useful for clients whose menu contents are changing dynamically and should be made consistent only when the menu is about to be displayed.

Shape The shape of its first child plus its shadow.

MultiBrowser

Leaf

A browser on a collection of text strings that allows multiple items to be selected.

Contents (`Items` takes precedence):

`Items` (TextList)
The contents of the browser.
Example: `(Items "a" "b" "c")`

`From` (Text)
If present, this names a resource from which the initial browser contents will be taken, one item per line.

Initial choices (`Value` takes precedence):

`Value` (CardinalList)
The positions of selected items.
Example: `(Value 1 3 5 2).`

`Select` (TextList)
The list of initially selected items.
Example: `(Items "c" "a")`

`Quick` (Boolean, `FALSE`)
If true, every selection action is reported as an event. Otherwise, only double-click actions are reported.

Behavior Displays items vertically, with a scrollbar at the left. The left button modifies the selection: If the item under the cursor is not currently selected, it becomes selected; if it is currently selected, it is deselected. Dragging sets the state of the additional items to the state it gave to the first item. Middle and right buttons clear any existing selection, and select the item under the cursor. Dragging selects additional items as the mouse passes over them; retreating unselects items. Autoscrolling is implemented, and it continues to select or unselect items as they scroll by.

When `Quick` is true, every selection action also generates an event, on the up-click. Otherwise, an event is not generated until the second up-click of a double-click.

Shape At minimum, large enough to hold its scrollbar plus the single string `"XXXX"` in the font being used, plus a shadow. Infinitely stretchable in both dimensions.

See Also Browser

Numeric Leaf

An interactor for integer values. Numeric has an editable displayed number, as well as "increment" and "decrement" buttons.

`Value` GP (Integer)
The currently displayed number.

`AllowEmpty` (Boolean, `FALSE`)
If true, the component supports a distinct "empty" state. See Notes.

`Min` GP (Integer, `FIRST(INTEGER)`)
The minimum allowed value.

`Max` GP (Integer, `LAST(INTEGER)`)
The minimum allowed value.

`HideButtons` (Boolean, `FALSE`)
If true, the numeric interactor appears without increment and decrement buttons.

`TabTo` (Symbol)
If given, this is the name of the component to which the keyboard focus wil be transferred when the user types Tab.

`FirstFocus` (Boolean, `FALSE`)
If this Numeric is in a subwindow or TSplit-child, then when that component appears, the keyboard focus will go to this Numeric, and its number field will be selected in replace-mode. See the note about the TypeIn's FirstFocus property.

Behavior The increment button (+) increments the number, whereas the decrement button (−) decrements it, up to the respective limits. The number field is editable as a single-line TypeIn. Typing Return in the number field checks the number; if it is out of range, it is forced to the nearest acceptable value. Increment, decrement, and Return all generate an event.

Shape The shape depends on the ShadowSize and Font in effect. When the default shadow and font are used, the size of a Numeric is 76 by 19 pixels.

Notes `GetInteger` can be used to retrieve the current value, and `PutInteger` can be used to set it.

When AllowEmpty is true, emptiness is a special, out-of-band state for the interactor. In this state the increment and decrement functions are disabled. The value of an empty Numeric is reported as `FIRST(INTEGER)`.

Emptiness can be tested explicitly by `NumericVBT.IsEmpty`, and can be set by `NumericVBT.SetEmpty`.

PageButton

Filter

A button that switches to the next or previous child of a TSplit. Its target is a TSplit, which may be specified by the `For` property; otherwise the target is the nearest TSplit ancestor of the PageButton itself.

`For`	(Symbol) If given, names the target, which must be a TSplit.
`Back`	(Boolean, `FALSE`) If true, the button advances backward among children of the TSplit, otherwise it advances forward.

Behavior PageButton is like a Button, but before generating an event, it switches the children of a TSplit. However, if the TSplit is not Circular, and if the last (first) child is already being displayed, nothing happens and no event is generated.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

Notes If a PageButton is a descendant of the TSplit it controls, it will naturally vanish when activated, due to the nature of TSplits.

You may specify more than one PageButton for the same TSplit.

See Also LinkButton, PageMButton, and TSplit

PageMButton

Filter

This is the “menu-style” equivalent of PageButton. It switches to the next or previous child of a TSplit. Its target is a TSplit, which may be specified by the `For` property; otherwise the target the nearest TSplit ancestor of the PageMButton itself.

`For`	(Symbol) If given, names the target, which must be a TSplit.
`Back`	(Boolean, `FALSE`) If true, the button advances backward among children of the TSplit, otherwise it advances forward.

Behavior PageMButton is like an MButton, but before generating an event, it switches the children of a TSplit. However, if the TSplit is not Circular, and if the last (first) child is already being displayed, nothing happens and no event is generated.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

See Also LinkMButton, PageButton, and TSplit

Pixmap Leaf

Displays a pixmap, centered in the space available. The image in the pixmap is a resource in "pnm" format, representing one of three types of images: "pbm" is a 1-bit-deep (bitmap) image; "pgm" is a greyscale image; "ppm" is a color image.

`Main` P (Text)
The name of a resource containing the image for a pixmap. The image must be in "pnm" format.

`Accurate` (Boolean, `FALSE`)
If the image is is greyscale or color (pgm or ppm), this property determines how each RGB value in the pixmap should be displayed on a color-mapped display.

`NeedsGamma` (Boolean, `FALSE`)
If the image is is greyscale or color (pgm or ppm), this property indicates whether to let Trestle gamma-correct the colors.

Notes The current foreground and background colors are used for bitmap (pbm) images. In all three formats, if the image is smaller than the space available, the current background color is used for the surrounding space.

Shape The shape of the pixmap, with infinite stretchability in both dimensions.

See Also Texture

PopButton

Filter

A button that pops up an overlapping subwindow when clicked. Its target must be specified with the For property.

`For` (Symbol)
If given, this names the target. The named component must be either an overlapping (non-background) child of a `ZSplit`, or a descendant of something that is. In the latter case, the actual target will be the `ZSplit` child, not the named descendant.

Behavior Like a Button, but before generating an event, it pops up its target (or brings the target to the top of it sibling, overlapping windows, if it is already visible).

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

See Also PopMButton

$\boxed{\textbf{PopMButton}}$ Filter

This is the “menu-style” equivalent of PopButton.

`For`	(Symbol) If given, this names the target. The named component must be either an overlapping (non-background) child of a `ZSplit`, or a descendant of something that is. In the latter case, the actual target will be the `ZSplit` child, not the named descendant.

Behavior Like an MButton, but before generating an event, it pops up its target (or brings the target to the top of it sibling, overlapping windows, if it is already visible).

See Also PopButton

Radio

Filter

Unites those descendants that are Choice components into "radio buttons." Within a Radio, at most one Choice component is selected at any given time.

`Value`	(Symbol, `""`) The name of the Choice element currently selected. Note that the application can clear the selection, so that no member of the group is selected, but the user cannot. The current selection can be accessed via `FormsVBT.GetChoice` and `FormsVBT.PutChoice`.

Behavior An event is generated whenever the user changes which element of the group is selected. If the Choice element has an attached event procedure, it will be called; otherwise, the event falls through to the Radio group, which may also have an attached procedure. Thus an event can occur in the name of the Radio group, though the group itself has no interactive behavior.

Shape The shape of its child.

See Also Boolean and Choice

Ridge Leaf

Displays a ridged, three-dimensional divider bar with the specified size in the principal direction (horizontal or vertical) of its parent.

`Main`	(Real, `1.5`) The size in the principal direction.

Shape The principal direction is explicitly specified; the other direction has zero preferred and minimum size and is infinitely stretchable, thereby taking on the parent's shape.

Notes Must be a child of a horizontal or vertical split (`HBox`, `HVTile`, `VBox`, or `VTile`).

See Also Bar and Chisel

Rim — Filter

Displays a rim around its child. A rim is like a border, but uses the background color rather than the foreground color.

`Pen`	(Real, `1.0`) The thickness of the rim.
`Texture`	(Text) The name of a pixmap-resource used for the border's texture, which defaults to `Pixmap.Solid`.

Shape The shape of its child, plus twice the value of Pen in each dimension.

See Also Border and Frame

Scale

Filter

Scale provides a filter that changes the *resolution*, not the size, of its child. Both graphics and fonts are scaled.

`HScale` GP (Real, `1.0`)
The horizontal scaling factor.

`VScale` GP (Real, `1.0`)
The vertical scaling factor.

`Auto` (Boolean, `FALSE`)
Dynamically set the scaling such that the child's natural size always fills its domain.

`AutoFixed` (Boolean, `FALSE`)
Like Auto, but always use set the horizontal and vertical scaling factors to the same number.

Shape The shape of the child.

Notes Auto takes precedence over AutoFixed, which takes precedence over HScale or VScale.

There are two ways you can use a "Scale" component: With the HScale and VScale properties, the "Scale" allows you to explicitly set a horizontal and vertical scale factor. Alternatively, with Auto the scale factors are set so that the child's natural size always fills the screen real estate it's been given. A varient of Auto is AutoFixed: here, the child is scaled by the same amount both horizontally and vertically. The amount is chosen so that the natural size of child just fits in the larger direction given and fits fine in the other direction.

You should only retrive and modify the values of HScale and VScale if the component was created without Auto or AutoFixed.

Scale does not change the size of the child, just the size of the "pixels." Graphic elements will be scaled fairly precisely. Fonts will be scaled to the nearest available font. If you are scaling components that include text, for best results, HScale and VScale should have the same value.

If you are specifying a Font or LabelFont in a component that is going to be scaled, you should use the "long form" of the font's name in order to specify the point size; e.g.,

```
(Font
  (Family "fixed")
  (WeightName "medium")
  (Slant "r")
  (Width "normal")
  (PointSize 120))
```

Scroller

Leaf

An integer-valued scroll bar interactor. The full range of the scroll bar is gray, and the "thumb" is a white rectangular stripe somewhere within the scroll bar. The scroll bar represents the interval `[Min .. Max]`, and the thumb represents the subinterval `[Value .. (Value+Thumb)]`.

`Value` GP (Integer, `50`)
The current value; always between Min and Max-Thumb, inclusive.

`Min` GP (Integer, `0`)
The minimum value allowed.

`Max` GP (Integer, `100`)
The maximum value allowed. Value is within Min and Max-Thumb.

`Thumb` GP (Cardinal, `0`)
A non-negative number no greater than Max-Min.

`Step` GP (Cardinal, `1`)
The amount to increment or decrement Value when "continuous scrolling."

`Vertical` (Boolean, `FALSE`)
If true, the scroll bar is oriented vertically, from south (Min) to north (Max). Otherwise, the scroll bar goes from west to east.

CONTINUED…

Behavior The user can adjust the position of the thumb with the mouse. For the sake of explanation, suppose that the scroll bar is adjacent and attached (via an application program) to a column containing the numbers Min through Max. At any given time, Thumb+1 of the numbers (i.e., Value through Value+Thumb) are visible in the attached view.

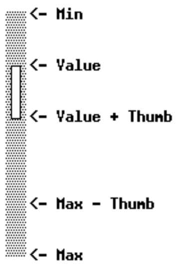
The semantics of the mouse are as follows: A left click scrolls the view towards its end by moving the number at the mouse so it becomes the first number visible in the view. A right click scrolls the view towards its beginning by bringing the first number visible in the view to the position of the mouse. A middle click scrolls the view to the mouse by bringing the top of the thumb to the position of the mouse. Holding the left or right button without moving the mouse will cause (after a short time) continuous scrolling to begin. If you then drag the mouse, any continuous scrolling is terminated and the view scrolls with the mouse.

An event is generated after each time the Value of the scroll bar is changed. That can happen after any click, while continuous scrolling is in effect, and while dragging the mouse. When continuous scrolling causes the thumb to reach its limit, the scroll bar doesn't continue to generate events, since the value is no longer changing.

It is not unreasonable for the application to modify properties of the scroll bar (the thumb, in particular) while processing an event.

Notes The scroll bar does not allow canceling.

Shape Vertical sliders have a minimum size of 13x27 pixels, with infinite vertical stretch. Horizontal sliders have a minimum size of 27x13 pixels, with infinite horizontal stretch.



Shape

Filter

The Shape filter is used for putting size-constraints on a component.

`Height`	(Size)
`Width`	(Size)

See Also Sections 3.6 and 3.6.1

Source

Filter

A Source component is a filter, used for drag-and-drop actions. It generates an event on an uncancelled upclick. A typical event-handler will then inquire the `ActiveTarget` property to determine whether the Source is over a Target and take some action if it is. For example, a source-icon representing a file might delete the file when it is dragged and released over a target-icon representing a "trash can." A tiling window manager might make every window both a Source and Target to permit windows to be exchanged.

`ActiveTarget` G — (Text) The name of the component over which the Source is located. This property is ignored when the Source is defined; it only has a value while the callback associated with the Source is active. The call to `FormsVBT.GetTextProperty` raises an `Error` exception if the Source is not over a valid Target on an uncancelled upclick or if the Target does not have a name.

Behavior If the Source is located over a Target when the mouse-button is down, the Source's `hit` method is called.

See Also Target

Stable Filter

The Stable filter is used to mask out changes to its child's preferred size.

Shape The max and min are its child's max and min size. The preferred size is the projection of its own size into the child's size range. Its own size is its current size if this is non-empty, or its last non-empty size otherwise.

Notes A Stable is part of a ZChassis. In this way, whenever the user changes the size of a subwindow, that size will take precedence over any new size preferences given by the ZChassis's child. A Stable is also inserted automatically by Trestle whenever a window is installed.

Target Filter

A Target is a filter that marks its child as a destination for a Source component.

Behavior By default, a Target component inverts its highlighting when there is a Source component overlapping it.

See Also Source

Text

Leaf

Displays a single-line text string. By default, the string is centered horizontally and vertically.

	Contents (`Main` takes precedence):
`Main` GP	(Text) The text.
`From`	(Text) The name of a resource from which the text will be taken.
	Centering choices (mutually exclusive):
`Center`	(Boolean, `TRUE`) Causes the text to be centered.
`LeftAlign`	(Boolean, `FALSE`) Causes the text to be left-aligned.
`RightAlign`	(Boolean, `FALSE`) Causes the text to be right-aligned.
`Margin`	(Real, `2.0`) Forces this many points of margin on the east and west sides of the text.

Shape The bounding rectangle of the text when rendered in the current LabelFont, plus `2 * Margin` in each axis.

Notes `Text` components use the `LabelFont` property, not the `Font` property.

TextEdit — Leaf

A multi-line, editable text with a scrollbar.

	Contents (`Value` takes precedence):
`Value`	(Text, `""`) The contents.
`From`	(Text) The name of a resource from which the text will be taken.
`ReadOnly` GP	(Boolean, `FALSE`) If true, the text will not be editable.
`Clip`	(Boolean, `FALSE`) If true, the long lines will be clipped, not wrapped.
`TurnMargin`	(Real, `2.0`) If long lines are wrapped, then a small grey bar will appear at the end of the first line and the beginning of the next to indicate that the line was wrapped. TurnMargin specifies the width of the grey bar.
`NoScrollbar`	(Boolean, `FALSE`) If true, there will be no scrollbar or thin line to the left of the text area.
`FirstFocus`	(Boolean, `FALSE`) If true, and if this TextEdit is in a subwindow or TSplit-child, then when that component appears, this TextEdit will acquire the keyboard focus. See the note about the TypeIn's FirstFocus property.
`Position` GP	(Cardinal, N/A) The position of the cursor.
`Length` GP	(Cardinal, N/A) The number of characters in the text.

Notes The Position and Length properties are unusual because they cannot be specified in the s-expression; they may only be accessed at runtime using `FormsVBT.GetIntegerProperty` and `FormsVBT.PutIntegerProperty`.

Notes For details on the editing commands, see the description of TextPort in the *VBTkit Reference Manual*[2].

Notes This form produces an object that is a subtype of `TextEditVBT.T`, which contains a `TextPort.T` and (optionally) a `TextPort.Scrollbar`. To override methods such as `filter` on the textport, the client should use the `realize` method of the `FormsVBT.T`; see Section 4.7, page 69 for an example.

See Also TypeIn and Typescript

Texture — Leaf

Displays a rectangle in some texture.

`Main`	(Texture) The name of a pixmap-resource used for the texture. The default is to use `Pixmap.Solid` for the texture.
`LocalAlign`	(Boolean, `FALSE`) If true, the texture is aligned to the northwest corner of the underlying VBT. Otherwise, it's aligned with the coordinate origin. Note that all other interactors that use textures are necessarily aligned with the coordinate origin.

Shape Zero size and infinitely stretchable in both dimensions.

See Also Pixmap

TrillButton

Filter

A button, generates an event on the down-click and continues to generate events while the mouse is held down over the button.

Behavior Highlights on down-click, and generates an event. If held long enough, generates events repeatedly until released, canceled (by chording), or moved outside the domain of the button. When moved outside the button and still held, events are suspended until the mouse is returned to the domain of the button. At that point, the button is re-highlighted and event generation is resumed. The button is unhighlighted when the button is released or canceled.

Notes The initial hold-period and the repeat-period should ultimately be governed by an application-independent user profile.

Shape The natural shape of its child (i.e., with no shrink or stretch), plus the border around the button.

TSplit Split

Organizes its children temporally: exactly one child is visible at any given time.

`Value` GP (Cardinal, 0)
Which child is currently shown. The first child is numbered 0.

`Which` (Symbol)
The name of the currently visible child. If both Value and Which are specified, they must refer to the same child.

`Circular` (Boolean, `FALSE`)
If true, makes the TSplit view its children as a circular rather than a linear list, thereby changing the behavior of PageButton.

`Flex` (Boolean, `FALSE`)
If true, the TSplit will change shape to fit the shape of the child on display at the moment. Otherwise switching the visible child never changes the TSplits's shape. A change of shape can lead to resizing that cascades throughout the entire form, so use with care.

Behavior TSplit has no direct interactive behavior, but the user can change which child is shown by using PageButton and LinkButton.

Shape If Flex is false, the natural width and height are separately computed as the maximum of the natural widths and heights of the children. If Flex is true, shape is identical to the shape of the currently displayed child.

See Also LinkButton, LinkMButtonm, PageButton, and PageMButton

TypeIn

Leaf

A single-line editable text.

Contents (`Value` takes precedence):

`Value` GP	(Text, `""`) The current text.
`From`	(Text) If present, names a resource from which the initial text will be taken.

`ReadOnly`	(Boolean, `FALSE`) If true, the text-area will not be editable.
`ExpandOnDemand`	(Boolean, `FALSE`) If true, the text area will grow and shrink vertically, as required, to contain the entire text.
`FirstFocus`	(Boolean, `FALSE`) If true, and if this TypeIn is in a subwindow or TSplit-child, then when that component appears, this TypeIn will acquire the keyboard focus, and its text will be selected in replace-mode. See the notes below.
`TabTo`	(Symbol) If given, this is the name of the component to which the keyboard focus wil be transferred when the user types Tab.
`TurnMargin`	(Real, `2.0`) If long lines are wrapped, then a small grey bar will appear at the end of the first line and the beginning of the next to indicate that the line was wrapped. TurnMargin specifies the width of the grey bar.

CONTINUED…

Behavior This is a text editor, normally used for small type-in fields.

Typing Return generates an event.

Extensive application control can be exercised by direct calls on procedures in the `TextPort` interface.

Shape The width is initially 30 times the width of the widest character in the font. It has infinite shrinkability and stretchability. It is initially one line high. If `ExpandOnDemand` is false, then it always keeps that height; otherwise, the height changes to accomodate the entire text, but never less than one line. In any case, there is no vertical stretch or shrink.

Notes For details on the editing commands, see the description of the TypeinVBT interface in the *VBTkit Reference Manual*[2].

Notes The FirstFocus property only works when the TypeIn is in a subwindow or TSplit-child. If you'd like a TypeIn in the top-level window to grab the keyboard when the mouse first enters the window, you can override the position method of the form to grab the focus the first time (and only the first time!) that the mouse enters the window. Here's how to do that.

Here's how to subclass the form:

```
TYPE
   MyForm = FormsVBT.T OBJECT
     graphFocus: BOOLEAN := TRUE;
     firstFocus: TEXT; (* name of widget to grab focus *)
   OVERRIDES
     position := MyPosition;
   END;

And here is what the position method looks like:
\begin{verbatim}
  PROCEDURE MyPosition(self: MyForm; READONLY cd: VBT.PositionRec) =
    BEGIN
      FormsVBT.T.position(self, cd);
      IF self.grabFocus THEN
        FormsVBT.TakeFocus(self, self.firstFocus, cd.time);
self.grabFocus := FALSE
      END
    END MyPosition
```

See Also TextEdit and Typescript

Typescript Leaf

This is like a TextEdit component, but the underlying VBT class provides a reader and a writer for accessing the text. The lines of text that have been read become read-only as far as the editor is concerned. It is useful for “transcripts.” With a small amount of Modula-3 code, you can connect the reader and writer to pipes that run a command interpreter. Typescripts always have scrollbars.

`ReadOnly`	(Boolean, `FALSE`) If true, the text-area will not be editable.
`Clip`	(Boolean, `FALSE`) If true, the long lines will be clipped, not wrapped.
`TurnMargin`	(Real, `2.0`) If long lines are wrapped, then a small grey bar will appear at the end of the first line and the beginning of the next to indicate that the line was wrapped. TurnMargin specifies the width of the grey bar.
`FirstFocus`	(Boolean, `FALSE`) If true, and if this component is in a subwindow or TSplit-child, then when that component appears, this component will acquire the keyboard focus, and its text will be selected in replace-mode. See the note about the TypeIn’s First-Focus property.

Notes For details on the editing commands, see the description of TextPort in the *VBTkit Reference Manual*[2]. The following code shows the Modula-3 code for accesses the underlying reader and writer:

```
WITH v = FormsVBT.GetVBT(fv, "typescript") DO
  rd := TypescriptVBT.GetRd(v);
  wr := TypescriptVBT.GetWr(v);
END;
```

See Also TextEdit and TypeIn

VBox Split

Organizes its children vertically, in order from top to bottom. If it is taller than the sum of its children's heights, the excess is distributed equally among all stretchable children, as far as they will stretch. If all stretchability is exceeded, the excess will be given to the last child. If it is shorter than the sum of its children's heights, it clips on the bottom, perhaps making some children entirely invisible. All children have width equal to the width of the VBox. Section 3.6 explains the layout model in detail.

Shape The height is the sum of its children's heights; the width is the maximum of the children's widths, but is stretchable only if all of them are.

See Also HBox

Viewport Filter

A Viewport is a filter that provides scrollbars (horizontal and/or and vertical) to let you scroll over its child-component when the child's preferred size is bigger than the Viewport's size.

`Step`	(Cardinal, 10) The number of pixels to move while auto-scrolling.
	Scrolling choices (mutually exclusive):
`HorAndVer`	(Boolean, `TRUE`) This puts a horizontal and vertical scrollbar on every view, and a "reset" button in the southwest corner that moves the northwest corner of the child to the northwest corner of the view.
`HorOnly`	(Boolean, `FALSE`) Places a horizontal scrollbar at the bottom of the Viewport, and none at the left side.
`VerOnly`	(Boolean, `FALSE`) Places a vertical scrollbar on the left side of the Viewport, and none at the bottom.

Behavior Meta-left-click in a scrollbar splits the view, and meta-right-click removes the current view.

See Also Scroller

VPackSplit — Split

Organizes its children vertically, in order from top to bottom, like paragraphs in a newspaper. The children in any given column have their west boundaries aligned, and all children that are first in their column have their north borders equal to the north border of the parent. A child is vertically clipped only if its requested vertical size exceeds the parent's vertical size; in this case the child will be alone in its column.

`HGap`	(Real, 2.0) This many points separate each column of children.
`VGap`	(Real, 2.0) This many points separate each child vertically.
`Background`	(Text, "White") Inter-children (HGap), inter-row (VGap), and end-of-line spaces are displayed in this texture.

Shape The shape is unconstrained in the vertical axis.

See Also HPackSplit

VTile Split

Organizes its children vertically, in order from top to bottom, with an adjusting bar between each child. The adjusting bar allows the user to move the boundary between the children, subject to the size range allowed by each child.

Shape The height is the sum of its children's heights; the width is the maximum of the children's widths, but is stretchable only if all of them are.

See Also HTile

ZBackground Filter

ZBackground is a filter that should be put around the first (background) child of a ZSplit. This filter will clip highlighting that takes place within the background child from the other children of the ZSplit.

See Also ZSplit, ZChassis, and ZChild

ZChassis Filter

A handy setup for a standard, titled, draggable subwindow (a non-background child of a ZSplit). The top of the subwindow contains a banner with a *close button*, a *title* that can be used to drag the window, and a *grow button*.

`At` (List, `0.5 0.5`)
This determines the initial position of the subwindow. See ZChild.

`Open` (Boolean, `FALSE`)
If true, the subwindow is initially visible. See ZChild.

`Title` (Sx, `(Text "<Unnamed>")`)
This is the text inside the draggable part of the title bar.

`NoClose` (Boolean, `FALSE`)
If true, the close button is omitted.

Shape Shape of its child plus borders, frames, and the title bar.

See Also ZSplit and ZChild

Except for some details of the feedback and the handling of keywords, ZChassis could be defined by a macro:

```
(Macro ZChassis ((Open   FALSE)
                 (At     (0.5 0.5))
                 (Title  "<Untitled>")
                 child)
  `(ZChild
     (Open ,Open)
     (At ,@At)
     (Stable
       (Border
         (VBox
           (HBox
             (CloseButton "C")
             Bar
             (Shape (Width + Inf) (ZMove ,Title))
             Bar
             (ZGrow "G"))
           Bar
           (Frame ,child))))))
```

ZChild — Filter

A hook on which to hang various properties (such as At and Name) that control the behavior of an overlapping subwindow.

`At`	(At, `0.5 0.5`) Position, as discussed in Section 3.7.
`Open`	(Boolean, `FALSE`) If true, the subwindow is initially visible. Otherwise, it is invisible until opened by user or program action (typically via PopButton or PopMButton). Open subwindows are useful for achieving layouts that cannot be achieved by a hierarchy of HBoxes and VBoxes.

Notes In practice, most overlapping subwindows begin with a ZChild with properties. ZChild itself has no interactive behavior or appearance, but because it is a HighlightVBT, it ensures that Buttons and other descendants that also highlight by HighlightVBT don't visually interfere with other overlapping subwindows.

See Also ZSplit and ZChassis

$\boxed{\textbf{ZGrow}}$ Filter

This button has the side effect of reshaping its nearest subwindow-ancestor. A ZGrow button is therefore useful in a ZChild.

See Also ZChassis and ZSplit

ZMove

Filter

A button that has the side effect of repositioning its nearest ancestor that's a non-background child of a ZSplit. A ZMove button is therefore useful in a ZChild (see ZChassis).

See Also ZChassis and ZSplit

ZSplit

Split

Organizes its children as overlapping subwindows. The first child is special: it is the background, and it underlies all overlapping windows and defines the shape of the ZSplit itself. The background child should be enclosed in a ZBackground form. It's recommended that non-background children of ZSplit be either ZChild or ZChassis, since those interactors support position control, non-interfering highlighting, and so on.

Shape The shape of its background child.

See Also CloseButton, ZChassis, ZChild, ZMove, and ZGrow

B. Miscellaneous Interfaces

This appendix describes interfaces that are of interest to FormsVBT programmers.

The `ColorName` interface describes the names that are permitted in FormsVBT color-expressions, e.g., `(BgColor "VividTomato")`. While these names are useful to all FormsVBT programmers, the procedures in the interface will be useful primarily to VBTkit programmers.

The `XTrestle` interface provides utility procedures for handling the display and geometry command-line parameters that use the X server.

You should use use `XParam` interface instead of `XTrestle` if your application installs more than one top-level window.

The `FVTypes` interface provides the type definitions of the VBT classes implementing each FormsVBT component. Clients that override the `realize` method will need to access this interface.

The `Rsrc` interface describes how constant data (texts, pixmaps, form-descriptions, etc.) can be combined with a program.

B.1 The ColorName Interface

The `ColorName` interface provides a standard mapping between color names and linear RGB triples. The implementation recognizes the following names, based on those found in `/usr/lib/X11/rgb.txt`:

AliceBlue	ForestGreen	MintCream	SandyBrown
AntiqueWhite †	Gainsboro	MistyRose †	SeaGreen †
Aquamarine †	GhostWhite	Moccasin	Seashell †
Azure †	Gold †	NavajoWhite †	Sienna †
Beige	Goldenrod †	Navy	SkyBlue †
Bisque	GoldenrodYellow	NavyBlue	SlateBlue †
Black	Gray ‡	OldLace	SlateGray †
BlanchedAlmond	Green †	OliveDrab †	SlateGrey
Blue †	GreenYellow	OliveGreen †	Snow †
BlueViolet	Grey ‡	Orange †	SpringGreen†
Brown †	Honeydew †	OrangeRed †	SteelBlue †
Burlywood †	HotPink †	Orchid †	Tan †
CadetBlue †	IndianRed †	PapayaWhip	Thistle †
Chartreuse †	Ivory †	PeachPuff †	Tomato †
Chocolate †	Khaki †	Peru	Turquoise †
Coral †	Lavender	Pink †	Violet
CornflowerBlue	LavenderBlush †	Plum †	VioletRed †
Cornsilk †	LawnGreen	Powderblue	Wheat †
Cyan †	LemonChiffon †	Purple †	White
DeepPink †	LimeGreen	Red †	WhiteSmoke
DeepSkyBlue †	Linen	RosyBrown†	Yellow †
DodgerBlue †	Magenta †	Royalblue†	YellowGreen
Firebrick †	Maroon †	SaddleBrown	
FloralWhite	MidnightBlue	Salmon †	

The dagger (†) indicates that the implementation recognizes a name along with the suffixes `1–4`; e.g., `Red`, `Red1`, `Red2`, `Red3`, and `Red4`.

The double dagger (‡) indicates that the implementation also recognizes the names with the suffixes `0` through `100`. That is, `Gray0`, `Gray1`, ..., `Gray100`, as well as `Grey0`, `Grey1`, ..., `Grey100`.

In addition, the name of a color C from this list can be prefixed by one or more of the following modifiers:

Term	*Meaning*
`Light` `Pale`	1/3 of the way from C to white
`Dark` `Dim`	1/3 of the way from C to black
`Drab` `Weak` `Dull`	1/3 of the way from C to the gray with the same brightness as C
`Vivid` `Strong` `Bright`	1/3 of the way from C to the purest color with the same hue as C
`Reddish`	1/3 of the way from C to red
`Greenish`	1/3 of the way from C to green
`Bluish`	1/3 of the way from C to blue
`Yellowish`	1/3 of the way from C to yellow

Each of these modifiers can be modified in turn by the following prefixes, which replace "1/3 of the way" by the indicated fraction:

Term	*Degree*	*% (approx.)*
`VeryVerySlightly`	1/16 of the way	6%
`VerySlightly`	1/8 of the way	13%
`Slightly`	1/4 of the way	25%
`Somewhat`	3/8 of the way	38%
`Rather`	1/2 of the way	50%
`Quite`	5/8 of the way	63%
`Very`	3/4 of the way	75%
`VeryVery`	7/8 of the way	88%
`VeryVeryVery`	15/16 of the way	94%

The modifier `Medium` is also recognized as a shorthand for `SlightlyDark`. (But you cannot use `VeryMedium`.)

```
INTERFACE ColorName;

IMPORT Color, TextList;

EXCEPTION NotFound;

PROCEDURE ToRGB (name: TEXT): Color.T RAISES {NotFound};
```

Give the `RGB.T` *value described by* `name`*, ignoring case and whitespace. A cache of unnormalized names is maintained, so this procedure should be pretty fast for repeated lookups of the same name.*

```
PROCEDURE NameList (): TextList.T;
```

Return a list of all the "basic" (unmodified) color names known to this module, as lower-case `TEXT`*s, in alphabetical order.*

```
END ColorName.
```

B.2 The XTrestle Interface

`XTrestle` checks for X-style "`-display`" and "`-geometry`" command-line switches and installs a top-level window accordingly. If your application install more than one top-level window, you may find the routines in the `XParam` interface helpful.

```
INTERFACE XTrestle;

IMPORT TrestleComm, VBT;

EXCEPTION Error;

PROCEDURE Install (v          : VBT.T;
                   applName   : TEXT    := NIL;
                   inst       : TEXT    := NIL;
                   windowTitle: TEXT    := NIL;
                   iconTitle  : TEXT    := NIL;
                   bgColorR   : REAL    := -1.0;
                   bgColorG   : REAL    := -1.0;
                   bgColorB   : REAL    := -1.0;
                   iconWindow : VBT.T   := NIL   )
  RAISES {TrestleComm.Failure, Error};
<* LL.sup = VBT.mu *>
```

This is like `Trestle.Install` *except that the locking level is different and the command line is parsed for X-style* `-display` *and* `-geometry` *options.*

```
END XTrestle.
```

The syntax of these switches is described in the X manpage and in *The X Window System* [9].

If there is a `-display` argument, it will be made the default Trestle connection for those procedures in the `Trestle` interface that take a `Trestle.T` as a parameter.

The `TrestleComm.Failure` exception is raised if a call to `Trestle` raises that exception. The `Error` exception is raised if the parameter following `-display` or `-geometry` contains any syntax errors (or is missing).

B.3 The XParam Interface

The `XParam` interface provides utilities for handling X-style `-display` and `-geometry` command-line arguments. If your application installs a single top-level window, the `XTrestle` interface may be more appropriate than this interface.

```
INTERFACE XParam;

IMPORT Point, Rect, Trestle, TrestleComm;
```

Here are routines for manipulating the `-display` argument:

```
TYPE
  Display = RECORD
              hostname: TEXT     := "";
              display : CARDINAL := 0;
              screen  : CARDINAL := 0;
              DECnet  : BOOLEAN  := FALSE
            END;

PROCEDURE ParseDisplay (spec: TEXT): Display RAISES {Error};
<* LL = arbitrary *>
```

Return a parsed version of the `-display` *argument in* `spec`.

For example, if `spec` contains the string `myrtle.pa.dec.com:0.2`, the record returned would be

```
Display{hostname := "myrtle.pa.dec.com",
        display := 0, screen := 2, DECnet := FALSE}
```

```
PROCEDURE UnparseDisplay (READONLY d: Display): TEXT;
<* LL = arbitrary *>
```

Return the text-version of the `-display` *argument* `d`.

Here are routines for manipulating the `-geometry` argument:

```
CONST Missing = Point.T{-1, -1};

TYPE
  Geometry =
    RECORD
      vertex := Rect.Vertex.NW;  (* corner for displacement *)
      dp     := Point.Origin;    (* displacement *)
      size   := Missing;         (* width, height *)
    END;
```

```
PROCEDURE ParseGeometry (spec: TEXT): Geometry RAISES {Error};
<* LL = arbitrary *>
```

Return a parsed version of the `-geometry` *argument in* `spec`.

For example, if `spec` contains the string `1024x800-0-10`, the returned record would be

```
Geometry {Rect.Vertex.SE,
          Point.T {0, 10},
          Point.T {1024, 800}}
```

The `size` field defaults to `Missing`. The horizontal and vertical displacements default to `Point.Origin` (no displacement). The displacements are always positive values; use the `vertex` field to find out from which corner they are to be offset.

```
PROCEDURE UnparseGeometry (READONLY g: Geometry): TEXT;
<* LL = arbitrary *>
```

Return the text-version of the `-geometry` *argument* `g`.

```
PROCEDURE Position (          trsl: Trestle.T;
                              id  : Trestle.ScreenID;
                     READONLY g   : Geometry          ): Point.T
  RAISES {TrestleComm.Failure};
<* LL.sup = VBT.mu *>
```

Return the position specified by `g` *in the screen coordinates for the screenID* `id` *on the window system connected to* `trsl` *(cf.* `Trestle.GetScreens`*). The value of* `g.size` *must not be* `Missing`*, unless* `g.vertex` *is the northwest corner.*

Here is the definition of the `Error` exception:

```
TYPE
  Info = OBJECT
           spec : TEXT;
           index: CARDINAL
         END;
  GeometryInfo = Info BRANDED OBJECT END;
  DisplayInfo  = Info BRANDED OBJECT END;
```

```
EXCEPTION Error(Info);
```

Parsing errors are reported with the text (`spec`*) and position (*`index`*) of the first illegal character in the text.*

```
END XParam.
```

An example

Here is an example of how to use this interface to install a VBT `v` as a top level window, obeying the display and geometry arguments given to the application. It relies on the `Params` interface, which provides the number of arguments passed to the program, `Params.Count`, and a procedure to retrieve the value of the `n`th argument, `Params.Get(n)`.

```
EXCEPTION Error (TEXT);
VAR
  display, geometry: TEXT := NIL;
  d: XParam.DisplayRec;
  g: XParam.Geometry;
  i: CARDINAL := 1;
BEGIN
  LOOP
    IF i >= Params.Count - 1 THEN EXIT END;
    WITH argument = Params.Get (i) DO
      IF Text.Equal (argument, "-display") THEN
        display := Params.Get (i + 1);
        TRY d := XParam.ParseDisplay (display)
        EXCEPT XParam.Error (info) =>
          RAISE Error ("Illegal -display argument: "
                       & info.spec)
        END;
        INC (i, 2)
      ELSIF Text.Equal (argument, "-geometry") THEN
        geometry := Params.Get (i + 1);
        TRY
          g := XParam.ParseGeometry (geometry);
          IF g.size = XParam.Missing THEN
            WITH shapes = VBTClass.GetShapes (v, FALSE) DO
              g.size.h := shapes [Axis.T.Hor].pref;
              g.size.v := shapes [Axis.T.Ver].pref
            END
          END
        EXCEPT XParam.Error (info) =>
          RAISE Error ("Illegal -geometry argument: "
                       & info.spec);
        END;
        INC (i, 2)
      ELSE INC (i)
      END          (* IF *)
```

```
      END             (* WITH *)
    END;              (* LOOP *)
```

At this point, if `display` is non-`NIL`, then `d` contains the information from the `-display` argument. Similarly, if `geometry` is non-`NIL`, then `g` contains the information from the `-geometry` argument. If the window-size specificiation was missing, the preferred shape of the window is used.

Finally, we now process the `display` and `geometry` information:

```
    VAR
      trsl := Trestle.Connect (display);
      screen: CARDINAL;
    BEGIN
      TrestleImpl.SetDefault (trsl);
      Trestle.Attach (v, trsl);
      Trestle.Decorate (v, ...);
      IF geometry = NIL THEN
        Trestle.MoveNear (v, NIL)
      ELSE
        StableVBT.SetShape (v, g.size.h, g.size.v)
        IF d = NIL THEN
          screen := Trestle.ScreenOf (v, Point.Origin).id
        ELSE
          screen := d.screen
        END;
        Trestle.Overlap (
          v, screen, XParam.Position(trsl, screen, g))
      END       (* IF *)
    END         (* BEGIN *)
  END;          (* BEGIN *)
```

The call to `TrestleImpl.SetDefault` establishes the value of the `-display` argument as the default Trestle connection. The call to `StableVBT.SetShape` is used to control the size of a top-level window. The `TrestleImpl` and `StableVBT` interfaces are part of Trestle.

B.4 The FVTypes Interface

This interface declares a type for each component in the language. A client wishing to subclass the VBT used by a component should be sure that the VBT returned by the overrideVBT method is a subtype of type listed here.

```
INTERFACE FVTypes;

IMPORT AudioVBT, AnchorSplit, AnchorHelpSplit, BooleanVBT, BorderedVBT, ChoiceVBT,
       FileBrowserVBT, Filter, FlexVBT, Font, FormsVBT,
       GuardedBtnVBT, HVSplit, HighlightVBT, ListVBT,
       MenuSwitchVBT, NumericVBT, PackSplit, PaintOp, PixmapVBT,
       ProperSplit, ReactivityVBT, ScaleFilter, ScrollerVBT, Shadow,
       ShadowedVBT, ShadowedBarVBT, SourceVBT,
       SplitterVBT, StableVBT, SwitchVBT, TSplit, TextEditVBT,
       TextPort, TextureVBT, TextVBT, TrillSwitchVBT, TypeinVBT,
       TypescriptVBT, VBT, VideoVBT, ViewportVBT, ZChassisVBT,
       ZGrowVBT, ZMoveVBT, ZChildVBT, ZTilps;
IMPORT StubImageVBT AS ImageVBT;

TYPE
  FVAny = VBT.Leaf;              (* just an alias *)
  FVAnyFilter = Filter.T;        (* just an alias *)
  FVAnySplit = ProperSplit.T;    (* just an alias *)

  FVAudio = AudioVBT.T BRANDED OBJECT END;
  FVBar = FlexVBT.T BRANDED OBJECT END;
  FVBoolean <: BooleanVBT.T;
  FVBorder = BorderedVBT.T BRANDED OBJECT END;
  FVBrowser =
    ListVBT.T BRANDED OBJECT END; (* requires a UniSelector *)
  FVButton <: SwitchVBT.T;
  FVChisel = ShadowedBarVBT.T BRANDED OBJECT END;
  FVChoice <: ChoiceVBT.T;
  FVCloseButton <: PublicCloseButton;
  FVDirMenu = FileBrowserVBT.DirMenu BRANDED OBJECT END;
  FVFileBrowser <: FileBrowserVBT.T;
  FVFill = FlexVBT.T BRANDED OBJECT END;
  FVFilter = ReactivityVBT.T BRANDED OBJECT END;
  FVFrame = ShadowedVBT.T BRANDED OBJECT END;
  FVGeneric = FlexVBT.T BRANDED OBJECT END;
  FVGlue = FlexVBT.T BRANDED OBJECT END;
  FVGuard <: GuardedBtnVBT.T;
  FVHBox <: HVSplit.T;
  FVHPackSplit = PackSplit.T;
```

```
FVHTile <: SplitterVBT.T;
FVHelp <: AnchorHelpSplit.T;
FVHelper = FileBrowserVBT.Helper BRANDED OBJECT END;
FVImage <: ImageVBT.T;
FVIntApply <: IntApplyPublic;
FVLinkButton <: SwitchVBT.T;
FVLinkMButton <: MenuSwitchVBT.T;
FVMButton <: MenuSwitchVBT.T;
FVMenu <: AnchorSplit.T;
FVMultiBrowser =
  ListVBT.T BRANDED OBJECT END; (* requires a MultiSelector *)
FVNumeric <: NumericVBT.T;
FVPageButton <: PublicPageButton;
FVPageMButton <: PublicPageMButton;
FVPixmap = PixmapVBT.T BRANDED OBJECT END;
FVPopButton <: SwitchVBT.T;
FVPopMButton <: MenuSwitchVBT.T;
FVRadio = PublicRadio;
FVRidge = ShadowedBarVBT.T BRANDED OBJECT END;
FVRim = BorderedVBT.T BRANDED OBJECT END;
FVScale = ScaleFilter.T BRANDED OBJECT END;
FVScroller <: ScrollerVBT.T;
FVShape = FlexVBT.T BRANDED OBJECT END;
FVSource <: SourceVBT.T;
FVStable = StableVBT.T BRANDED OBJECT END;
FVTSplit = PublicTSplit;
FVTarget = Filter.T BRANDED OBJECT END;
FVText = TextVBT.T BRANDED OBJECT END;
FVTextEdit =
  TextEditVBT.T BRANDED OBJECT END; (* requires a Port *)
FVTexture = TextureVBT.T BRANDED OBJECT END;
FVTrillButton <: TrillSwitchVBT.T;
FVTypeIn <: TypeinVBT.T;
FVTypescript = TypescriptVBT.T BRANDED OBJECT END;
FVVBox <: HVSplit.T;
FVVTile <: SplitterVBT.T;
FVVideo = VideoVBT.T BRANDED OBJECT END;
FVViewport = ViewportVBT.T BRANDED OBJECT END;
FVZBackground = HighlightVBT.T BRANDED OBJECT END;
FVZChassis <: ZChassisVBT.T;
FVZChild = ZChildVBT.T BRANDED OBJECT END;
FVZGrow = ZGrowVBT.T BRANDED OBJECT END;
FVZMove = ZMoveVBT.T BRANDED OBJECT END;
```

```
  FVZSplit = ZTilps.T BRANDED OBJECT END;

TYPE UniSelector <: ListVBT.UniSelector;
```

If you create a subtype of `FVBrowser`*, its* `.selector` *field must be* `NIL` *or a subtype of* `FVTypes.UniSelector`*.*

```
TYPE MultiSelector <: ListVBT.MultiSelector;
```

If you create a subtype of `FVBrowser`*, its* `.selector` *field must be* `NIL` *or a subtype of* `FVTypes.MultiSelector`*.*

```
TYPE
  Port <: PublicPort;
  PublicPort =
    TextPort.T OBJECT
    METHODS
      init (textedit      : FVTextEdit;
            reportKeys    : BOOLEAN;
            font          : Font.T;
            colorScheme   : PaintOp.ColorScheme;
            wrap, readOnly: BOOLEAN;
            turnMargin    : REAL                 ): Port;
    END;
```

If you create a subtype of `FVTextEdit`*, its* `.tp` *field must be* `NIL` *or a subtype of* `FVTypes.Port`*.*

```
TYPE
  PublicCloseButton =
    SwitchVBT.T OBJECT
    METHODS
      init (ch: VBT.T; shadow: Shadow.T): FVCloseButton
    END;

  PublicPageButton = SwitchVBT.T OBJECT
                     METHODS
                       init (ch       : VBT.T;
                             shadow   : Shadow.T;
                             backwards: BOOLEAN;
                             tsplit   : FVTSplit  ): FVPageButton
                     END;

  PublicPageMButton =
    MenuSwitchVBT.T OBJECT
    METHODS
      init (ch       : VBT.T;
            shadow   : Shadow.T;
```

```
            backwards: BOOLEAN;
            tsplit   : FVTSplit  ): FVPageMButton
    END;

  PublicRadio = Filter.T OBJECT radio: ChoiceVBT.Group END;

  PublicTSplit = TSplit.T OBJECT circular := FALSE END;

  IntApplyPublic =
    Filter.T OBJECT
    METHODS
      init (fv      : VBT.T;
            ch      : VBT.T;
            name    : TEXT;
            property: TEXT    := NIL): FVIntApply
            RAISES {FormsVBT.Error};
      (* raises an error if NOT ISTYPE(fv, FormsVBT.T) OR
         NOT(ISTYPE(ch, FVNumeric) OR ISTYPE(ch, FVScroller)) *)
    END;

END FVTypes.
```

B.5 The Rsrc interface

Resources are arbitrary files that are associated with applications. Resources can be bundled into an application using the `m3bundle` facility. They may also be found in the file system.

This interface supports retrieval of resources using a *search path*. A search path is a list of elements, and each element is either a *path* or a *bundle*. A path is a directory, implemented as a `Pathname.T`. It should already be fully expanded by having called `Pathname.Expand`. A bundle is a `Bundle.T` object, typically created by `m3bundle`.

```
INTERFACE Rsrc;

IMPORT List, Rd, Thread;

TYPE
  Path = List.T; (* of Pathname.T or Bundle.T *)

EXCEPTION NotFound;

PROCEDURE Open (name: TEXT; path: Path): Rd.T RAISES {NotFound};
```

Search each element of `path`*, from front to back, for the first occurrence of the resource called* `name` *and return a reader on the resource. If the path element is a string* `s`*, then a reader is returned if*

```
FileStream.OpenRead(s & "/" & name)
```

is successful. If the path element is a bundle `b`*, a reader is returned if*

```
TextRd.New(Bundle.Get(b, name))
```

is successful. The `NotFound` *exception is raised if no element of* `path` *yields a successful reader on* `name`*. It is a checked runtime error if* `path` *contains an element that is neither a string nor a bundle.*

```
PROCEDURE Get (name: TEXT; path: Path): TEXT
  RAISES {NotFound, Rd.Failure, Thread.Alerted};
```

A convenience procedure to retrieve the contents of the resource `name` *as a* `TEXT`*.* `Get` *is logically equivalent to*

```
VAR rd := Open(name);
BEGIN
  TRY
    RETURN Rd.GetText(rd, LAST(CARDINAL))
  FINALLY
    Rd.Close(rd)
  END
END;
```

The implementation is slightly more efficient, because it takes advantage of `Bundle.Get` *procedure which returns the contents of the bundle element as a* `TEXT`*. The* `Rd.Failure` *exception is raised if* `Rd.GetText` *or* `Rd.Close` *report a problem. The* `Thread.Alerted` *can be raised by the call to* `Rd.GetText`*.*

```
PROCEDURE BuildPath (a1, a2, a3, a4: REFANY := NIL): Path;
```

Build a `Path` *from the non-*`NIL` *elements. Each element must be either a* `Bundle.T` *or a* `TEXT`*. If a* `TEXT`*, the string is passed to* `Pathname.Expand` *and the result is used, if it's non-*`NIL`*.*

Note: Currently, `Pathname.Expand` *is not implemented;* `TEXT`*s are expanded as follows: The text is assumed to be the name of a directory, unless it starts with a dollar sign. In the latter case, it is assumed to be environment variable and it's expanded using* `Env.Get`*.*

```
END Rsrc.
```

C. An Annotated Example

In this appendix, we present a complete example of a non-trivial form, the one used for FormsEdit itself. This form uses nearly all of the FormsVBT components, as well as macros.

It will be easier to understand this form if you can run the FormsEdit program at the same time, to see what each part of the description looks like in the actual application.

The design of a user interface is not easy. We make no claims about this particular one; it has been used by a fair number people for over a year, although details have changed. You may have quite different preferences in fonts, colors, and layout. At the very least, FormsEdit will make it easy for you to “fix” this form!

In this appendix, we present the form hierarchically. We show “line numbers” along the left edge, and each line containing an ellipsis is explained in more detail in a subsequent section, where the line numbers use decimal points.

The following diagram shows the overall structure of the form, which is fairly typical: a top-level filter of some sort with the global properties; some macros; and a ZSplit with a background and a dozen or so subwindows. While the entire file is fairly long, the structure is simple.

```
(Shape %top ...
  (Macro TLA ...)
  (Macro TRA ...)
  (Macro SEP ...)
  (Macro BOX ...)
  (Macro COMMAND ...)
  (Macro FINDER ...)
  (Macro YESNO ...)
  (Macro CONFIRM ...)
  (Macro FILEDIALOG ...)
  (ZSplit
    (ZBackground
     (VBox (Glue 3)
           (HBox %menubar ...)
           (Glue 3)
           Ridge
           (TextEdit %buffer)
           (FINDER ...)))
    (ZChassis %manpage ...)
    (ZChild %notFound ...)
    (ZChild %aboutFE ...)
    (ZChassis %errorPopup ...)
    (ZChassis %PPwidthNumeric ...)
    (ZChassis %snapshotDialog ...)
    (ZChassis %dumpTablePopup ...)
    (FILEDIALOG %OpenDialog ...)
    (FILEDIALOG %SaveAsDialog ...)
    (CONFIRM %quitConfirmation ...)
    (CONFIRM %switchConfirmation ...)
    (CONFIRM %closeConfirmation ...)
    (YESNO %overwriteConfirmation ...)
    (YESNO %RevertDialog ...)))
```

C.1 The top-level filter

The outermost form is usually a filter where you can place global properties. In this case, we use a Shape filter so that the editing windows can start out with a similar size.

```
1.0   (Shape %top
1.1     (Width  425 - 200 + Inf)
1.2     (Height 300 - 200 + Inf)
1.3     (LabelFont (Family "new century schoolbook"))
1.4     (Font "fixed")
1.5     (BgColor "PaleYellow")
1.6     (LightShadow "VeryVeryLightBlue")
1.7     (DarkShadow "Blue")
        ...)
```

On lines 1.1 and 1.2, we establish an initial size for the editor window, 425 points wide and 300 points tall. It can shrink to 200x200, and it can grow arbitrarily large. On lines 1.3 and 1.4, we specify the initial fonts for labels (Text forms) and for all the editable-text areas (TextEdit and TypeIn forms). Since the user will want to align columns of text (i.e., pretty-print the form), it is appropriate to use a fixed-width font. We also include the default colors here.

C.2 Simple macros

The first macros are just shorthand: TLA for “Text LeftAlign” and so on.

```
2 (Macro TLA BOA (x) `(Text LeftAlign ,x))
3 (Macro TRA BOA (x) `(Text RightAlign ,x))
4 (Macro SEP () '(VBox (Glue 3) Ridge (Glue 3)))
```

TLA and TRA are simple, 1-argument macros that are used several times throughout this form. BOA stands for “By Order of Argument,” which means that the arguments are passed by position, not by keyword. This allows us to write `(TLA "Open")`, for example, instead of `(TLA (x "Open"))`.

SEP, on line 4, is the simplest form of macro, effectively a constant, since it takes no arguments. We use it to separate groups of items within a Menu.

C.3 A recursive macro

The BOX macro produces a series of nested, double-bordered boxes.

```
5.0 (Macro BOX (pens child)
5.1   (IF (= pens '())
5.2       child
5.3       `(Border
5.4          (Pen ,(List.Nth pens 0))
5.5            (Rim (Pen ,(List.Nth pens 1))
5.6              (BOX (pens ,(List.NthTail pens 2))
5.7                   (child ,child))))))
```

BOX is a recursive macro. It generates expressions of the form

```
(Border (Pen ...)
  (Rim (Pen ...)
    (Border (Pen ...)
      (Rim (Pen ...)
        ...))))}
```

The `pens`-argument is a list of pen-widths; it must have an even number of elements. The even-numbered widths are used for the Borders; the odd-numbered widths are used for the Rims. For example,

```
(BOX (pens (2 4)) (child "Hello!"))
```

expands into

```
(Border (Pen 2) (Rim (Pen 4) "Hello!"))
```

The expression

```
(BOX (pens (2 4 5 10)) (child "Hello!"))
```

expands into

```
(Border (Pen 2)
  (Rim (Pen 4)
    (Border (Pen 5)
      (Rim (Pen 10)
        "Hello!"))))
```

Line 5.1 tests whether there are any pen-widths left in the list. If not, the expansion is simply the "child," on line 5.2. If there are pen-widths in the list, then the first two are used in lines 5.4 and 5.5, and the rest are passed recursively on line 5.6.

C.4 A macro for menu-items

The COMMAND macro generates menu-items that have some text on the left (the name of the command, such as “Open”), some filler-space in the middle, and a TSplit containing all the possible keybindings on the right. A keybinding is the name of a keyboard-equivalent for the command, such as **M-o**. FormsEdit allows you to change text-editing models, by means of a menu described in Section C.11 on page 178. When you do so, it changes all the keybindings in all the menus. In the Emacs model, for example, the keybinding for “Open” is written **M-o**, but in the Ivy model, it’s **oO**.

```
6.0  (Macro COMMAND BOA (name label k1 k2 k3 k4 (type (MButton)))
6.1    `(,@type
6.2       ,name
6.3       (HBox
6.4         (TLA ,label)
6.5         Fill
6.6         (TSplit
6.7           %,(SxSymbol.FromName
6.8                (Text.Cat "Model_"
6.9                   (SxSymbol.Name (List.Nth name 1))))
6.10          (TRA ,k1)
6.11          (TRA ,k2)
6.12          (TRA ,k3)
6.13          (TRA ,k4)))))
```

The macro generates MButtons by default, as we see at the end of line 6.0. The button contains an HBox (line 6.3) that has the left-aligned command-label on the left (line 6.4) and the keybinding on the right (lines 6.6–6.13).

The arguments `k1–k4` are the keybindings for the four models. We use the TSplit on line 6.6 to switch among them. The name of the TSplit is constructed by concatenating the string `"Model_"` to the name of the menu-item, e.g., `"Model_Open"`. When the user changes the text-editing model, the application calls `FormsVBT.PutInteger(fv, "Model_Open", n)` to set the TSplit to nth child, where n is the index that corresponds to the model.

Arguments are passed to this macro by position (BOA), and calls are written as

```
(COMMAND %foo ...)
```

That’s equivalent to writing `(COMMAND (Name foo) ...)`, so the first parameter, `name`, is bound to the list `(Name foo)`. To extract the symbol `foo` from that, we call `(List.Nth name 1)` on line 6.9. Passing that symbol to `SxSymbol.Name` produces the string `"foo"`, which we then concatenate to the string `"Model_Open"` on line 6.8.

C.5 A macro for a Finder-dialog

To provide a user interface for text-searching, we use a “Finder,” a form that contains a type-in field for the search-string, buttons for finding the first, next, or previous occurrence of that string, and a button to hide the form. See Figure C.1. The application searches through the text (the buttons tell it where to start and in which direction to look). If the search succeeds, the matching text is highlighted.

We use this form both in the main editing window and in the help-window, which is why we make it a macro.

```
7.0  (Macro FINDER (first next prev typein show close)
7.1    `(TSplit
7.2       Flex
7.3       Circular
7.4       (LightShadow "White")
7.5       (BgColor "VeryPaleBlue")
7.6       (Shape (Height 0) "")
7.7       (VBox %,show
7.8         Ridge
7.9         (Glue 2)
7.10        (HBox
7.11          (Shape (Width + 0) "Find:")
7.12          (Shape (Width 0 + 3000)
7.13            (Frame Lowered
7.14              (TypeIn %,typein (BgColor "VeryVeryPaleBlue"))))
7.15          (Glue 5)
7.16          (Shape (Width 0 + 1000)
7.17            (Rim (Pen 1) (Button %,first "First")))
7.18          (Shape (Width 0 + 1000) (ShadowSize 2.5)
7.19            (Button %,next "Next"))
7.20          (Shape (Width 0 + 1000)
7.21            (Rim (Pen 1) (Button %,prev "Prev.")))
7.22          (Glue 20)
7.23          (Rim (Pen 1) (PageButton %,close "C")))
7.24        (Glue 2))))
```

The FINDER macro expands into a TSplit with two children. The first child, on line 7.6, has no size at all; since the TSplit has the Flex property (line 7.2), each child gets the size it wants, so “displaying” the first child is displaying nothing.

Figure C.1: *The Finder dialog*

The second child is a VBox, beginning on line 7.7 (see Figure C.1). Since the Finder-dialog appears at the bottom of the editing and Help windows, we separate it from the rest with a Ridge (a 3-D bar) and 2 points' worth of Glue (lines 7.8–7.9). Underneath them is an HBox that contains a type-in field and three buttons: First, Next, and Previous. The Next button is the default, so it has the same effect as typing Return in the type-in field. To indicate that it is the default, it has no Rim, as the First and Prev buttons do (lines 7.17, 7.21). Instead, it uses the same amount of space for its shadow-size (line 7.18): 1 point to match the Rims' Pen-size, plus 1.5 points, which is the default shadow-size.

The (Width 0 + 3000) property on the framed type-in component (line 7.12), and the (Width 0 + 1000) properties on the three buttons, have the effect of diving the horizontal space into six equal regions (3 + 1 + 1 + 1), the first three of which are given to the type-in. These proportions will persist even if the subwindow grows or shrinks.

The "close button," marked "C," is a PageButton (line 7.23). Since the TSplit in which it appears has the Flex property, clicking this button selects the first child, the one that has no size, so the effect is to make the entire TSplit disappear.

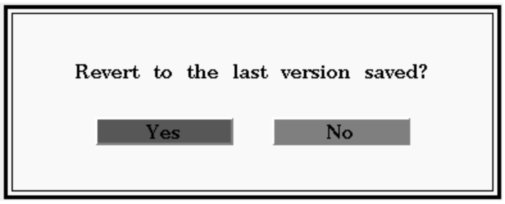


Figure C.2: *A yes/no dialog.*

C.6 A macro for yes/no dialogs

A yes/no dialog is a simple form containing two buttons; see Figure C.6 for an example. There are two places in the form where a yes/no dialog is needed (lines 31 and 32); we use this macro to ensure that the dialogs look the same.

```
8.0  (Macro YESNO (Name msg yesName noName)
8.1    `(ZChild %,Name
8.2       (BgColor "VeryPaleGray")
8.3       (LightShadow "White")
8.4       (DarkShadow "VeryDarkGray")
8.5       (Shape (Width 300)
8.6         (BOX (pens (2 2 1 26))
8.7           (child
8.8             (VBox
8.9               ,msg
8.10              (Glue 20)
8.11              (HBox
8.12                Fill
8.13                (Button %,yesName (BgColor "Red")
8.14                  (Shape (Width 80) "Yes"))
8.15                Fill
8.16                (CloseButton %,noName (BgColor "Green")
8.17                  (Shape (Width 80) "No"))
8.18                Fill)))))))
```

The call to BOX on line 8.6 produces a 2-point Border around a 2-point Rim, which in turn encloses a 1-point Border around a 26-point Rim, which surrounds the VBox beginning on line 8.8.

The VBox contains a question (line 8.9), some filler, and two equally spaced buttons (8.13 and 8.16). We use the convention that the "safe" option is always green, and the "dangerous" option is red. In our case, the safe option is also the do-nothing option, so we use a CloseButton for "No" (line 8.16).

C.7 A macro for confirmation dialogs

A confirmation dialog is similar to a yes/no dialog, but it offers the user two choices on how to perform some action, plus a third choice of not performing the action at all. For example, the form on line 30 asks the user whether changes should be saved before closing the file. The choices are: yes, close the file, but save the changes first; no, close the file, but discard the changes; and don't close the file at all. The third choice is often simply labeled "Cancel," but a more descriptive label (e.g., "Don't close") may be more helpful.

```
9.0  (Macro CONFIRM (Name question yesName noName
9.1                  cancelName cancelLabel)
9.2    `(ZChild %,Name
9.3       (BgColor "VeryPaleBlue")
9.4       (LightShadow "White")
9.5       (DarkShadow "VeryDarkBlue")
9.6       (Shape (Width 300)
9.7         (BOX (pens (2 2 1 26))
9.8           (child
9.9             (VBox
9.10              ,question
9.11              (Glue 20)
9.12              (HBox
9.13                Fill
9.14                (VBox
9.15                  (Button %,yesName (BgColor "Green")
9.16                    (Shape (Width 80) "Yes"))
9.17                  (Glue 10)
9.18                  (Button %,noName (BgColor "Red")
9.19                    (Shape (Width 80) "No")))
9.20                (Glue 20)
9.21                (VBox
9.22                  Fill
9.23                  (Filter %,cancelName
9.24                    (CloseButton (Shape (Width 80) ,cancelLabel))))
9.25                Fill)))))))
```

The arguments are the name for the subwindow, the question being asked (e.g., "Save changes before quitting?"), the names of the "yes," "no," and "cancel" button, and the text for the cancel button (e.g., "Don't quit").

Note that as in the YESNO macro on lines 8.0–8.18, green is used for the "safe" button, red for the "dangerous" button, but now the text of those buttons is reversed: the "safe" button says "No", and the "dangerous" button says "Yes", because the question being asked in a confirmation dialog is always "dangerous."

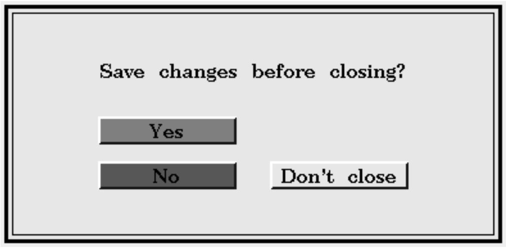


Figure C.3: *A confirmation dialog*

C.8 A macro for a file-chooser

As we mentioned in the description of the FileBrowser component on page 94, it is a good idea to combine a FileBrowser, a DirMenu, and a Helper, with activation and cancellation buttons, in a standard arrangement. We do that with the following macro.

```
10.0  (Macro FILEDIALOG
10.1    (Name BgColor DarkShadow Title fbName OKName OKLabel
10.2     cancelName (cancelLabel "Cancel") helperName
10.3     (ReadOnly FALSE) (other ()))
10.4    `(ZChassis %,Name
10.5       (BgColor ,BgColor)
10.6       (LightShadow "White")
10.7       (DarkShadow ,DarkShadow)
10.8       (Title ,Title)
10.9       (Shape (Width 300 - 200 + Inf) (ShadowSize 2)
10.10        (Rim
10.11          (Pen 10)
10.12          (VBox
10.13            (HBox
10.14              (Shape (Width 150 + Inf) (Height 150 + Inf)
10.15                (VBox (LabelFont "fixed")
10.16                  (DirMenu (For ,fbName))
10.17                  (Glue 6)
10.18                  (Frame Lowered (BgColor "VeryPaleGray")
10.19                    (FileBrowser %,fbName))))
10.20             Fill
10.21              (Shape (Width 100)
10.22                (VBox
10.23                 Fill
10.24                  (Button %,OKName ,OKLabel)
10.25                  (Glue 10)
10.26                  (Filter
10.27                    (CloseButton %,cancelName ,cancelLabel)))))
10.28            (Glue 6)
10.29            (HBox
10.30              (Shape (Width 30) "File:")
10.31              (Frame Lowered
10.32                (Helper %,helperName FirstFocus (For ,fbName)
10.33                  (BgColor "VeryPaleGray"))))
10.34            ,@other)))))
```

While the overall appearance is standard, the names of the components, the colors, the labels, and other aspects may vary, so those are all passed in as parameters to the macro. Figure C.4 shows what this looks like for the "Save As..." dialog in Section C.24 on page 193.

The fixed-width label-font is used on line 10.15 to make it easier to read filenames. Filename-punctuation such as periods and slashes are often hard to read in small, variable-width fonts.

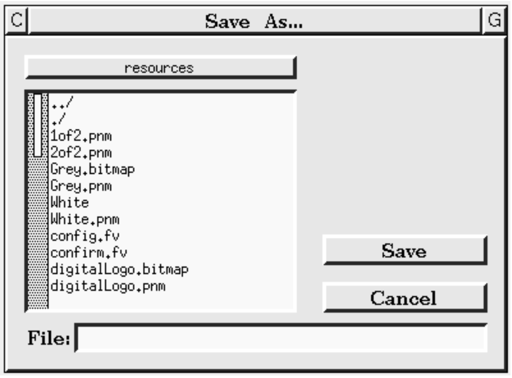


Figure C.4: *The 'Save As..." file-dialog*

As in the example on line 9.22, the Filter on line 10.26 makes it possible to de-activate the Cancel button via `FormsVBT.MakeDormant`.

C.9 The background child

```
(ZSplit
  (ZBackground
    (VBox (Glue 3)
          (HBox %menubar ...)
          (Glue 3)
          Ridge
          (TextEdit %buffer)
          (FINDER ...)))
   ...)
```

The background child of this ZSplit is a VBox with a menubar, the main text-editing area,[1] and the Finder-dialog.

[1] The most complex component has the shortest description in the form!

C.10 The menubar

```
14.0  (HBox %menubar
14.1    (Glue 5)
14.2    (Menu ...)
14.3    (Glue 5)
14.4    (Menu ...)
14.5    (Glue 5)
14.6    (Menu ...)
14.7    (Glue 5)
14.8    (Menu ...)
14.9    (Glue 5)
14.10   Fill
14.11   (TSplit %modified
14.12     (Text "")
14.13     (Pixmap (Color "Red") "bnote.pbm"))
14.14   (Text %shortname "")
14.15   Fill
14.16   (Button %parse (BgColor "VeryPaleBlue")
14.17     (Text (Margin 10) "Do It"))
14.18   (Glue 5))
```

The menubar is an HBox with four Menus, a TSplit for the note-icon (14.11–14.13), the filename (14.14), and the "Do it" Button.

This TSplit displays either an empty string (14.12) or a small red warning note (14.13), depending on whether there are unsaved changes to the text in the buffer. The application switches between these choices by calling `FormsVBT.PutInteger(fv, "modified", n)`, where n is the index of the desired TSplit-child.

The name of the file is inserted into the menubar by assigning it to Text component on line 14.14, via `FormsVBT.PutText(fv, "shortname", ...)`. The Fill's on lines 14.10 and 14.15 cause the TSplit and the filename to be centered between the menus and the button.

C.11 The quill-pen menu

```
14.2.0 (Menu
14.2.1   (Shape (Width 40) (Height 13.5) (Pixmap "pen.pbm"))
14.2.2     (Border
14.2.3       (VBox
14.2.4         (PopMButton (For aboutFE) (TLA "About FormsEdit..."))
14.2.5         (SEP)
14.2.6         (COMMAND %Help "Help..."
14.2.7           "oH" "M-h" "c-h" "M-h" (PopMButton (For manpage)))
14.2.8         Ridge
14.2.9         (Radio %Model
14.2.10          (HBox
14.2.11            (Glue 10)
14.2.12            (VBox
14.2.13              "Editing Model"
14.2.14              (Choice %ivyModel MenuStyle (TLA "Ivy"))
14.2.15              (Choice %emacsModel MenuStyle (TLA "Emacs"))
14.2.16              (Choice %macModel MenuStyle (TLA "Mac"))
14.2.17              (Choice %xtermModel MenuStyle (TLA "Xterm")))))
14.2.18        Ridge
14.2.19        (COMMAND %quit2 "Quit" "oQ" "M-q" "c-q" "M-q")))))
```

The quill-pen icon was created using `bitmap(1)`, converted to "pnm" format by `anytopnm(1)`, and saved in the resource-file named `pen.pbm`. All the menu-buttons have a width of 40 points.

Line 14.2.4 shows a simple pop-menu-button containing a left-aligned string. When the button is released, FormsVBT automatically opens the subwindow whose name in this form is `aboutFE`. The subwindow itself is defined on line 21.

The call to `(SEP)` on line 14.2.5 produces the VBox that separates groups within a menu. It was defined on line 4; see Section C.2 on page 166.

The first "command" button is the Help button on lines 14.2.6–14.2.7. As it happens, it is a PopMButton, not an MButton; by passing a seventh argument, `(PopMButton (For manpage))`, we avoid getting the default value for the `type` parameter; see line 6.0 in Section C.4 on page 168. The name of the button is `Help`. The four keynames are the keyboard shortcuts for the Ivy, Emacs, Mac,[2] and Xterm models, in that order.

[2] We use "c-h" for the Mac, since there is no ISO Latin-1 character that corresponds to the Mac "command" or "cloverleaf" icon.

This macro-call expands into the following expression:

```
(PopMButton (Name Help)
  (HBox (Text LeftAlign "Help...")
        Fill
        (TSplit (Name Model_Help)
           (Text RightAlign "oH")
           (Text RightAlign "M-h")
           (Text RightAlign "c-h")
           (Text RightAlign "M-h"))))
```

The 4 radio-buttons on lines 14.2.14–14.2.17 allow the user to choose the editing model. The group is indented slightly (line 14.2.11).

C.12 The File menu

The File menu has the standard assortment of items: Open, Save, Save As..., etc. The form describes all the visual aspects of the menu; the application interprets each of the commands by attaching an event-handler to each name.

```
14.4.0  (Menu
14.4.1    (Shape (Width 40) "File")
14.4.2     (Border
14.4.3       (Shape (Width 110)
14.4.4         (VBox
14.4.5           (COMMAND %new "New" "oN" "M-n" "c-n" "M-n")
14.4.6           (COMMAND %openMButton "Open..."
14.4.7             "oO" "M-o" "c-o" "M-o"
14.4.8             (PopMButton (For OpenDialog)))
14.4.9           (SEP)
14.4.10          (MButton %close (TLA "Close"))
14.4.11          (Filter (COMMAND %save "Save" "oS" "M-s" "c-s" "M-s"))
14.4.12          (PopMButton %saveasMButton (For SaveAsDialog)
14.4.13            (TLA "Save As..."))
14.4.14          (PopMButton %revertbutton (For RevertDialog)
14.4.15            (TLA "Revert To Saved"))
14.4.16          (SEP)
14.4.17          (PopMButton %ppwidthPopMButton (For PPwidthNumeric)
14.4.18            (TLA "PP setup..."))
14.4.19          (COMMAND %PPrint "PPrint" "oP" "M-p" "c-p" "M-p")
14.4.20          (SEP)
14.4.21          (COMMAND %quit "Quit" "oQ" "M-q" "c-q" "M-q"))))))
```

The Filter on line 14.4.11 enables the application to gray-out the Save button, which it does when no changes have been made to the buffer, sincethere's no point in saving an unmodified file.

The Quit button on line 14.4.21 is a duplicate of the item on line 14.2.19 at the bottom of the quill-pen menu. A Quit button normally appears at the bottom of the File menu, as it is here, but some users expect to see it at the bottom of the top-left menu, whatever that may be called. The application attaches the same event-handler to both quit-buttons.

C.13 The Edit Menu

The Edit menu has a standard set of items, plus three additional items for searching.

```
14.6.1   (Menu
14.6.2     (Shape (Width 40) "Edit")
14.6.3       (Border
14.6.4         (Shape (Width 100)
14.6.5           (VBox
14.6.6             (COMMAND %undo "Undo" "cZ" "C-_" "c-z" "M-z")
14.6.7             (COMMAND %redo "Redo" "csZ" "M-_" "c-Z" "M-Z")
14.6.8             (SEP)
14.6.9             (COMMAND %cut "Cut" "oX" "C-w" "c-x" "M-x")
14.6.10            (COMMAND %copy "Copy" "oC" "M-w" "c-c" "M-c")
14.6.11            (COMMAND %paste "Paste" "oV" "C-y" "c-v" "M-v")
14.6.12            (COMMAND %clear "Clear" "" "" "" "")
14.6.13            (COMMAND %selectAll "SelectAll"
14.6.14              "oA" "M-a" "c-a" "M-a")
14.6.15            (SEP)
14.6.16            (COMMAND %findMButton "Find..."
14.6.17              "oF" "" "c-f" "M-f"
14.6.18              (LinkMButton (For FindInBuffer2)))
14.6.19            (COMMAND %findNext "Find Next" "c," "C-s" "" "")
14.6.20            (COMMAND %findPrev "Find Prev" "cM" "C-r" "" ""))))))
```

The “Find” button on 14.6.18 is a not a PopMButton controlling a subwindow, although that would be more typical. Instead, it is a LinkMButton that selects the second child (see lines 7.7–7.23) of the Finder-dialog at the bottom of the window. The Finder is a TSplit; when its second child appears, the main window shrinks somewhat, but nothing is hidden. The typical pop-up window would *overlap* the main window and could easily obscure the text that the user has highlighted after a successful search.

C.14 The Misc Menu

```
14.8.0   (Menu
14.8.1     (Shape (Width 40) "Misc")
14.8.2       (Border
14.8.3         (VBox
14.8.4           (PopMButton %dumpTable (For dumpTablePopup)
14.8.5             "Show the named VBTs...")
14.8.6           (PopMButton %snapshot  (For snapshotDialog)
14.8.7             "Show current snapshot...")
14.8.8           (PopMButton (For errorPopup)
14.8.9             "Show last error message")
14.8.10          (SEP)
14.8.11          (Filter %rescreenFilter (VBox %rescreenMenu)))))
```

The “dumpTable” and “snapshot” buttons bring up windows that give information about the form; they are defined on lines 24–25.

Error messages are reported in a green subwindow in the bottom-right corner (see line 22), but the application automatically closes this window after 5 seconds. The button on line 14.8.8 gives the user a way to bring it back without a 5-second timeout, to read the message more carefully.

Note that the VBox on line 14.8.11 is empty. The application dynamically inserts MButtons here, one pair for each display-screen, for moving the editor window or the result window. On a 1-screen workstation, no buttons are inserted, and the VBox is grayed-out via the Filter. (Some window managers allow windows to be dragged between screens, which is more convenient than using these buttons.)

C.15 The Finder-dialog

The Finder at the bottom of the main window is generated by the following call:

```
(FINDER
  (show FindInBuffer2)
  (first bhelpfindfirst)
  (next bhelpfindnext)
  (prev bhelpfindprev)
  (typein bhelpfindtext)
  (close bhelpfindclose))
```

The layout is fixed (see Figure C.1); all the parameters are names assigned to the various components. The application attaches event-handlers to these names to perform the search.

C.16 The Help subwindow

```
19.0  (ZChassis %manpage
19.1    (BgColor "VeryPaleBlue")
19.2    (Title "formsedit help")
19.3    (VBox
19.4      (HBox
19.5        (Menu
19.6          (Shape (Width 40) "Edit")
19.7          (VBox
19.8            (COMMAND %mpcopy "Copy" "oC" "M-w" "c-c" "M-c")
19.9            (COMMAND %mpselectAll "SelectAll"
19.10             "oA" "M-a" "c-a" "M-a")
19.11           (SEP)
19.12           (COMMAND %mpfindMButton "Find..."
19.13             "oF" "" "c-f" "M-f"
19.14             (LinkMButton (For FindDialog)))
19.15           (COMMAND %mpfindNext "Find Next"
19.16             "c," "C-s" "" "")
19.17           (COMMAND %mpfindPrev "Find Prev"
19.18             "cM" "C-r" "" "")))
19.19       Fill)
19.20     Ridge
19.21     (Shape (Width 360 + Inf) (Height 150 + Inf)
19.22       (TextEdit ReadOnly %manpagetext))
19.23     (FINDER
19.24       (show FindDialog)
19.25       (first helpfindfirst)
19.26       (next helpfindnext)
19.27       (prev helpfindprev)
19.28       (typein helpfindtext)
19.29       (close helpfindclose))))
```

This is the subwindow that pops up when the user clicks the Help button on line 14.2.6. (See Figure C.5.) Its structure is similar to that of the main window. It contains a menubar (HBox) with one menu, Edit, which has items appropriate for a read-only buffer (e.g., Copy but not Cut). The Edit menu also includes a LinkMButton for another Finder-dialog, just as the main editing window does. Below the menubar is a read-only text-editor, which the application fills with the manpage for FormsEdit. At the bottom of the subwindow is the Finder-dialog, which is shown in the open position in Figure C.5.

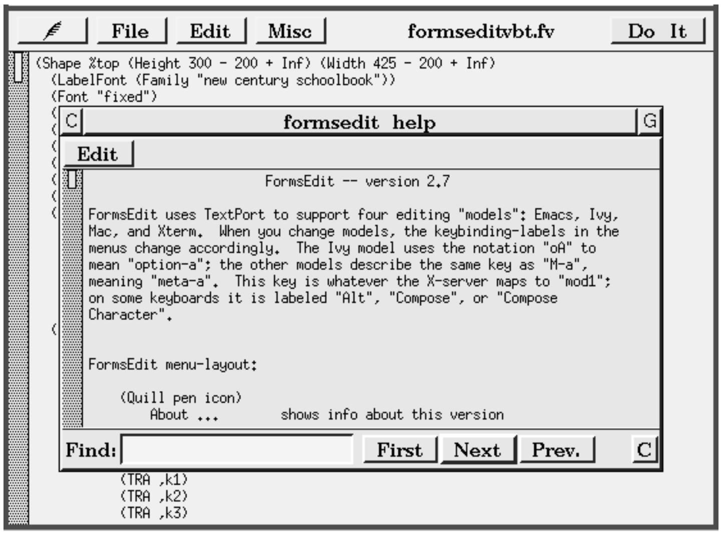


Figure C.5: *The "manpage" window*

C.17 A disappearing subwindow

```
20.0  (ZChild %notfound (BgColor "Red") (Color "White")
20.1    (Rim (Border "Not found")))
```

This is a small subwindow that the application displays—there is no PopButton for this component—whenever a search has failed, either in the editor window or in the "manpage" window. The application removes it after 2 seconds.

Figure C.6: *The "About FormsEdit..." window*

C.18 The About... window

```
21.0  (ZChild %aboutFE (BgColor 0.8 0.8 1)
21.1    (BOX (pens (1.5 2 1 12))
21.2      (child
21.3        (VBox
21.4          (Pixmap (Color "Blue") "digitalLogo.pbm")
21.5          (Glue 6)
21.6          "FormsEdit version 2.7"
21.7          "Written by Jim, Marc, and Steve."
21.8          "Copyright \251 1993 Digital Equipment Corp."
21.9          "Send comments to meehan@src.dec.com"
21.10         (Glue 6)
21.11         Ridge
21.12         Ridge
21.13         (Glue 6)
21.14         (HBox
21.15           Fill
21.16           (CloseButton (BgColor "VeryPaleBlue")
21.17                        (LightShadow "White")
21.18             (Text (Margin 5) "Close"))
21.19           Fill)))))
```

This is a completely static window. Note that the CloseButton (21.16) does not need a `For`-property, since it closes the subwindow that contains it. The use of the BOX macro (21.1) produces the double-bordered effect. (The 4-character sequence `\251` in the middle of the text on line 21.8 is converted into a single character by the S-expression reader; that character's code is 251 in octal, which is the ISO Latin-1 standard code for the copyright symbol, ©.)

C.19 The error-message subwindow

```
(ZChassis %errorPopup
  (At 1. 1. SE)
  (BgColor "VeryPaleGreen")
  (Title "Error")
  (LightShadow "White")
  (DarkShadow "DarkGreen")
  (Shape (Width 300 + Inf - 200) (Height 50 + Inf - 50)
    (TextEdit %stderr ReadOnly)))
```

The error-message window is displayed by the application whenever there is a parsing error. The application also removes it after 5 seconds. (The Misc menu has an item to make this window reappear.)

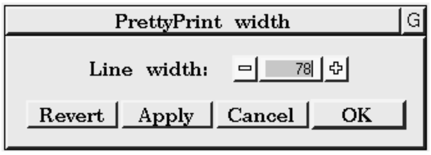


Figure C.7: *The "PrettyPrint width" window*

C.20 The pretty-print-width subwindow

```
23.0  (ZChassis %PPwidthNumeric
23.1    NoClose
23.2    (BgColor "PaleGold")
23.3    (Title (Text "PrettyPrint width"))
23.4    (At 0.1 0.1 NW)
23.5    (Shape (Width 250 + Inf)
23.6      (Rim
23.7        (Pen 10)
23.8        (VBox
23.9          (HBox
23.10           Fill
23.11           (Shape (Width 80) "Line width:")
23.12           (Glue 10)
23.13           (Shape (Width 70)
23.14             (Numeric %ppwidth FirstFocus
23.15               (BgColor "VeryPaleGold")
23.16               =78 (Min 30) (Max 200)))
23.17           Fill)
23.18         (Glue 10)
23.19         (HBox
23.20           (Shape (Width 0 + 1000)
23.21             (Rim (Pen 1) (Button %ppwRevert "Revert")))
23.22           (Shape (Width 0 + 1000)
23.23             (Rim (Pen 1) (Button %ppwApply "Apply")))
23.24           (Shape (Width 0 + 1000)
23.25             (Rim (Pen 1) (CloseButton "Cancel")))
23.26           (Shape (Width 0 + 1000) (ShadowSize 2.5)
23.27             (Button %ppwOK "OK")))))))
```

Some people would argue that having a pretty-printer built in to the editor is an essential tool in one's programming environment, regardless of the source language. It is especially helpful when the entire program is a single expression. (A parenthesis-balancer would help, too!) This pop-up window allows the user to specify the maximum line-width that the pretty-printer will use, measured in characters. (The pretty-printer is available as the "PPrint" item in the file menu; see line 14.1.19.) It is sometimes helpful, especially on a 2-screen workstation, to make the window as wide as possible and then to increase the pretty-print width; 150 is a reasonable maximum.

This subwindow uses the NoClose property (see line 23.1), which removes the button labeled “C” from the top-left corner. We do this so that we can be more precise about the side-effects of closing the window. There are two ways to close it. Clicking the Cancel button closes it without permanently changing the desired width, that is, the width that will be used on all subsequent calls to the pretty-printer. Clicking OK, or typing Return in the Numeric type-in, will invoke the pretty-printer with the new width, setting that to be the desired width, and finally closing the subwindow.

On line 23.14, the FirstFocus property has the effect that the type-in field within the Numeric will grab the keyboard focus whenever this subwindow pops up, and it selects the text (the width) in replace-mode, to make it easy for the user to type a new width, hit Return, and have the form prettyprinted.

Note that the first three buttons have a 1-point Rim around them, but that the last button, OK, does not. Instead, it has a shadow that is 1 point larger than the default (which is 1.5 points). The effect is to make the OK button stand out a little more: it is the “default” button, so it has the same effect as typing Return in the Numeric. In 2-D style, as on a monochrome screen, “default” buttons are usually given a black border. This is the same convention we used in the Finder window; see Section C.5 on page 169.

C.21 The snapshot subwindow

```
24.0  (ZChassis %snapshotDialog
24.1    (At 0.1 0.9 0.2 0.8 Scaled)
24.2    (BgColor "VeryPaleTurquoise")
24.3    (Title (Text (BgColor "White") (Color "DarkTurquoise")
24.4      "Current Snapshot"))
24.5    (Shape (Height 250 - 100 + Inf)
24.6      (TextEdit %SnapshotText ReadOnly)))
```

This is a subwindow used primarily for debugging "snapshot" and "restore" operations. See Section 4.5 on page 67.

C.22 The named-components subwindow

```
25.0  (ZChassis %dumpTablePopup
25.1    (BgColor "PaleGold")
26.2    (At 0.1 0.9 0.2 0.8 Scaled)
26.3    (Title (Text (BgColor "White") (Color "Blue") "Named VBTs"))
26.4    (Shape (Height 300 - 100 + Inf)
26.5      (TextEdit %VBTtable ReadOnly)))
```

This is also primarily a debugging window, but it's useful for debugging layout problems. Each named component in the form is described on a separate line that includes its type, its size-range, and its actual size, in both the horizontal and vertical dimensions. For example, the following line appears for the ZChassis on line 23:

```
PPwidthNumeric : FVTypes.FVZChassis
        H: [330, 330, 100001] = 330. V: [113, 113, 114] = 113.
```

The name of the component is `PPwidthNumeric`. Its runtime type is `FVTypes.FVZChassis`. Its horizontal size-range has a "lo" and a "pref" of 330 pixels, with a very large "hi" value (essentially unlimited stretchability). Its actual width is 330. Its vertical size-range has no stretchability; "lo" and "pref" are 113 pixels, and "hi" is 114. Its actual size is 113. (The actual size will be 0 for components that are not visible.)

C.23 The open-file dialog

```
26.0  (FILEDIALOG %OpenDialog
26.1    (BgColor "VeryPaleGreen")
26.2    (DarkShadow "RatherDarkGreen")
26.3    (Title "Open an existing file")
26.4    (fbName openfile)
26.5    (ReadOnly TRUE)
26.6    (OKName open)
26.7    (OKLabel "Open")
26.8    (cancelName cancelOpen)
26.9    (helperName fbh)
26.10   (other
26.11     ((Glue 6)
26.12      (HBox
26.13        (Radio =newwindow
26.14          (VBox
26.15            (Choice %reuse (TLA "Use this window"))
26.16            (Choice %newwindow (TLA "Open a new window"))))
26.17        Fill
26.18        (Radio =fvonly
26.19          (VBox
26.20            (Choice %fvonly (TLA "*.fv only"))
26.21            (Choice %notfvonly (TLA "Any file"))))))))
```

This is a standard file-dialog, produced by the macro on line 10, described in section C.8 on page 173.

As an addition to the standard file-dialog controls, this one allows the user to specify whether a new pair of windows (editor and result) should be used when displaying a new file. When this macro-call is expanded, the expressions on lines 26.11–26.21 are simply appended (see line 10.34) to the forms inside the VBox that starts on line 10.12.

The radio buttons on lines 26.15 and 26.16 are used by the application to determine whether to open the new file in a separate window.

The radio buttons on lines 26.20 and 26.21 are used to control the value that is passed to the procedure (`FileBrowserVBT.SetSuffixes`) that reads directories.

C.24 The save-as dialog

This window, shown in Figure C.4 on page 175, is simpler than the preceding open-file dialog: it has no “other” parameter, so it contains exactly what the macro specifies. It uses a different background color to distinguish it from the open-file dialog.

```
27.0  (FILEDIALOG %SaveAsDialog
27.1    (BgColor "VeryPaleBlue")
27.2    (DarkShadow "Blue")
27.3    (Title "Save As...")
27.4    (fbName saveasfile)
27.5    (OKName saveas)
27.6    (OKLabel "Save")
27.7    (cancelName cancelsaveas)
27.8    (helperName sfbh))
```

C.25 The confirmation dialogs

The three "confirmation" subwindows all perform a similar function, so the use of a macro makes it easy to give them a similar appearance. All that varies is the text they display. The macro itself is described in Section C.7 on page 172.

```
28.0  (CONFIRM %quitConfirmation
28.1    (question "Save changes before quitting?")
28.2    (yesName saveandquit)
28.3    (noName quitAnyway)
28.4    (cancelName dontquit)
28.5    (cancelLabel "Don't quit"))
29.0  (CONFIRM %switchConfirmation
29.1    (question "Save changes before switching?")
29.2    (yesName saveandswitch)
29.3    (noName switchAnyway)
29.4    (cancelName cancelSwitch)
29.5    (cancelLabel "Don't switch"))
30.0  (CONFIRM %closeConfirmation
30.1    (question "Save changes before closing?")
30.2    (yesName saveandclose)
30.3    (noName closeAnyway)
30.4    (cancelName cancelClose)
30.5    (cancelLabel "Don't close"))
```

C.26 The yes/no dialogs

The “yes/no” subwindows are similar to the “confirmation” windows. The macro is described in Section C.6 on page 171.

```
31.0  (YESNO %overwriteConfirmation
31.1    (msg "That file already exists. Overwrite it?")
31.2    (yesName overwrite)
31.3    (noName dontOverwrite))
32.0  (YESNO %RevertDialog
32.1    (yesName revert)
32.2    (noName dontRevert)
32.3    (msg "Revert to the last version saved?"))
```

Acknowledgments

Gidi Avrahami implemented an embryonic FormsVBT prototype in during the summer of 1988. Ken Brooks helped to implement the original FormsVBT system in Modula-2+ during 1989.

Bibliography

[1] Gideon Avrahami, Kenneth P. Brooks, and Marc H. Brown. A Two-View Approach To Constructing User Interfaces. *Computer Graphics*, 23(3):137–146, July 1989. A videotape of the system was part of the Video Program at the CHI'90 conference. The CHI '90 Video Program is available in the SIGGRAPH Video Review series.

[2] Edited by Marc H. Brown and James R. Meehan. VBTkit reference manual. Technical report, DEC Systems Research Center, in preparation.

[3] Samuel P. Harbison. *Modula-3*. Prentice Hall, 1992.

[4] Shiz Kobara. *Visual Design with OSF/Motif*. Addison Wesley, 1991.

[5] Mark S. Manasse and Greg Nelson. Trestle Reference Manual. Technical Report 68, DEC Systems Research Center, December, 1991.

[6] Mark S. Manasse and Greg Nelson. Trestle tutorial. Technical Report 69, DEC Systems Research Center, May 1, 1992.

[7] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[8] Randy Pausch, Nathaniel R. Young II, and Robert DeLine. SUIT: The Pascal of User Interface Toolkits. In *Proc. of the ACM Symposium on User Interface Software and Technology*, pages 117–125, November 1991.

[9] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System, 2nd edition*. Digital Press, 1990.