VBTkit Reference Manual: A Toolkit for Trestle

Edited by Marc H. Brown and James R. Meehan

Printed on April 26, 1996

Abstract

This manual describes VBTkit, a Modula-3 user interface toolkit based on the Trestle window system toolkit. VBTkit provides a library of “widgets” and the support software that makes it easy to customize these widgets and to construct more widgets.

Contents

Contributors

The VBTkit software reflects the efforts of many people at SRC. Development of the software began in early 1988, and new pieces are still being developed. The major contributors, in alphabetical order, are as follows:

- **Andrew Birrell** implemented `ListVBT` and preliminary versions of radio buttons and check boxes.
- **Ken Brooks** implemented early versions of the text-editing modules.
- **Marc H. Brown** implemented the 3-D look and feel (Section 3.1), the multi paradigm (Section 2), buttons (Sections 4 and 5), subwindows (Section 6), and some of the utility routines.
- **Mark R. Brown** implemented the Ivy text editor. This isn't part of VBTkit *per se*, but formed the inspiration for many of the advanced features found in VBTkit's editing models.
- **Pat Chan** implemented an earlier version of `ScrollerVBT`. Pat also implemented Ivy with Mark Brown.
- **Luca Cardelli** implemented a Trestle-based application builder in 1987. That work influenced the material in this document in many ways.
- **John DeTreville** implemented the `VText` package, which is the heart of the VBTkit's text-editing facilities.
- **Steve Glassman** implemented `Image`, `ScaleFilter`, `ViewportVBT` and `XTrestle`. Most importantly, Steve has discovered and fixed numerous infelicities in a variety of modules.
- **Mark Manasse** helped implement `AutoRepeat`, `ReactivityVBT`, and `ViewportVBT`. He was a valuable sounding board in the development of the multi-filter and multi-split paradigms.
- **Jim Meehan** implemented the text-editing modules (Section 8).
- **Jorge Stolfi** implemented the color modules (Section 14).

1 Introduction

This document is a programmer's reference manual for VBTkit, a Modula-3 user interface toolkit based on the Trestle window system toolkit. VBTkit provides a library of "widgets" and the support software that makes it easy to customize these widgets and to construct more widgets.

This reference manual is not self-contained; we assume you are already familiar with Trestle. If you are not, you should read the *Trestle Tutorial* [4] before continuing. You are advised to have a copy of the *Trestle Reference Manual* [3] in hand as you read this. Most of the material in this manual are logical extensions to Chapters 4, 5, and 6 of the *Trestle Reference Manual*.

1.1 Roadmap

Section 3 describes the facilities for giving a "three-dimensional" look. The basic data types are defined in the `Shadow` interface. A `ShadowedVBT` displays a 3-D border around a VBT, and a `ShadowedBarVBT` displays a horizontal or vertical line in 3-D.

Section 2 describes the ways that one can build a VBTkit widget out of other VBTs. For example, a `ViewportVBT` puts scrollbars around a child VBT. If the child's minimum acceptable size is larger than the domain it has been given, the scrollbars are used to see parts of the child. Logically, a `ViewportVBT` has a single child. Internally, a `ViewportVBT` comprises nearly a dozen VBTs, including two `ScrollerVBT`s, three `HVSplit`s, one `JoinedVBT`, two `FlexVBT`s, and so on.

We call a VBT with a single logical child a "multi-filter." Similarly, a "multi-split" is a VBT with zero more logical children. The common operations are multi-filters and multi-splits are done using the `MultiFilter` and `MultiSplit` interface. The `MultiClass` interface is used for building composite VBTs.

Composite VBTs are at the heart of the buttons that VBTkit provides; you are advised to study the `Multi`-related interfaces before reading about VBTkit buttons.

Section 5 describes VBTkit buttons. They are composed of three elements: contents, gesture, and feedback. By *contents*, we mean the "thing that's inside the button." Often this is just a piece of text (e.g., a `TextVBT`), but it can be any arbitrary VBT. Each subtype of ButtonVBT defines what constitutes a "button press"; e.g., does it react on an up-click or a down-click? This is called the *gesture*. Finally, by *feedback*, we mean the visual indication of the state of the button; e.g., when the button is activated, does it change its colors? The feedback is a multi-filter and a subclass of `FeedbackVBT`. Feedbacks are described in **Section 4**.

There are two fundamental differences between Trestle buttons and VBTkit buttons. First, in VBTkit the gesture is independent of the visual feedback. The gesture is defined by a subtype of `ButtonVBT`, and the feedback is defined by a subtype of `FeedbackVBT`. These VBTs are composed using the `MultiFilter` interface. (It is not unusual for a feedback VBT to be a composite VBT itself.) Second, the action procedure ("callback") that is invoked when the user activates a VBTkit button is a *method* of the VBT subclass, whereas in Trestle, it is a *procedure* that is specified when a button subclass is initialized. To ensure that there is no confusion, we call VBTkit buttons *switches*. VBTkit also adds considerably to the collection of buttons provided by Trestle. The switches provided by VBTkit includes guarded buttons (they require a double click to be activated), trill buttons (they continue to invoke the callback while the mouse button is down), Boolean check boxes, radio choices, and "drag and drop" buttons with semantic feedback.

Section 6 describes support for subwindows. A subwindow is like a top-level window but it is not installed in the window manager. The bad news is that user-gestures for controlling the subwindow (moving, reshaping, closing) are not the same as for top-level windows. The good news is that the user interface is quite obvious, and the subwindows are automatically controlled by the top-level window. For example, when the top level window is iconized, all of its subwindows also disappear (without cluttering up the window manager's icon box).

Section 7 contains the `PixmapVBT` and `Image` interfaces. A `PixmapVBT` is a VBT that displays a pixmap. The `Image` interface contains utilities for creating pixmaps, from files (stored in Jef Poskanzer's "pnm" format) and from the contents of an arbitrary VBT.

Section 8 describes the text-editing facilities in VBTkit. The primary VBT class is a `TextPort`. It provides a full-screen editor, with the most common commands bound to keys. It is rare to use a `TextPort` directly, however. Typically, one uses a `TextEditVBT`—a `TextPort` with a scrollbar. A popular subclass of `TextEditVBT` is a `TypescriptVBT`. Section 8 also describes a number of interfaces that are of interest to programmers implementing subclasses of text-editors.

Section 9 describes various leaf VBTs. (Internally, these VBTs may be composed of many VBTs; however, they do not have any logical children.) These include `ListVBT`, a text browser that displays a scrollable list of items, typically strings; `FileBrowserVBT`, a file browser that displays the files in a directory and allows the user to traverse the directory hierarchy; `NumericVBT`, a widget for specifying an integer in a given range, comprising a type-in field with increment and decrement buttons on its sides, and `ScrollerVBT`, a scrollbar.

Section 10 describes the filters and multi-filters provided by VBTkit. A `FlexVBT` places size constraints on its child. You can give an explicit size (just like Trestle's `RigidVBT`), or add some stretch or shrinkage to the child's preferred size. A `ReactivityVBT` can make its child unresponsive to mouse and keyboard activity, optionally also "graying-out" the child. A `ScaleFilter` magnifies or de-magnifies its descendants. A `ViewportVBT` places scrollbars around the child so the entire contents of the child can be viewed, even if the `ViewportVBT` is allocated less screen space than the child has specified that it needs.

Section 11 describes the splits and multi-splits provided by VBTkit. There is only one: A `SplitterVBT` is like an `HVSplit`, but automatically puts adjusting bars (`HVBars`) between each child. Because a `SplitterVBT` is a multi-split, the client never deals with the `HVBars` (unless it accesses the children of the `SplitterVBT` using the `Split` interface rather than the `MultiSplit` interface, of course).

Section 12 contain some utilities for processing the standard X11 `-geometry` and `-display` command-line arguments.

Section 13 contains the various utility interfaces. The `AnyEvent` interface encapsulates different types of Trestle events into a single parameter. It is needed by clients of `NumericVBT` and `FileBrowserVBT`, since those widgets allow different types of user gestures (e.g., a double-click and a carriage return) to invoke the same callback method. The `AutoRepeat` interface is useful for causing a procedure to be invoked repeatedly at certainly time intervals. The `AutoRepeat` is use for implementing trill buttons (`TrillSwitchVBT` and continuous scrolling (`ScrollerVBT`). The `Rsrc` interface provides functions for an application to locate the resources it needs at runtime. These resources are stored in files, and they include cursors, pixmaps, and text files. You may need to use routines in `Pts` to convert between points (as used by VBTkit text-editing widgets) and millimeters (used by most other widgets). Finally, the `VBTColors` interface is used to record the primary background and foreground colors associated with a `VBT`. This is useful for ensuring that an outline of a subwindow has a good chance of being noticed as it is dragged.

Section 14 describes some of the utilities for manipulating color. Most noteworthy is the `ColorName` interface that takes a description like "Somewhat-MurkyYellow" and returns the red, green, and blue components that will display the color. This section also defines mappings between the RGB and HSV color models.

The Appendices contain material about VBTkit text-editing. **Appendix A** contains a description of the user interface text-editing models support by VBTkit. The models include Emacs, Ivy, Macintosh, and Xterm. If you'd like to customize the editing model, you'll need to use the `TextPortClass` interface in **Appendix A.2**.

2 Composing VBTs

A *multi* is a VBT with *logical children* that may or may not correspond to its children in the VBT hierarchy. There are two types of multis: a *multi-filter* has a single logical child, and a *multi-split* has any number of logical children. Typically, logical children of multi `m` are also VBT-descendants of `m`, but this is not necessary.

Consider how one might implement a check-box widget, `CheckboxVBT`. Logically, a check box is a VBT class with a single child—typically a piece of text that appears to the right of the check box. The box itself is a pixmap whose appearance changes dynamically. This can be implemented by a `TSplit` with two children `PixmapVBT`s. The `TSplit` is placed next to the text using an `HVSplit`, and a `ButtonVBT` is placed around the `HVSplit` in order to make it responsive to mouse clicks. So, the VBT structure is

```
(ButtonVBT
   (HVSplit
      (TSplit PixmapVBT PixmapVBT)
      child))
```

At a VBT level, the `CheckboxVBT` comprises 5 VBTs, plus the child. However, it would suffice for a client of a `CheckboxVBT` to to think in terms of a single child (e.g., the piece of text that appears to the right of the check box).

A multi-filter allows a `CheckboxVBT` to act as a single entity to its clients. The child of a `CheckboxVBT c` is retrieved by calling

```
MultFilter.Child(c)
```

and the call

```
MultiFilter.Replace(c, new)
```

replaces the child of `c` by `new`.

A `SplitterVBT` is a good example of a multi-split. It is built as an `HVSplit` with `HVBar`s automatically inserted between children. The client of a `SplitterVBT` doesn't care about the `HVBars` at all. Invoking

```
MultiSplit.Nth(v,i)
```

retrieves child `i` of the `SplitterVBT` (which is actually child `2*i` of the `HVSplit`), whereas invoking

```
Split.Nth(v,i)
```

retrieves child `i` of the `HVSplit`. Invoking

```
MultiSplit.Delete(v, ch)
```

deletes child `ch` and its adjacent `HVBar`, whereas invoking

```
Split.Delete(v, ch)
```

deletes a single child `ch` of the `HVSplit`.

An important feature of multi-filters and multi-splits is that they allow the implementor of a multi to hide the actual VBT structure from clients. Not only does this simplify the abstraction presented to clients, but it also frees the implementor to change the VBT structure without affect clients. Thus, unless you are a wizard (or you are feeling very lucky), don't use procedures from the `Split` or `Filter` interfaces on a multi; instead use the corresponding procedures from the `MultiSplit` and `MultiFilter` interfaces respectively. The procedures in `MultiSplit` and `MultiFilter` work like their `Split` and `Filter` counterparts if the argument is not a multi.

We document the fact that a VBT class is a multi-split or multi-filter using the `SUBTYPE` pragma. In general, this is intended to suggest a subtype relationship where none actually exists (because the Modula-3 type system does not have multiple inheritance).

For example, here's what the `CheckboxVBT` interface would look like:

```
INTERFACE CheckboxVBT;
IMPORT ButtonVBT, VBT;
TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = ButtonVBT.T OBJECT METHODS
    <* LL <= VBT.mu *>
    init (ch: VBT.T):T
  END;
...
END CheckboxVBT.
```

The pragma indicates that a `CheckBoxVBT` is a kind of multi-filter, i.e., that it has one logical child. In fact, it is actually a subtype of `ButtonVBT.T`.

Clients wishing to implement their own multi-filters or multi-splits should refer to the `MultiClass` interface.

2.1 The MultiSplit Interface

The `MultiSplit` interface defines operations that are common to all multi-splits, such as enumerating and deleting children.

```
INTERFACE MultiSplit;

IMPORT Point, VBT;

EXCEPTION NotAChild;
```

```
TYPE T = VBT.T;
```

A `MultiSplit.T` is a `VBT.T` with a `MultiClass.Split` in its property set.

All of the procedures in this interface can accept either a `MultiSplit.T` or a `Split.T` as the first argument. If the first argument is not a `MultiSplit.T`, the procedure just calls the corresponding procedure in the `Split` interface, re-raising any `Split.NotAChild` exceptions as `NotAChild` exceptions.

Unlike the procedures in the `Split` interface, the procedures here do not perform any VBT operations. For example, `Split.Delete(v, ch)` deletes the child `ch` of split `v`, detaches `ch`, and marks `v` for redisplay, whereas `MultiSplit.Delete` just deletes the multi-child `ch` of multi-split `v`, without detaching `ch` or marking `v` for redisplay. The `MultiClass` methods of `v` that implement the `Delete` functionality will most likely manipulate the VBT tree using `Split.Delete` (or other calls to `Split` and `Filter` as appropriate), so that `v` will be marked and `ch` will be detached, as one would expect.

```
PROCEDURE Succ (v: VBT.T; ch: VBT.T): VBT.T
  RAISES {NotAChild};
<* LL >= {VBT.mu} *>
```

Return the child of `v` that follows the child `ch`.

The successor of `NIL` is the first child; the successor of the last child is `NIL`; the successor of `NIL` is `NIL` if there are no children.

```
PROCEDURE Pred (v: VBT.T; ch: VBT.T): VBT.T
  RAISES {NotAChild};
<* LL >= {VBT.mu} *>
```

Return the child of `v` that precedes the child `ch`.

More precisely, `Pred(v,ch) = x` iff `Succ(v,x) = ch`.

```
PROCEDURE NumChildren (v: VBT.T): CARDINAL
  RAISES {NotAChild};
<* LL >= {VBT.mu} *>
```

Return the number of children of `v`.

```
PROCEDURE Nth (v: VBT.T; n: CARDINAL): VBT.T;
<* LL >= {VBT.mu} *>
```

Return the child of `v` with index `n`.

More precisely, `Nth(v, n)` is the child of `v` with `n` predecessors, or `NIL` if `v` has at most `n` children.

```
PROCEDURE Index (v: VBT.T; ch: VBT.T): CARDINAL
  RAISES {NotAChild};
```

```
<* LL >= {VBT.mu} *>
```

Return the index of `v`'s child `ch`.

In other words, `Index(v, ch)` is the value `n` such that `Nth(v, n) = ch`. It is always true that `Index(v, NIL)` equals `NumChildren(v)`.

```
PROCEDURE Locate (v: VBT.T; READONLY pt: Point.T): VBT.T;
<* LL.sup = VBT.mu *>
```

Return the child of `v` that would receive a mouse click at point `pt`, or `NIL` if there is no such child.

```
PROCEDURE Delete(v: T; ch: VBT.T)
  RAISES {NotAChild};
<* LL.sup = VBT.mu *>
```

Delete the child `ch` of `v`.

```
PROCEDURE Replace (v: VBT.T; ch, new: VBT.T)
  RAISES {NotAChild};
<* LL.sup = VBT.mu *>
```

Replace child `ch` of `v` with `new`.

```
PROCEDURE Insert (v: VBT.T; pred, new: VBT.T)
  RAISES {NotAChild};
<* LL.sup = VBT.mu *>
```

Add `new` as a child of `v` following `pred`.

Some multi-splits can accommodate only a bounded number of children. Whenever `Insert(v,pred,new)` is applied to a multi-split `v` that cannot accommodate an additional child, then `pred` (or the original first child, if `pred=NIL`) is deleted from the multi-split. The precise semantics are defined by the individual multi-splits.

```
PROCEDURE Move (v: VBT.T; pred, ch: VBT.T)
  RAISES {NotAChild};
<* LL.sup = VBT.mu *>
```

Move child `ch` of `v` to follow `pred`. `ch` and, if non-`NIL`, `pred`, must be children of `v`.

```
PROCEDURE AddChildArray (
    v: VBT.T;
    READONLY new: ARRAY OF VBT.T);
<* LL.sup = VBT.mu *>
```

Insert the non-`NIL` elements of `new` at the end of `v`'s list of children.

Procedure `AddChildArray` is equivalent to

```
    pred := Pred(v, NIL);
    FOR i := FIRST(new) TO LAST(new) DO
      IF new[i] # NIL THEN
        Insert(v, pred, new[i]);
        pred := new[i]
      END
    END

PROCEDURE AddChild (
    v: VBT.T;
    n0, n1, n2, n3, n4, n5, n6, n7, n8, n9: VBT.T := NIL);
<* LL.sup = VBT.mu *>
```

Insert the non-`NIL` *parameters as children to* `v`.

Procedure `AddChild` is equivalent to

```
    AddChildArray(v,
      ARRAY OF VBT.T{n0, n1, n2, n3, n4, n5, n6, n7, n8, n9})

END MultiSplit.
```

2.2 The MultiFilter Interface

The `MultiFilter` interface defines the functionality that is common to all clients of multi-filters; namely, retrieving and changing a multi-filter's multi-child.

A multi-filter is a multi-split with at most one child. Thus, you can use the procedures in the `MultiSplit` interface on a VBT that is a multi-filter. The semantics of the `MultiSplit` procedures on a multi-filter should be obvious, with the following exceptions: `MultiSplit.Move` on a multi-filter is a no-op, and `MultiSplit.Insert` on a multi-filter replaces the child, if any.

```
INTERFACE MultiFilter;

IMPORT VBT;

TYPE T = VBT.T;
```

A `MultiFilter.T` *is a* `VBT.T` *with a* `MultiClass.Filter` *in its property set.*

The following procedures can accept either a `MultiFilter.T` or a `Filter.T` as the first argument. If the first argument is not a `MultiFilter.T`, the procedure just calls the corresponding procedure in the `Filter` interface.

```
PROCEDURE Child (v: VBT.T): VBT.T;
<* LL.sup = VBT.mu *>
```

Return the child of `v`*, or* `NIL` *if there is no child.*

```
PROCEDURE Replace (v, ch: VBT.T): VBT.T;
<* LL.sup = VBT.mu *>
```

Replace `v`*'s child by* `ch` *and return* `v`*'s old child.*

`MultiFilter.Replace` is similar to `MultiSplit.Replace`, except that it returns the old multi-child instead of taking the old multi-child as an argument, and if `ch` is `NIL` it is similar to `MultiSplit.Delete`.

```
END MultiFilter.
```

2.3 The MultiClass Interface

An arbitrary VBT is made into a multi by providing a set of methods for maintaining the logical structure. The methods are used for replacing, inserting, traversing, and performing other common operations on the children.

In a language with multiple inheritance, multis would simply inherit different methods from different parent-types. In Modula-3, however, we achieve this effect by creating an instance `mc` of type `MultiClass.T`, and attaching `mc` to a VBT `v` by way of `v`'s property set. The object `mc` points back to `v` via the field `mc.vbt`.

Clients defining their own multis can make a VBT `v` "into" a multi by calling `Be(v,mc)` during the initialization of the VBT. They must call `BeChild` on each new child when it is inserted, and `UnChild` when a child of a multi is deleted. `MultiFilter.Replace`, `MultiSplit.Replace`, and `MultiSplit.Insert` all do this automatically, and `MultiSplit.Insert` calls `BeChild`.

```
INTERFACE MultiClass;

IMPORT RefList, VBT;

TYPE
  T = ROOT OBJECT
        vbt: VBT.T;               (* READONLY *)
      METHODS
        <* LL = VBT.mu *>
        replace (ch, new: VBT.T);
        insert  (pred, new: VBT.T);
        move    (pred, ch: VBT.T);
        succ    (ch: VBT.T): VBT.T;
        pred    (ch: VBT.T): VBT.T;
        nth     (n: CARDINAL): VBT.T;
        index   (ch: VBT.T): CARDINAL;
      END;
```

2.3.1 The MultiSplit methods

The methods implement the behavior in the `MultiSplit` interface.

The method call `mc.replace(ch,new)` implements the operation

```
MultiSplit.Replace(mc.vbt, ch, new)
```

and the call `mc.replace(ch,NIL)` implements

```
MultiSplit.Delete(mc.vbt, ch)
```

Before calling the method, the generic code in the `MultiSplit` interface checks that `ch` is a multi-child of `mc.vbt`, and, if `new` is not `NIL`, calls `BeChild(mc.vbt, new)`. After calling the method, the generic code calls `UnChild(mc.vbt, ch)`, if `ch` was not `NIL`.

Similarly, the method call `mc.insert(pred,new)` implements the operation

```
MultiSplit.Insert(mc.vbt, pred, new)
```

Before calling the method, the generic code in `MultiSplit` checks that `pred` is a multi-child of `mc.vbt` and calls `BeChild(mc.vbt, new)`. If `new` is `NIL`, `MultiSplit.Insert` raises a runtime exception.

The default methods for `replace` and `insert` are both equal to `NIL`, so every multi-split needs to override these methods.

The method call `mc.move(pred, ch)` implements

```
MultiSplit.Move(mc.vbt, pred, ch)
```

Before calling the method, the generic code in `MultiSplit` verifies that `ch` and `pred` are both multi-children of `mc.vbt` (or `NIL`, in the case of `pred`). The call to `mc.move` is avoided if `pred=ch` or `mc.succ(pred)=ch`.

The default `move` method for a `MultiClass.T` object `mc` is simply a call to `mc.replace(ch, NIL)` followed by a call to `mc.insert(pred, ch)`.

This default method is naive on two fronts. One, it is not particularly efficient since the tree of VBTs is typically being manipulated twice. Two, and more importantly, some multi-splits will take action as part of the `replace` method (e.g., reallocating the screen layout of its children) that is not "undone" by the subsequent call to the `insert` method.

The method calls

```
mc.succ(ch)
mc.pred(ch)
mc.nth(n)
mc.index(ch)
```

all implement the corresponding operations in the `MultiSplit` interface. The default `pred`, `nth` and `index` methods are implemented by repeatedly calling the `succ` method. The default `succ` method finds the successor of `ch` for the `MultiClass.T` object `mc` by a depth-first walk of `mc.vbt`'s descendants, starting after `ch`, and stopping at the first VBT `w` for which `IsChild(mc.vbt, w)` returns `TRUE`, or when all of `mc.vbt`'s descendants have been visited, in which case, `ch` has no successor so `NIL` is returned. In practice, the default `succ` method seems to work nearly all of the time; however, there is often a more efficient way to implement a `succ` method for any particular multi-split.

2.3.2 The MultiFilter methods

```
TYPE
  Split <: T;
  Filter <: Split;
```

The default methods for a `Filter` are the same as for a `Split`, except that the `insert` method has a default. Thus, you only need to override the `replace` method of a multi-filter.

The default method call `mc.insert(pred, new)` is

```
mc.replace (mc.succ(pred), new)
```

Also, the `move` method is never run; the generic code in `Split.Move` ensures this.

2.3.3 Procedures for creating multis

```
PROCEDURE Be (v: VBT.T; mc: T);
<* LL.sup <= VBT.mu *>
```

Make `v` into a multi by storing `mc` on `v`'s property set and setting `mc.vbt` to `v`.

```
PROCEDURE Resolve (v: VBT.T): T;
<* LL.sup < v *>
```

Return the multiclass of `v`, that is, the `mc` for which `Be(v,mc)` was previously called. Return `NIL` if there is no such `mc`.

```
PROCEDURE BeChild (v: VBT.T; ch: VBT.T);
<* LL.sup < ch *>
```

Make `ch` into one of `v`'s children that is exposed to the client via the `MultiSplit` or `MultiFilter` interfaces. It is possible for `ch` to be a child of more than one multi, and it is possible that `ch` is not related to `v` in the VBT hierarchy.

```
PROCEDURE UnChild (v: VBT.T; ch: VBT.T);
<* LL.sup < ch *>
```

Unmark `ch` as one of `v`'s children that is exposed to the client via the `MultiSplit` or `MultiFilter` interfaces.

```
PROCEDURE IsChild (v: VBT.T; ch: VBT.T): BOOLEAN;
<* LL.sup < ch *>
```

Return `TRUE` iff `BeChild(v,ch)` was previously invoked and `UnChild(v,ch)` has not been subsequently called.

```
PROCEDURE Parents (ch: VBT.T): RefList.T (* of VBT.T *);
<* LL.sup < ch *>
```

Return a list of VBTs for which `IsChild(v,ch)` is `TRUE`. The list may be `NIL`.

```
END MultiClass.
```

3 The 3-dimensional look and feel

This section describes the facilities for giving a "three-dimensional" look. (See Kobara [2] for information on this style.) The basic data types are defined in the `Shadow` interface. A `ShadowedVBT` displays a 3-D border around a VBT, and a `ShadowedBarVBT` displays a horizontal or vertical line in 3-D.

3.1 The Shadow Interface

The `Shadow` interface contains the basic definitions for VBT classes that implement a Motif-like, 3-D look. There are two basic primitives: a 3-D border, and a 3-D vertical or horizontal line. The style, size, and colors of the shadow are specified by data structures defined in this interface.

A 3-D border can give the visual illusion of "raising" an object above the background, "lowering" an object into the background, drawing a "ridge" above the background, or chiseling a "groove" into the background. A 3-D line has the visual effect of being either a "ridge" above the background or a "groove" chiseled into the background (see Figure 3.1).

These visual effects are actually quite simple to accomplish by drawing parts of the 3-D border or 3-D line using a dark variant of the background color, and by drawing other parts using a light variant of the background color.

For example, to give the impression that an object is raised above its background, the north and west borders are drawn using a light color, whereas the south and east border are drawn in a dark color. To draw a "ridge," the north and west shadows start out in the light color, and, halfway, switch to the dark color. Analogously, the south and east shadows start out dark and switch to a light color.

The following chart summarizes the visual effects:

Style	*North/West*	*South/East*
Flat	Background	Background
Raised	Light	Dark
Lowered	Dark	Light
Ridged	Light/Dark	Dark/Light
Chiseled	Dark/Light	Light/Dark

For maximum effectiveness, the child's background should be a color whose saturation level is about 50%, and the light and dark shadows should be colors with the same hue and lightness, but with saturation levels of 25% and 75% respectively.

On a monochrome display, the 3-D borders and lines appear flat and 50% of the size they'd be on non-monochrome displays. Also, those VBTkit widgets that use 3-D borders for feedback (say, a button that gives the effect of lowering its contents when depressed) are implemented in such a way as to give feedback in a non-3-D manner (e.g., the `ShadowedFeedbackVBT` interface in Section 4.2).

You can force VBTkit widgets to use a non-3-D style of feedback by specifying a shadow size that is negative. Such widgets will draw borders and lines with 50% of the absoluate value of the shadow size. (You should also be sure to set the light and dark shadow to be the same as the foreground color.)

```
INTERFACE Shadow;

IMPORT PaintOp, VBT;

TYPE
  T = PaintOp.ColorScheme OBJECT
        size: REAL;
        light, dark, both, reversed: PaintOp.T;
      END;

TYPE
  Style = {Flat, Raised, Lowered, Ridged, Chiseled};

PROCEDURE New (size : REAL      := 0.5;
               bg   : PaintOp.T := PaintOp.Bg;
               fg   : PaintOp.T := PaintOp.Fg;
               light: PaintOp.T := PaintOp.Fg;
               dark : PaintOp.T := PaintOp.Fg): T;
<* LL = arbitrary *>
```

Return a newly allocated `Shadow.T`. The `size`, `light`, and `dark` fields of the new `Shadow.T` are copies of the parameters, respectively. The `both` field is computed from `PaintOp.Pair(light, dark)`, and the `reversed` field is computed from `PaintOp.Pair(dark, light)`.

The `size` is specified in millimeters. All of the paint ops must be tints, arranged so that on a monochrome screen `bg` draws in background, while `fg`, `light`, and `dark` draw in foreground.

```
PROCEDURE Supported (shadow: T; v: VBT.T): BOOLEAN;
<* LL.sup < v *>
```

Return whether `shadow` should appear 3-D on `v`. Two conditions must hold: `v` must be on screen whose depth is greater than 1, and `shadow.size` must be positive.

Finally, we have the definition for a “default” shadow:

```
VAR (* CONST *) None: T;
```

This variable is really a constant for

```
New(0.0, PaintOp.Bg, PaintOp.Fg, PaintOp.Fg, PaintOp.Fg)
```

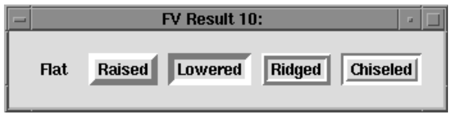

Figure 1: *ShadowStyles, with size = 4 points.*

Because `None` is not a constant, it cannot be the default value of a procedure argument. Therefore, we adopt the following convention: when a parameter whose type is `Shadow.T` has a default value of `NIL`, the procedure will use `Shadow.None` instead.

```
END Shadow.
```

3.2 The ShadowedVBT Interface

A `ShadowedVBT.T` is a filter whose parent's screen consists of the child's screen surrounded by a 3-D border. The style, size, and colors of the shadow can be set dynamically. The parent's shape is determined from the child's shape by adding the size of the shadow.

```
INTERFACE ShadowedVBT;

IMPORT Filter, Shadow, VBT;

TYPE
  T <: Public;
  Private <: Filter.T;
  Public = Private OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (ch: VBT.T;
            shadow: Shadow.T := NIL;
            style: Shadow.Style := Shadow.Style.Flat): T;
    END;
```

The call `v.init(...)` initializes `v` as a `ShadowedVBT` with child `ch` and the given `shadow` and `style`. When `Shadow.Support(shadow, v)` is `TRUE`, each dimension of `v` exceeds the corresponding dimension of `ch` by `2 * ABS(shadow.size)`; otherwise, each dimension of `v` exceeds the corresponding dimension of `ch` by `2 * ABS(shadow.size/2)`. If `shadow=NIL`, it defaults to `Shadow.None`.

```
PROCEDURE Set (v: T; shadow: Shadow.T);
<* LL.sup = VBT.mu *>
```

Change the size and colors of `v`'s shadow and mark `v` for redisplay.

```
PROCEDURE SetStyle (v: T; style: Shadow.Style);
<* LL.sup = VBT.mu *>
```

Change the style of `v`'s shadow, and mark `v` for redisplay.

```
PROCEDURE Get (v: T): Shadow.T;
<* LL.sup = VBT.mu *>
```

Return `v`'s shadow.

```
PROCEDURE GetStyle (v: T): Shadow.Style;
<* LL.sup = VBT.mu *>
```

Return `v`'s shadow style.

```
END ShadowedVBT.
```

3.3 The ShadowedBarVBT Interface

A `ShadowedBarVBT.T` is a leaf-VBT that displays a horizontal or vertical 3-D line.

The following chart summarizes the visual effects:

Style	*top (vertical)* *left (horizontal)*	*bottom (vertical)* *right (horizontal)*
Flat	Background	Background
Raised	Light	Dark
Lowered	Dark	Light
Ridged	Light	Dark
Chiseled	Dark	Light

```
INTERFACE ShadowedBarVBT;

IMPORT Axis, Shadow, VBT;

TYPE
  T <: Public;
  Public = VBT.Leaf OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (axis  : Axis.T;
                   shadow: Shadow.T := NIL;
                   style           := Shadow.Style.Flat): T;
```

```
                END;
```

The call `v.init(...)` initializes `v` as a `ShadowedBarVBT` with the `axis` orientation and with the given `shadow` and `style`. The default `shadow` is `Shadow.None`. If the `shadow.size` along the `axis` dimension results in an odd number of pixels, the extra pixel goes to the top half.

When `Shadow.Supported(shadow, v)` is `TRUE`, the shape of `v` is `ABS(shadow.size)` in the primary axis, and unconstrained in the other dimension. Otherwise, the shape of `v` in the primary axis is `ABS(shadow.size/2)`.

```
PROCEDURE Set (v: T; shadow: Shadow.T);
<* LL.sup = VBT.mu *>
```

Change the size and colors of `v`'s shadow and mark `v` for redisplay.

```
PROCEDURE SetStyle (v: T; style: Shadow.Style);
<* LL.sup = VBT.mu *>
```

Change the style of `v`'s shadow and mark `v` for redisplay.

```
END ShadowedBarVBT.
```

4 Providing Visual Feedback

4.1 The FeedbackVBT Interface

A `FeedbackVBT` is a filter that provides some visual feedback for its child.

The essence of a `FeedbackVBT` are its `normal` and `excited` methods. The `normal` method is intended for giving permanent feedback, whereas the `excited` method is used for displaying transitory feedback (e.g., while a button is pressed). In addition, a feedback maintains a *state* flag to distinguish between an "on" and "off" state (e.g., for use by a `BooleanVBT`).

Clients should not invoke a `FeedbackVBT`'s `normal` and `excited` methods directly. Instead, use the procedures `Normal` and `Excited` in this interface. The state of a `FeedbackVBT` is set using the `SetState` procedure; it is queried using the procedure `GetState`.

The default `normal` and `excited` methods are no-ops. A `FeedbackVBT` by itself is not very useful; subtypes are expected to override these methods with something useful. Also, VBTkit switches that use `FeedbackVBTs` assume that the `FeedbackVBT` is a multi-filter, not simply a filter.

```
INTERFACE FeedbackVBT;

IMPORT Filter, VBT;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (ch: VBT.T): T;
             <* LL = VBT.mu *>
             normal  ();
             excited ();
           END;
```

The call `v.init(ch)` initializes `v` as a `FeedbackVBT` with VBT child `ch`. The default `normal` and `excited` methods are no-ops.

```
PROCEDURE Normal (v: T);
<* LL.sup = VBT.mu *>
```

Invoke v's `normal` method.

```
PROCEDURE Excited (v: T);
<* LL.sup = VBT.mu *>
```

Invoke v's `excited` method.

```
PROCEDURE SetState (v: T; state: BOOLEAN);
```

```
<* LL.sup = VBT.mu *>
```

Record the `state` and then invoke whichever of `v`'s methods, `normal` or `excited`, was most recently invoked. If neither method has ever been invoked, the `normal` method is invoked.

```
PROCEDURE GetState (v: T): BOOLEAN;
<* LL.sup = VBT.mu *>
```

Return the value of the most recent call to `SetState`. The initial state is `FALSE`.

```
END FeedbackVBT.
```

4.2 The ShadowedFeedbackVBT Interface

A `ShadowedFeedbackVBT` is a multi-filter feedback that displays a 3-D border as visual feedback to another VBT.

```
INTERFACE ShadowedFeedbackVBT;

IMPORT FeedbackVBT, Shadow, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public =
    FeedbackVBT.T OBJECT
    METHODS
      <* LL <= VBT.mu *>
      init (ch             : VBT.T;
            shadow         : Shadow.T := NIL;
            onStyle                   := Shadow.Style.Lowered;
            onExcitedStyle            := Shadow.Style.Raised;
            offStyle                  := Shadow.Style.Raised;
            offExcitedStyle           := Shadow.Style.Lowered): T
    END;
```

The call `v.init(ch, shadow, ...)` initializes `v` as a `ShadowedFeedbackVBT`. The internal structure of `v` includes a `ShadowedVBT` for displaying the shadow `shadow` around `ch`. The default `normal` and `excited` methods change the style of the shadow, taking into account the state of `v`. For example, when `FeedbackVBT.GetState(v)` returns `FALSE`, the `excited` method uses the value of `offExcitedStyle`.

On a monochrome screen (whenever `Shadow.IsSupport(v, shadow)` is false), `ch` is inverted by the default `normal` method when the state is "on" and by the `excited` method when the state is "off."

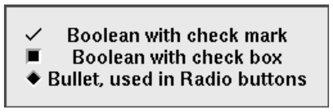

Figure 2: *MarginFeedbackVBTs*

The default parameters to the `init` method generate a feedback that is appropriate for buttons. The following procedure generates a feedback that is appropriate for use by menu buttons:

```
PROCEDURE NewMenu (ch: VBT.T; shadow: Shadow.T := NIL): T;
<* LL <= VBT.mu *>
```

Return a `ShadowedFeedbackVBT` appropriate for menu buttons. The `normal` method always uses `Shadow.Style.Flat`; the `excited` method always uses `Shadow.Style.Lowered`.

```
END ShadowedFeedbackVBT.
```

4.3 The MarginFeedbackVBT Interface

A `MarginFeedbackVBT` is a multi-filter feedback that provides visual feedback to the left of another VBT. This interface defines a handful of useful "left-hand sides."

```
INTERFACE MarginFeedbackVBT;

IMPORT FeedbackVBT, Shadow, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = FeedbackVBT.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (ch, marginVBT: VBT.T): T
           END;
```

The following procedures create some popular types of `MarginFeedbackVBTs`. See Figure 4.3.

```
PROCEDURE NewCheck  (ch: VBT.T; shadow: Shadow.T := NIL): T;
<* LL.sup <= VBT.mu *>

PROCEDURE NewBox (ch: VBT.T; shadow: Shadow.T := NIL): T;
```

```
<* LL.sup <= VBT.mu *>

PROCEDURE NewBullet (ch: VBT.T; shadow: Shadow.T := NIL): T;
<* LL.sup <= VBT.mu *>

END MarginFeedbackVBT.
```

4.4 The BiFeedbackVBT Interface

A `BiFeedbackVBT` is a multi-filter feedback that is used for composing two arbitrary feedbacks. The default `normal` and `excited` methods of a `BiFeedbackVBT` invoke the corresponding methods on the two feedbacks. The `BiFeedbackVBT` itself doesn't have any visual appearance.

```
INTERFACE BiFeedbackVBT;

IMPORT FeedbackVBT, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = FeedbackVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (f1, f2: FeedbackVBT.T; ch: VBT.T): T;
           END;
```

The call `v.init(f1, f2, ch)` initializes `v` as a `BiFeedbackVBT`. The multi-child of `v` is `ch`. The internal structure of `v` is as follows: The VBT-child of `v` is `f1`, the multi-child of `f1` is `f2`, and the multi-child of `f2` is `ch`. (Recall that it is legal and meaningful for a VBT to have multiple multi-parents, as `ch` will have.) When the `init` method is called, both `f1` and `f2` must be childless.

```
END BiFeedbackVBT.
```

5 Buttons

Buttons in VBTkit differ from buttons in Trestle (i.e., `ButtonVBT` and its subclasses). VBTkit buttons are referred to as *switches*.

There are three primary differences:

1. Trestle buttons are passed an `action` procedure. VBTkit switches define a `callback` method, which makes it easier to define a subclass of a switch. The `callback` method is invoked whenever the `action` procedure would be.
2. Trestle buttons provide visual feedback as part of their `pre`, `cancel`, and `post` methods. VBTkit switches assume that their `VBT`-child is a feedback multi-filter; the `pre`, `cancel`, and `post` methods invoke the feedback's `normal` and `excited` methods appropriately. This facilitates a "mix and match" of button-styles with gestures (e.g., putting a radio button within a menu).
3. Trestle buttons are filters. VBTkit switches are multi-filters. The multi-child of the feedback is the switch's multi-child. This makes is it easy to create buttons with sophisticated visual feedback, while retaining the model that a button is a "filter with an arbitrary child." Clients should use the `MultiFilter` interface to access the arbitrary multi-child of a switch, and the `Filter` interface to access the feedback.

A switch `s` follows these three conventions:

1. `s` has a `callback` method. This method will be invoked whenever a button would have invoked its `action` procedure.
2. The VBT-child of `s` is a feedback, `f`. The default methods are as follows: `s.pre()` invokes `Feedback.Excited(f)`, and `s.post()` and `s.cancel()` both invoke `Feedback.Normal(f)`.
3. `s` is also a multi-filter. Its multi-child is defined to be the multi-child of `f`.

Although all VBTkit switches follow these conventions and are subtypes of `ButtonVBT.T`, few are subtypes of `SwitchVBT.T`. A `MenuSwitchVBT`, for example, is a subtype of `MenuBtnVBT.T`.

This section defines the following switches:

- `SwitchVBT`, `MenuSwitchVBT`, and `QuickSwitchVBT` are switch versions of basic Trestle buttons. (See the *Trestle Reference Manual* [3], sections 5.6–5.9.)

- `AnchorSplit` is a switch version of Trestle's `AnchorBtnVBT`. There, the "menu" is a data field, and "anchor" is the child of the button, but in an `AnchorSplit`, which is a multi-split, the "menu" and the "anchor" are children.
- A `TrillSwitchVBT` is an auto-repeating version of a basic button. Trestle has no counterpart.
- A `GuardedBtnVBT` is button that forces the user to click to remove the guard. Trestle has no counterpart.
- A `SourceVBT` implements VBTkit's "drag-and-drop" buttons. Trestle has no counterpart.
- `BooleanVBT` and `ChoiceVBT` implement check boxes and radio buttons, respectively. Trestle has no counterparts.

5.1 The SwitchVBT Interface

A `SwitchVBT` is a switch version of Trestle's `ButtonVBT`.

```
INTERFACE SwitchVBT;

IMPORT ButtonVBT, FeedbackVBT, MultiClass, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = ButtonVBT.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (f: FeedbackVBT.T): T;
             <* LL.sup = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;
```

The call `v.init(f)` initializes `v` as a `SwitchVBT` with child `f`. The multi-child of `f` is marked as `v`'s multi-child too.
The default `callback` method is a no-op.

The following type is useful for creating switches that have the same internal structure as a `SwitchVBT.T`; namely, a `Filter.T` whose child is a `FeedbackVBT.T`.

```
TYPE
  MC <: MultiClass.Filter;
```

The following procedures are useful for some VBTkit switches to use as their default `ButtonVBT` methods:

```
PROCEDURE Pre (v: ButtonVBT.T);
<* LL.sup = VBT.mu *>
```

Equivalent to: `Feedback.Excited (Filter.Child(v))`

```
PROCEDURE Post (v: ButtonVBT.T);
<* LL.sup = VBT.mu *>
```

Equivalent to: `Feedback.Normal (Filter.Child(v))`

```
PROCEDURE Cancel (v: ButtonVBT.T);
<* LL.sup = VBT.mu *>
```

Equivalent to: `Feedback.Normal (Filter.Child(v))`

```
END SwitchVBT.
```

5.2 The QuickSwitchVBT Interface

A `QuickSwitchVBT` is a switch version of Trestle's `QuickBtnVBT`.

```
INTERFACE QuickSwitchVBT;

IMPORT FeedbackVBT, QuickBtnVBT, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = QuickBtnVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (f: FeedbackVBT.T): T;
             <* LL = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;

END QuickSwitchVBT.
```

5.3 The MenuSwitchVBT Interface

A `MenuSwitchVBT` is a switch version of Trestle's `MenuBtnVBT`.

```
INTERFACE MenuSwitchVBT;
```

```
IMPORT FeedbackVBT, MenuBtnVBT, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = MenuBtnVBT.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (f: FeedbackVBT.T): T;
             <* LL.sup = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;

END MenuSwitchVBT.
```

5.4 The AnchorSplit Interface

An `AnchorSplit` is a multi-split version of `AnchorBtnVBT`. The first child is the *anchor* that is displayed (such as a text string or an icon). The second child is the *menu* that is displayed when the anchor is activated. Attempts to give an anchor-split more than two children cause the extra children to be lost.

At initialization time, the feedback for the anchor is specified. It must be a childless multi-filter. Also at initialization time, a frame is specified that will surround the menu. The frame is also a childless multi-filter.

```
INTERFACE AnchorSplit;

IMPORT AnchorBtnVBT, FeedbackVBT, MultiFilter, VBT;

TYPE
  <* SUBTYPE T <: MultiSplit.T *>
  T <: Public;
  Public = AnchorBtnVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (f           : FeedbackVBT.T;
                   menuFrame   : MultiFilter.T;
                   n           : CARDINAL        := 0;
                   anchorParent: VBT.T           := NIL;
                   hfudge                        := 0.0;
                   vfudge                        := 0.0  ): T;
           END;
```

The call `v.init(...)` initializes `v` as an `AnchorSplit`. The feedback `f` and the multi-filter `menuFrame` must have no multi-children. That is, calling `MultiFilter.Child(f)` and `MultiFilter.Child(menuFrame)` must both return NIL. The other parameters, n, anchorParent, hfudge, and vfudge are the same as in AnchorBtnVBT.

```
END AnchorSplit.
```

5.5 The TrillSwitchVBT Interface

A TrillSwitchVBT.T is a switch version of Trestle's TrillBtnVBT.

Actually, a TrillBtnVBT does not exist. If it existed, it would be a button that generates events repeatedly while the mouse is down and in its domain. When the mouse leaves the domain, events generation is suspended until the mouse returns.

The implementation uses the AutoRepeat interface for repeatedly generating events. That interface defines the parameters that control how frequently events are generated, and how long to wait before starting to auto-repeat.

```
INTERFACE TrillSwitchVBT;

IMPORT ButtonVBT, FeedbackVBT, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = ButtonVBT.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (f: FeedbackVBT.T): T;
             <* LL.sup = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;

END TrillSwitchVBT.
```

5.6 The GuardedBtnVBT Interface

A GuardedBtnVBT protects its child against unintentional mouse clicks. While the guard is displayed, mouse clicks are not forwarded. To remove the guard, click on the button. The guard is restored after the next upclick, chord-cancel, or when the mouse leaves the domain of the VBT.

Typically, a GuardedBtnVBT is placed above a "dangerous" button, like one that terminates an application. This forces the user to click twice to terminate the application—the first time to remove the guard, and the second time to invoke the button that terminates the application.

A GuardedBtnVBT is much closer to being a VBTkit switch than a Trestle button. There's a callback method (invoked when the guard is removed), and the guard is a multi-filter. However, the client does not provide a feedback; it is hard-wired into the `GuardedBtnVBT` implementation.

```
INTERFACE GuardedBtnVBT;

IMPORT ButtonVBT, PaintOp, VBT;

TYPE
  T <: Public;
  <* SUBTYPE T <: MultiFilter.T *>
  Public =
    ButtonVBT.T OBJECT
    METHODS
      <* LL <= VBT.mu *>
      init (ch: VBT.T; colors: PaintOp.ColorScheme := NIL): T;
      <* LL = VBT.mu *>
      callback (READONLY cd: VBT.MouseRec);
    END;

END GuardedBtnVBT.
```

5.7 The SourceVBT Interface

A `SourceVBT` is used to implement a "drag-and-drop" paradigm. The object being dragged is the *source* and an object into which the source may be dropped is the *target*.

As a subclass of `ButtonVBT`, a `SourceVBT` has `pre`, `post`, and `cancel` methods. In addition, it has `during`, `callback`, and `hit` methods. The methods are called as follows: The `pre` method is invoked on the first click in the VBT; the `post` method is called on an uncanceled upclick; the `cancel` method is called whenever the mouse is chord-canceled; the `during` method is called whenever the mouse has moved (and remained on the same screen) since the last call to `during` or `pre`. A new VBT cage containing the current cursor position will be set before calls to `pre` and `during`. The `callback` method is called after the `post` method, as long as the mouse was over an "acceptable target" when the upcplick happened.

The heart of drag-and-drop is implemented by the default `during` method: Recall that the `during` method is invoked each time the mouse moves while the button is down and not chord-cancelled. The default `during` method looks to see if the mouse is over a VBT marked as a *target*. If so, then the `SourceVBT`'s `hit` method is invoked to see if the target is acceptable for the source. If so, an `excited` method on the target is invoked to give feedback, and eventually, a target's `normal` method is called to remove the feedback. If the target is not acceptable, nothing happens.

```
INTERFACE SourceVBT;
```

```
IMPORT ButtonVBT, FeedbackVBT, HighlightVBT, PaintOp, VBT;
```

5.7.1 Sources

```
TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public =
    ButtonVBT.T OBJECT
    METHODS
      <* LL <= VBT.mu *>
      init (f: FeedbackVBT.T): T;
      <* LL = VBT.mu *>
      during   (READONLY cd: VBT.PositionRec);
      callback (READONLY cd: VBT.MouseRec);
      hit (target: VBT.T; READONLY cd: VBT.PositionRec):
           BOOLEAN;
    END;
```

The call `v.init(...)` initializes `v` as a `SourceVBT`. The default `pre` method changes the cursor to a starburst and calls `SwitchVBT.Pre`. The default `during` method calls the `hit` method whenever it is on a location controlled by a VBT that is a target. If the `hit` method returns `TRUE`, the target's `excited` method is called. As the mouse moves from target to target, the previous trarget's `normal` method is called before another target's `excited` method is invoked. The `post` and `cancel` methods invoke the current target's `normal` method, restore the original cursor, and call `SwitchVBT.Post` and `SwitchVBT.Cancel` respectively. It's guaranteed that a target's `excited` and `normal` methods are called in non-nested pairs.

The default `hit` method always returns `TRUE`. The default `during` and `callback` methods are no-ops.

```
PROCEDURE GetTarget (v: T): Target;
<* LL.sup = VBT.mu *>
```

If the mouse is not over a valid target, or if the most recent call to `v.hit(target, cd)` returned `FALSE`, then return `NIL`; otherwise return `target`. This procedure is intended to be called by a `callback` method to find out the current target.

5.7.2 Targets

```
TYPE Target = VBT.T;
```

A target is a VBT on which `BeTarget` has been invoked.

```
TYPE
  TargetClass <: TargetClassPublic;
  TargetClassPublic =
    ROOT OBJECT
      vbt: VBT.T;  (* READONLY; set by BeTarget *)
      source: T;   (* READONLY; for use by normal/excited *)
    METHODS
      <* LL = VBT.mu *>
      normal  ();
      excited ();
    END;
```

A `TargetClass` determines the feedback when a target's `excited` method is called. The `source` field can be read by the `normal` and `excited` methods, but clients may find `GetSource` more convenient to use.

The default `normal` and `excited` methods are no-ops.

```
PROCEDURE BeTarget (w: VBT.T; class: TargetClass);
<* LL.sup < w *>
```

Make `w` into a target for a `SourceVBT`. As a target, `w` may be passed to some `SourceVBT`'s `hit` method.

```
PROCEDURE TargetClassOf (w: Target): TargetClass;
<* LL.sup < w *>
```

Return the `class` argument for which there was a previous call to `BeTarget(w, class)`, or `NIL` if there was no such call.

```
PROCEDURE GetSource (w: Target): T;
<* LL.sup = VBT.mu *>
```

Called by a target's `normal` or `excited` methods to find out the `SourceVBT` causing the method to be invoked.

```
PROCEDURE GetHighlighter (v: T): HighlightVBT.T;
<* LL.sup = VBT.mu *>
```

Returns the `HighlightVBT` above the nearest Trestle-installed ancestor of `v`. This is typically called by a `normal` or `excited` method.

Here are three `TargetClass` objects that may be useful. Each of these use the `op` parameter for painting in the `HighlighVBT`.

```
PROCEDURE NewInserterTarget (op := PaintOp.TransparentSwap): TargetClass;
<* LL = arbitrary *>
```

Displays a grid over itself when excited. Appropriate for an adjusting bar in a tiling window manager. The parent of the target must be an `HVSplit`, and grid has a minimum size in the `HVSplit`'s axis.

```
PROCEDURE NewSwapTarget (op := PaintOp.TransparentSwap): TargetClass;
<* LL = arbitrary *>
```

Displays a grid over itself when excited. This target is appropriate for a non-adjusting bar in a tiling window manager.

```
PROCEDURE NewTarget (op := PaintOp.TransparentSwap): TargetClass;
<* LL = arbitrary *>
```

Inverts itself when excited. This target class is a general-purpose target.

```
END SourceVBT.
```

5.8 The BooleanVBT Interface

A `BooleanVBT` is a multi-filter that maintains a Boolean state for its VBT-child.

When the `action` procedure of the button would normally be invoked, the value of the state of the `BooleanVBT` is toggled and the `callback` method on the `BooleanVBT` is invoked.

The multi-child of a `BooleanVBT` is defined to be the multi-child of the `ButtonVBT`.

```
INTERFACE BooleanVBT;

IMPORT ButtonVBT, HighlightVBT, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = HighlightVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (button: ButtonVBT.T): T;
             <* LL = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;
```

The call `v.init(...)` initializes `v` as a `BooleanVBT` with an initial state of `FALSE`. The default `callback` method is a no-op.

Warning: This call modifies the `action` field of `button`.

```
PROCEDURE Put (v: T; state: BOOLEAN);
<* LL.sup = VBT.mu *>
```

Set `v`*'s state.*

```
PROCEDURE Get (v: T): BOOLEAN;
<* LL.sup = VBT.mu *>
```

Returns `v`*'s current state.*

```
END BooleanVBT.
```

5.9 The ChoiceVBT Interface

A `ChoiceVBT` multi-filter behaves in concert with other `ChoiceVBT`s to implement *radio buttons*. Abstractly, a `ChoiceVBT` `v` consists of

`state(v)`	`TRUE` or `FALSE`
`group(v)`	a set of `ChoiceVBT`s (the *radio group*)

A group `g` consist of

`selection(g)`	the one member of `g` whose state is `TRUE`, or `NIL` if there is no such member.

`state(v)` is defined as `v = selection (group (v))`.

Structurally, a `ChoiceVBT` is identical to a `BooleanVBT`: it is a multi-filter that maintains a Boolean state for its VBT-child. All events are forwarded to the VBT-child.

When the `action` procedure of the button would normally be invoked, the value of the state of the `ChoiceVBT` is toggled and the `callback` method on the `ChoiceVBT` is invoked.

The multi-child of a `ChoiceVBT` is defined to be the multi-child of the `ButtonVBT`.

```
INTERFACE ChoiceVBT;

IMPORT BooleanVBT, ButtonVBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = BooleanVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (button: ButtonVBT.T; group: Group): T;
           END;
```

The call `v.init(...)` initializes `v` as a `ChoiceVBT` with an initial state of `FALSE`. It is added to the radio group `group`.

```
TYPE Group <: ROOT;
```

A `Group` is a set of `ChoiceVBT`s.

A `ChoiceVBT` `v` is added to a group when `v` is initialized. When `v` is discarded, it is removed from its group.

```
PROCEDURE Get (v: T): T;
<* LL.sup = VBT.mu *>
```

Return `selection(group(v))`

```
PROCEDURE Put (v: T);
<* LL.sup = VBT.mu *>
```
Equivalent to `selection(group(v)) := v`

```
PROCEDURE Clear (v: T);
<* LL.sup = VBT.mu *>
```
Equivalent to `selection(group(v)) := NIL`

```
PROCEDURE Selection (group: Group): T;
<* LL.sup = VBT.mu *>
```
Return `selection(group)`

```
END ChoiceVBT.
```

6 Subwindows

This section describes the facilities in VBTkit for *subwindows*. Technically, a subwindow is any non-background child of a `ZSplit`. Informally, a subwindow is just like a top-level window but it is not installed in the window manager.

We recommend that you use a `ZChildVBT` or `ZChassisVBT` filter for subwindows, and a `ZBackground` filter for the background child of a `ZSplit`. The `ZChildVBT` gives you a powerful notation for specifying a location within the `ZSplit` where the subwindow should appear, and whether the subwindow should be visible or invisible. A `ZChassisVBT` is a subtype of `ZChildVBT` and provides a "chassis" for subwindow that contains widgets for moving, resizing, and closing the subwindow.

The `ZGrowVBT` and `ZMoveVBT` interface define switches that have the side-effect of resizing and repositioning its nearest ancestor that is a subwindow. These are used by `ZChassis`.

The `ZSplitUtils` contains some utility procedures for clients operating with subwindows. Finally, the `ZTilps` multi-split is just like a `ZSplit`, but considers its children from bottom to top.

6.1 The ZChildVBT Interface

An `ZChildVBT.T` is a VBT that is typically used as a subwindow.

A `ZChildVBT` is a subclass of a `HighlightVBT` that insulates any highlighting done in the `ZChildVBT` from highlighting done in other subwindows. Clients should use a `ZBackgroundVBT` around the background child in order to insulate highlighting in the background child from highlighting in the subwindows.

There are two alternate ways to initialize a `ZChildVBT`. Each allows the client to specify whether the subwindow should be initially visible ("mapped") and how the subwindow should be reshaped when the parent `ZSplit` is reshaped.

The method call `v.init(...)` allows the client to specify where the center or a corner of `v` should be placed, relative to the parent, either in absolute distance (in millimeters) from the parent's northwest corner (`CoordType.Absolute`), or as percentages of the parent's width and height (`CoordType.Scaled`). The default is to align the center of `v` with the center of the parent. The size of `v` is its preferred sizes in both the horizontal and vertical dimensions.

The method call `v.initFromEdges(...)` allows the client to specify the edges of `v`, either in absolute distance (in millimeters) from the parent's northwest corner (this is the only case in which the client specifies the absolute size of `v`), or as percentages of the parent's width and height.

The implementation will not pop up a subwindow with its northwest corner north or west of the visible portion of the `ZSplit` parent; it will override the specified position as necessary to bring it into view. It is a checked runtime error to specify scaled coordinates (percentages) that are outside the range 0.0–1.0. If the specified position is overriden, or if the subwindow is not entirely visible when the subwindow is first made visible, the implementation will also override the reshape method so that the subwindow will be repositioned using the information specified when it was initialized.

Finally, in order for the reformatting to meet specifications above, the client must call `Inserted` after the subwindow is inserted as a child of a `ZSplit`; the client must call `Moved` after the subwindow is repositioned by the user; and the client must call `Grew` after the size of the subwindow is changed by the user.

```
INTERFACE ZChildVBT;

IMPORT HighlightVBT, VBT, ZSplit;

TYPE
  Location = {NW, NE, SW, SE, Center};
  CoordType = {Absolute, Scaled};
  T <: Public;
  Public = HighlightVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (ch  : VBT.T;
                   h, v          := 0.5;
                   loc           := Location.Center;
                   type          := CoordType.Scaled;
                   shaper: ZSplit.ReshapeControl := NIL;
                   open: BOOLEAN                 := TRUE): T;
             initFromEdges (ch: VBT.T;
                            w, e, n, s: REAL;
                            type := CoordType.Absolute;
                            shaper: ZSplit.ReshapeControl := NIL;
                            open := TRUE): T;
           END;

PROCEDURE InitiallyMapped (v: VBT.T): BOOLEAN;
```

If `v` is a `ZChild`, return the value of `open` when it was initialized. Otherwise, return `TRUE`.

```
PROCEDURE Pop (v: VBT.T; forcePlace := FALSE);
```

Map `v` and lift it to the top of its parent's children. If `forcePlace` is set, position `v` at its initial location.

```
PROCEDURE Inserted (v: VBT.T);
```

The client must call this procedure after `v` has been inserted into a `ZSplit`. This procedure sets a `ReshapeControl` object on `v`. If `v` isn't a `ZChildVBT`, the `ReshapeControl` tries to center `v`, subject to the contraint that its northwest corner is just visible. If `v` is a `ZChild`, the `ReshapeControl` will follow `vbt`'s initial position until `v` is moved by the user (usually because

`Moved` is called). At that point, the northwest corner of `v` will remain at that position relative to its parent, until the user moves it again.

```
PROCEDURE Moved (v: VBT.T);
```

The client must call this procedure after `v` has been moved by a user. If `v` is a `ZChildVBT`, this procedure notes that `v` has been moved by the user, so that the next time it is reshaped, `v` will retain its current position relative to its parent. If `v` isn't a `ZChildVBT`, this procedure is a no-op.

```
PROCEDURE Grew (v: VBT.T; w, h: INTEGER);
```

The client must call this procedure after the size of `v` has been changed to `w`-by-`h` (in pixels) by a user. If `v` is a `ZChildVBT`, this procedure notes that `v` has a new shape and calls `VBT.NewShape` to tell the parent `ZSplit`. Subsequently, `v` will report its shape as `w`-by-`h`. If `v` is not a `ZChildVBT`, this procedure is a no-op.

Finally, here are a few `ZSplit` reshape controllers that are sometimes useful:

```
VAR (*CONST*)
  Scaled: ZSplit.ReshapeControl;
  ScaledHFixed: ZSplit.ReshapeControl;
  ScaledVFixed: ZSplit.ReshapeControl;
  ScaledHVFixed: ZSplit.ReshapeControl;
```

`Scaled` reshapes the child by scaling the old child domain to occupy the same relative position of the changing parent domain. `ScaledHFixed` does the same, and then insets the west and east edges so that the child's width is not changed. Similarly, `ScaledVFixed` scales the child's domain and then insets the north and south edges. `ScaledHVFixed` insets both the north and south edges and the west and east edges so the size of the child's domain stays fixed. In other words, `ScaledHVFixed` can be used to reposition the center point of the child without changing its size.

```
END ZChildVBT.
```

6.2 The ZChassisVBT Interface

A `ZChassisVBT` multi-filter provides a *chassis* for a subwindow. The visual display of the chassis is hard-wired into this module; in particular, it won't look like a top-level window of most of the common X window managers. The top of the chassis is a banner containing (from left to right) a *close button*, *draggable title*, and a *grow button*. (See Figure 6.2.)

Clicking on the close button unmaps the `ZChassisVBT`, thereby causing it to disappear. Dragging the title allows the user to reposition the `ZChassisVBT` within its parent. Clicking on the grow button allows the user to change the size of the `ZChassisVBT`, subject to its size constraints. That is, the user isn't allowed to make the interior of the chassis smaller or larger than its reported bounds along each dimension.

```
INTERFACE ZChassisVBT;

IMPORT Shadow, VBT, ZChildVBT, ZSplit;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public =
    ZChildVBT.T OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (ch      : VBT.T;
            title   : VBT.T;
            shadow  : Shadow.T := NIL;
            closable: BOOLEAN  := TRUE;
            open    : BOOLEAN  := TRUE;
            h, v               := 0.5;
            loc                := ZChildVBT.Location.Center;
            type               := ZChildVBT.CoordType.Scaled;
            shaper: ZSplit.ReshapeControl := NIL):
            T;
      initFromEdges (ch        : VBT.T;
                     title     : VBT.T;
                     w, e, n, s: REAL;
                     shadow    : Shadow.T := NIL;
                     closable  : BOOLEAN  := TRUE;
                     open      : BOOLEAN  := TRUE;
                     type := ZChildVBT.CoordType.Absolute;
                     shaper: ZSplit.ReshapeControl := NIL): T;

      <* LL = VBT.mu *>
      callback (READONLY cd: VBT.MouseRec);
    END;

END ZChassisVBT.
```

The call `v.init(...)` initializes `v` as a `ZChassisVBT`. It is assumed that `v` will be a subwindow. The interior of the chassis, `ch`, is `v`'s child in the multi-child sense.

An alternative method, `v.initFromEdges`, also initializes `v`, using different information for specifying the initial location of the subwindow. (See the `ZChildVBT` interface on page 33 for details of the `h`, `v`, `loc`, and `type` parameters

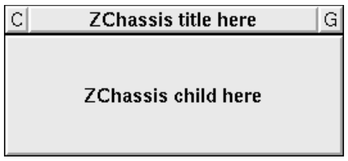

Figure 3: *A ZChassis.*

to `init`, as well as for details of the `w`, `e`, `n`, `s`, and `type` parameters to `initFromEdges`.)

A close button is displayed iff `closable` is set. The grow button is implemented with a `ZGrowVBT`. `title` also functions as a drag bar. It is implemented by a `ZMoveVBT`. The looks of these buttons is governed by the `shadow` parameter.

If `open` is `TRUE`, then `v` will be visible when it is inserted as a child of its parent `ZSplit`.

In the current implementation, a chassis has the following general structure (using FormsVBT notation):

```
(Stable
  (Border
    (VBox
      (HBox (CloseButton "C")
            (ZMove title)
            (ZGrow "G"))
      (Frame ch)))))
```

See Figure 6.2.

However, don't try to traverse the VBT tree directly; it is subject to change. To retrieve the contents of a chassis `v`, use `MultiFilter.Child(v)`.

`v.callback(cd)` is invoked when the close button is activated. The default method is a no-op.

A `ZChassisVBT`'s move, grow, and close buttons are not effective unless the `ZChassis` is a non-background child of a `ZSplit`.

6.3 The ZBackgroundVBT Interface

A `ZBackgroundVBT` is a filter that should be put around the background child of a `ZSplit`. This filter will insulate highlighting that takes place within the background child from highlighting in the other children of the `ZSplit`. The implementation is merely a `HighlightVBT`, but it's easier to remember the purpose of that highlighter by calling it a `ZBackgroundVBT` instead.

In order for `ZChassisVBT` to display an outline of a subwindow that is visible against the background when it is moved or resized, you should use the `VBTColors` interface to associate the primary background and foreground colors of the contents of the `ZBackgroundVBT`.

```
INTERFACE ZBackgroundVBT;

IMPORT HighlightVBT;

TYPE
  T = HighlightVBT.T BRANDED OBJECT END;

END ZBackgroundVBT.
```

6.4 The ZMoveVBT Interface

A `ZMoveVBT` is a switch that has the side-effect of repositioning its nearest ancestor subwindow.

If the initial mouse click is unshifted, the subwindow is lifted to the top of its sibling; otherwise, the subwindow keeps its current top-to-bottom ordering among its siblings. As the mouse is moved, the cursor is changed to give appropriate feedback, and an outline of the subwindow is moved to show where it will be repositioned on an uncancelled upclick. On an uncancelled upclick or chord-cancel, the outline is removed.

```
INTERFACE ZMoveVBT;

IMPORT Rect, SourceVBT;

TYPE T <: SourceVBT.T;
```

The following procedure is useful for subclasses, such as `ZGrowVBT`, to control the shape of the outline of `v`'s subwindow as the mouse is being dragged.

```
PROCEDURE MoveAndHighlight (v: T; READONLY rect: Rect.T);
<* LL = VBT.mu *>
```

Show the outline of `v` as `rect`. Should only be called by the `during` method of a subclass.

The default `during` method calls `MoveAndHighlight` with `rect` equal to the domain of the subwindow being moved, translated by an appropriate amount to reflect the mouse movement since the initial mouse click.

On an uncancelled upclick, the default `post` method moves the subwindow to the rectangle last specified to `MoveAndHighlight` and calls `ZChildVBT.Moved` and `ZChildVBT.Grew`.

The highlighter used for displaying an outline of the subwindow contain `v` is the `HighlightVBT` returned by `SourceVBT.GetHighlighter(v)`. An appropriate paint op is constructed by examing the colors of the background child of the subwindow's parent. Those colors are found using the `VBTColors` interface; be sure to use that interface to record the background child's primary foreground and background colors.

```
END ZMoveVBT.
```

6.5 The ZGrowVBT Interface

A `ZGrowVBT` is a switch that has the side effect of reshaping its nearest ancestor subwindow.

If the initial mouse click is unshifted, the subwindow is lifted to the top of its sibling; otherwise, the subwindow keeps its current top-to-bottom ordering among its siblings. As the mouse is moved, the cursor changes to give appropriate feedback, and an outline of the subwindow is displayed to show the shape the subwindow will acquire on an uncancelled upclick. The shape of the subwindow is not actually changed until the uncancelled upclick. The outline is removed on an uncancelled upclick or on a chord-cancel.

```
INTERFACE ZGrowVBT;

IMPORT FeedbackVBT, ZMoveVBT;

TYPE
  <* SUBTYPE T <: MulitFilter.T *>
  T <: Public;
  Public = ZMoveVBT.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (f: FeedbackVBT.T): T;
           END;

END ZGrowVBT.
```

6.6 The ZSplitUtils Interface

The `ZSplitUtils` interface contains utility procedures for working with `ZSplit`s. The `ZChildVBT` interface contains some additional utility procedures that are oriented for children of `ZSplit`s that are used as subwindows.

```
INTERFACE ZSplitUtils;

IMPORT VBT;

PROCEDURE FindZChild (v: VBT.T): VBT.T;
```

Return the lowest (possibly improper) ancestor of `v` whose parent is a `ZSplit.T` and which is not the `ZSplit.T`'s background child. If no such `VBT` is found, return `NIL`. There's a good chance that the `VBT` returned is a `ZChildVBT.T`, but this is not required.

```
END ZSplitUtils.
```

6.7 The ZTilps Interface

A `ZTilps.T` multi-split is like a `ZSplit`, except that its children are stored from bottom to top. For example, `MultiSplit.Nth(v,0)` returns the background child of the `ZTilps`.

```
INTERFACE ZTilps;

IMPORT ZSplit;

TYPE
  <* SUBTYPE T <: MultiSplit.T *>
  T <: Public;
  Public = ZSplit.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (saveBits := FALSE; parlim := -1): T
           END;
```

The call `v.init(...)` initializes `v` as a `ZTilps` and returns `v`. See the `ZSplit` interface for a description of `saveBits` and `parlim`.

```
END ZTilps.
```

7 Images

This section describes the facilities in VBTkit for displaying images. A `PixmapVBT` is a VBT class that displays a `Pixmap.T`, a screen-independent specification of a pixmap. And the `Image` interface contains utilities for building screen-independent pixmaps from screen-dependent pixmaps and from descriptions stored in files.

7.1 The PixmapVBT Interface

A `PixmapVBT.T` is a VBT that displays a pixmap.

The minimum size of a `PixmapVBT` is just large enough to display its pixmap (surrounded by any margins that were supplied when the `PixmapVBT` was created). Its preferred size is the same as its minimum size, and its maximum size is very large.

```
INTERFACE PixmapVBT;

IMPORT VBT, PaintOp, Pixmap;

TYPE
  T <: Public;
  Public =
    VBT.Leaf OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (pm: Pixmap.T;
            halign, valign: REAL := 0.5;
            hmargin, vmargin: REAL := 0.0;
            op: PaintOp.T := PaintOp.BgFg;
            bg: PaintOp.T := PaintOp.Bg): T
    END;
```

The call `v.init(...)` initializes `v` as a `PixmapVBT` displaying pixmap `pm` using the paint op `op`, and returns `v`.

If `halign = 0.0`, the west boundary of the pixmap will be indented by the given `hmargin` (in millimeters) from the west boundary of the `VBT`; if `halign = 1.0`, the east boundary of the pixmap will be inside the east boundary of the `VBT` by the given `hmargin`; for other values of `halign`, the horizontal position of the text is computed by linear interpolation. In particular, `halign = 0.5` centers the pixmap horizontally. The vertical position is determined by `vmargin` and `valign` in a similar way.

If the domain of `v` is larger than the pixmap, the background is painted using the tint `bg`.

When the pixmap has depth 1, `op` should be a pair of tints. Otherwise, a good choice for `op` is `PaintOp.Copy`.

```
PROCEDURE Put (v: T; pm: Pixmap.T);
<* LL.sup = VBT.mu *>
```

Change the pixmap displayed by `v` *to* `pm`*, and mark* `v` *for redisplay.*

```
PROCEDURE SetColors (v : T;
                     op: PaintOp.T;
                     bg: PaintOp.T   := PaintOp.Bg);
<* LL.sup = VBT.mu *>
```

Change the `op` *and* `bg` *of* `v`*, and mark* `v` *for redisplay.*

```
END PixmapVBT.
```

7.2 The Image Interface

An `Image.T` is a screen-independent specification of an *image*. An image is a pixmap that includes specifications for both color and resolution. It is rendered consistently across screen types in terms of its colors and size.

```
INTERFACE Image;

IMPORT Pixmap, Rd, ScrnPixmap, Thread, TrestleComm,
  VBT, Word, Wr;

TYPE T = Pixmap.T;

EXCEPTION Error;
```

An `Image.Raw` is an array of pixels, with both resolution and color information. It is like a `Pixmap.Raw`, with the addition of resolution and color information. An `Image.T` is built from an `Image.Raw` pixmap using procedures in this interface.

There are three types of `Image.Raw` pixmaps: The `RawBitmap` represents bitmaps (1-bit deep pixmaps); the `RawPixmap` represents pixmaps that do not have a color table; and the `RawPixmapCMap` represents pixmaps that use a color table.

```
TYPE
  Pixel = Word.T;
  Mode = {Stable, Normal, Accurate};
  RGB = RECORD r, g, b: REAL END;

TYPE
  Raw = OBJECT
      width, height: INTEGER;
      xres, yres: REAL := 75.0; (* in pixels per inch *)
    METHODS
      get (h, v: INTEGER): Pixel;
      set (h, v: INTEGER; pixel: Pixel);
    END;

  RawBitmap = Raw BRANDED OBJECT END;

  RawPixmap = Raw OBJECT
      needsGamma := TRUE;
      colorMode  := Mode.Normal;
    END;

  RawPixmapCMap = RawPixmap OBJECT
      colors: REF ARRAY OF RGB;
    END;
```

If `pm` is a `Raw` pixmap, then `pm` contains `pm.height` rows, and each row contains `pm.width` elements. These fields are read-only after they are initialized. The pixels are accessed with (0,0) in the northwest corner and `(width-1,height-1)` in the southeast corner. The `pm.xres` and `pm.yres` fields specify the resolution at which `pm` was designed. The `get` and `set` methods retrieve and store individual elements of the pixmap.

Each subtype of `Raw` can interpret a "pixel" in whatever way it chooses. The three subtypes defined here do the following:

- If `pm` is a `RawBitmap` pixmap, then it is guaranteed that the method `pm.get` will return a 0 or 1. In the call `pm.set(h,v,pixel)`, only the least significant bit of `pixel` is used.
- If `pm` is a `RawPixmap`, the pixels in `pm` encode an RGB value each of whose components is 8 bits. An `(r,g,b)` triple is stored as

  ```
  r * 256 * 256 + g * 256 + b
  ```

  and each of `r`, `g`, and `b` is between 0 and 255. The field `pm.needsGamma` indicates whether to let Trestle gamma-correct the colors. The `pm.colorMode` field determines how each RGB value in the pixmap should be displayed on color-mapped display.
- If `pm` is a `RawPixmapCMap`, the pixels in `pm` are used as an index into the color table stored in the field `pm.colors`.

The colors used to display a colored pixmap `pm` depends on a number of factors. The `pm.colorMode` field is used to match colors in the pixmap with colors in the color table, as described in the `ScrnPixmap` interface. The matching depends on other applications running, on other pixmaps being displayed, and on the depth of the screen.

The current implementation does not perform any dithering, except on monochrome screens. On monochrome screen, a very crude "thresholding" is performed: if the brightness of the color is more than 50% of the maximum brightness, the screen's foreground color is used. Otherwise, the screen's background color is used.

7.2.1 Retrieving and storing "raw" pixmaps

An `Image.Raw` can be built from a reader containing an image in Jef Poskanzer's "portable anymap file" ("pnm") format, and a "pnm" description of an `Image.Raw` can be stored into a writer.

There are many tools available in the public domain for manipulating images in "pnm" format and for converting between that format and other formats (e.g., GIF, X11, Macintosh PICT, HP PaintJet, and so on).

There are three types of "pnm" files:

- "pbm" – portable bitmap file
- "pgm" – portable graymap file
- "ppm" – portable pixmap file

Each of these format has two variants: "raw" and "ASCII." In the "ASCII" version, pixel values are stored as ASCII decimal numbers. In the "raw" version, pixel values must be less than 256 and are stored as plain bytes.

```
PROCEDURE FromRd (rd: Rd.T): Raw
  RAISES {Thread.Alerted, Rd.Failure, Error};
<* LL = arbitrary *>
```

Returns an `Image.Raw` *from the reader* `rd` *containing an image in "pnm" format. Pixels in "ppm" files are normalized to 8 bits per channel and intensity values of "pgm" files are normalized to 8 bits.*

```
PROCEDURE ToWr (raw: Raw; wr: Wr.T)
  RAISES {Thread.Alerted, Wr.Failure};
<* LL = arbitrary *>
```

Store an ASCII description of `raw` *into the writer* `wr` *using "pnm" format.*

Procedures `FromRd` and `ToWr` are not guaranteed to be idempotent because pixel values are normalized by `FromRd` to be 8 bits. Also, the "pnm" format produced by `ToWr` is either ASCII "pbm" for subtypes of `RawBitmap` or ASCII "ppm" for subtypes of `RawPixmap`, whereas procedure `FromRd` can accept these formats as well as the "raw" variants and grayscale formats ("pgm").

The more serious limitation of using "pnm" format is that "pnm" does include any information about the pixmap resolution or color matching. `FromRd` will use the default resolution of a `Raw` and the default color parameters of a `RawPixmap`; `ToWr` simply ignores the resolution and color fields.

7.2.2 Creating "raw" pixmaps from a VBT

`FromVBT` captures the information in an arbitrary VBT into an `Image.Raw` of particular dimensions:

```
PROCEDURE FromVBT(v : VBT.T; width, height: REAL): Raw
   RAISES {TrestleComm.Failure};
<* LL = VBT.mu *>
```

Return a screen-independent `Raw` that describes `v` when `v` is scaled to be `width` by `height` millimeters.

The current implementation of `FromVBT` will cause `v` to be redisplayed multiple times: First `v` is detached from its parent `pm` (unless `pm = NIL`). Next, `v` is installed in an offscreen Trestle window, with an appropriate `ScaleFilter` inserted to make `v` the correct size. A call to `VBT.Capture` creates a screen-dependent version of the offscreen window. At this point, `v` is detached from the offscreen window, and reattached to `pm` (unless `pm = NIL`). Each time that `v` changes its parent, various VBT methods (reshape, rescreen, redisplay, and so on) are called.

The following procedure converts a screen-dependent pixmap (such as that returned by `VBT.Capture`), into one that is screen-independent:

```
PROCEDURE FromScrnPixmap (
    spm: ScrnPixmap.T;
    st: VBT.ScreenType): Raw RAISES {TrestleComm.Failure};
<* LL.sup <= VBT.mu *>
```

Returns a screen-independent `Raw` that describes the pixmap `spm` when displayed on `st`. Any field of `Raw` that cannot be computed from `spm` and `st` is given its default value. For example, the `needsGamma` and the `colorMode` fields of pixmaps that are deeper than 1-bit.

7.2.3 Building an image from "raw" pixmaps

The remaining procedures in this interface create an `Image.T` from an `Image.Raw` pixmap:

```
PROCEDURE Unscaled (raw: Raw): T;
<* LL.sup <= VBT.mu *>
```

Returns a pixmap that will display as `raw`. The pixels in `raw` will not be scaled regardless of the screen's resolution.

For example, consider a pixmap `pm` whose dimensions are 150 wide by 50 high. On a 75dpi screen (a typical 1993-vintage monitor), the pixmap `pm` would appear 2 inches wide and 2/3 inches high. On a high-resolution monitor of 300 dpi, `pm` would appear 1/2 inch wide and 1/6 inch high. The `pm.xres` and `pm.yres` fields are ignored.

If you want `pm` always to appear as 2 inches by 2/3 inches, regardless of the pixel density of the monitor, you'd use `Scaled` instead:

```
PROCEDURE Scaled (raw: Raw): T;
<* LL.sup <= VBT.mu *>
```

Return a pixmap that will display as `raw`*, scaled for the screen's resolution. The horizontal and vertical dimensions are scaled independently.*

The current implementation scales pixmaps by non-negative integer amounts: horizontally by `ROUND(dpiX/pm.xres)` and vertically by `ROUND(dpiY/pm.yres)`, where `dpiX` and `dpiY` are the horizontal and vertical resolution of the screen, respectively, expressed in dots-per-inch.

In the example above, suppose that `pm.xres` and `pm.yres` were both 75. On a 300 dpi screen, `pm` would appear 2 inches wide and 2/3 inches high. Each pixel in `pm` would appear as a block of 4x4 screen pixels. If the screen were 250 dpi horizontally and 175 dpi vertically, then `pm` would appear $1\frac{1}{2}$ inches wide and $1\frac{1}{3}$ inches high. Each pixel in `pm` would appear as a block of 3x2 screen pixels.

Procedure `ScaledN` allows you to provide a collection of pixmaps, each at a different resolution, and scales the most appropriate pixmap:

```
PROCEDURE ScaledN (READONLY raws: ARRAY OF Raw;
                   tolerance: REAL     := 0.25;
                   maxScale : CARDINAL := 4     ): T;
<* LL.sup <= VBT.mu *>
```

Return a pixmap which will scale and display pixmap `raws[i]`*, where* `i` *is chosen so that* `raws[i]` *has the "most appropriate" resolution.*

Specifically, `i` is chosen such to minimize the *scale factor* (the amount that a "raw" pixmap must be scaled) while remaining within the given error `tolerance`.

The scale factor of pixmap `pm` is

```
MAX (dpiX/pm.xres, dpiY/pm.yres)
```

where `dpiX` and `dpiY` are the horizontal and vertical resolutions of the screen, respectively, expressed in dots-per-inch.

For a given scale factor `s`, the error is

```
ABS (MAX ((dpiX - MAX(s, maxScale) * pm.xres) / dpiX,
          (dpiY - MAX(s, maxScale) * pm.yres) / dpiY))
```

If none of the pixmaps in the `raws` array satisfies the tolerance, then the pixmap giving the smallest error is chosen.

The purpose of `tolerance` and `maxScale` is to allow the user control over the interpretation of "most appropriate" when chosing the pixmap.

- A small `tolerance` ensures a small error, which can mean a larger scale factor.

  For example, suppose the screen has a resolution of 300 dpi and pixmaps that are 150 and 250 dpi. When `tolerance <` 1/6, then `ScaledN` chooses the 150 dpi pixmap with a scale factor equal to 2, rather than the 250 dpi pixmap with a scale factor equal to 1.

- A small `maxScale` makes it less likely that a very low-resolution pixmap (which happens to give very small error) is chosen over a higher-resolution pixmap (which gives a larger error).

  For example, suppose the screen has a resolution of 300 dpi and pixmaps that are 50 and 200 dpi. If `tolerance >` 1/3, then `ScaledN` always chooses the 200 dpi pixmap, because the error, (300 - 200)/300=1/3, is within the tolerance and the scale factor for 200 dpi is less than the scale factor for the 50 dpi pixmap. However, when `tolerance <` 1/3, the 50 dpi pixmap is chosen unless `maxScale <=` 4.

```
END Image.
```

8 Text Editing

The principal VBT for text-editing is a TextPort. It has a subtype, TypeinVBT, for single-line "type-in boxes." A TextPort is also combined with a scrollbar to form a TextEditVBT, which has a subtype, TypescriptVBT, for transcripts and command-shells.

The `TextPortClass` interface in Appendix A.2 is the starting point for clients wishing to define subclasses of text-editors. You might also wish to look at the implementation of the text-editors already provided: `EmacsModel` (Appendix A.3), `IvyModel` (Appendix A.4), `MacModel` (Appendix A.5), and `XtermModel` (Appendix A.6).

The VBTkit package also provides a number of interfaces, not described in this manual, that are intended to help clients implement subclasses of text-editors. Here are the interfaces of interest:

- `Key` defines constants for the `VBT.KeySym`s of some common non-graphic keys.
- `KeyTrans` provides some standard mapping between keyboard keys and ASCII characters.
- `MTextUnit` implements tools for treating the underlying text as sequences of characters, lines, or paragraphs.

8.1 The TextPort Interface

A textport is a `VBT` that allows the user to type and edit text.

The methods and procedures in this interface fall into several categories, each dealing with different aspects of the text-editor.

Appearance The client can choose the font, colors, margins, and whether long lines should be clipped or wrapped. The fonts and colors can be changed dynamically.

Access to the text There are procedures to read and write subsequences of the text, to read and set the current "type-in" point (cursor position), to get the length of the text, and to make the text read-only.

Keybindings and Text-Selections A textport is initialized with a *model*, an object (defined in the `TextPortClass` interface) that establishes the connection between keystrokes and editing operations, and the connection between mouse-gestures, the cursor position, local selections (including highlighted regions), and global selections such as the "clipboard" (`VBT.Source`). Four such models are implemented—Emacs, Ivy, Xterm, and Mac—corresponding to different editing paradigms. The choice of model can be changed dynamically. The client may override the `filter` method to intercept keystrokes.

Feedback A textport has callback-methods that are invoked when the text changes, when the user types Return or Tab, when the textport gains or loses the keyboard focus, when the visible region changes, and when errors are detected. All these methods have defaults.

The locking level for all procedures is `LL <= VBT.mu` except as noted.

```
INTERFACE TextPort;

IMPORT Font, PaintOp, VBT, VText;

TYPE
  T <: Public;
  Public = VBT.Leaf OBJECT
           METHODS
             init (hMargin, vMargin := 0.5;
                   font             := Font.BuiltIn;
                   colorScheme: PaintOp.ColorScheme := NIL;
                   wrap                             := TRUE;
                   readOnly                         := FALSE;
                   turnMargin                       := 0.5;
                   model := Model.Default): T;

             <* LL.sup = VBT.mu *>
             filter         (cd: VBT.KeyRec);
             getFont        (): Font.T;
             setFont        (font: Font.T);
             getColorScheme (): PaintOp.ColorScheme;
             setColorScheme (c: PaintOp.ColorScheme);
             getModel       (): SpecificModel;
             setModel       (model: Model);
             getReadOnly    (): BOOLEAN;
             setReadOnly    (flag: BOOLEAN);

             (* callbacks *)
             modified     ();
             returnAction (READONLY cd: VBT.KeyRec);
             tabAction    (READONLY cd: VBT.KeyRec);
             focus (gaining: BOOLEAN; time: VBT.TimeStamp);
             error        (msg: TEXT);

           END;
```

The call `v.init(...)` initializes `v` as a `TextPort.T` and returns it.

The parameters `hMargin` and `vMargin` indicate how much whitespace to leave around the text, expressed in millimeters.

`colorScheme` is used for painting the text. If the parameter is `NIL`, then `PaintOp.bgFg` will be used.

If `wrap` is `TRUE`, then text will be wrapped across line boundaries; otherwise it will be clipped. If it is wrapped, then `turnMargin` specifies the width (in millimeters) of the gray bar placed at the end of the first line and the beginning of the second, indicating that the text has been wrapped.

If `readOnly` is `TRUE`, then the text cannot be changed through the user interface (keyboard and mouse). The procedures `Replace`, `Insert`, `SetText`, and `PutText` bypass the read-only protection, but these are not called by internal routines. In all other descriptions in this interface, the words *replace*, *insert*, *delete*, and so on should be understood as having the restriction that `v` is not read-only.

If `model` is `Model.Default`, then the current value of `DefaultModel` will be used. `DefaultModel` is defined below.

`v.getModel()` returns the name of the current model; note that the return value cannot be `Model.Default`. The call `v.setModel(...)` changes the current model; its parameter may be `Model.Default`, in which case the value of `DefaultModel` will be used.

The call `v.setFont(font)` changes the font used for displaying the text.

The call `v.setColorScheme(c)` changes the colors used for displaying the text.

The implementation calls `v.focus(gaining, time)` whenever `v` gains or loses the keyboard focus. If `gaining` is `TRUE`, then `v` is about to gain the keyboard focus (and `time` is a valid event-time); i.e., this method is called *before* the selection feedback is established, so it is reasonable to call `Select` (below) or put up some other indication. If `gaining` is `FALSE`, then `v` has just lost the keyboard focus (and `time` is *not* valid), so it reasonable to take down whatever indicated that the focus had been acquired. It is not within the power of the `focus` method to prevent `v` from gaining or losing the focus. The default for this method is a no-op.

The implementation calls `v.error(msg)` whenever an exception is raised for which there is no particular remedy, such as an `Rd.Failure`. The value of `msg` will be a short description of the error, typically the name of the procedure where the exception was raised. No method or procedure defined in this interface raises exceptions, but the client may wish to override this method in order to report the error in a popup window, for example. The default for this method is a procedure that tests whether the environment-variable named `TEXTPORTDEBUG` is set (to any value); if so, it writes the message to `Stdio.stderr`.

8.1.1 Access to the text

The textport's initial read-only status depends on the `readOnly` parameter to the `init` method. The `getReadOnly` method returns it; the `setReadOnly` method sets it.

```
PROCEDURE GetText (v    : T;
                   begin: CARDINAL := 0;
                   end  : CARDINAL := LAST (CARDINAL)): TEXT;
<* LL.sup = VBT.mu *>
```

Returns a sub-sequence of the text in `v`*. The result will be empty if*

```
        begin >= Length(v)
```

Otherwise the range of indexes of the subsequence is

```
        [begin .. MIN (end, Length (v)) - 1]
```

```
PROCEDURE SetText (v: T; t: TEXT);
```

Replace the current contents of `v` *with* `t`*. This procedure does not test the read-only status of* `v`*.*

```
PROCEDURE PutText (v: T; t: TEXT);
```

Append `t` *to the current contents of* `v`*. This procedure does not test the read-only status of* `v`*.*

```
PROCEDURE Replace (v: T; begin, end: CARDINAL; newText: TEXT);
```

Replace the text between positions `begin` *and* `end` *in* `v` *with* `newText`*. If* `begin` *and* `end` *are beyond the end of the text, they are taken to refer to the end of the text. This procedure does not test the read-only status of* `v`*.*

```
PROCEDURE Insert (v: T; text: TEXT);
```

If there is a replace-mode selection (see Section A.4.2, page 117), replace it with `text`*; otherwise insert* `text` *at the type-in point. In either case, this is a no-op if* `text` *is the empty string. This procedure does not test the read-only status of* `v`*.*

```
PROCEDURE Index (v: T): CARDINAL;
```

Return the current "type-in" position.

```
PROCEDURE Seek (v: T; n: CARDINAL);
```

Set the "type-in" position to `n`*.*

```
PROCEDURE Length (v: T): CARDINAL;
```

Return the number of characters in `v`*'s text.*

```
PROCEDURE Newline (v: T);
```

Insert a newline character at the type-in point.

```
PROCEDURE NewlineAndIndent (v: T);
```

Insert a newline character and enough spaces to match the indentation of the previous line. As it leaves a blank line, it will delete all spaces from that line so as to leave it truly empty.

```
PROCEDURE IsVisible (v: T; pos: CARDINAL): BOOLEAN;
```

Test whether the character at position `pos` is visible.

8.1.2 Models

```
TYPE
  Model = {Default, Ivy, Emacs, Mac, Xterm};
  SpecificModel = [Model.Ivy .. Model.Xterm];

VAR DefaultModel: SpecificModel := Model.Emacs;
```

The initial value of `DefaultModel` depends on the environment variable named `TEXTPORTMODEL`; if that is set to `emacs`, `ivy`, `mac`, or `xterm` at startup time, then `DefaultModel` will be set accordingly. If it is not defined, or is defined as some other value, then the initial value of `DefaultModel` will be `Model.Emacs`. See the `EmacsModel`, `IvyModel`, `XtermModel`, and `MacModel` interfaces in Appendices A.3–A.6 for details on keybindings, mouse-clicks, and selections.

```
PROCEDURE ChangeAllTextPorts (v: VBT.T; newModel := Model.Default);
```

For each textport `p` that is a descendent of VBT `v`, call `p.setModel(newModel)`.

8.1.3 Keybindings

The `TextPort` interface allows clients a great deal of flexibility in handling keystrokes. `v.key(cd)` proceeds in three steps:

In step 1, it tests whether `cd.wentDown` is true, whether `v` has the keyboard focus, and whether `v`'s domain is non-empty. If all three conditions are true, it proceeds to step 2.

In step 2, it passes `cd` to the model's `keyfilter` object, which handles low-level tasks such as converting "Escape + character" into "meta-character" (in Emacs mode), 8-bit "compose character" operations, and so on. The model may actually contain a *chain* of keyfilters (see the `KeyFilter` interface), each implementing some translation.

In step 3, the model passes `cd` (possibly changed by the keyfilters) to the textport's `filter` method. Clients who wish to intercept keystrokes usually do so at this point, by overriding the `filter` method, rather than by overriding the `key` method, so that they can take advantage of the low-level conversions.

In the default `filter` method, there are several mutually exclusive possibilities, tested in this order:

- If the key is Return, then if the `shift` modifier is on, we insert a newline; if the `option` modifier is on, we insert a newline but leave the cursor in place; otherwise, we invoke `v.returnAction(cd)`, another callback method. Its default method calls `NewlineAndIndent(v, cd)`.
- If the key is Tab, we invoke `v.tabAction(cd)`. The default method inserts 4 spaces.
- If the key is an "arrow" key, we call the model's `arrowKey` method, which moves the cursor one character forward, one character backward, one line up, or one line down, as appropriate.
- If the `control` modifier is on, we call the model's `controlChord` method.
- If the `option` modifier is on, we call the model's `optionChord` method.
- If the key is Backspace or Delete, we delete the previous character, or the current primary selection, if that is non-empty and in replace-mode.
- If the key is an ISO Latin-1 graphic character, we insert it into the text.
- Otherwise, we ignore it.

Finally, we call `Normalize(v)`, except in the `controlChord` and `optionChord` cases.

Clients can specialize the handling of keys, therefore, by overriding the textport's `key`, `filter`, `returnAction`, or `tabAction` methods, and by overriding the model's `controlChord`, `optionChord`, or `arrowKey` methods.

The following procedures give the client access to the keyboard focus:

```
PROCEDURE TryFocus (v: T; t: VBT.TimeStamp): BOOLEAN;
```

Try to acquire the keyboard focus and the primary selection, and report whether it succeeded.

```
PROCEDURE HasFocus (v: T): BOOLEAN; <* LL.sup = VBT.mu *>
```

Test whether `v` *has the keyboard focus.*

8.1.4 Selections

With various keyboard and mouse-gestures, the user may delimit a range of text, known as a *local selection*. The `TextPort` interface defines two local selections, called *primary* and *secondary*. The mechanism for doing this depends entirely on the textport's model. (In fact, only the Ivy model implements secondary selection.) The type-in point is always at one end or the other of the primary selection.

Primary selections in non-readonly textports may be in *replace mode*, also called *pending-delete mode*. This means that any text that is inserted will replace the primary selection, and that the Backspace and Delete keys will delete it.

Independent of the local selections are the two *global selections* defined by Trestle: `VBT.Source` and `VBT.Target`. On X window systems, these are defined by the X server, and are shared across applications. The Source selection, for example, is effectively the "clipboard." Globals selections are "owned" by one program at a time; in Trestle programs, they are owned by one `VBT` at a time. While every textport may have a primary and secondary local selection, at most one can own Source, and at most one can own Target. The *contents* of a global selection are controlled by its owner.

The correspondence between local and global selections also depends entirely on the model. Every model implements an operation called **Copy**, which is defined as follows: the textport acquires ownership of Source, and copies the Primary selection so that it is the contents of Source.

Some models establish an *alias* between a local and a global selection, which means that when that textport owns the global selection, the contents of the global selection are *identical with* the contents of the local selection.

In the Ivy model, for example, Primary is an alias for Target, and Secondary is an alias for Source. In the Xterm model, Primary is an alias for Source. The other models do not use aliasing at all; they implement **Copy** by making a separate copy of the local selection. In those models, the contents of the global selection are not visible; i.e., they are not displayed in the textport.

Local selections are usually highlighted in some way. The highlighting obeys the following conventions, applied in this order:

1. A replace-mode Primary selection is highlighted with black text on a light red background. (On monochrome screens, it is highlighted with "inverse video": white text on a dark background.)
2. If a Source selection is visible (i.e., if it is aliased with a local selection), it is highlighted with a thin, green underline. (On monochrome screens, it is a thin, black underline.)
3. A Primary selection that is neither a replace-mode selection nor a Source selection (e.g., a selection in the Emacs model), is underlined with a thick line. On color screens, there is a further distinction: in a read-only text, the underline is blue; otherwise, the underline is red.

A selection is represented by a pair of inclusive indexes (`begin` and `end`) into the text. The current selection-indices can be retrieved via the `GetSelection` procedure.

```
TYPE SelectionType = {Primary, Secondary};

PROCEDURE Select (v    : T;
                          time : VBT.TimeStamp;
                          begin: CARDINAL        := 0;
                          end  : CARDINAL        := LAST (CARDINAL);
                          sel         := SelectionType.Primary;
                          replaceMode := FALSE;
                          caretEnd    := VText.WhichEnd.Right   );
```

Make a selection in `v`, at event-time `time`. If `begin` and/or `end` are beyond the end of the text, they will be clipped to the end of the text. Acquire ownership of the corresponding `VBT.Selection`; if `sel` is `SelectionType.Primary`, acquire ownership of the keyboard focus as well.

The parameters `replaceMode` and `caretEnd` are relevant only if the value of `sel` is `SelectionType.Primary`. If `replaceMode` is `TRUE` and the entire selection is writable, then `Insert` and `VBT.Write` will *replace* the selected text; otherwise, they cause the new text to be *inserted* at whichever end of the primary selection is specified by `caretEnd`.

```
PROCEDURE IsReplaceMode (v: T): BOOLEAN;
```

Return `TRUE` if the primary selection is in replace mode.

```
TYPE Extent = RECORD l, r: CARDINAL END;

CONST NotFound = Extent {LAST (CARDINAL), LAST (CARDINAL)};

PROCEDURE GetSelection (v: T; sel := SelectionType.Primary):
  Extent;
```

Return the extent of the most recent selection in `v`. If there is no such selection, return `NotFound`.

```
PROCEDURE GetSelectedText (v: T; sel := SelectionType.Primary):
  TEXT;
<* LL.sup = VBT.mu *>
```

Return the text of the most recent selection in `v` if there is one, or the empty string otherwise.

```
PROCEDURE PutSelectedText (v: T;
                           t: TEXT;
                           sel := SelectionType.Primary);
<* LL.sup = VBT.mu *>
```

Replace the text of the most recent selection in `v`, if there is one, with `t`. If there is no such selection, this is a no-op.

8.1.5 Feedback

A textport maintains a "modified" flag. Any operation that changes the text will cause this flag to be set to `TRUE`. If it was previously `FALSE`, then the implementation calls `v.modified()` *after* the change has already happened to `v`. The default is a no-op. The `IsModified` and `SetModified` procedures set and test this flag, respectively.

```
PROCEDURE IsModified (v: T): BOOLEAN;
```

Return the value of the "modified" flag for `v`. Any change to the text will cause the flag to be set to `TRUE`.

```
PROCEDURE SetModified (v: T; value: BOOLEAN);
```

Set the value of the "modified" flag for `v`. This will not invoke `v.modified`, even if `value` is `TRUE`.

A textport also maintains a scrollbar (optional). See the `TextEditVBT` interface in Section 8.3.

```
PROCEDURE Normalize (v: T; to := -1);
```

Scroll `v` if necessary to ensure that position `to` is visible. If `to < 0`, it refers to the current type-in point. If `to` is larger than the length of the text, normalizes to the end of the text.

8.1.6 Direct access to the text

```
PROCEDURE GetVText (v: T): VText.T;
```

For wizards only: extract the underlying `VText`. It is legal to create and manipulate highlighting intervals on it. It is legal to run readers on it, provided you can be sure that you are locking out concurrent change (for example, by holding `VBT.mu`). It is not legal to modify it directly. It is not legal to scroll it directly either, because that will leave the scrollbar incorrect.

```
END TextPort.
```

8.2 The TypeinVBT Interface

```
INTERFACE TypeinVBT;

IMPORT Font, PaintOp, TextPort, VBT;

TYPE
  T <: Public;
  Public = TextPort.T OBJECT
             <* LL.sup = VBT.mu *>
             tabNext: VBT.T := NIL
           METHODS
             init (expandOnDemand   := FALSE;
                   hMargin, vMargin := 0.5;
                   font             := Font.BuiltIn;
                   colorScheme: PaintOp.ColorScheme := NIL;
                   wrap                             := TRUE;
                   readOnly                         := FALSE;
                   turnMargin                       := 0.5;
                   model := TextPort.Model.Default): T;
           END;

END TypeinVBT.
```

TypeinVBT overrides the returnAction, tabAction, key, and shape methods.

The default returnAction method is a no-op, but most clients will override this method.

The TextPort's height is initially set to the height of the tallest character in the current font. Its default width is 30 times the width of the widest character in the current font. The default height is one line, but if expandOnDemand is TRUE, then SELF will expand (and contract) vertically as the text requires, so that the entire text is visible in the window.

The default tabAction method tests whether SELF.nextTab is NIL. If so, it calls the parent-method, TextPort.T.tabAction. If not, it sends a miscellaneous code of type VBT.TakeSelection with the VBT.KBFocus selection to SELF.nextTab, i.e., it asks the nextTab VBT to take the keyboard focus. In addition, if that VBT is itself a TextPort, then it selects all the text in the TextPort in replace-mode.

8.3 The TextEditVBT Interface

A TextEditVBT combines a textport with a scrollbar.

```
INTERFACE TextEditVBT;

IMPORT TextPort, TextPortClass, VBT;

TYPE
  T <: Public;
  Public = Private BRANDED OBJECT
             (* READONLY after init *)
             tp: TextPort.T := NIL;
             sb: Scrollbar  := NIL;
           METHODS
```

```
                <* LL.sup = VBT.mu *>
                init (scrollable := TRUE): T
              END;
      Private <: VBT.T;
      Scrollbar <: TextPortClass.Scrollbar;
```

The call `v.init()` initializes `v` as a `TextEditVBT` and returns `v`. If the textport, `v.tp`, is `NIL`, then a new textport will be allocated, initialized (with default parameters), and assigned to `v.tp`. If `scrollable` is `FALSE`, then there will be no scrollbar. If `scrollable` is `TRUE` but `v.sb` is `NIL`, then a new scrollbar will be allocated, initialized as a vertical scrollbar with the textport's color scheme, and assigned to `v.sb`.

If `v` is scrollable, then the default layout will contain the scrollbar, a thin vertical bar, and the textport, laid out horizontally.

```
    END TextEditVBT.
```

8.4 The TypescriptVBT Interface

A `TypescriptVBT` is a subtype of `TextEditVBT`, with additional features to make it serve as a "glass teletype" with a memory.

Abstractly, a typescript contains

`reader(v)`	an intermittent, unseekable reader
`writer(v)`	a buffered, unseekable writer
`readingThread(v)`	a thread

`reader(v)` provides the client with input that the user typed. `writer(v)` is used to display output. The reader and writer are paired such that the writer is flushed whenever a seek blocks on the reader. The writer is not flushed at every newline.

All input to the typescript, once it has been read, and all output, become part of the *history* of the typescript, and is not modifiable; it remains until the client deletes it by calling `ClearHistory`. Selections that lie fully or partially within the history region are never "replace-mode" selections (see Section A.4.2, page 117). Any attempt to type or insert text in the history region becomes an insertion at the end of the typescript instead.

`readingThread(v)` is initially `NIL`. When a client reads from `v`, `readingThread(v)` is set to `Thread.Self()`. The `handleInterrupt` method (see below) alerts `readingThread(v)`. This is useful when the reading thread is blocked waiting for input.

A typescript's textport, `v.tp`, must be of type `TypescriptVBT.Port` (which is a subtype of `TextPort.T`). The textport's `returnAction` method makes the text of the current type-in region available to the reader and no longer editable. The textport's `setReadOnly` method is a no-op.

Typescripts do not allow the use of Undo and Redo.

```
INTERFACE TypescriptVBT;

IMPORT Rd, TextEditVBT, TextPort, VBT, Wr, Thread;

TYPE
  T <: Public;
  Public = TextEditVBT.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init             (scrollable := TRUE): T;
             interrupt        (time: VBT.TimeStamp);
             handleInterrupt  (time: VBT.TimeStamp);
             terminate        ();
             setThread        (thread: Thread.T := NIL);
           END;
  Port <: TextPort.T;
```

The call `v.init()` initializes `v` as an empty typescript.

It is a checked runtime error if `v.tp` is `NIL` or is not of type `TypescriptVBT.Port`, which is a subtype of `TextPort.T`.

The call `v.interrupt(time)` simulates an interrupt by flushing any pending type-in, writing the characters `^C`, and then calling `v.handleInterrupt(time)`.

The call `v.handleInterrupt(time)` alerts `readingThread(v)`.

After `v.terminate()` is called, subsequent attempts to read from `v` will causes it to report end of file, and `v` becomes unresponsive to further user input, although it will continue to display output written to its output stream. This is appropriate when `v` is being discarded.

The call `v.setThread(thread)` changes `readingThread(v)`. This can be used to protect `Thread.Self()` from being alerted after it has finished reading from `reader(v)`. Subsequent reads on `reader(v)` will reset the `readingThread(v)` to `Thread.Self()`.

```
TYPE
  Reader <: PublicReader;
  PublicReader = Rd.T OBJECT METHODS typescript (): T END;

  Writer <: PublicWriter;
  PublicWriter = Wr.T OBJECT METHODS typescript (): T END;

PROCEDURE GetRd (v: T): Reader;
```

Get the input stream for `v`*. By definition,*

```
        GetRd(v).typescript() = v
```

```
PROCEDURE GetWr (v: T): Writer;
```

Get the output stream for `v`*. By definition,*

```
        GetWr(v).typescript() = v
```

```
PROCEDURE GetHistory (v: T): TEXT; <* LL <= VBT.mu *>
```

Return the "history" text of `v`*.*

```
PROCEDURE ClearHistory (v: T);  <*  LL <= VBT.mu *>
```

Clear the "history" text of `v`*.*

```
END TypescriptVBT.
```

9 Miscellaneous Leaf VBTs

9.1 The ListVBT Interface

A `ListVBT` defines a VBT class for displaying a list (or table) of items. Each item is in a *cell*. All cells are the same size. They are displayed in a single vertical column, with a scrollbar.

The `ListVBT` itself deals with the details of being a VBT, maintains a table that maps a cell-number to a cell-value, and maintains the *selection*, a distinguished subset of the cells. It uses subsidiary objects to handle the details of what cells look like on the screen (`Painter`), and how the list responds to mouse clicks (`Selector`).

This interface contains basic versions of each of the subsidiary objects:

- `TextPainter`, which treats cells' values as `TEXT` and paints them.
- `UniSelector`, which maintains at most one selected cell, adjusted by mouse clicks.
- `MultiSelector`, which uses mouse clicks for selection, but permits multiple cells to be selected.

The client can subclass these, or provide entirely different ones. A client that wishes to take actions in response to mouse clicks should subclass a `Selector`. Similarly, a client that wishes to display objects other than text strings should subclass `Painter`.

9.1.1 Locking levels

`ListVBT` is internally synchronized; it can safely be called from multiple threads. All `ListVBT.T` methods have `LL.sup < list`. In addition, `VBT.mu < list` for any `list` of type `ListVBT.T`.

VBT methods call `Selector` methods with `LL.sup = VBT.mu`. `Selector` methods are permitted to call `ListVBT.T` methods.

`ListVBT.T` methods call `Painter` methods with the `ListVBT.T`'s internal mutex held. `Painter` methods must not call any of the `ListVBT.T` methods; their locking level is such that `LL.sup = list`.

The `TextPainter` class uses its own internal lock for font information; `TextPainter.setFont(v,font)` has `LL.sup < v`.

9.1.2 The type ListVBT.T

```
    INTERFACE ListVBT;

    IMPORT Font, PaintOp, Rect, VBT;
```

```
TYPE Cell = INTEGER;
```

The number of a cell; the first cell-number is 0.

```
TYPE
  T <: Public;
  Private <: VBT.Split;
  Public = Private OBJECT
             painter : Painter  := NIL;
             selector: Selector := NIL;
           METHODS
             init             (colors: PaintOp.ColorQuad): T;
             setValue         (this: Cell; value: REFANY);
             getValue         (this: Cell): REFANY;
             count            (): CARDINAL;
             insertCells      (at: Cell; n: CARDINAL);
             removeCells      (at: Cell; n: CARDINAL);
             selectNone       ();
             selectOnly       (this: Cell);
             select           (this: Cell; selected: BOOLEAN);
             isSelected       (this: Cell): BOOLEAN;
             getAllSelected   (): REF ARRAY OF Cell;
             getFirstSelected (VAR this: Cell): BOOLEAN;
             scrollTo         (this: Cell);
             scrollToShow     (this: Cell);
             reportVisible    (first: Cell; num: CARDINAL);
           END;
```

In the following descriptions, `v` is an object of type `ListVBT.T`, and a value `n` is said to be *in range* if

$0 \leq$ `n < v.count()`

`v.painter` is the list's painter; the client may read but not assign to this field, although the client may provide a value at allocation time. If the actual painter has methods allowing it to be modified, the client is welcome to call them, although the client and painter are then responsible for provoking any necessary repaints.

`v.selector` is the list's selector; client may read but not assign to this field, although the client may provide a value at allocation time. If the actual selector has methods allowing it to be modified, the client is welcome to call them, although the client and selector are then responsible for any necessary adjustments to the set of selected cells.

The call `v.init(colors)` initializes `v` as a `ListVBT` and returns `v`. It must be called before any other method. `colors` is passed intact to the scroller; `colors.fg` is used for a bar that separates the cells from the scroller. If `v.painter = NIL` when this method is called, `init` will allocate and initialize a `TextPainter`. If `v.selector = NIL`, `init` will allocate and initialize a `UniSelector`. Neither the painter nor the selector need have been initialized before this method is called. The list initially has no cells (and no selection).

In the call `v.setValue(this,value)`, if `this` is in range, then record `value` as the value of the cell `this`; otherwise do nothing.

In the call `v.getValue(this)`, if `this` is in range, then return the previously recorded value of the cell `this`; otherwise return `NIL`.

The call `v.count()` returns the number of cells.

The call `v.insertCells(at,n)` inserts `n` cells, starting at

```
MAX (0, MIN (at, v.count()))
```

Previously existing cells at and beyond `at` are renumbered appropriately, and selections are relocated appropriately. The VBT will be repainted in due course. The new cells' values are all `NIL`, and they are not selected.

The call `v.removeCells(at, n)` removes all cells in the range

```
[MAX (0, MIN (at, v.count ())) ..
  -1 + MIN (at + n, v.count ())]
```

Subsequent cells are renumbered appropriately. The VBT will be repainted in due course.

The call `v.selectNone()` makes the set of selected cells be empty.

In the call `v.selectOnly(this)`, if `this` is in range, make the set of selected cells be exactly `this`; otherwise make the list of selected cells be empty. Equivalent to

```
v.selectNone(); v.select(this,TRUE)
```

In the call `v.select(this,selected)`, if `this` is in range and `selected` is `TRUE`, add `this` to the set of selected cells (without complaint if it's already selected); otherwise if `this` is in range and `selected` is `FALSE`, remove it from the set of selected cells (again without complaint). The VBT will be repainted as necessary in due course.

The call `v.isSelected(this)` returns `TRUE` if `this` is in range and is a selected cell; otherwise it returns `FALSE`.

The call `v.getAllSelected()` returns the set of selected cells. If there are none, it returns a non-`NIL` `REF` to an array of length 0.

The call `v.getFirstSelected(this)` assigns to `this` the lowest-numbered selected cell and returns `TRUE`; if there are no selected cells, it returns `FALSE`.

The call `v.scrollTo(this)` adjusts the list's scrolling position to place

```
MAX (0, MIN (this, v.count () - 1) )
```

at the top of `v`'s domain.

The call `v.scrollToShow(this)` adjusts the list's scrolling position to make `this` visible.

The `ListVBT` will call `v.reportVisible(first, num)` whenever the set of visible cells changes (either because of scrolling or because of reshaping). (A cell is "visible" if it is within the domain of the `ListVBT`; it may not be visible to the user if other windows obscure the `ListVBT`.) The argument `first` is the index of the first visible cell, and `num` is the number of visible cells. The default for this method is a no-op; override it if you need the information it provides. The locking level of the method is `LL.sup = v` (that is, the `ListVBT` itself is locked when the method is called, so the method mustn't operate on `v`).

9.1.3 The Painter

Here is the definition of a `Painter`. In the comments about its methods, `v` is the VBT in which the painting is to take place; it is the `ListVBT.T` or a subtype of it. Recall that `LL.sup = list` for all methods, other than `init`.

```
TYPE
  Painter = OBJECT
            METHODS
              init   (): Painter;
              height (v: VBT.T): INTEGER;
              paint (v        : VBT.T;
                     r        : Rect.T;
                     value    : REFANY;
                     index     : CARDINAL;
                     selected: BOOLEAN;
                     bad      : Rect.T    );
              select (v        : VBT.T;
                      r        : Rect.T;
                      value    : REFANY;
                      index    : CARDINAL;
                      selected: BOOLEAN );
              erase (v: VBT.T; r: Rect.T);
            END;
```

The call `p.init()` initializes `p` as a `Painter` and returns `p`.

The call `p.height(v)` returns the pixel height of each cell if painted in `v`. The list caches the result of this call, so it needn't be very efficient. It is called only when the list has a non-empty domain. It gets re-evaluated whenever the list's screen changes.

The call `p.paint(v, r, value, index, select, bad)` paints the cell with the given index and value in the given rectangle (whose height will equal that returned by `p.height()`, and some part of which will be visible). If `selected` is `TRUE`, highlight the painted cell to indicate that it is in the set of selected cells. `bad` is the subset of `r` that actually needs to be painted; `bad` is wholly contained in `r`.

The call `p.select(v, r, value, index, selected)` changes the highlight of the cell with the given index and value, according to `selected`, to show whether it is in the set of selected cells. The cell has previously been painted; its selection state has indeed changed. It's OK for this method to be identical to `paint`, but it might be more efficient or cause less flicker, e.g. by just inverting `r`.

The call `p.erase(v, r)` paints the given rectangle to show that it contains no cells. Typically, this just fills it with the background color used when painting cells.

9.1.4 TextPainter

Perhaps the most common type of `Painter` is a `TextPainter`. It displays cells whose values are text strings. Here is its public definition:

```
TYPE
  TextPainter <: TextPainterPublic;

  TextPainterPublic =
    Painter OBJECT
    METHODS
      init (bg       := PaintOp.Bg;
            fg       := PaintOp.Fg;
            hiliteBg := PaintOp.Fg;
            hiliteFg := PaintOp.Bg;
            font     := Font.BuiltIn): TextPainter;
      setFont (v: VBT.T; font: Font.T); <* LL.sup < v *>
    END;
```

The call `p.init(...)` initializes `p` as a `TextPainter` and returns `p`. Unselected cells are painted with `fg` text on `bg`; selected cells are painted with `hiliteFg` text on `hiliteBg`; erased areas are painted with `bg`. Text is drawn using `font`.

After the call `p.setFont(v, font)`, the `TextPainter` uses `font` for subsequent painting of values; the call also marks `v` for redisplay. `v` should be the relevant `ListVBT.T`.

9.1.5 The Selector

Here is the definition of `Selector`. Recall that `LL.sup = VBT.mu` for all methods other than `init`.

```
TYPE
  Selector =
    OBJECT
    METHODS
      init        (v: T): Selector;
      insideClick  (READONLY cd: VBT.MouseRec; this: Cell);
      outsideClick (READONLY cd: VBT.MouseRec);
      insideDrag   (READONLY cd: VBT.PositionRec; this: Cell);
      outsideDrag  (READONLY cd: VBT.PositionRec);
    END;
```

The call `s.init(v)` initializes `s` as a `Selector` and returns `s`. The `ListVBT` `v` need not have been initialized before this method is called.

The call `s.insideClick(cd, this)` is called on a `FirstDown` mouse click inside the cell, or on any mouse click inside the cell while we have the mouse focus. On any click other than `LastUp`, the list itself has set a cage so that it receives position reports during subsequent drags.

The call `s.outsideClick(cd)` is called when there is a `FirstDown` click in the `ListVBT` that is not in a cell, or on any mouse click not in a cell while we have the mouse focus. On any click other than `LastUp`, the list itself has set a cage so that it receives position reports during subsequent drags.

The call `s.insideDrag(cd)` is called if the list has received a `FirstDown` click and a subsequent position report with the mouse not in any cell. The list itself has set a cage so that it receives further position reports.

The call `s.outsideDrag(cd)` is called if the list has the mouse focus and receives a subsequent position report with the mouse in this cell. The list itself has set a cage so that it receives further position reports.

9.1.6 UniSelector and MultiSelector

One common class of `Selector` is a `UniSelector`. It maintains the invariant that there is at most one selected cell. On an `insideClick` firstDown, or an `insideDrag`, it removes any previous selection and then selects this cell. Its other methods do nothing. Here is its declaration:

```
TYPE
  UniSelector <: Selector;
```

The other common class of `Selector` is `MultiSelector`. It permits multiple cells to be selected. On an `insideClick` firstDown, it remembers this cell as the *anchor*; if this is not a shift-click, it calls `selectNone` and inverts the selection state of this cell. On an `insideDrag`, it makes the selection state of all cells between this cell and the anchor be the same as that of the anchor. Here is its declaration:

```
TYPE
  MultiSelector <: Selector;

END ListVBT.
```

9.2 The FileBrowserVBT Interface

A `FileBrowserVBT` displays the files in a directory, and allows the user to traverse the file system and to select one or more files. There are two additional widgets that can be associated with a `FileBrowserVBT`. A *helper* is a type-in field that displays the pathname of the directory and allows the user to type new pathnames. A *directory-menu* is a menu containing the names of each level in the directory tree, with the root at the bottom; you can go to any level in the tree by selecting the appropriate item in the menu.

There are two user-actions, selecting and activating.

- The user may *select* items, either by single-clicking on an item to select just that one, or by single-clicking and dragging to select a range. Shift-clicking adds to the selection. A change in selection is reported to the client by invoking the `selectItems` method. The client can read the current selection by calling `GetFile` or `GetFiles`.

- The user may *activate* an item, either by double-clicking on it, or by typing its name in the helper followed by Return.

  Activation of a *file* is reported to the client by invoking the `activateFile` method, whose default is a no-op.

  Activation of a *directory* is reported by invoking the `activateDir` method, whose default behavior is to call `Set` to display the activated directory.

  The client can distinguish between a double-click and Return by looking at the `AnyEvent.T` passed to the activation method. A double-click will be reported as an `AnyEvent.Mouse`, and Return will be reported as an `AnyEvent.Key`.

Directories are indicated in the display by showing some text (e.g., "(dir)") after the name, but that is not part of the pathname returned by `getValue`, `GetFile`, `GetFiles`, or the value passed to `activateDir`.

A background thread calls `Refresh(v)` for every open filebrowser `v`, once per second, to see whether it needs to be updated (although a distributed filesystem may cause a substantial delay before the change is noticed).

`FileBrowserVBT` is internally synchronized.

```
INTERFACE FileBrowserVBT;

IMPORT AnchorSplit, AnyEvent, Font, ListVBT, PaintOp,
       Pathname, Shadow, TextList, TypeinVBT, VBT;

TYPE
  T <: Public;
  Public =
    ListVBT.T OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (font                      := Font.BuiltIn;
            colors: PaintOp.ColorQuad := NIL           ): T;
```

```
        <* LL.sup = VBT.mu *>
        selectItems  (event: AnyEvent.T);
        activateFile (filename: Pathname.T; event: AnyEvent.T);
        activateDir  (dirname : Pathname.T; event: AnyEvent.T);
        error        (err: E);
      END;
```

The call `v.init(...)` initializes `v` as a `FileBrowserVBT`. If `v.painter` is a subtype of `ListVBT.TextPainter`, `init` calls `v.paint.setFont(font)`. The `selector` field must be either `NIL` (in which case a new selector is created) or a subtype of `FileBrowserVBT.Selector`. The initial state of the filebrowser is the current working directory, as returned by `Process.GetWorkingDirectory`.

The implementation calls `v.selectItems(event)` when the user changes the selection using the mouse.

When the user double-clicks on a file in the browser, the implementation calls `v.activateFile(filename, event)`, where `filename` in the absolute pathname corresponding to the first selected item. If the user types Return in the helper, the implementation calls `v.activateFile(filename, event)`, where `filename` is either the pathname in the helper, if that was absolute, or absolute pathname corresponding to

```
    Pathname.Join (GetDir(v), 'helper text', NIL)
```

Don't forget that if `activateFile` is being called because of a double-click, multiple files might be selected in the browser, even though you are given only one in the `filename` parameter.

The implementation calls `v.activateDir(dir)` when a directory is activated. The normal action is simply to set `v` to view that directory, relative to `GetDir(v)`. If an error occurs during the activation, the `error` method is invoked.

The implementation calls `v.error(...)` when an error occurs during user action in `v`, and the `Error` exception cannot be raised (e.g., because it happened in a separate thread). Some examples of errors are as follows: the user has typed a nonexistent directory in the path; the current directory has become inaccessible; the user has no permission to read the directory. The default method is a no-op. By overriding this method, the client can provide better information to the user.

The `error` method is passed an `E` object containing information about the error that occurred. Here is its definition:

```
  EXCEPTION Error (E);
  TYPE
    E = OBJECT
          v   : T;
          text: TEXT       := "";
          path: Pathname.T := ""
    END;
```

The argument to the `Error` exception includes the `FileBrowserVBT` itself, along with a descriptive message and the pathname in question when the error occurred.

Finally, if you create a subtype of `FileBrowserVBT` (which is a subtype of `ListVBT.T`) and you specify a selector for it, it must be a subtype of `Selector`:

```
TYPE Selector <: ListVBT.MultiSelector;
```

9.2.1 The Helper

The FileBrowser's helper (see page 67) is a `TypeinVBT`. Once the user types in the helper, any selected items in the browser are unselected. If the user types Return in the browser, that will activate the name in the Helper.

If an error occurs during the activation, the `error` method of the filebrowser to which the helper is attached will be invoked.

```
TYPE Helper <: TypeinVBT.T;

PROCEDURE SetHelper (v: T; helper: Helper) RAISES {Error};
<* LL.sup = VBT.mu *>
```

Sets the helper for `v` to be `helper`, and fills it with `GetDir(v)`.

9.2.2 The Directory-Menu

The directory menu shows the name of each of the parent directories, going back to the root directory.

```
TYPE
  DirMenu <: PublicDirMenu;
  PublicDirMenu =
    AnchorSplit.T OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (font             := Font.BuiltIn;
            shadow: Shadow.T := NIL;
            n     : CARDINAL := 0             ): DirMenu;
      <* LL.sup = VBT.mu *>
      setFont (font: Font.T);
    END;
```

The `font` and `shadow` control the appearance of the text within the menu. As usual, if `shadow` is `NIL`, then `Shadow.None` is used instead. The parameter `n` is used by `AnchorSplit` to determine the `ZSplit` in which to install the menu.

```
PROCEDURE SetDirMenu (v: T; dm: DirMenu);
<* LL.sup = VBT.mu *>
```

Sets the directory-menu of `v` to be `dm` and fill it with the current directory.

9.2.3 FileBrowser options

A file browser can be "read-only":

```
PROCEDURE SetReadOnly (v: T; readOnly: BOOLEAN);
<* LL.sup = VBT.mu *>
```

Change the "read-only" mode of `v` to be `readOnly`.

If a file browser is "read-only" then in subsequent calls to

```
v.activateFile(filename)
```

`filename` is guaranteed to exist. Otherwise, the user can type the name of a non-existing file into the helper. A newly initialized `FileBrowserVBT` is not read-only.

By default all files in the directory are displayed, but the following procedure can be used to filter which files are shown:

```
PROCEDURE SetSuffixes (v: T; suffixes: TEXT);
<* LL.sup = VBT.mu *>
```

Specify which `suffixes` are to be displayed.

If `suffixes` is not the empty string, only files with the specified suffixes (and all directories) will be displayed. The format of `suffixes` is a sequence of suffixes (not including the period) separated by non-alphanumeric characters (e.g., spaces). The special suffix `$` indicates "files with no suffix." Calling `SetSuffixes` procedure does not force `v` to be redisplayed.

9.2.4 Setting the displayed directory

```
PROCEDURE Set (v       : T;
               pathname: Pathname.T;
               time    : VBT.TimeStamp := 0) RAISES {Error};
<* LL.sup = VBT.mu *>
```

Set the display state of v.

The `pathname` may be absolute or relative; if it's relative, it is relative to the current displayed directory.

If `pathname` refers to a non-existent or inaccessible directory, `Error` will be raised. The exception will also be raised if `pathname` refers to a non-existent file and `v` is read-only.

If `time` is not zero and there is a helper, then the helper will take the keyboard focus and will display its new contents in replace-mode, ready for the user to type something in its place.

```
PROCEDURE Unselect (v: T);
<* LL.sup = VBT.mu *>
```

Put `v` into the no-selection state, without changing the current directory. Equivalent to `v.selectNone()`.

```
PROCEDURE Refresh (v: T) RAISES {Error};
<* LL.sup = VBT.mu *>
```

Update the display without changing the directory.

If `v`'s domain is not empty, and its directory has been `Set`, and the directory has changed since the last time it was displayed, then `v` will be marked for redisplay. `Error` is raised only if the directory has become inaccessible for some reason; in this case, the browser goes to the empty state, so that if the client catches `Error` and takes no other action, the browser will be empty but not broken.

9.2.5 Retrieving selections from the browser

```
PROCEDURE GetFiles (v: T): TextList.T RAISES {Error};
<* LL.sup = VBT.mu *>
```

Return the current selections of `v`, or `NIL` if there are no selections. The list includes "full" pathnames; they satisfy `Pathname.Absolute`, but they may contain symbolic links. Use `FS.GetAbsolutePathname` to get a pathname with no symbolic links.

```
PROCEDURE GetFile (v: T): Pathname.T RAISES {Error};
<* LL.sup = VBT.mu *>
```

Return the first selection, or the empty string if there are no selections.

```
PROCEDURE GetDir (v: T): Pathname.T;
<* LL.sup = VBT.mu *>
```

Return the current displayed directory of `v`. Returns an empty string if `v` is in the "empty" state.

```
END FileBrowserVBT.
```

9.3 The NumericVBT Interface

A `NumericVBT` is a VBT class for displaying and changing an integer within some range. A `NumericVBT` has three parts (from left to right): a minus button, a type-in field, and a plus button. The type-in field is restricted to contain an integer within a specified range; it can be changed by editing (it uses the default editing model), or by typing Return, or by clicking on the plus or minus buttons. The plus/minus buttons are trill buttons, so clicking and holding will cause the value of the `NumericVBT` to continuously increment/decrement.

The `NumericVBT` has a `callback` method that is called each time the user types Return or click the plus or minus button. The default callback method is a no-op.

```
INTERFACE NumericVBT;

IMPORT AnyEvent, Filter, Font, Shadow, TypeinVBT,  VBT;

TYPE
  T <: Public;
  Public = Filter.T OBJECT
             typein: Typein := NIL; (* READONLY after init *)
           METHODS
             <* LL.sup <= VBT.mu *>
             init (min       : INTEGER  := FIRST (INTEGER);
                   max       : INTEGER  := LAST (INTEGER);
                   allowEmpty: BOOLEAN  := FALSE;
                   naked     : BOOLEAN  := FALSE;
                   font      : Font.T   := Font.BuiltIn;
                   shadow    : Shadow.T := NIL             ):
               T;
             callback (event: AnyEvent.T);
           END;
  Typein <: TypeinVBT.T;
```

The call to `v.init(...)` initializes `v` as a `NumericVBT` and returns `v`. The integer stored with `v`, referred to as "the value in" `v`, is constrained to be in the range

```
   [min .. MAX (min, max)]
```

The initial value in `v` is equal to `min`.

If `allowEmpty` is `TRUE`, then "empty" (no text in the type-in area) is a distinct and valid state, and can be tested by the procedure `IsEmpty`. The call `Get(v)` in the empty state will return `FIRST(INTEGER)`, regardless of whether this is in the valid range. Clicking the plus/minus buttons has no effect when `v` is in the empty state.

If `naked` is `TRUE`, then the numeric interactor appears as just a type-in field, without plus or minus buttons.

IF `v.typein` is `NIL` when `v.init(...)` is called, then a new `Typein` will be allocated and assigned to `v.typein`. Whether or not it was `NIL` at the time of the call, it will be initialized via

```
   v.typein(FALSE, 1.5, 1.5, font, shadow)
```

That is, it will not be expandable, its margins will be 1.5 mm, and `font` and `shadow` will determine its appearance.

The implementation calls

```
    v.callback(event)
```

when the user clicks the plus/minus buttons, or types Return in the type-in area. The `event` parameter reports the details of the event as either an `AnyEvent.Mouse` (clicking on the plus/minus buttons) or an `AnyEvent.Key` (typing Return in the type-in area). The value in `v` is changed before `v.callback` is invoked.

The value in `v` is range-checked before the callback is called, and in every call to `Get`. If the number is out of range, the nearest number in range will be written into the type-in area, and that value will be returned to the caller of `Get`.

```
PROCEDURE Put (v: T; n: INTEGER);
<* LL.sup = VBT.mu *>
```

Change the value in `v` to be

```
        MIN(GetMax(v), MAX(GetMin(v), n))
```

and display this value in the type-in area. Note that `v.callback` is not invoked.

```
PROCEDURE PutBounds (v: T; min, max: INTEGER);
<* LL.sup = VBT.mu *>
```

Change `v.min` to be `min` and `v.max` to be `MAX(min, max)`, and then call `Put(v, Get(v))`. The call to `Put` has the effect of projecting the value of `v` into the new bounds.

```
PROCEDURE Get (v: T)    : INTEGER; <* LL.sup = VBT.mu *>
```

Return the current value in `v`. This value is range-checked, in case the user typed an out-of-range value without typing Return.

```
PROCEDURE GetMin (v: T): INTEGER; <* LL.sup = VBT.mu *>
PROCEDURE GetMax (v: T): INTEGER; <* LL.sup = VBT.mu *>
```

Return the indicated value associated with `v`.

```
PROCEDURE SetEmpty (v: T);
<* LL.sup = VBT.mu *>
```

Set `v` to the empty state. This is a no-op unless `allowEmpty` was `TRUE` when `v` was initialized.

```
PROCEDURE IsEmpty (v: T): BOOLEAN;
```

```
<* LL.sup = VBT.mu *>
```

Test whether `v` is in the empty state. If `allowEmpty` was not `TRUE` when `v` was initialized, this procedure will always return `FALSE`.

```
PROCEDURE TakeFocus (v          : T;
                     time       : VBT.TimeStamp;
                     alsoSelect : BOOLEAN       := TRUE):
  BOOLEAN;
<* LL = VBT.mu *>
```

Cause the type-in area to grab the keyboard focus. If the focus could be grabbed and if `alsoSelect` is set, the type-in area will make its entire text the primary selection. Returns whether the keyboard focus could be acquired.

```
END NumericVBT.
```

9.4 The ScrollerVBT Interface

A `ScrollerVBT` is a scrollbar with an orientation along an *axis*. For the sake of brevity in this interface, we'll only talk about vertical scrollers. For horizontal scrollers, replace phrases like *top and bottom edges* by *left and right edges*, and so on.

Like a `NumericVBT`, a `ScrollerVBT` provides a *bounded-value* abstraction. That is, a `ScrollerVBT` has a *value* associated with it, and that value is guaranteed to stay within some bounds. Various user gestures change the value and invoke a `callback` method on the `ScrollerVBT`. The callback method can inquire the value of the scrollbar, and can change the value and bounds.

Visually, a scrollbar contains a *stripe* that spans some fraction of the height of the scrollbar and is slightly narrower than the scrollbar. The stripe represents the value of the scrollbar. Various user-gestures cause the stripe to move.

More specifically, the state of a `ScrollerVBT` consists of five integer quantities: `min`, `max`, `thumb`, `step`, and `value`. The `value` is guaranteed to stay in the range `[min .. max-thumb]`. Visually, the `value` is represented by the position (top edge) of a stripe in the scroller, and `thumb` by the length of the stripe. The amount that `value` should change when continuous scrolling is given by `step`, the *stepping* amount.

Although each `VBT` class that uses a `ScrollerVBT` is free to associate any meaning with the length of the stripe, the following convention is suggested for using scrollbars to view an object:

> The ratio of the height of the stripe to the height of the scrollbar should be the same as the ratio of the amount of the object visible vertically to its entire height. The position of top of the stripe reflects the position of top of the view of the object within the entire object.

Here is some terminology and the user-interface provided by a `ScrollerVBT`:

- To *scroll* means to left-click or right-click in the scrollbar.
- You need to release the button relatively quickly, or else you'll start *continuous scrolling.* You stop continuous scrolling by releasing the button, by chord-cancelling or by moving the mouse.
- When you move the mouse, you are then using *proportional scrolling.* This means that the more that you move the mouse vertically, the more the stripe will be moved in the direction of the mouse movement. You stop proportional scrolling by upclicking or chord-cancelling.
- The left and right buttons are inverses: the left button moves the stripe downward and the right button moves the stripe upward.
- You *thumb* with a middle-click. The top of the stripe moves to the position of the cursor. Thus, middle-click above the top of the stripe moves the stripe up, and middle-click below the top moves the stripe down.
- Middle-drag causes *continuous thumbing.* As you drag to a new position, the top of the stripe moves to match the current cursor position. You stop continuous thumbing by middle-upclicking or chord-canceling.

If you want a different user interface, you need to subclass various methods (e.g., a `thumb`, `scroll`, `autoscroll`) of the scrollbar. These methods are defined in the `ScrollerVBTClass` interface.

```
INTERFACE ScrollerVBT;

IMPORT Axis, PaintOp, VBT;

TYPE
  T <: Public;
  Private <: VBT.T;
  Public = Private OBJECT
           METHODS
             <* LL.sup = VBT.mu *>
             init (axis  : Axis.T;
                   min   : INTEGER;
                   max   : INTEGER;
                   colors: PaintOp.ColorQuad;
                   step  : CARDINAL            := 1;
                   thumb : CARDINAL            := 0  ): T;
             <* LL = VBT.mu *>
             callback (READONLY cd: VBT.MouseRec);
           END;
```

The call to `v.init(...)` initializes `v` as a `ScrollerVBT` in the `axis` orientation. It is displayed using `colors`.

The implementation calls `v.callback(cd)` after `v`'s value has been changed by the user; it is not called when the value is changed as the result of calls to `Put` or `PutBounds`. The default `callback` method is a no-op.

```
PROCEDURE Put (v: T; n: INTEGER);
<* LL.sup = VBT.mu *>
```

Change the value of `v`, projected to `[min .. max-thumb]`, and mark `v` for redisplay.

```
PROCEDURE PutBounds (v    : T;
                     min  : INTEGER;
                     max  : INTEGER;
                     thumb: CARDINAL  := 0);
<* LL.sup = VBT.mu *>
```

Set the bounds, project `v`'s value into `[min .. max-thumb]`, and mark `v` for redisplay.

```
PROCEDURE PutStep (v: T; step: CARDINAL);
<* LL.sup = VBT.mu *>
```

Change the amount that `v`'s value should change while continuous scrolling to `step`. If `step = 0`, scrolling will be disabled.

```
PROCEDURE Get      (v: T): INTEGER;  <* LL.sup = VBT.mu *>
PROCEDURE GetMin   (v: T): INTEGER;  <* LL.sup = VBT.mu *>
PROCEDURE GetMax   (v: T): INTEGER;  <* LL.sup = VBT.mu *>
PROCEDURE GetThumb (v: T): CARDINAL; <* LL.sup = VBT.mu *>
PROCEDURE GetStep  (v: T): CARDINAL; <* LL.sup = VBT.mu *>
```

Return the current `value`, `min`, `max`, `thumb`, and `step`.

```
END ScrollerVBT.
```

10 Miscellaneous Filters

10.1 The FlexVBT Interface

The `FlexVBT.T` is a filter whose shape is based on a *natural* size with some *stretch* and *shrink*. If a natural amount is left unspecified, the stretch and shrink are applied relative to the child's size. If a stretch or shrink is left unspecified, 0 is assumed. All units are specified in millimeters. See Figure 10.1 for examples.

This interface is similar to `RigidVBT`, but more powerful in that one can specify a size based on a child's size and can dynamically change the size specification. Also, it presents a slightly different model to the client: In `RigidVBT`, one thinks in terms of the low and high bounds of some range. Here, one thinks in terms of the amount thed natural size value can be stretched and shrunk.

```
INTERFACE FlexVBT;

IMPORT Axis, Filter, VBT;

CONST
  Large    = 99999.0;
  Missing  = -Large;
  Infinity = Large;

TYPE
  SizeRange = RECORD natural, shrink, stretch: REAL END;
  Shape     = ARRAY Axis.T OF SizeRange;
```

Some useful shapes are defined at the end of this interface.

```
TYPE
  T <: Public;
  Public = Filter.T OBJECT
           METHODS
             <* LL.sup <= VBT.mu *>
             init (ch: VBT.T; READONLY sh := Default): T
           END;
```

The call `v.init(ch, sh)` initializes `v` as a `FlexVBT` with child `ch` and shape specification `sh`. The default shape causes `v` to be a no-op: it will simply return the shape of its child as its own.

```
PROCEDURE FromAxis (          ch: VBT.T;
                              ax: Axis.T;
                     READONLY sh: SizeRange := DefaultRange): T;
<* LL.sup <= VBT.mu *>
```

Return a `FlexVBT` whose shape specification in the `ax` dimension is `sh` and whose shape in the other dimension is that of `ch`.

```
PROCEDURE Set (v: T; READONLY sh: Shape);
<* LL.sup = VBT.mu.v *>
```

Change the shape of `v` to `sh`, and notify `v`'s parent that `v`'s size has changed.

```
PROCEDURE Get (v: T): Shape;
<* LL.sup = VBT.mu.v *>
```

Get the shape of `v`.

```
PROCEDURE SetRange (v: T; ax: Axis.T; READONLY sr: SizeRange);
<* LL.sup = VBT.mu.v *>
```

Change the shape of `v` to `sr` along the `ax` axis, and notify `v`'s parent that `v`'s size has changed.

The rest of this interface defines some useful shapes: `Default` uses child's size; `Fixed` uses child's preferred, removing all shrink and stretch; `Stretchy` uses child's preferred and shrink, giving infinite stretch; and `Rigid` is a procedure to set a shape to a specified natural size, with neither stretch nor shrink.

```
CONST
  Default  = Shape{DefaultRange, DefaultRange};
  DefaultRange =
    SizeRange {natural := Missing,
               shrink  := Missing,
               stretch := Missing};

  Fixed   = Shape{FixedRange, FixedRange};
  FixedRange =
    SizeRange {natural := Missing,
               shrink  := 0.0,
               stretch := 0.0};

  Stretchy = Shape{StretchyRange, StretchyRange};
  StretchyRange =
    SizeRange {natural := Missing,
               shrink  := Missing,
               stretch := Infinity};

PROCEDURE RigidRange (natural: REAL): SizeRange;
<* LL = arbitrary *>
```

Return a `SizeRange` with the specified natural amount and with no stretch or shrink. Equivalent to

```
        SizeRange {natural, 0.0, 0.0}
```

```
PROCEDURE Rigid (hNat, vNat: REAL): Shape;
<* LL = arbitrary *>
```

Return a `Shape` *with the specified natural amounts long the horizontal and vertical axes and with no stretch or shrink. Equivalent to*

```
        Shape {SizeRange {hNat, 0.0, 0.0},
               SizeRange {vNat, 0.0, 0.0}}
```

```
END FlexVBT.
```

10.2 The ReactivityVBT Interface

A `ReactivityVBT` is a filter that can make its child active, passive, dormant, and invisible. The *active* state does nothing; mouse and keyboard events are relayed to child. The *passive* state doesn't allow mouse or keyboard events to go to the child. The *dormant* state doesn't send mouse or keyboard events to the child; it also grays out the child. The *vanish* state also doesn't send mouse or keyboard events to go to the child; in addition, it draws over the child in the background color, thereby making it invisible.

When the state of a `ReactivityVBT` is set, the caller also specifies a cursor to be used.

If a VBT-descendant of a `ReactivityVBT` is painted, it will appear correctly. For example, if the `ReactivityVBT` is in the vanished state, the descendant will not appear until the state changes; if the `ReactivityVBT` is in a dormant state, the descendant will be grayed out.

A `ReactivityVBT` also passes on any miscellaneous events to take the keyboard focus to the descendant that last acquired the keyboard focus.

```
INTERFACE ReactivityVBT;

IMPORT Cursor, ETAgent, PaintOp, Rect, VBT;

TYPE
  State = {Active, Passive, Dormant, Vanish};
  T <: Public;
  Public =
    ETAgent.T OBJECT
    METHODS
      <* LL.sup <= VBT.mu *>
      init (ch: VBT.T; colors: PaintOp.ColorScheme := NIL): T;
      <* LL = VBT.mu.v *>
      paintDormant (r: Rect.T; colors: PaintOp.ColorScheme);
    END;
```

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 $q + r$.

The call `v.init(..)` initializes `v` as a `ReactivityVBT` with child `ch` and with an initial state of `Active`. If `colors` is `NIL`, then `PaintOp.bgFg` is used instead. The `colors` are used to draw the vanished and dormant states.

The implementation calls `v.paintDormant(r, colors)` to paint the part of `ch` bounded by rectangle `r` when `v`'s state is `Dormant`. The "current colors" of `v` are passed as `colors`. Initially, the current colors are those that were specified when the `ReactivityVBT` was initialized. They can be changed using the `SetColors` procedure. The default method paints a `Pixmap.Gray` texture using `colors.transparentBg`.

```
PROCEDURE Set (v: T; state: State; cursor: Cursor.T);
<* LL.sup = VBT.mu *>
```

Change `v`'s state and cursor, and mark `v` for redisplay.

```
PROCEDURE Get (v: T): State;
<* LL.sup = VBT.mu *>
```

Retrieve `v`'s current state.

```
PROCEDURE GetCursor (v: T): Cursor.T;
<* LL.sup = VBT.mu *>
```

Retrieve `v`'s current cursor.

```
PROCEDURE SetColors (v: T; colors: PaintOp.ColorScheme);
<* LL.sup = VBT.mu *>
```

Change the colors that `v` uses for the `Dormant` and `Vanish` states. If `v` is currently in the `Dormant` or `Vanish` state, mark `v` for redisplay.

```
END ReactivityVBT.
```

10.3 The ScaleFilter Interface

A `ScaleFilter` is a multi-filter whose child's screentype is the same as the parent's except that the resolution is scaled.

```
INTERFACE ScaleFilter;

IMPORT VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Private <: VBT.T;
  Public = Private OBJECT
           METHODS
```

```
          <* LL.sup <= VBT.mu *>
          init (ch: VBT.T): T
        END;
```

The call `v.init(ch)` initializes `v` as a `ScaleFilter` with multi-child `ch` and with horizontal and vertical scale factors both equal to 1.0.

There are two ways you can use a `ScaleFilter`: Procedure `Scale` allows you to explicitly set a horizontal and vertical scale factor. Procedure `AutoScale` looks at the preferred size of the child and dynamically sets the scale factors such that the child's preferred size always fills its domain.

```
PROCEDURE Scale (v: T; hscale, vscale: REAL);
<* LL.sup = VBT.mu.v *>
```

Set `v`'s horizontal and vertical scale factors to be `hscale` and `vscale` respectively, and mark `v` for redisplay.

Thus, if the `v` has resolution of `px` and `py` horizontally and vertically, then the resolution of `v`'s multi-child will be `hscale*px` and `vscale*py`.

Note that the locking level of `Scale` does not require the full share of `VBT.mu`. Therefore, it can be called from `v`'s `reshape` or `rescreen` method, for example, since those methods are called with only `v`'s share of `VBT.mu` locked. This fact is useful for the implementation of procedure `AutoScale`:

```
PROCEDURE AutoScale (v: T; keepAspectRatio := FALSE);
<* LL.sup = VBT.mu *>
```

Set `v`'s scale factor such that the preferred size of `v`'s child `ch` is scaled to fit into `VBT.Domain(ch)`. If `keepAspectRatio` is `TRUE`, then `ch` is scaled by the same amount `f` both horizontally and vertically. The amount `f` is chosen so that the preferred size of `ch` just fits in the larger direction of `v` and fits fine in the other direction. In any event, `v` is marked for redisplay.

The call to `AutoScale` has the effect of causing `Scale` to be called each time that `v` is reshaped. Thus, it is important that `Scale` have a locking level of `VBT.mu.v` rather than simply `VBT.mu`.

```
PROCEDURE Get(v: T; VAR (* OUT *) hscale, vscale: REAL);
<* LL.sup = VBT.mu *>
```

Return `v`'s current horizontal and vertical scale factors.

If `Scale` was called more recently than `AutoScale`, then `Get` returns the values passed to `Scale`. On the other hand, if `AutoScale` was called more recently, then `Get` will return values that reflect scaling for `v`'s current domain.

```
END ScaleFilter.
```

10.4 The ViewportVBT Interface

A `ViewportVBT` is a multi-filter that displays multiple views of a child `VBT`, with optional horizontal and vertical scrollbars. When the child's preferred size is larger than the viewport's *interior* (that is, the screen of the viewport minus the scrollbars), the child is reformatted to its preferred size. Since only part of the child is visible, the user can pan the child using the scrollbars. When the child's preferred size is smaller than the viewport's screen, the child is reformatted to the size of the viewport interior, and the scrollbars are ineffective.

Views may be added or deleted under program control or by appropriate gestures in the scrollbar: Option Left click adds a new view after the view in which the user clicked. Option Right click removes the view (unless, of course, it would leave the viewport with zero views!).

```
INTERFACE ViewportVBT;

IMPORT Axis, HVSplit, Rect, Shadow, VBT;

TYPE
  <* SUBTYPE T <: MultiFilter.T *>
  T <: Public;
  Public = HVSplit.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (ch             : VBT.T;
                   axis           : Axis.T   := Axis.T.Ver;
                   shadow         : Shadow.T := NIL;
                   step           : CARDINAL := 10;
                   adjustableViews: BOOLEAN  := TRUE;
                   scrollStyle := ScrollStyle.AlaViewport;
                   shapeStyle  := ShapeStyle.Unrelated): T;
           END;
```

The call to `v.init(..)` initializes `v` as a `ViewportVBT.T`. The `axis` parameter says whether the views are arranged vertically or horizontally. `step` is the number of pixels to move while auto-scrolling. `shadow` gives the shadow for displaying scrollbars, resets and hvbars. When `adjustableViews` is `TRUE`, an `HVBar` will be inserted between views so the user can adjust the screen allocated to each view. `scrollStyle` and `shapeStyle` are explained below.

The internal structure of a viewport is a rather complex collection of `JoinedVBT`s, `HVSplits`, `ScrollerVBT`s, and others. It depends on the options with which the viewport was created. Be sure to use the `MultiFilter` interface to get at the child.

```
TYPE
  View = INTEGER;
```

A `View` is an internal ID for a view. The value is valid for the life of a view (i.e., until it is removed by a user gesture or by a call to `RemoveView`). Thereafter, the ID may be reused. The initial view created by the `init` method has a value of 0.

A viewport can be created with a number of different styles of scrollbars:

```
TYPE
  ScrollStyle =
    {HorAndVer,
     HorOnly,
     VerOnly,
     NoScroll,
     AlaViewport,
     Auto};
```

The styles are as follows:

- `HorAndVer` puts a horizontal and vertical scrollbar on every view. In addition, nestled between the scrollbars in the southwest corner, there's a little "reset" button that moves the northwest corner of the child to the northwest corner of the view.
- `HorOnly` places a scrollbar at the bottom.
- `VerOnly` places a scrollbar at the left side.
- `NoScroll` specifies that views will not have scrollbars.
- `AlaViewport` specifies that there is a scrollbar in the same axis as the viewport. Thus, `AlaViewport` for a vertical viewport is equivalent to `VerOnly`.
- `Auto` specifies that scrollbars appear only when the preferred size of the child exceeds the size of the viewport (in that dimension).

There are two possible shape-relationships between a viewport and its child:

```
TYPE ShapeStyle = {Unrelated, Related};
```

`Unrelated` makes the shape of the child equal to its preferred shape—completely unrelated to the viewport's current shape.

`Related` makes the child's shape equal to the viewport's shape in the non-axis direction of the viewport. In the viewport's axis direction, the child's preferred shape is used. For example, the width of the child in a `Vertical` viewport is the width of the viewport.

10.4.1 Panning the contents

```
PROCEDURE ScrollTo (         v    : T;
                    READONLY r    : Rect.T;
                             view : View     := 0;
                             force: BOOLEAN  := TRUE);
<* LL = VBT.mu *>
```

Scroll the viewport `v` so that rectangle `r` is visible in view `view`. Rectangle `r` will be roughly centered within `v`, but if `r` is too big to be seen entirely, its northwest corner will be made visible. If `force` is `FALSE` and `r` is already entirely visible, this procedure is a no-op.

```
PROCEDURE Normalize (v: T; w: VBT.T; view: View := 0);
<* LL = VBT.mu *>
```

If the domain of `w` is non-empty and it's entirely visible, do nothing. Otherwise, do a `ScrollTo` to `w`'s domain in view `view`.

At first blush, `Normalize` seems to be just a call to

```
ScrollTo(v, VBT.Domain(w), FALSE)
```

However, if `w` doesn't have a domain, as is the case when `w` has been recently installed and the `VBT` tree has not been redisplayed, a thread is forked to wait until it can acquire `VBT.mu` (recall that `Normalize` and `ScrollTo` have `LL = VBT.mu`). After the lock is acquired, all pending redisplays are satisfied, and then `ScrollTo` of `w`'s domain is invoked. Since the thread executes outside event-time, it explicitly causes all marked `VBT`s to be redisplayed after it calls `ScrollTo`.

10.4.2 Multiple views

```
PROCEDURE AddView (v: T; pred: View := -1; split := TRUE):
  View;
<* LL = VBT.mu *>
```

Add another view after the view `pred` (-1 means add as the first view) of the child. If `split` is `TRUE`, then the new view and the view `pred` will occupy the area previously occupied by the view `pred`. The area of all other views will be unchanged. The value returned is an internal ID for the view. This value may be reused after the view has been removed.

```
PROCEDURE RemoveView (v: T; view: View);
<* LL = VBT.mu *>
```

Remove the view `view` from `v`'s child. The ID for the initial view created by the `init` method is 0.

```
END ViewportVBT.
```

11 Miscellaneous Splits

11.1 The SplitterVBT Interface

A `SplitterVBT.T` is a parent window that partitions its screen into a row or column of children windows, depending on the *axis* of the split, with adjusting bars between all children. The adjusting bars allow the user to adjust the allocation of screen real estate among the splitter's children, subject to the size constraints of each child.

A `SplitterVBT` is subclass of an `HVSplit`, but through the `MultiSpit` interface, only the "interesting" children of the `HVSplit` are exposed. That is, adjusting bars are never exposed to the client: they are inserted automatically when a new child is added, and removed as necessary. To access all children, including the adjusting bars, use the `Split` interface instead. The `HVSplit` routines `Move`, `Adjust`, `FeasibleRange`, `AvailSize`, and `AxisOf` can be used.

```
INTERFACE SplitterVBT;

IMPORT Axis, HVSplit, PaintOp, Pixmap;

TYPE
  <* SUBTYPE T <: MultiSplit.T *>
  T <: Public;
  Public = HVSplit.T OBJECT
           METHODS
             <* LL <= VBT.mu *>
             init (hv      : Axis.T;
                   size    : REAL      := DefaultSize;
                   op      : PaintOp.T := PaintOp.BgFg;
                   txt     : Pixmap.T  := Pixmap.Gray;
                   saveBits: BOOLEAN   := FALSE;
                   parlim  : INTEGER   := -1            ): T;
           END;
```

The call `v.init(...)` initializes `v` as a `SplitterVBT` with no children. See the `HVSplit` interface for an explanation of parameters `saveBits` and `parlim`. See the `HVBar` interface for an explanation of the `size`, `op`, and `txt` parameters.

```
CONST
  DefaultSize = 2.0;

END SplitterVBT.
```

12 Installing Top-Level Windows

This section contains interfaces that support the processing of the X11 `-display` and `-geometry` command-line options. If your application is installing a single top-level window, the `XTrestle` interface will probably suffice; otherwise, you'll need to use the routines in `XParam` for processing the command-line options, and use routines in `Trestle` (not `XTrestle`) for installing the windows.

12.1 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* [5].

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).

12.2 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.
```

12.2.1 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.

13 Utilities

This section contains a variety of utility interfaces that clients of VBTkit and implementors of new VBTkit widgets might find useful.

13.1 The AnyEvent Interface

An `AnyEvent.T` is an object that can hold any of the Trestle event-time events. This object type is useful for `VBT` methods that are called in response to multiple styles of user gestures. For instance, the `callback` method of a `NumericVBT` is invoked either because a user clicked on the plus or minus button or because the user typed a carriage return in the type-in field. The Trestle event is passed to the `callback` method as an `AnyEvent.T`, and the `callback` method can then use a `TYPECASE` to differentiate button clicks from carriage returns, and to retrieve the data specific to each type of event.

The locking level is arbitrary for all procedures in this interface.

```
INTERFACE AnyEvent;

IMPORT VBT;

TYPE
  T = BRANDED OBJECT END;
  Key = T OBJECT key: VBT.KeyRec END;
  Mouse = T OBJECT mouse: VBT.MouseRec END;
  Position = T OBJECT position: VBT.PositionRec END;
  Misc = T OBJECT misc: VBT.MiscRec END;
```

The four subtypes of `AnyEvent.T` correspond to the four event-time Trestle events: keyboard, mouse, position, and miscellaneous.

```
PROCEDURE FromKey (
            READONLY event: VBT.KeyRec): Key;
PROCEDURE FromMouse (
            READONLY event: VBT.MouseRec): Mouse;
PROCEDURE FromPosition (
            READONLY event: VBT.PositionRec): Position;
PROCEDURE FromMisc (
            READONLY event: VBT.MiscRec): Misc;
```

Return `event` as an appropriate subtype of `AnyEvent.T`.

```
PROCEDURE TimeStamp (anyevent: T): VBT.TimeStamp;
```

Return the timestamp of the `anyevent`. It is a checked runtime error if `anyevent` is not a proper subtype of `AnyEvent.T`.

```
END AnyEvent.
```

13.2 The AutoRepeat Interface

The `AutoRepeat` interface provides support for calling a procedure repetitively. Auto-repeating typically takes place while a key or mouse button is held down, although there is no direct relation between `AutoRepeat` and `VBT`s.

When an auto-repeat object `ar` is activated, it forks a *timer thread* that calls `ar.repeat()` after `firstWait` milliseconds, and every `period` milliseconds thereafter. However, there is a flow-control mechanism: if the call to `ar.repeat()` has not returned by the time the next repetition is scheduled to take place, the timer thread will wait. That is, repetitions cannot queue up more than one deep.

An auto-repeat object `ar` is activated by a call to `Start(ar)`, terminated by a call to `Stop(ar)`, and resumed by a call to `Continue(ar)`.

All locking is handled within `AutoRepeat`; calls to `Start(ar)`, `Stop(ar)`, and `Continue(ar)` are serialized on a per-`ar` basis. These procedures may be called by a `repeat` method. Clients must not call the `repeat` method directly; it is called by the timer thread subject to client-calls to `Start`, `Stop`, and `Continue`. The `AutoRepeat` interface will never call a `repeat` method re-entrantly.

```
INTERFACE AutoRepeat;

TYPE
  Milliseconds = CARDINAL;

CONST
  DefaultFirstWait: Milliseconds = 500;
  DefaultPeriod   : Milliseconds = 66;

TYPE
  T <: Public;
  Public =
    Private OBJECT
    METHODS
      init (firstWait: Milliseconds := DefaultFirstWait;
            period   : Milliseconds := DefaultPeriod): T;
      repeat ();
      canRepeat(): BOOLEAN;
    END;
  Private <: ROOT;
```

The call `ar.init(firstWait, period)` initializes `ar` as an `AutoRepeat.T`, and it returns `ar`. The `firstWait` and `period` parameters are stored internally for use by the `Start` and `Continue` procedures.

The call `ar.canRepeat` should return `FALSE` whenever there's reason to suspect that a client might want to call `Stop` in the near future. The next call to `ar.repeat` will be suspended for `period` milliseconds. The default for this method always returns `TRUE`.

The `canRepeat` method is intended for situations when a `repeat` method takes more time than `period` milliseconds to complete. The problem with slow `repeat` methods is that the scheduler might decide to always run the timer thread (since it will want to call the `repeat` method as soon as the slow `repeat` method completes), thereby blocking another thread from being able to call `Stop`.

The default `repeat` method is a no-op.

```
PROCEDURE Start (ar: T);
```

Initiate auto-repeating for `ar`*.*

`Start(ar)` forks a timer thread that will wait `ar.firstWait` milliseconds before calling `ar.repeat()` the first time, then `ar.period` milliseconds between subsequent calls to `ar.repeat()`. This procedure is a no-op if `ar` is already running.

```
PROCEDURE Stop (ar: T);
```

Stop auto-repeating as soon as possible.

After calling `Stop(ar)`, the implementation will not call `ar.repeat()` again until a call to `Start(ar)` or `Continue(ar)` restarts auto-repeating. This procedure is a no-op if `ar` is not already running.

It is possible (but unlikely) that `ar.repeat()` is called one more time after a call to `Stop(ar)` returns. This can happen because calls to `ar.repeat` are not serialized with respect to the call to `Stop(ar)`. They are not serialized in order to allow a `repeat` method to call `Stop`.

```
PROCEDURE Continue (ar: T);
```

Resume auto-repeating immediately.

`Continue(ar)` is like `Start(ar)`, except rather than waiting `ar.firstWait` milliseconds as in the call to `Start(ar)`, the timer thread calls `ar.repeat` without waiting at all. Subsequent calls to `ar.repeat()` happen every `period` milliseconds, as usual. This procedure is a no-op if `ar` is already running.

```
END AutoRepeat.
```

13.3 The Rsrc Interface

Resources are arbitrary texts 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; each element is either a `Pathname.T` that refers to a directory, or a `Bundle.T`, typically created by `m3bundle`.

```
INTERFACE Rsrc;

IMPORT RefList, Rd, Thread;

TYPE Path = RefList.T; (* of Pathname.T or Bundle.T *)

EXCEPTION NotFound;

PROCEDURE Open (name: TEXT; path: Path): Rd.T
  RAISES {NotFound};
```

If `name` is an absolute pathname, then look for `name` in the file system: A reader is returned if

```
FileRd.Open(name)
```

is successeful; otherwise an exception is raised. If `name` is not an absolute pathname, then 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 pathname `p`, then a reader is returned if

```
FileRd.Open(Pathname.Join (p, name, NIL))
```

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 pathname 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`.

The procedure `Get` is logically equivalent to

```
VAR rd := Open(name, path);
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 it is a `TEXT`, is assumed to be the pathname of a directory, unless it starts with a dollar sign, in which case it is assumed to be environment variable whose value is the name of a directory; the value is retrieved using `Env.Get`. It is a checked runtime error of the pathname is not valid.

```
END Rsrc.
```

13.4 The Pts Interface

The `Pts` interface contains utilities to convert between points and pixels. VBTkit uses 72 points per inch and 25.4 millimeters per inch.

The locking level is arbitrary for all procedures in this interface.

```
INTERFACE Pts;

IMPORT Axis, VBT;

PROCEDURE ToScreenPixels (v: VBT.T; pts: REAL; ax: Axis.T):
  INTEGER;
```

Return the number of screen pixels that correspond to `pts` points on `v`'s screentype in the axis `ax`; or return `0` if `v`'s screentype is `NIL`. Equivalent to `ROUND (ToPixels (v, pts, ax))`

```
PROCEDURE ToPixels (v: VBT.T; pts: REAL; ax: Axis.T): REAL;
```

Return the number of pixels that correspond to `pts` points on `v`'s screentype in the axis `ax`; or return `0` if `v`'s screentype is `NIL`.

```
PROCEDURE FromPixels (v: VBT.T; pixels: REAL; ax: Axis.T): REAL;
```

Return the number of points that correspond to `pixels` pixels on `v`'s screentype in the axis `ax`; or return `0` if `v`'s screentype is `NIL`.

```
CONST
  PtsPerInch = 72.0;
  MMPerInch  = 25.4;

PROCEDURE FromMM (mm: REAL): REAL;
```

Convert from millimeters to points.

```
PROCEDURE ToMM (pts: REAL): REAL;
```

Convert from points to millimeters.

```
END Pts.
```

13.5 The VBTColors Interface

The `VBTColors` interface provides a way to associate a `VBT`'s background and foreground colors with the `VBT`. This information can be retrieved by some other `VBT` to compute a related color.

```
INTERFACE VBTColors;

IMPORT PaintOp, VBT;

PROCEDURE Put (v: VBT.T; colors: PaintOp.ColorScheme);
<* LL.sup < v *>
```

Store `colors` with `v`.

```
PROCEDURE Get (v: VBT.T): PaintOp.ColorScheme;
<* LL.sup < v *>
```

Return the colors stored by the most recent call to `Put`. If `Put` has never been called on `v`, return `PaintOp.bgFg`.

```
END VBTColors.
```

14 Color Utilities

This section describes the utilities that VBTkit provides for specifying colors. The `Color` interface defines two color models, RGB (Red, Green, Blue) and HSV (Hue, Saturation, Value), and contains procedures to convert between the color models. The `ColorName` interface provides routines to translate a color name, such as "VeryPaleCornflowerBlue," into an RGB triple. The locking level is arbitrary for all procedures in these interfaces.

14.1 The Color Interface

A `Color.T` describes a color as a mixture of the three color TV primaries (Red, Green and Blue), in a linear scale (proportional to luminous power), where 0.0 = black and 1.0 = maximum screen intensity.

The set of all colors with RGB coordinates in the range 0.0–1.0 is the *unit RGB cube*. The colors along the main diagonal of the unit cube (from (0,0,0) to (1,1,1)) contain equal amounts of all three primaries; they represent gray levels. RGB triples outside the unit cube cannot be displayed on typical color monitors, but are still legal as far as this interface is concerned, make perfect physical sense, and are useful in some color computations.

This interface also provides routines to convert colors between the HSV (Hue, Saturation, Value) and RGB color models.

```
INTERFACE Color;

TYPE
  T = RECORD r, g, b: REAL;  END;

CONST
  (* The vertices of the unit RGB cube: *)
  Black   = T{0.0, 0.0, 0.0};
  Red     = T{1.0, 0.0, 0.0};
  Green   = T{0.0, 1.0, 0.0};
  Blue    = T{0.0, 0.0, 1.0};
  Cyan    = T{0.0, 1.0, 1.0};
  Magenta = T{1.0, 0.0, 1.0};
  Yellow  = T{1.0, 1.0, 0.0};
  White   = T{1.0, 1.0, 1.0};
```

The following procedures are useful for converting a color into a shade of gray:

```
PROCEDURE Brightness (READONLY rgb: T): REAL;
```

Return the intensity of `rgb` *in a linear scale. The formula used is*

`0.239 * rgb.r + 0.686 * rgb.g + 0.075 * rgb.b`

clipped to the range 0.0–1.0.

An `HSV` is a color represented as a (Hue, Saturation, Value) triple. The HSV color model is somewhat more intuitive than the RGB color model. It's based on mimicking the way that an artist mixes paint: "He chooses a pure hue, or pigment and lightens it to a *tint* of that hue by adding white, or darkens it to a *shade* of that hue by adding black, or in general obtains a *tone* of that hue by adding some mixture of white and black."

So, varying hue corresponds to selecting a pure color along a color wheel where 0 is red, .167 is yellow, .333 is green, .5 is cyan, .667 is blue, and .833 is magenta, and 1.0 is red again. Decreasing the saturation (from 1 down to 0) corresponds to adding white. Decreasing the value (from 1 down to 0) corresponds to adding black.

This interface provides procedures to map between RGB and HSV color models. Note that white and black have indeterminate hue and saturation. Pure colors have saturation=1 and value=1, whereas grey levels have saturation=0, value=brightness, and indeterminate hue.

```
TYPE
  HSV = RECORD h, s, v: REAL END;
```

The following procedures convert between RGB and HSV color models:

```
PROCEDURE ToHSV (READONLY rgb: T): HSV;
```

Convert from RGB to HSV coordinates. By convention, gray colors (including white and black) get hue=0.0. In addition, black gets saturation=0.0.

```
PROCEDURE FromHSV (READONLY hsv: HSV): T;
```

Convert from HSV to RGB coordinates. If value=0 (black), saturation and hue are irrelevant. If saturation=0 (gray), hue is irrelevant.

```
END Color.
```

14.2 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.
```

A Text-editing Interfaces

A.1 Meta, Option, and Compose keys

The editing commands listed in the interfaces for the text-editing models are described in terms of "control," "meta," and "option" keys. The "control" modifier should be familiar to users. "Meta" and "option" are two names that refer to the same modifier in VBTkit applications; the Emacs, Mac, and Xterm models uses the term "meta," and the Ivy model uses "option."

There are two ways to type meta/option characters. The first is to hold down the key that generates the modifier known as `mod1`, and then to type the character. (The notion of a "modifier" is defined by the X-server. Other modifiers are `shift`, `lock`, `control`, and `mod2`–`mod5`. The utility named `xmodmap(1)` can be used to display and alter the relationship between physical keys and the information that the X-server provides to an application. On some keyboards, this key is labeled "Alt" or "Compose"; consult your system manager for more information.)

The second way to type a meta/option character is to type and release the Escape key, and then type the character. This is implemented only in the Emacs model.

In the Emacs, Ivy, and Xterm models, the meta/option key can also be used as a "prefix" key for composing extended-ASCII (8-bit) characters. If you type and release the meta/option key, then the next two characters you type will be "composed" into an extended character. While these two characters are being read, the cursor-shape will change to two counterclockwise arrows (the `XC_exchange` cursor). For example, if you type Meta, then "c", then "o," the result will be the ISO Latin-1 character for the copyright symbol, ©. If the font you are using does not have this character, you will see an ASCII representation for the character code, in octal, e.g., `\251`.

A.2 The TextPortClass interface

The `TextPortClass` interface reveals more of the representation of a textport, and it defines the object-type (`Model`) that is used to implement keybindings and selection-controls. Four subtypes of models are implemented: Ivy, Emacs, Mac, and Xterm. `TextPort.Model` is an enumeration type for the four names, but `TextPortClass.Model` is the type of the actual object attached to a textport, to which user-events (keys, mouse clicks, position reports) are sent.

In this interface, the variable `v` always refers to a textport, and the variable `m` always refers to a model.

Unless otherwise noted, the locking level of all procedures in this interface is `v.mu`.

```
INTERFACE TextPortClass;
```

```
IMPORT Font, KeyFilter, PaintOp, Rd, ScrollerVBTClass,
       TextPort, Thread, VBT, VTDef, VText;

REVEAL TextPort.T <: T;

TYPE VType = {Focus, Source, Target};
```

Constants for the three Trestle selections used here.

```
CONST
  Primary   = TextPort.SelectionType.Primary;
  Secondary = TextPort.SelectionType.Secondary;
  Focus     = VType.Focus;
  Source    = VType.Source;
  Target    = VType.Target;

TYPE
  Pixels = CARDINAL;
  T =
    TextPort.Public OBJECT
      mu: MUTEX;                 (* VBT.mu < mu *)
      <* LL = mu *>
      m             : Model              := NIL;
      readOnly      : BOOLEAN;
      vtext         : VText.T;

      font          : Font.T;
      fontHeight    : Pixels             := 0;
      charWidth     : Pixels             := 0;
      scrollbar     : Scrollbar          := NIL;
      typeinStart   : CARDINAL;

      thisCmdKind   : CommandKind;
      lastCmdKind   : CommandKind;
      wishCol       : CARDINAL;

      cur           : UndoRec;

      owns := ARRAY VType OF BOOLEAN {FALSE, ..};

      <* LL.sup = VBT.mu.SELF *>
      lastNonEmptyWidth: Pixels := 0;

    METHODS
      <* LL = SELF.mu *>
      getText       (begin, end: CARDINAL): TEXT;
      index         (): CARDINAL;
      isReplaceMode (): BOOLEAN;
      length        (): CARDINAL;
      normalize     (to := -1);

      replace (begin, end: CARDINAL; newText: TEXT):
             TextPort.Extent;
    unsafeReplace (begin, end: CARDINAL; newText: TEXT):
                TextPort.Extent;
    insert          (t: TEXT);
    unsafeInsert    (t: TEXT);
    getKFocus       (time: VBT.TimeStamp): BOOLEAN;
    newlineAndIndent ();
    findSource (time      : VBT.TimeStamp;
                loc                       := Loc.Next;
                ignoreCase                := TRUE      );
    notFound ();
    (* All of these call SELF.error. *)
    vbterror   (msg: TEXT; ec: VBT.ErrorCode);
    vterror    (msg: TEXT; ec: VTDef.ErrorCode);
    rdfailure  (msg: TEXT; ec: REFANY);
    rdeoferror (msg: TEXT);
    (* We release SELF.mu around the following callbacks. *)
    ULreturnAction (READONLY cd: VBT.KeyRec);
    ULtabAction    (READONLY cd: VBT.KeyRec);
    ULfocus        (gaining: BOOLEAN; time: VBT.TimeStamp);
    ULmodified     ();
    ULerror        (msg: TEXT);
  END;
```

`v.font` is the current font. `v.fontHeight` is the height of a (maximal) character. `v.charwidth` is the width of a (maximal) character. `v.scrollbar` contains the scrollbar that is updated when the visible region of text changes, and vice versa.

`v.typeinStart` is meaningful only for typescripts, where it indicates the point that divides the "history" part of the transcript, which is read-only, from the current command line, which is not. See the `TypescriptVBT` interface. For non-typescripts, this field is always zero.

`v.thisCmdKind` and `v.lastCmdKind` allow the interpretation of a command to depend on the previous command. Currently, the only commands that depend on context are the "vertical" commands that call `UpOneLine` and `DownOneLine`. The column to which they move is stored in `v.wishCol`.

`v.cur` holds the information needed to reverse or reinstate the effects of editing operations that change the text.

`v.owns[vtype]` is `TRUE` when `v` owns the `VBT.Selection` corresponding to `vtype`: keyboard focus, Source selection, or Target selection.

`v.lastNonEmptyWidth` is used by the `shape` and `reshape` methods.

`v.replace` tests `v.readOnly`; if that is `TRUE`, then it returns the constant `TextPort.NotFound`. Otherwise it calls `v.unsafeReplace`, which is the only routine that actually alters the underlying text. (The "unsafe" methods are those that do not test `v.readOnly`.)

`v.insert` calls `v.replace`; i.e., it is safe.

`v.notFound` is called when a search fails; see `FindAndSelect`, below. The default method is a no-op.

```
TYPE
  CommandKind = {VertCommand, OtherCommand};
  Scrollbar = ScrollerVBTClass.T OBJECT
                textport: T
              METHODS
                update ()        <* LL = SELF.textport.mu *>
              END;
```

A.2.1 Models

A `TextPortClass.Model` is the object that interprets keyboard and mouse events. The model can be replaced via `v.setModel`.

Keybindings

Trestle calls `v.key(cd)`, which calls `m.keyfilter.apply(v,cd)`, as described on page 52. A keyfilter is essentially a linked list of objects, each of which implements some low-level character translation such as "quoted insert" or "compose character." The last link calls `v.filter(cd)`, which calls `m.controlChord` or `m.optionChord` for "command-keys", or `m.arrowKey` for cursor-keys.

Text-selections

As explained on page 53, the model interprets keyboard and mouse events to establish the local selections, Primary and Secondary, which are subsequences of the text, usually highlighted in some way. The model also deals with the global selections, Source and Target, which may be owned ("acquired") by any VBT or by an external program such as an Xterm shell. The owner of a global selection controls its contents; `read` and `write` calls are forwarded to the owner.

A particular model may establish an "alias" relationship between a local selection and a global selection, which means that if the textport owns the global selection, then its contents are identical with (mapped to) the local selection. For example, in an Xterm shell, and therefore in the Xterm model, Primary is an alias for Source, which means that when you click and drag to highlight a region, that defines not only the local Primary selection but the global Source selection as well. Any program that asks to read the Source selection will be given a copy of the highlighted text.

In Ivy, Primary is an alias for Target, and Secondary is an alias for Source. (Ivy users therefore have a hard time understanding the distinction between local and global selections, since they are wired together.)

A Primary selection in a non-readonly textport may be in "replace mode" (or "pending-delete mode"). In this mode, insertions replace the entire selection; Backspace deletes the entire selection.

Selection-related editing operations

The standard editing operations such as Cut, Copy, and Paste, are defined not merely in terms of the underlying text, but also in terms of the effects they have on the local and global selections. Indeed, they are not functions at all; Copy does not return a copy of anything.

Copy If the Primary selection is not empty, then acquire Source, and unless Primary is an alias for Source, make a copy of the Primary selection as the contents of Source. (If Primary is an alias for Source, no copy is needed.)

Paste If the Primary selection is not empty and is in replace-mode, then replace the Primary selection with the contents of Source. Otherwise, insert the contents of Source at the type-in point.

Clear Delete the contents of the Primary selection.

Cut This is defined as **Copy** followed by **Clear**.

Select All Extend the Primary selection to include the entire text.

```
TYPE
  Model <: PublicModel;
  PublicModel =
    OBJECT
      v: T;
      selection := ARRAY TextPort.SelectionType OF
                     SelectionRecord {NIL, NIL};
      dragging := FALSE;
      dragType := TextPort.SelectionType.Primary;
      approachingFromLeft: BOOLEAN;
      keyfilter          : KeyFilter.T
    METHODS
      <* LL = SELF.v.mu *>
      init  (cs: PaintOp.ColorScheme; keyfilter: KeyFilter.T):
             Model;
      close ();
      seek  (position: CARDINAL);

      (* Keybindings *)
      controlChord (ch: CHAR; READONLY cd: VBT.KeyRec);
      optionChord  (ch: CHAR; READONLY cd: VBT.KeyRec);
      arrowKey     (READONLY cd: VBT.KeyRec);

      (* Mouse and Selection-controls *)
      mouse    (READONLY cd: VBT.MouseRec);
      position (READONLY cd: VBT.PositionRec);
      misc     (READONLY cd: VBT.MiscRec);
```

```
          read     (READONLY s   : VBT.Selection;
                             time: VBT.TimeStamp): TEXT
                 RAISES {VBT.Error};
          write    (READONLY s   : VBT.Selection;
                             time: VBT.TimeStamp;
                             t   : TEXT           )
                 RAISES {VBT.Error};

          cut   (time: VBT.TimeStamp);
          copy  (time: VBT.TimeStamp);
          paste (time: VBT.TimeStamp);
          clear ();
          select (time : VBT.TimeStamp;
                  begin: CARDINAL       := 0;
                  end  : CARDINAL       := LAST (CARDINAL);
                  sel                   := Primary;
                  replaceMode           := FALSE;
                  caretEnd              := VText.WhichEnd.Right);

          getSelection    (sel := Primary): TextPort.Extent;
          getSelectedText (sel := Primary): TEXT;
          putSelectedText (t: TEXT; sel := Primary);
          takeSelection (READONLY sel : VBT.Selection;
                                  type: TextPort.SelectionType;
                                  time: VBT.TimeStamp          ):
                        BOOLEAN;
          highlight (rec: SelectionRecord; READONLY r: IRange);
          extend    (rec: SelectionRecord; left, right: CARDINAL)
        END;
```

`m.init(...)` initializes a Model `m`. The default method stores `keyfilter` and returns `m`.

`m.close()` releases the `VBT` selections (Source, Target, and KBFocus) and deletes highlighting intervals.

`m.seek(position)` sets the type-in point.

The type `TextPort.T` overrides the `VBT` `mouse`, `position`, `misc`, `read`, and `write` methods with procedures that lock `v.mu` and call `m.mouse`, `m.position`, etc. Note that the signatures are not identical to their Trestle counterparts. `v.position` checks `m.dragging` and `cd.cp.gone` before calling `m.position`.

Clients must override the `read` method with a procedure that returns a text if `m` owns the selection `s`; otherwise it should call the default method, which calls `VBT.Read(s, time)`. `time` is valid when the caller is a user-event procedure such as `Paste`; it will be 0 when called from `v.read`, but in that case, `m` owns the selection, so `time` is not needed.

Similarly, clients must override the `write` method. `write` is called by `v.write`, which ensures that `v.readOnly` is `FALSE` before calling `m.write`.

If there is a non-empty Primary selection, then `m.copy(time)` arranges for that text to become the Source selection. Otherwise, it is a no-op; in particular, if the Primary selection is empty, `copy` must not acquire the Source selection. There is no default method for `copy`; the client must override this method.

The default for `m.cut(time)` is `m.copy(time); m.clear()`.

The default for `m.paste(time)` is `m.insert(m.read(VBT.Source, time))`.

`m.clear()` deletes the Primary selection. Its default method is

```
    m.putSelectedText ("", TextPort.SelectionType.Primary)
```

`m.insert(t)` implements `TextPort.Insert`. The default method replaces the Primary selection, if there is one, with `t`; otherwise, it inserts `t` at the type-in point. Clients may wish to override this in order to alter the highlighting.

`m.extend(rec,...)` extends the highlighting for the given selection.

A.2.2 Selections

```
TYPE
  SelectionRecord = OBJECT
                      type := TextPort.SelectionType.Primary;
                      interval   : VText.Interval;
                      cursor     : CARDINAL;
                      mode       : VText.SelectionMode;
                      anchor     : TextPort.Extent;
                      alias      : VBT.Selection;
                      replaceMode                   := FALSE
                    END;
```

Each local selection is represented by a `SelectionRecord`. `type` indicates whether this is a Primary or Secondary selection. `interval` describes the range of text and the highlighting. `mode` indicates whether this selection includes a character (point), word, line, paragraph, or the entire text. `anchor` is the range that stays fixed when we extend a selection. `replaceMode` indicates whether the selection was created with a replace-mode gesture or with `TextPort.Select(..., replaceMode := TRUE)`.

```
PROCEDURE ChangeIntervalOptions (v: T; rec: SelectionRecord)
  RAISES {VTDef.Error};
```

Change the highlighting according to the conventions specified in the `TextPort` *interface (see page 54).*

```
TYPE IRange = RECORD left, middle, right: CARDINAL END;

PROCEDURE GetRange (         v    : T;
                    READONLY cp   : VBT.CursorPosition;
```

```
                                    mode: VText.SelectionMode ):
  IRange;
<* LL = v.mu *>
```

Return an `IRange` indicating the boundaries of the character, word, paragraph, etc., that contains the position `cp`. The `middle` field of the result will be equal to either the `left` field or the `right` field, depending on which end the cursor was nearer.

A.2.3 Cursor-motion

```
PROCEDURE ToPrevChar (v: T);
PROCEDURE ToNextChar (v: T);
```

Move the cursor (type-in point) left or right one char.

```
PROCEDURE ToStartOfLine (v: T);
PROCEDURE ToEndOfLine   (v: T);
```

Move the cursor to start or end of line.

```
PROCEDURE UpOneLine   (v: T);
PROCEDURE DownOneLine (v: T);
```

Move the cursor up or down one line.

```
PROCEDURE ToOtherEnd (v: T);
```

Move the cursor to other end of the Primary selection.

```
PROCEDURE FindNextWord (v: T): TextPort.Extent;
PROCEDURE FindPrevWord (v: T): TextPort.Extent;
```

Locate the "next" or "previous" word.

In `FindNextWord`, we scan right from the current position until we reach an alphanumeric character. Then we continue scanning right until we reach the first non-alphanumeric character; that position defines the right end of the extent. Then we scan left until we find a non-alphanumeric character. That position, plus 1, defines the left end of the extent.

If the initial position is in the middle of a word, then the extent actually covers the *current* word, but on successive calls, it covers each following word in turn.

`FindPrevWord` works the same as `ToNextWord`, except that all the scanning directions are reversed.

"Alphanumeric characters" include the ISO Latin-1 characters, such as accented letters.

A.2.4 Deletion commands

All these procedures return an `Extent` indicating the range of characters that were deleted, or `TextPort.NotFound` if no characters were deleted.

```
PROCEDURE DeletePrevChar (v: T): TextPort.Extent;
PROCEDURE DeleteNextChar (v: T): TextPort.Extent;
```

```
PROCEDURE DeleteToStartOfWord (v: T): TextPort.Extent;
PROCEDURE DeleteToEndOfWord   (v: T): TextPort.Extent;
```

Delete from the current position to the beginning of the previous word (as defined in `ToPrevWord`) or the end of the "next" word (as defined in `ToNextWord`).

```
PROCEDURE DeleteToStartOfLine (v: T): TextPort.Extent;
```

Delete from the cursor to the beginning of the current line, or delete the preceding newline if the cursor is already at the beginning of the line.

```
PROCEDURE DeleteToEndOfLine (v: T): TextPort.Extent;
```

Delete to the end of line. If the cursor is at the end, delete the newline.

```
PROCEDURE DeleteCurrentWord (v: T): TextPort.Extent;
```

Delete the word containing the cursor.

```
PROCEDURE DeleteCurrentLine (v: T): TextPort.Extent;
```

Delete line containing the cursor.

A.2.5 Other modification commands

```
PROCEDURE SwapChars(v: T);
```

Swap the two characters to the left of the cursor.

```
PROCEDURE InsertNewline(v: T);
```

Insert a newline without moving the cursor.

A.2.6 Searching

```
TYPE Loc = {First, Next, Prev};
```

```
PROCEDURE Find (v          : T;
                pattern    : TEXT;
                loc                 := Loc.Next;
                ignoreCase          := TRUE      ):
```

```
  TextPort.Extent;
```

Search for `pattern` *in the text of* `v`*. The search proceeds either forward from the beginning of the text (*`Loc.First`*), forward from* `v.index()` *(*`Loc.Next`*, the default), or backward from* `v.index()` *(*`Loc.Prev`*). If* `ignoreCase` *is* `TRUE`*, the case of letters is not significant in the search.*

```
PROCEDURE FindAndSelect (v          : T;
                         pattern    : TEXT;
                         time: VBT.TimeStamp;
                         loc                := Loc.Next;
                         ignoreCase         := TRUE       );
```

Call `Find(v, pattern, loc, ignoreCase)`*. If the search was successful, then select the found text in replace-mode. Otherwise, call* `v.notFound()`*.*

A.2.7 Scrolling the display

```
PROCEDURE ScrollOneLineUp (v: T)
  RAISES {VTDef.Error, Rd.EndOfFile, Rd.Failure,
          Thread.Alerted};
PROCEDURE ScrollOneLineDown (v: T)
  RAISES {VTDef.Error, Rd.EndOfFile, Rd.Failure,
          Thread.Alerted};
PROCEDURE ScrollOneScreenUp (v: T)
  RAISES {VTDef.Error, Rd.EndOfFile, Rd.Failure,
          Thread.Alerted};
PROCEDURE ScrollOneScreenDown (v: T)
  RAISES {VTDef.Error, Rd.EndOfFile, Rd.Failure,
          Thread.Alerted};
```

Move the displayed text up or down by either a line or screen. This doesn't move the selections or the cursor, so the `TextPort` may not be normalized when done. A "screen" contains `MAX(1, n-2)` lines, where `n` is the number of displayed lines.

A.2.8 Managing the "Undo" stack

The "Undo" stack records all the editing changes made to the `TextPort`. These changes can be undone; once undone, they can be redone. There is no built-in limit to the number of changes that are recorded. A sequence of insertions of graphic characters (i.e., plain typing) counts as one "edit."

```
TYPE UndoRec <: ROOT;

PROCEDURE AddToUndo (v: T; begin, end: CARDINAL; newText: TEXT);
<* LL = v.mu *>
```

This is called by `v.unsafeReplace(begin, end, newText)` *to record a change to the underlying text.*

```
PROCEDURE Undo (v: T); <* LL = v.mu *>
```

Reverse the effect of the last editing command.

```
PROCEDURE Redo (v: T); <* LL = v.mu *>
```

Reinstate the effect of the last editing command.

```
PROCEDURE ResetUndo (v: T); <* LL < v.mu *>
```

Clear the "Undo" stack. (Nothing in the implementation calls this procedure.)

```
PROCEDURE UndoCount (v: T): CARDINAL; <* LL < v.mu *>
```

Return the number of changes that can be undone.

```
PROCEDURE RedoCount (v: T): CARDINAL; <* LL < v.mu *>
```

Return the number of undone changes that can be redone.

A.2.9 Compose-character filtering

```
TYPE Composer <: KeyFilter.ComposeChar;
```

This type overrides the `feedback` *method to change the cursor-shape to* `XC_exchange` *during character-composition, and the standard "text pointer" otherwise.*

A.2.10 Miscellany

```
PROCEDURE TextReverse (t: TEXT): TEXT;
PROCEDURE TextLowerCase (t: TEXT): TEXT;

CONST
  VBTErrorCodeTexts = ARRAY VBT.ErrorCode OF
                        TEXT {
                        "event not current", "timeout",
                        "uninstalled", "unreadable",
                        "unwritable", "unowned selection",
                        "wrong type"};

END TextPortClass.
```

A.3 The EmacsModel Interface

```
INTERFACE EmacsModel;

IMPORT KeyFilter, TextPortClass;

TYPE
```

```
    T <: TextPortClass.Model;
    EscapeMetaFilter <: KeyFilter.T;

  END EmacsModel.
```

In the Emacs model, there is only a Primary selection. It is not an alias for either Source or Target.

The model supports a single *region*, which is delimited by the *mark* and the *point*. Control-space and control-@ set the mark; the point is the same as the current cursor position, which is changed by mouse-gestures, cursor-keys, or control-keys. When the region is established by cursor-keys or control-keys, it is not highlighted. If the region is highlighted, then any gesture that extends it will extend the highlighting as well.

A single left-click sets the point and ensures that the current selection is not in replace-mode. If you then drag the mouse, the location of the downclick becomes the mark, and the point is set to the current position of the mouse. When the region is defined by dragging, it is highlighted. A double left-click sets both the mark and the point.

The Cut and Copy commands make a copy of the text in the region (i.e., the Primary selection); it becomes the Source selection. Middle-click and meta-w call Copy.

Right-click extends and highlights the current selection.

The control- and meta-keys in the Emacs model are not case-sensitive; control-shift-a, for example, has the same effect as control-a. The Emacs model supports "Escape + character" as an alternate way to type "meta-character," and ISO Latin-1 character composition. See Section A.1 for an explanation of "meta" keys and composition.)

control-space	set the mark
control-a	move to the beginning of the line
control-b	move to the previous character
meta-b	move to the previous word
control-d	delete the next character
meta-d	delete the next word
control-e	move to the end of the line
control-f	move to the next character
meta-f	move to the next word
control-h	delete the previous character, and move left
meta-h	delete to the start of the current word
control-i	invoke the `tabAction` callback
control-j	insert a newline
control-k	delete to the end of the line, and make that the source selection
control-m	invoke the `returnAction` callback
control-n	move down one line

control-o	insert a newline without moving the cursor
control-p	move up one line
control-q	insert the next character ("quoted insert")
control-r	search backward for the current source selection
control-s	search forward for the current source selection
control-t	swap the current and previous characters
control-v	scroll up one screen
meta-v	scroll down one screen
control-w	**Cut**
meta-w	**Copy**
control-y	**Paste**
control-z	scroll up one line
meta-z	scroll down one line
control-_	**Undo**
meta-_	**Redo**
meta-¡	move to the beginning of the buffer
meta-¿	move to the end of the buffer
meta-leftArrow	move to the previous word (like meta-b)
meta-rightArrow	move to the next word (like meta-f)

A.4 The IvyModel Interface

```
INTERFACE IvyModel;

IMPORT TextPortClass;

TYPE T <: TextPortClass.Model;

END IvyModel.
```

`TextPort` was originally designed after an editor called Ivy [6] that was developed at SRC. Ivy was written in Modula-2 and included a wealth of features; the Ivy model, documented here, implements only a small subset of them.

The Ivy model supports both local text-selections, Primary and Secondary. Primary is an alias for Target, and Secondary is an alias for Source.

There are two ways of acquiring the Source selection. The usual way is to make a Secondary selection (since Secondary is an alias for Source) by shift- or control-clicking to select a point, word, line, paragraph, or buffer. The second way is to use the Copy command (option-C) or the Cut command (option-X). These commands make a copy of the Primary selection; the copy becomes the Source selection, but it is not displayed.

The following list shows the Ivy keybindings. The Ivy model also supports ISO Latin-1 character composition. See Section A.1 for an explanation of "option" keys and composition.

Return	invoke the `returnAction` method
shift-Return	call `Newline`
option-Return	insert a newline after the cursor
Backspace	delete primary selection or the previous character
option-Backspace	swap the two previous characters
control-A	delete previous character
control-B	delete whole line
control-C	delete to start of line
option-C	**Copy**
control-D	delete to the start of the current word
control-E	**Move**: replace target with source, and clear source
control-F	delete to the end of the current word
control-G	delete whole word
control-H	swap the selection *boundaries*
control-I	move to the next word
control-J	move to previous character
control-K	move to next character
control-L	move to first non-blank and select line
control-M	find previous occurrence
option-M	find previous occurrence of primary
control-N	find next occurrence of primary
option-N	find first occurrence of primary
control-O	move up 1 row in the current column
control-P	move down 1 row in the current column
control-Q	**Clear** (delete the Primary selection)
control-R	**Swap**: exchange the selected *text*
control-S	delete the next character
control-U	move to the previous word
control-V	delete to end of line
option-V	**Paste**
control-W	**Paste**
option-X	**Cut**
control-Y	move to opposite end of selection
control-Z	**Undo**
control-shift-Z	**Redo**
control-,	find next occurrence
control-;	move to end of line and select line
control-Space	normalize

A.4.1 The Ivy selection model

The following table shows the mouse-gestures that establish the Primary selection; if the Shift or Control key is held down, these same gestures establish the Secondary selection.

click	Left	to select a point between characters
double-click	Left	to select a single line
triple-click	Left	to select the entire buffer
drag	Left	to change the selected point
click	Middle	to select a single word
double-click	Middle	to select a single paragraph
triple-click	Middle	to select the entire buffer
drag	Middle	to change the selected word or paragraph
click	Right	to extend the current selection
double-click	Right	to reduce the selection-unit
drag	Right	to extend the current selection

A selection is a sequence of "units"; a unit is a point, a word, a line, a paragraph, or the entire buffer. Double-clicking the right mouse-button reduces the unit of the current selection from buffer to paragraph, from paragraph to line, from line to word, and from word to point.

A single left-click selects the point (zero-length interval) between two characters. If you move the mouse and then right-click, the selection is extended to include all the characters between that point and the new position of the mouse. If you do *not* move the mouse, then a right-click extends the selection to include the character nearest that point.

A "word" is a maximal non-empty character sequence containing (1) only letters and digits, or (2) one or more space and tab characters, or (3) a single character that is not a letter, a digit, a space, or a tab.

A "line" is a non-empty character sequence containing at most one newline, whose first character either is the first character of the buffer or immediately follows a newline, and whose final character is either a newline or the last character in the buffer.

A "paragraph" is a sequence of lines—either a maximal sequence of non-blank lines or a maximal sequence of blank lines. (A blank line contains only spaces, tabs, and at most one newline.)

A.4.2 Replace-mode selection

When a Primary selection in a non-readonly buffer is extended, the selection becomes what is called a replace-mode selection, and its highlighting changes from a red underline to a pale red background. If you type after making a replace-mode selection, the first character you type will replace the selection. If you use the Copy or Move commands, the Secondary selection will replace the Primary selection.

A.5 The MacModel Interface

```
INTERFACE MacModel;
```

```
    IMPORT TextPortClass;

    TYPE T <: TextPortClass.Model;

    END MacModel.
```

The Mac model supports only a single selection, Primary. Is it not an alias for either Source or Target. A Primary selection in a non-readonly textport is always in replace-mode.

The conventions for the Mac model are taken from Apple's *Human Interface Guidelines* [1, pages 106-114].

The first unmodified downclick establishes the *anchor point.* If the user then drags the mouse, the upclick establishes the *active end*; the range between the anchor point and the active end is the Primary selection, and it is highlighted. If the user releases the mouse without dragging, that establishes the *type-in point*, and there is no selection or highlighting.

Shift-downclick extends (or reduces) the primary selection and establishes the new active end.

Double-clicking selects a word; dragging after a double-click extends the selection in word-size increments.

The Mac model implements the following Apple guidelines:

> When a Shift-arrow key combination is pressed, the active end of the selection moves and the range over which it moves becomes selected. ... Option-Shift-Left Arrow selects the whole word that contains the character to the left of the insertion point (just like double-clicking on a word).
>
> In a text application, pressing Shift and either Left Arrow or Right Arrow selects a single character. Assuming that the Left Arrow key was used, the anchor point of the selection is on the right side of the selection, the active end on the left. Each subsequent Shift-Left Arrow adds another character to the left side of the selection. A Shift-Right Arrow at this point shrinks the selection.
>
> Pressing Option-Shift and either Left Arrow or Right Arrow ... selects the entire word containing the character to the left of the insertion point. Assuming Left Arrow was pressed, the anchor point is at the right end of the word, the active end at the left. Each subsequent Option-Shift-Left Arrow adds another word to the left end of the selection...
>
> When a block of text is selected, either with a pointing device or with cursor keys, pressing either Left Arrow or Right Arrow deselects the range. If Left Arrow is pressed, the insertion point goes to the beginning of what had been the selection. If Right Arrow is pressed, the insertion point goes to the end of what had been the selection.

> [From page 83] When the user chooses Cut, ... the place where the selection used to be becomes the new selection. ... In text, the new selection is an insertion point [and the highlighting is removed].
>
> Paste ... inserts the contents of the Clipboard [Source] into the document, replacing the current selection [i.e., Primary selections are always replace-mode]. If there is no current selection, it's inserted at the insertion point.... After a Paste, the new selection is ... an insertion point immediately after the pasted text. [In either case, there is no highlighting.]

In documentation from Apple, Mac keybindings are typically described in terms of "command" and "option" modifiers. DEC keyboards and the X server do not use those terms, but a correspondence can be established. The Mac model uses the value of environment variable `MacCommandModifier` to name the X-modifier that the user would like to behave as if it were the "command" key. The choices are:

`lock`, `control`, `mod1`, `mod2`, `mod3`, `mod4`, and `mod5`

(Case is not significant in these names.) The default is `control`. Consult the manpage for `xmodmap(1)` for more information on these modifiers.

Similarly, the Mac model uses the environment variable `MacOptionModifier` to name the X-modifier that the user would like to behave as if it were the "option" key. The choices are the same as in the list above. The default is `mod1`.

The following commands are implemented in the Mac model:

command-c	**Copy**
command-v	**Paste**
command-x	**Cut**
command-z	**Undo**
command-shift-z	**Redo**

The Mac model supports the Apple standards for typing extended characters, insofar as the resulting characters are defined for ISO Latin-1. For example, option-g produces the copyright symbol, ©, but option-shift-7, which produces a double dagger, ‡, on the Macintosh, produces no key in the Mac model, since the double-dagger is not in ISO Latin-1. The Mac model supports all the two-character sequences, such as option-e followed by "a" to produce "a" with an acute accent, á. The complete table appears on page ??.

A.6 The XtermModel Interface

```
INTERFACE XtermModel;
```

```
IMPORT TextPortClass;

TYPE T <: TextPortClass.Model;

END XtermModel.
```

The Xterm model, patterned after **`xterm(1)`**, supports a single selection, Primary, which is an alias for Source. The Primary selection is never in replace-mode. The Xterm model is not influenced by commands in the user's `.Xdefaults` file.

A single-left-click establishes the keyboard focus and insertion point, but it does not change (acquire) the selection. A double-left-click selects the current word; a triple-left-click selects the current line. More clicks rotate among these three options.

Single-left-click and drag selects a range of characters. Double-left-click and drag selects a range of words, and triple-left-click and drag selects a range of lines.

Middle-click pastes the current source selection at the insertion point, which need not be at the end of the text (as it would be for a "typescript").

Right-click extends the current selection, re-highlighting it if needed.

The shift key has no effect on the mouse; it is ignored, so that shift-left-click, for example, has the same effect as left-click. The control and meta ("option") keys, however, are not ignored; they cause the mouse-clicks to be no-ops, and they have different keybindings. Control-left-click, for example, has no effect.

The only keybindings that are supported are these:

control-u	delete everything from the current position to the beginning of the line
control-z	**Undo**
control-shift-z	**Redo**
meta-x	**Cut**
meta-c	**Copy**
meta-v	**Paste**

Note that Copy does very little; since Primary is an alias for Source, nothing is actually copied.

A.7 The KeyFilter Interface

```
INTERFACE KeyFilter;

IMPORT VBT;

TYPE
  T = OBJECT
        next: T
```

```
      METHODS
        apply (v: VBT.T; cd: VBT.KeyRec)
      END;
  Composer <: T OBJECT
              METHODS
                feedback (v: VBT.T; composing: BOOLEAN)
              END;
  ComposeChar <: Composer;
  Diacritical <: Composer;

PROCEDURE IsModifier (c: VBT.KeySym): BOOLEAN;
```

Test whether `c` is a "modifier" key, such as Shift, Control, or Meta. Such keys are usually ignored by `Composer`s. Equivalent to:

```
    KeyboardKey.Shift_L <= c AND c <= KeyboardKey.Hyper_R
```

```
END KeyFilter.
```

A `KeyFilter`'s `apply` method takes a `VBT.T` and a `VBT.KeyRec` and may pass them on to the `KeyFilter` in its `next` field, possibly having altered the `KeyRec` in the process. For example, a "transparent" filter would simply call `SELF.next.apply(v,cd)`. An "upper-case filter" (transducer) would convert lower-case characters to upper-case before passing them on.

A `KeyFilter` may also maintain an internal state, and it is not required to call `SELF.next.apply` on every call. Various "character composition" schemes, for example, involve typing one character (e.g., a key labeled "Compose Character") followed by two others, which are all "composed" to produce a single character. That is, they effectively implement a "look-ahead" reader.

A `Composer` is a subtype that provides a `feedback` method; the intention is that the `apply` method calls `SELF.feedback(v, TRUE)` when it sees a key that begins a multi-character sequence, and `SELF.feedback(v, FALSE)` when it sees a key that ends a sequence. The default `feedback` method is a no-op, but a client may wish to override that in order to provide a visual cue to the user that key-composition is in effect (e.g., changing the cursor). Otherwise, the user might not understand why typed character are not being "echoed."

Two types of `Composers` are provided, `ComposeChar` and `Diacritical`. `ComposeChar` produces the ISO Latin-1 (8-bit, extended ASCII) characters, using the VT220 style of composition: when the filter sees a `Keyrec` whose `whatChanged` field is `KeyboardKey.MultiKey`, it calls `SELF.feedback(v,TRUE)`; after two more `KeyRec`s have been passed to it, it looks for those two keys in an internal table. If it finds a character, then it passes it to `SELF.next.apply`. For example, on many keyboards, there is a key labeled `Compose` or `Compose Character`, which produces the `MultiKey` code. When you type that key, followed by "c" and "o", the filter passes the character for the copyright symbol, ©, to the `next` filter. If there is no entry in the table, the filter does not pass anything to the `next` filter. In any case, it always returns to its initial state.

For some users, the “Compose” key is also the “meta” or “option” key. Holding this key down and typing “a”, for example, produces a `KeyRec` with the `mod1` modifier (which Trestle represents as `VBT.Modifier.Option`). When the `ComposeChar` filter sees a `KeyRec` with this modifier, it assumes that the user is *not* composing an 8-bit character, so it calls `SELF.feedback(v,FALSE)` and `SELF.next.apply(v,cd)`, and it returns to its initial state.

A `Diacritical` filter also produces 8-bit characters. The filter looks at 2-character sequences; comma followed by “c”, for example, produces an “c” with a cedilla, ç. If the sequence is not defined, such as comma followed by space, then filter passes both characters to the `next` filter; i.e., when it receives the second `KeyRec`, it makes *two* calls to `SELF.next.apply`. (This is why the `KeyFilter` uses a `next` field instead of merely returning a `KeyRec`.)

Here is an example showing the intended use of this interface. Assume that `TextEditingVBT` is a subtype of `VBT` used for typing text, such as `TypeinVBT.T` or `TextPort.T`. A client would override the `key` method in order to filter the keys delivered to the supertype's `key` method.

```
TYPE
  MyTextEditor =
    TextEditingVBT.T OBJECT
        comp: KeyFilter.ComposeChar
      OVERRIDES
        key := Key
      END;
  Parent = Keyfilter.T OBJECT
             OVERRIDES
                apply := ApplyParent
             END;

PROCEDURE Key (v: MyTextEditor; READONLY cd; VBT.KeyRec) =
  BEGIN
    IF cd.wentDown AND cd.whatChanged # VBT.NoKey THEN
      v.comp.apply (v, cd)
    END
  END Key;

PROCEDURE ApplyParent (self : MyParent;
                       v    : VBT.T;
                       cd   : VBT.KeyRec) =
  BEGIN
    TextEditingVBT.T.key (v, cd)
  END ApplyMyParent;

VAR editor := NEW (MyTextEditor,
                    comp := NEW (KeyFilter.ComposeChar,
                                 next := NEW (Parent)));
```

A `ComposeChar` object is not case-sensitive where there is no ambiguity. For example, `c` and `o` can be combined to produce the copyright symbol, ©; so can `C` and `O`, `c` and `O`, or `C` and `o`. By contrast, `e` and ‘ can be combined to produce a lower-case `e` with a grave accent, è, but `E` and ‘ produce an upper-case `E` with a grave accent, È.

Unless both of the characters are alphanumeric, they can be combined in either order. So ‘ and `e` have the same effect as `e` and ‘, but `o` and `c` do *not* combine to form the copyright symbol.

A.7.1 Composed Characters

The following table shows the two-character combinations (in the left column) that are "composed" by a `KeyFilter.ComposeChar` object to produce an "extended ASCII", ISO-Latin-1 character. Where possible, that character is shown in the middle column, and a description of the character appears in the right column.

two spaces		non-breaking space
`!!`	¡	inverted exclamation point
`??`	¿	inverted question mark
`C/` or `C$`		cent sign
`L-` or `L$`	£	pound sign
`XO` or `G$`		currency sign
`Y-` or `Y$`		Yen sign
`||`		broken bar
`SO`	§	section sign
`""`	¨	diaeresis
`CO`	©	copyright sign
`A_` or `SA`	ª	feminine ordinal indicator
`O_` or `SO`	º	masculine ordinal indicator
`<<`	$\ll$	left angle-quotation mark
`>>`	$\gg$	right angle-quotation mark
`-,` or `NO`	¬	not sign
`--`	$-$	hyphen
`RO`		registered trademark sign
`-^` or `__`	¯	macron
`0^` or `DE`	^{0}	ring above, degree sign
`+-`	±	plus-minus sign
`++`	#	number-sign
`1^` or `S1`	^{1}	superscript 1
`2^` or `S2`	^{2}	superscript 2
`3^` or `S3`	^{3}	superscript 3
`''`	´	acute accent
`'<space>`	’	apostrophe
`/U` or `*M`	μ	Greek small letter mu
`P|` or `PG`	¶	paragraph
`.^` or `..`	·	middle dot
`,,`	¸	cedilla
`14`	$\frac{1}{4}$	one quarter
`12`	$\frac{1}{2}$	one half
`34`	$\frac{3}{4}$	three quarters

`` A` ``	À	A with grave accent
`A'`	Á	A with acute accent
`A^`	Â	A with circumflex
`A~`	Ã	A with tilde
`A"`	Ä	A with diaeresis
`` a` ``	à	a with grave accent
`a'`	á	a with acute accent
`a^`	â	a with circumflex
`a~`	ã	a with tilde
`a"`	ä	a with diaeresis
`A*` or `oA`	Å	A with ring above
`a*` or `oa`	å	a with ring above
`AE`	Æ	capital diphthong AE
`ae`	æ	small diphthong ae
`C,`	Ç	C with cedilla
`c,`	ç	c with cedilla
`` E` ``	È	E with grave accent
`E'`	É	E with acute accent
`E^`	Ê	E with circumflex
`E"`	Ë	E with diaeresis
`` e` ``	è	e with grave accent
`e'`	é	e with acute accent
`e^`	ê	e with circumflex
`e"`	ë	e with diaeresis
`` I` ``	Ì	I with grave accent
`I'`	Í	I with acute accent
`I^`	Î	I with circumflex
`I"`	Ï	I with diaeresis
`` i` ``	ì	i with grave accent
`i'`	í	i with acute accent
`i^`	î	i with circumflex
`i"`	ï	i with diaeresis
`N~`	Ñ	N with tilde
`n~`	ñ	n with tilde

O`	Ò	O with grave accent
O'	Ó	O with acute accent
O^	Ô	O with circumflex
O~	Õ	O with tilde
O"	Ö	O with diaeresis
O/	Ø	O with oblique stroke
o/	ø	o with oblique stroke
o`	ò	o with grave accent
o'	ó	o with acute accent
o^	ô	o with circumflex
o~	õ	o with tilde
o"	ö	o with diaeresis
U`	Ù	U with grave accent
U'	Ú	U with acute accent
U^	Û	U with circumflex
U"	Ü	U with diaeresis
u`	ù	u with grave accent
u'	ú	u with acute accent
u^	û	u with circumflex
u"	ü	u with diaeresis
Y'	Ý	Y with acute accent
y'	ý	y with acute accent
y"	ý	y with diaeresis
ss	ß	small German letter sharp s
xx or mu [sic]	×	multiplication sign
-:	÷	division sign
D-		capital Icelandic letter ETH
d-		small Icelandic letter ETH
TH or |P		capital Icelandic letter thorn
th or |p		small Icelandic letter thorn

A.7.2 Extended characters in the Mac model

The Mac model does not use the character-compositions described in the previous section. Instead, it composes characters according to the following tables.

key	plain	shift	option	option	key	plain	shift	option	option
cap				shift	cap				shift
A	a	A	å	Å	1	1	!	¡	/
B	b	B			2	2	@		*currency*
C	c	C	ç	Ç	3	3	#	£	<
D	d	D		Î	4	4	$	*cents*	>
E	e	E		´	5	5	%		
F	f	F		Ï	6	6	ˆ	§	
G	g	G	©		7	7	&	¶	
H	h	H		Ó	8	8	*		
I	i	I		ˆ	9	9	(	ª	·
J	j	J		Ô	0	0	)	º	,
K	k	K	0		‘	`	˜		`
L	l	L	¬	Ò	-	-	_		
M	m	M	μ	Â	=	=	+		±
N	n	N		˜	[	[	{		
O	o	O	ø	Ø	]	]	}		
P	p	P			\	\	—	<<	>>
Q	q	Q			;	;	:		Ú
R	r	R			’	’	”	æ	Æ
S	s	S	ß	Í	,	,	<		
T	t	T			.	.	>		
U	u	U			/	/	?	÷	¿
V	v	V							
W	w	W							
X	x	X							
Y	y	Y		Á					
Z	z	Z							

option-	plain	shift	option-	plain	shift
E			I		
A	á	Á	A	â	Â
E	é	É	E	ê	ê
I	í	Í	I	î	Î
O	ó	Ó	O	ô	Ô
U	ú	Ú	U	û	Û
SPACE	´	´	SPACE	ˆ	ˆ

option-N	plain	shift	option-U	plain	shift	option-‘	plain	shift
A	ã	Ã	A	ä	Ä	A	à	À
N	ñ	Ñ	E	ë	Ë	E	è	è
O	õ	Õ	I	ï	Ï	I	ì	Ì
SPACE	˜	˜	O	ö	Ö	O	ò	Ò
			U	ü	Ü	U	ù	Ù
			Y	ÿ	Ÿ	SPACE	`	`
			SPACE	¨	¨			

A.7.3 Diacritical marks

A `KeyFilter.Diacritical` object uses a simpler scheme for producing a subset of the extended ASCII characters. The following table shows the 2-character sequence that you type, and the resulting character. If you type a character in the first column, such as comma, and then a character that is *not* in the second column, such as space, the key-filter will produce the two characters you typed.

‘	a	à	‘	A	À
‘	e	è	‘	E	È
‘	i	ì	‘	I	Ì
‘	o	ò	‘	O	Ò
‘	u	ù	‘	U	Ù
‘	‘	`			
’	a	á	’	A	Á
’	e	é	’	E	É
’	i	í	’	I	Í
’	o	ó	’	O	Ó
’	u	ú	’	U	Ú
’	y	ý	’	Y	Ý
’	’	´			
^	a	â	^	A	Â
^	e	ê	^	E	Ê
^	i	î	^	I	Î
^	o	ô	^	O	Ô
^	u	û	^	U	Û
^	^	ˆ			
"	a	ä	"	A	Ä
"	e	ë	"	E	Ë
"	i	ï	"	I	Ï
"	o	ö	"	O	Ö
"	u	ü	"	U	Ü
"	y	ÿ			
"	"	¨			
~	a	ã	~	A	Ã
~	n	ñ	~	N	Ñ
~	o	õ	~	O	Õ
~	~	˜			
,	c	ç	,	C	Ç
,	,	¸			

References

[1] Apple Computer Co. *Human Interface Guidelines: The Apple Desktop Interface.* Apple Computer Co., 1987.

[2] Shiz Kobara. *Visual Design with OSF/Motif.* Addison Wesley, 1991.

[3] Mark S. Manasse and Greg Nelson. Trestle reference manual. Technical Report 68, DEC Systems Research Center, December, 1991.

[4] Mark S. Manasse and Greg Nelson. Trestle tutorial. Technical Report 69, DEC Systems Research Center, May, 1992.

[5] Robert W. Scheifler, James Gettys, and Ron Newman. *X Window System, 2nd edition.* Digital Press, 1990.

[6] Mary-Claire van Leunen, Mark R. Brown, and Patrick Chan. Ivy reference manual. Technical report, DEC Systems Research Center, forthcoming.

Index