1

The VIBRANT Portable Interface Development Library

Jonathan A. Kans, Information Engineering Branch, NCBI, NLM, NIH

VIBRANT is a high-level, multi-platform user interface development library written in C and distributed as part of the NCBI Software Development ToolKit. Vibrant acts as an intermediary between an application and the underlying windowing system toolkit.The philosophy behind Vibrant is that everything in the published user interface guidelines for the various windowing systems (i.e., the generic behavior of windows, menus, buttons, etc., to which all programs should conform) is taken care of automatically, without needing any attention from the programmer. Vibrant frees the programmer from maintaining resource files, explicitly specifying the positions of interface objects, and writing an event loop.A program written with Vibrant calls functions that create windows, menus, and the various graphical control objects that reside in them. The first parameter is typically the parent object for that control. The programmer may also write a "callback" function for a given object. The name of the callback is typically passed as the last parameter. When the user manipulates the object (and thus changes its "value"), the callback function is automatically executed.In addition to such standard interface objects as windows, menus, lists, and text boxes, Vibrant provides a universal drawing object called a "slate". A slate can contain one or more "panels". Each panel can have instance-specific callbacks for click, drag, hold, release, and draw functions. By using Vibrant's portable drawing functions in panels, an application can present arbitrarily complex drawings in a completely portable manner. A text document display and a graphical viewer have been implemented using panels.Although Vibrant is intended to allow biologists to write programs without becoming experts in any of the native windowing system toolboxes, it is sufficiently powerful for more complex applications. *Entrez*, an NCBI information retrieval program, is written with Vibrant, and is source code-identical on the Macintosh, PC/Windows, UNIX and VMS machines.

Simplifying assumptions central to VibrantThe purpose of Vibrant is to allow scientists to concentrate on designing and implementing algorithms, and then, with a minimum of effort, quickly incorporate them into user-friendly programs that can be run on computers found in molecular biology laboratories.The hierarchical relationship seen by users (e.g., radio buttons always "belong" to a particular group, which in turn belongs to a particular window) is explicitly specified when the programmer creates interface objects. The "parent" object is always created prior to its "children".The generic behavior and appearance of objects, published in the user interface guidelines, is handled automatically by Vibrant. This includes selecting windows, highlighting controls, tabbing between text boxes, and clearing radio buttons.The specific behavior of a program is centered around "callback" functions. These can be assigned to any interface object, and are executed when the user manipulates the object to change its "value".Positions of objects are calculated by Vibrant, using layout specifications in the program, eliminating the need to create and maintain resource files. The program can easily alter the assigned positions of objects, or align multiple objects with one another, when building complex dialogs.Certain dialog box objects represent program parameters (e.g., strings, numbers, Boolean toggles, choices from a set). Converting between the object and the actual parameter value is performed with simple functions (e.g., `SetValue`, `GetStatus`, `SetTitle`).Choices from a set are referenced by integer value. These can be represented by radio buttons in a group, or by items in a menu choice group, popup list, or scrolling list box. Changing the implementation requires only changing the names of the function calls and callback parameter types.Because of the object-oriented internal design of Vibrant, functions used to manipulate object appearance (e.g., `Show`, `Hide`, `Enable`, `Disable`, `SetTitle`, `GetValue`) can be applied to any interface object.

Solutions to issues of portabilityThe underlying toolkits upon which Vibrant sits have completely different designs. A number of design decisions, or "tricks", were needed in Vibrant to allow portable code to be written. Programs written with Vibrant are source code-identical on the Macintosh, PC/Windows, and UNIX and VMS machines with X11/Motif.Vibrant objects are variants of "`Handle`" types. A handle may really be a pointer, an indirect pointer, or an int, depending upon the platform. Vibrant declares each object to be a `HNDL` (near pointer) to a unique (dummy) C structure. This means that the compiler can detect and warn of an attempt to place a button in a menu, but will allow a button to be placed in a group or directly in a window.typedef struct button {
  VoidPtr dummy;
} HNDL ButtoN;typedef struct menu {
  VoidPtr dummy;
} HNDL MenU;typedef struct window {
  VoidPtr dummy;
} HNDL GrouP, HNDL WindoW;By assigning one device context to each parent window, having child windows use the parent's device context, and keeping track of the "current" window and device context in static variables, the Windows device context becomes the functional equivalent of the Macintosh port. This allowed the creation of the slate, a universal drawing environment, on otherwise very different graphics systems. Vibrant's drawing functions do not need the current context as an explicit parameter, even though the underlying Windows toolbox calls do need to be told which device context to use.On the Macintosh, selection of a font name, size, and style are independent operations. Under Windows, a different font descriptor must be created for each combination of name, size, and style. The portable solution in Vibrant is to create a `FonT` object that specifies one unique combination of name, size, and style, and to change fonts on each machine by calling `SelectFont` with the desired font object.

The structure of a Vibrant programVibrant is written with the NCBI CoreLib, which provides low-level portable types and functions. The application `Main` function (capital M) returns an `Int2`. (Vibrant contains the C `main` function.) The only header that needs to be included is `<vibrant.h>`, which itself includes `<ncbi.h>`. The application does not see any of the low-level toolbox-specific symbols or functions.The convention for naming Vibrant objects is that the initial and last letter are capitalized. The "standard" types include `BaR`, `ButtoN`, `ChoicE`, `DoC`, `GrouP`, `IcoN`, `IteM`, `ListT`, `MenU`, `PaneL`, `PopuP`, `PrompT`, `RepeaT`, `SlatE`, `SwitcH`, `TexT`, `VieweR`, and `WindoW`. Of these, the icon, repeat, and switch objects (implemented in the `vibextra.c` file) provide good examples of how to use slates to make novel interface objects without needing direct access to the underlying toolkits.A Vibrant program creates windows, populates them with interface objects, and then calls `ProcessEvents` to turn control over to the user. The remainder of the program resides in the callbacks, which are executed in response to user-driven events.It is generally most convenient to have a separate routine for creating the menus, in this case named `SetupMenus`. The code to create other interface objects is not shown in this example.#include <vibrant.h>Int2 Main (void)

```
{
  WindoW  w;

#ifdef WIN_MAC
  SetupMenus (NULL);
#endif
  w = FixedWindow (-50, -33, -10, -10, "Test", NULL);
#ifndef WIN_MAC
  SetupMenus (w);
#endif

  /* Populate the window with interface objects */

  Show (w);
  ProcessEvents ();
  return 0;
}
```

Menu placement and font specification are usually the only cases in which platform-dependent code must be written. On the Macintosh, the program typically creates the apple menu, and places the About box item and desk accessory group in it. The `NULL` parent specifies that the menu bar is on the desktop and not in a particular window.static void SetupMenus (WindoW w)

```
{
  MenU  m;

#ifdef WIN_MAC
  m = AppleMenu (NULL);
  CommandItem (m, "About...", AboutProc);
  SeparatorItem (m);
  DeskAccGroup (m);
#endif
  m = PulldownMenu (w, "File");
#ifndef WIN_MAC
  CommandItem (m, "About...", AboutProc);
  SeparatorItem (m);
#endif
  CommandItem (m, "Quit/Q", QuitProc);
}
```

At least one callback should call the `QuitProgram` function. This will cause the `ProcessEvents` loop to be exited, finishing any statements in the remainder of the `Main` function. All callbacks are passed the item handle of the manipulated object as a parameter. It is therefore possible for several objects to be served by a single callback in some situations.static void QuitProc (IteM i)

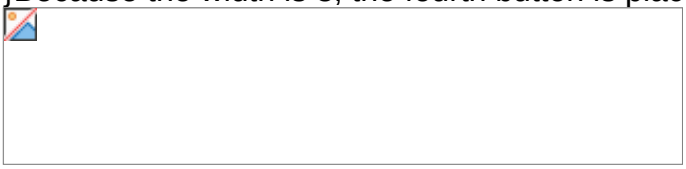
```
{
  QuitProgram ();
}
```

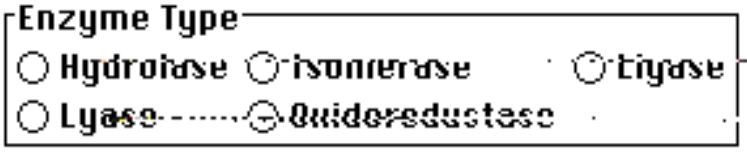
Other important Vibrant or CoreLib functions, several of which will be described later, will be used in a reasonable number of applications. These include `GetInputFileName` and `GetOutputFileName` for file dialog boxes, `GetAppParam` and `SetAppParam` for configuration file parameters, `Metronome` for setting an application timer, and `Message` and `Beep` for posting message windows or alerting the user. The CoreLib provides portable type definitions (e.g., `Int2`, `CharPtr`) and platform symbols (e.g., `WIN_MSWIN`, `WIN_MOTIF`), and portable functions for string manipulation (e.g., `StringLen`, `TO_UPPER`), memory allocation (e.g., `MemNew`, `MemFree`), file access (e.g., `FileOpen`, `FileRead`), byte stores, error handling, and date and time display.

Vibrant windows and variablesThe remaining sections will give specific examples of interface object creation and manipulation. The code examples provided were actually used to produce the figures.In addition to interface objects and the `FonT` type, Vibrant provides portable `PoinT`, `RecT`, and `RegioN` types. The rectangular coordinates of any object in a window (or the window itself) are obtained by calling `ObjectRect`. Given a Vibrant object, its immediate parent object is returned by `Parent`, and the window that is its ultimate ancestor is returned by `ParentWindow`.Global variablesA number of useful global variables are available in Vibrant. `screenRect` holds the rectangular coordinates of the computer screen. `systemFont` and `programFont` are predefined font variables. `stdLineHeight` and `stdCharWidth` are constants that refer to dimensions on `systemFont`. `dblClick` and `shftKey` may be set during list and panel clicks. `updateRgn` and `updateRect` are set in response to panel draw (expose) events.WindowsThe window is the only Vibrant interface object that does not have a parent object. It is at the top of the hierarchy. Window creation functions have parameters for the position and size of the window, a title, a close callback, and (for document windows) a resize callback.Vibrant provides several different kinds of windows. A document window can be resized by the user, while a fixed window cannot be.WindoW DocumentWindow (Int2 left, Int2 top, Int2 width,
Int2 height, CharPtr title,
WndActnProc close, WndActnProc resize);WindoW FixedWindow (Int2 left, Int2 top, Int2 width,
Int2 height, CharPtr title,
WndActnProc close);A modal window is used when certain information must be provided before program execution can proceed, and it blocks the use of "lower" windows until it is dismissed (by calling `Remove`).WindoW ModalWindow (Int2 left, Int2 top, Int2 width,
Int2 height, WndActnProc close);

A floating window stays above all others, but is not considered the active window, and so does not interfere with the normal operation of other windows. It is useful for a palette of icon buttons.WindoW FloatingWindow (Int2 left, Int2 top, Int2 width,
Int2 height, WndActnProc close);Vibrant makes no distinction between classes of windows in terms of what objects can go in them. (There is no concept of a dialog window separate from other kinds of windows.) Any object can go in any window.The window is also the only object that is not shown (made visible) automatically upon creation. If you don't want to show it right away, but want to create and populate another window, you should first call `RealizeWindow` on the current window.The simplest way of using a window is to ask it to size itself around its "child" objects by giving negative parameters for the width and height. When the window is first shown (or realized), the actual size is calculated. If the left and top parameters are negative, their absolute values are taken as percentages of remaining screen space (e.g., "`-50`" specifies the center).The window drawing contextA given window can be moved to the front by calling `Select`. This also makes it the "current" window, i.e., the window in which drawing will occur. The current window is returned by `CurrentWindow`, and can be changed (without bringing it to the front) via `UseWindow`.The functions `SavePort` and `RestorePort` incorporate these concepts, and are used internally in calls that change the appearance of objects. This allows a callback triggered by a button in one window to change the contents of a dialog text box in another window (by calling `SetTitle`) without the text unintentionally appearing in the wrong window.To set the context for slates, the `Select` function must also be called, since under Motif each slate is a separate DrawingArea (Xlib Window). Like the Windows device context, the current Xlib Window (needed as a parameter for some of the X11 drawing functions) is stored in a static variable. Again, the appearance-changing functions of standard Vibrant objects (i.e., the internal functions that implement `SetTitle` and `SetValue` for each object) already contain any necessary code to save and restore the context.

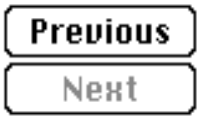
Positioning of objectsAutomatic positioning with groupsBy default, an object in a window is placed below the previous object. The group object allows the program to control the automatic positioning of objects. (The group also acts as the parent of a set of radio buttons, and enforces the principle that only one button in a given set can be selected.)GrouP NormalGroup (GrouP prnt, Int2 width, Int2 height,
CharPtr title, GrpActnProc actn);GrouP HiddenGroup (GrouP prnt, Int2 width, Int2 height,
GrpActnProc actn);The width and height parameters of a group determine the layout of child objects. If the width is positive and the height is 0, objects are laid out horizontally, and the position "breaks" to the next row after each set of n objects. If the width is 0 and the height is positive, they are laid out vertically, and the position "advances" to the next column after n objects.If either width or height are negative, objects are laid out as above, but their borders are not aligned with one another. If both width and heigth are 0, successive objects are placed at the same position. In this case you would typically hide all but one of the superimposed objects at any given time.The sizes and positions of objects are automatically adjusted as new objects are added. In the example below, a group is populated with radio buttons.{
```
  GrouP  g;

  g = NormalGroup (w, 3, 0, "Enzyme Type", ChangeType);
  RadioButton (g, "Hydrolase");
  RadioButton (g, "Isomerase");
  RadioButton (g, "Ligase");
  RadioButton (g, "Lyase");
  RadioButton (g, "Oxidoreductase");
  RadioButton (g, "Transferase");
```
}Because the width is 3, the fourth button is placed on the second line.



The title of the fifth button is longer than previous width of the second column. When it is added, previous items in the same column expand, items in succeeding columns move over, and the group enlarges. 

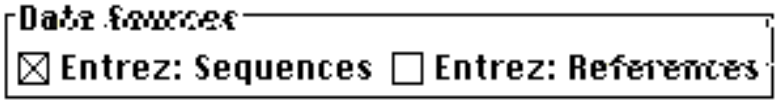
The borders between a group and its objects, and the spacing between internal objects can also be set. Measurements are in pixels.void SetGroupMargins (GrouP g, Int2 xMargin, Int2 yMargin);void SetGroupSpacing (GrouP g, Int2 xSpacing, Int2 ySpacing);Additional object alignmentIn most cases placing objects in groups is sufficient to produce nicely aligned dialog boxes. However, in more complex cases, it is necessary to adjust the positions after the fact. This is done with the `AlignObjects` function, which takes a variable number of arguments.void AlignObjects (Int2 align, ...);The first argument specifies the kind of alignment. `ALIGN_LEFT` will align the left margins of all object parameters to each other. Margins are moved to the maximum value found in the original objects. `ALIGN_RIGHT` does the same for right margins, while `ALIGN_JUSTIFY` separately aligns both left and right margins. `ALIGN_CENTER` will center objects horizontally. There are equivalent values for vertical alignment.The remaining parameters are Vibrant interface objects (of type `Handle`). All objects must be type cast to `(HANDLE)`, which prevents the arguments from being erroneously promoted to the wrong number of bytes under certain platforms. The argument list is terminated by `NULL`.Finally, `AlignObjects` should not be called until after the parent groups of its object parameters have been completely populated.Explicit control over object positions is available, if really needed.void GetPosition (Handle object, RectPtr rct);void SetPosition (Handle object, RectPtr rct);

The menagerie of Vibrant objectsPush buttons and check boxesThe push button is used to trigger a callback under user control. It has no persistent value. The callback is executed when the button is pressed. The `DefaultButton` is like a push button, but the callback is also executed when the user presses the Return key. The check box may take a callback, but its Boolean status may also be tested from other callbacks with `GetStatus`.ButtoN PushButton (GrouP prnt, CharPtr title, BtnActnProc actn);ButtoN DefaultButton (GrouP prnt, CharPtr title, BtnActnProc actn);ButtoN CheckBox (GrouP prnt, CharPtr title, BtnActnProc actn);The initial size of the push button is determined by the pixel width of the button title and a standard button height. The initial size and position may change if the button is in a group.




```
{
  ButtoN  b;
  GrouP   g;

  g = HiddenGroup (w, 0, 2, NULL);
  PushButton (g, "Previous", PrevProc);
  b = PushButton (g, "Next", NextProc);
  Disable (b);
}
```

The initial size of the check box, which represents a Boolean toggle, is also calculated by the pixel width of the title and a standard height.




```
{
  GrouP   g;

  g = NormalGroup (w, 2, 0, "Data Sources", NULL);
  esBtn = CheckBox (g, "Entrez: Sequences", NULL);
  SetStatus (esBtn, TRUE);
  erBtn = CheckBox (g, "Entrez: References", NULL);
}
```

Displaying static prompts (labels)The static prompt takes width and height parameters in pixels. If the width is 0, then the width of the text string (plus 2 pixels) is used. (This is calculated internally by calling `StringWidth` after having called `SelectFont` on the font parameter.) If the height is 0, then the actual value is calculated by calling `LineHeight`. The global variables `dialogTextHeight` and `popupMenuHeight` may be used to center prompts vertically next to dialog text or popup list objects.PrompT StaticPrompt (GrouP prnt, CharPtr title, Int2 pixwidth,

Int2 pixheight, FonT font, Char just);The prompt justification parameter is a character, either '`l`', '`c`', or '`r`', for left-, center-, or right-justified, respectively. (This convention also applies to the `DrawString` function that is used to draw character strings in slates.)Entering and editing text stringsAlthough the popularity of "point and click" interfaces has reduced the need to remember arcane acronyms in order to run computer programs, there are still cases where typing at the keyboard is the most efficient way of entering data. The Vibrant text object is used in these situations.The `DialogText` object can contain a single line of text, with no tabs or returns. If the user presses the tab key while in a dialog text object, Vibrant will attempt to find and select the next available dialog text.If multiple lines of text are needed, the `ScrollText` object may be used. The font used for scrolling texts can also be specified. The predefined font variable `programFont` is a non-proportional font that is useful for displaying program code or email messages in scrolling texts.A text object can be assigned a callback function that is triggered whenever the user changes the contents of the object (by inserting or deleting characters). The contents of the text object can be obtained with `GetTitle`, and can be changed under program control with `SetTitle`.TexT DialogText (GrouP prnt, CharPtr dfault, Int2 charWidth,

TxtActnProc actn);TexT ScrollText (GrouP prnt, Int2 width, Int2 height, FonT font, Boolean wrap, TxtActnProc actn);

The `HiddenText` has an additional callback that is triggered when the user presses the tab key. This object can be used to simulate spreadsheets, where having multiple dialog texts (and attempting to "scroll" them) would be inefficient. The hidden text object is usually superimposed over a slate (or the panel-based document display object).TexT HiddenText (GrouP prnt, CharPtr dfault, Int2 charWidth,

TxtActnProc actn, TxtActnProc tabProc);Cutting and pasting requires knowledge of the currently selected text object. The `CurrentText` function returns this information. `CutText`, `CopyText`, `PasteText`, and `ClearText`, which usually reside in menu command item callbacks, would typically be passed the current text.The example below demonstrates the use of `AlignObjects` to align the right margins of two text objects, even though they are not members of the same group. The left margins of another two dialog texts are aligned because the pixel widths of the "**Volume**" and "**Journal**" prompt have been explicitly set to be the longest of the two. The first dialog text is currently selected. At the next key stroke the entire selection will be deleted and replaced by the character that was typed.


```
{
  GrouP  g
  GrouP  h;
  TexT   jour;
  TexT   pgs;
  TexT   vol;
  Int2   wid;

  h = HiddenGroup (w, 0, 2, NULL);
  g = HiddenGroup (h, 2, 0, NULL);  SetGroupSpacing (g, 13, 2);
  wid = MAX (StringWidth ("Journal"), StringWidth ("Volume")) + 2;
  StaticPrompt (g, "Journal", wid, dialogTextHeight, NULL, 'l');
  jour = DialogText (g, "J Mol Biol", 5, NULL);
  g = HiddenGroup (h, 4, 0, NULL);  SetGroupSpacing (g, 13, 2);
  StaticPrompt (g, "Volume", wid, dialogTextHeight, NULL, 'l');
  vol = DialogText (g, "215", 3, NULL);
  StaticPrompt (g, "Pages", 0, dialogTextHeight, NULL, 'l');
  pgs = DialogText (g, "403-10", 6, NULL);
  AlignObjects (ALIGN_RIGHT, (HANDLE) jour, (HANDLE) pgs, NULL);
  Select (jour);
}
```

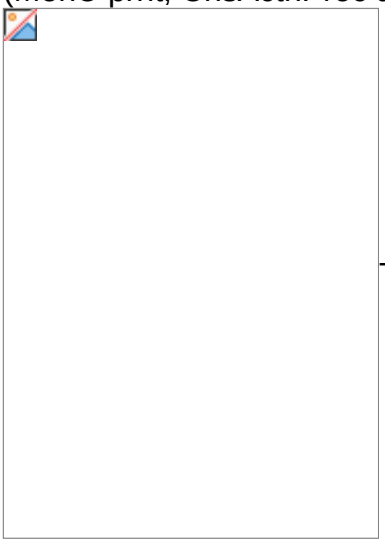
13

Menus and menu itemsA pulldown menu can reside in a window menu bar, or in the Macintosh desktop menu bar (passing `NULL` as the parent). `SubMenu` creates a menu item that controls a cascading sub menu, and takes a menu as its parent.MenU PulldownMenu (WindoW prnt, CharPtr title);MenU SubMenu (MenU prnt, CharPtr title);The `CommandItem` is the equivalent of a push button, and the `StatusItem` is the equivalent of a check box.IteM CommandItem (MenU prnt, CharPtr title, ItmActnProc actn);IteM StatusItem (MenU prnt, CharPtr title, ItmActnProc actn);The `SeparatorItem` places a horizontal bar in the menu. It has no function, but provides visual separation of unrelated sets of items.void SeparatorItem (MenU prnt);The menu choice group will be discussed in the next section. This example shows three `StatusItems` and two menu `ChoiceGroups` in sub menus.

{

```
  ChoicE  c;
  IteM    i;
  MenU    m;
  MenU    s

  m = PulldownMenu (w, "Preferences");
  persistItem = StatusItem (m, "Parents Persist", NULL);
  SetStatus (persistItem , TRUE);
  i = StatusItem (m, "Show Sequence", ShowSeqProc);
  SetStatus (i, TRUE);
  timerItem = StatusItem (m, "Use Timer", NULL);
  s = SubMenu (m, "Chars Per Line");
  c = ChoiceGroup (s, ChangeCharsProc);
  ...
}
```

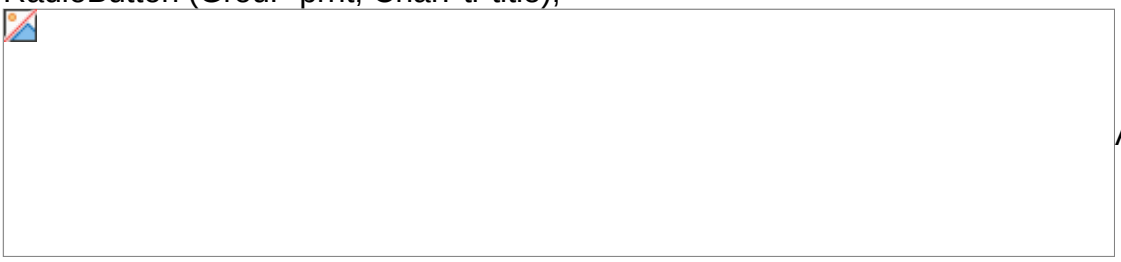
14
Choices from a setVibrant considers certain objects to represent choices from a set. The decision of which implementation to use depends upon such factors as how much "real estate" the object takes up on the window and whether the number of choices can be very large. In each case a pair of functions is used, one to create the set, the other to add choices to the set.Menu choice groupThe first example shows three menu `ChoiceGroups`, visually delimited by separator items. Items are entered with the `ChoiceItem` function.ChoicE ChoiceGroup (MenU prnt, ChsActnProc actn);IteM   ChoiceItem (ChoicE prnt, CharPtr title);



The default value of each independent choice group has been set to 1 with `SetValue`. Initially, no choice is checked, and the value is 0.{

```
  ChoicE  c;
  MenU    m;

  m = PulldownMenu (w, "Options");
  c = ChoiceGroup (m, ChangeArticle);
  ChoiceItem (c, "MEDLINE Report");
  ChoiceItem (c, "MEDLARS Format");
  ChoiceItem (c, "MEDLINE ASN.1");
  SetValue (c, 1);
  c = ChoiceGroup (m, ChangeReport);
  ...
}
```

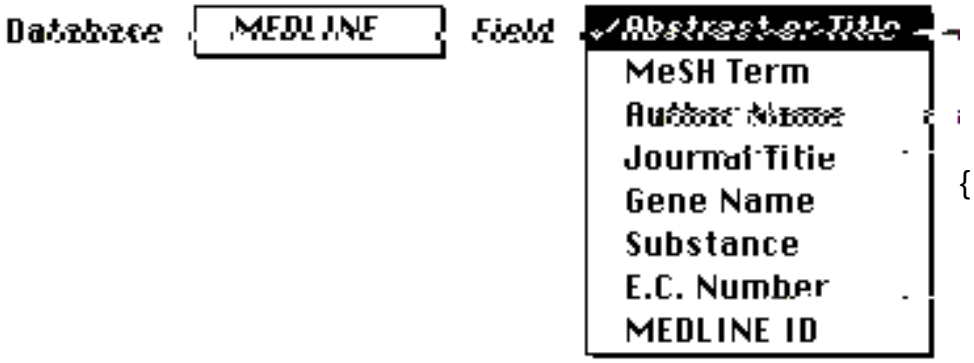
15
Group of radio buttonsThis next example shows a `NormalGroup` containing several `RadioButtons`. A normal group displays a title at the upper left hand corner of its bounding box. As radio buttons are added, the group adjusts the positions and widths of the previously-entered buttons so that all items line up nicely. The "`4, 0`" specification indicates that the buttons should be added horizontally, with a new row being used after every four items.ButtoN RadioButton (GrouP prnt, CharPtr title);



As is the case with the menu choice group, the initial value of the group of radio buttons has been set with `SetValue`. A value of 1 specifies that the first button is chosen, while 0 would indicate that no buttons are chosen.static CharPtr divisions [] = {
  "Bacterial", "EST", "Fungal", "Invertebrate",
  "Mammalian", "Patent", "Phage", "Plant",
  "Primate", "Rodent", "Structural RNA",
  "Synthetic DNA", "Unannotated", "Vector",
  "Vertebrate", "Viral", NULL
};{
  GrouP g;
  Int2   i;

  g = NormalGroup (w, 4, 0, "Division", ChangeDivProc);
  for (i = 0; divisions [i] != NULL; i++) {
    RadioButton (g, divisions [i]);
  }
  SetValue (g, 1);
}Groups can contain of any kind of object that a window can have (i.e., all objects except menus and their choice and item objects), and the group will automatically align its child objects.While groups can have titles, and menu choice groups can appear to have titles if they are in sub menus, popup lists and scrolling lists do not have titles. The static prompt is generally used to label these objects.

Popup listThe third example of choices from a set is implemented as a popup list. This has the advantages of taking up very little "real estate" and of displaying only the currently-selected value. Unlike items in menus, which can only be viewed by selecting the menu with the mouse, the popup list is visible in a window. The popup on the left shows the normal appearance, while the one on the right is being selected with the mouse.PopuP PopupList (GrouP prnt, Boolean macLike, PupActnProc actn);void PopupItem (PopuP prnt, CharPtr title);

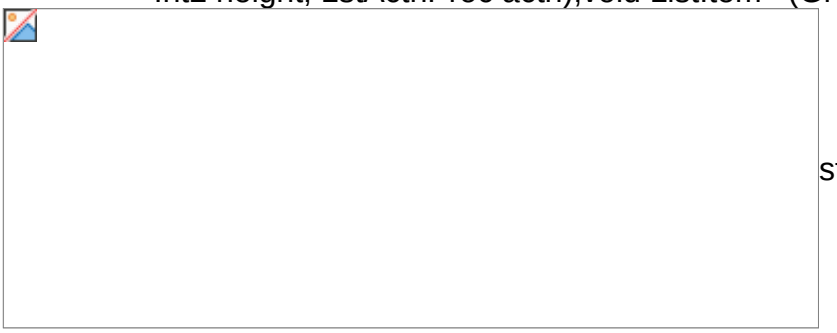



```
  GrouP  h;
  PopuP  p;

  h = HiddenGroup (w, 4, 0, NULL);
  SetGroupSpacing (h, 10, 2);
  StaticPrompt (h, "Database", 0, popupMenuHeight, systemFont, 'l');
  p = PopupList (h, TRUE, ChangeDbaseProc);
  PopupItem (p, "MEDLINE");
  PopupItem (p, "Protein");
  PopupItem (p, "Nucleotide");
  SetValue (p, 1);
  StaticPrompt (h, "Field", 0, popupMenuHeight, systemFont, 'l');
  p = PopupList (h, TRUE, ChangeFldProc);
  PopupItem (p, "Abstract or Title");
  PopupItem (p, "MeSH Term");
  ...
  PopupItem (p, "MEDLINE ID");
  SetValue (p, 1);
}
```

As with the menu choice group and the group of radio buttons, the initial values of the popup lists have been set with `SetValue`. `SetGroupSpacing` has been used to increase the number of pixels between objects in the hidden group to 10 (the default is 3 horizontally and 2 vertically).

17

Scrolling listThe final example uses a single choice scrolling list. This object is most appropriate when the number of items can be quite large, or when the longest text string may be very wide. The width and height of the scrolling list are specified using the global variables `stdCharWidth` and `stdLineHeight` (precomputed attributes of `sysemFont`) as coordinates.ListT SingleList (GrouP prnt, Int2 width,
Int2 height, LstActnProc actn);void ListItem (GrouP prnt, CharPtr title);



static Boolean ReadProc (CharPtr term, Int4 special, Int4 total)

```
{
  ListItem (lst, term);
  return TRUE;
}

{
  GrouP  h;

  h = HiddenGroup (w, 0, 2, NULL);
  StaticPrompt (h, "Organism", 0, 0, systemFont, 'c');
  lst = SingleList (h, 20, 6, ChangeLstProc);
  EntrezTermListByTerm (TYP_AA, FLD_ORGN, "Drosophila",
                        50, ReadProc, NULL);
  SetValue (lst, 4);
}
```

It should come as no surprise that `SetValue` has been used to preselect the fourth item of the scrolling list.Setting the value of a list will automatically cause it to scroll so that the current value is visible. The scroll offset can be changed under program control with `SetOffset (list, 0, line)`. Note that a scroll offset of 0 is when the first item appears in the first line of the list.

Creating novel interface objects with slatesThe slate is used to extend the repertoire of Vibrant objects by allowing the application to respond directly to update (expose) and mouse events.Displaying a pictorial iconThe `IconButton` is a control that allows display of an application-specified drawing in a window. Although implemented with a slate, it cannot have scroll bars or extra instance-specific data.IcoN IconButton (GrouP prnt, Int2 pixwidth, Int2 pixheight,

IcnActnProc draw, IcnChngProc inval,
IcnClckProc click, IcnClckProc drag,
IcnClckProc hold, IcnClckProc release);The icon object can retain an integer value, a Boolean status, and a text string. These can be obtained and changed with the usual functions (e.g., `SetValue`, `GetTitle`). The application may change some of these settings at certain times, usually in the callbacks that respond to mouse events. Changing any of the settings triggers an invalidation operation. This marks the icon as needing erasure and redrawing at the next update event.If the inval parameter is `NULL`, then the entire object rectangle is invalidated. The application can supply an inval function to be triggered instead, if it is desired that only certain parts of an icon should be invalidated, to prevent unwanted flicker when changing the settings. (The application might also prefer to call `ScrollRect` instead of `InvalRect`.)In the example shown below, the icon button can appear in two states, indicated by the direction of the arrow. Clicking on the icon and then releasing the mouse within the icon toggles between the two states.   The icon is created with the width and height in pixels, and with callbacks for drawing, invalidation, and mouse release. The invalidation callback is used to prevent flicker of the rectangular frame around the arrows. An icon must have a draw callback in order to be visible. IconButton (w, 32, 22, DrawIcon, InvalIcon,

NULL, NULL, NULL, ReleaseIcon);

The example draw callback function gets the rectangular coordinates of the icon, draws (frames) the rectangle, and copies a bitmap into the center of the icon. The callback determines which bitmap to draw by examining the Boolean status of the icon.static void DrawIcon (IcoN i)

```
{
  RecT  r;

  ObjectRect (i, &r);
  FrameRect (&r);
  InsetRect (&r, 12, 11);
  if (GetStatus (i)) {
    CopyBits (&r, upArrow);
  } else {
    CopyBits (&r, downArrow);
  }
}
```

The release callback changes the Boolean status of the icon if the mouse was located within the icon boundaries when the mouse button was released. Setting the status (or changing the integer value or text string) triggers the invalidation function, and then forces the resulting update event to be processed immediately.static void ReleaseIcon (IcoN i, PoinT pt)

```
{
  RecT  r;

  ObjectRect (i, &r);
  if (PtInRect (pt, &r)) {
    SetStatus (i, (Boolean) (! GetStatus (i)));
  }
}
```

The default invalidation routine retrieves the bounding rectangle, expands it by one pixel, and then invalidates the resulting rectangle. This behavior is overridden in this example by an invalidation callback, which shrinks the bounding rectangle by one pixel before invalidation to ensure that the visible rectangle frame does not flicker.static void InvalIcon (IcoN i, Nlm_Int2 newval, Nlm_Int2 oldval)

```
{
  RecT  r;

  ObjectRect (i, &r);
  InsetRect (&r, 1, 1);
  InvalRect (&r);
}
```

The `CopyBits` function is used to draw inside panels. The form of the bitmaps is identical among all platforms. A binary 1 indicates "foreground" color (typically black) and a binary 0 indicates "background" color (typically white). Vibrant provides functions to change these colors (e.g., `Red`, `SelectColor`). Bitmaps are displayed using the current drawing "mode", which is either `CopyMode`, `MergeMode`, `InvertMode` or `EraseMode`. The current pattern (e.g., `Solid`, `Medium`) and line style (e.g., `Solid`, `Dashed`) have no effect on bitmaps.static Uint1 upArrow [] = {
0x08, 0x1C, 0x3E, 0x7F, 0x1C,
0x1C, 0x1C, 0x1C, 0x1C, 0x1C
};static Uint1 downArrow [] = {
0x1C, 0x1C, 0x1C, 0x1C, 0x1C,
0x1C, 0x7F, 0x3E, 0x1C, 0x08
};Under Windows the bits are inverted before being sent to the low-level `BitBlt` function, while under Motif the bytes are flipped (least significant bit becomes most significant bit) before being sent to `XCopyPlane`. Vibrant will also compensate properly if the number of bytes per row is odd, so no extra "padding" bytes are required.The full spectrum of Vibrant portable drawing functions may be used in the draw callback.Arbitrary drawing with slates and panelsThe slate allows an application to present arbitrarily complex drawings in a completely portable manner. (The icon button, described above, is an example of a general-purpose graphical object implemented with a slate.) Messages or events that are treated by other objects at the generic level (i.e., drawing requests, mouse clicks, and key presses) are passed by slates to application callback functions in a platform-independent manner.A slate can have one or more panels, and each panel can have instance-specific callbacks for click, drag, hold, release, and draw functions. Vibrant provides a variety of drawing procedures that can be used inside panels regardless of the machine on which the program is running.A panel can be used to create a new object that solves a general problem (such as display of tabular text, or display of bitmap icons). Since the slate allows multiple panels, these building blocks can be combined (even

superimposed) to create a complex slate from simpler components. This can be much more efficient than having to modify (and thus understand) a copy of the code for each component in order to make a custom object.An even better way of addressing a general problem, and yet being amenable to customization, is to allow the application to specify an additional draw callback, sometimes known as a "hook". The panel draw callback would perform its general drawing functions, then call the specific function to do any custom drawing. This turns out to be even easier to use than having multiple, superimposed, independent panels in a slate.The simplest versions of a slate merges a slate and panel in one object instance. These include the `SimplePanel` and the more general `AutonomousPanel`. The autonomous panel can have scroll bars, extra instance data, a reset function to free that data, and the ability to override the Vibrant class functions in order to allow such functions as `SetTitle` and `GetValue` to apply to particular slate/panels (henceforth called panels).PaneL SimplePanel (GrouP prnt, Int2 pixwidth, Int2 pixheight,
PnlActnProc draw);PaneL AutonomousPanel (GrouP prnt, Int2 pixwidth, Int2 pixheight,
PnlActnProc draw, SltScrlProc vscrl,
SltScrlProc hscrl, Int2 extra,
PnlActnProc reset, GphPrcsPtr classPtr);`SetPanelClick` attaches mouse response callbacks to a panel.void SetPanelClick (PaneL p, PnlClckProc click, PnlClckProc drag,
PnlClckProc hold, PnlClckProc release);The click, drag, and release callbacks can be used to do such things as drawing a "marquee", which consists of a dotted rectangle containing the the location of the original mouse click and the current mouse position.Panel callbacks are passed the panel handle and the point as parameters. For this task, the callbacks must know the first point and the current point.static PoinT curpnt;
static PoinT fstpnt;

The click callback sets the first point and current point to this initial point. It then draws a (tiny) dotted rectangle around this point.static void ClickProc (PaneL p, PoinT pt)

```
{
  fstpnt = pt;
  curpnt = pt;
  CopyMode ();
  CommonFrameProc (p);
}
```

The drag callback sets `InvertMode` and draws over (thus erasing) the previously-drawn rectangle. It then updates the current point static variable and draws the new rectangle.static void DragProc (PaneL p, PoinT pt)

```
{
  InvertMode ();
  CommonFrameProc (p);
  curpnt = pt;
  CommonFrameProc (p);
}
```

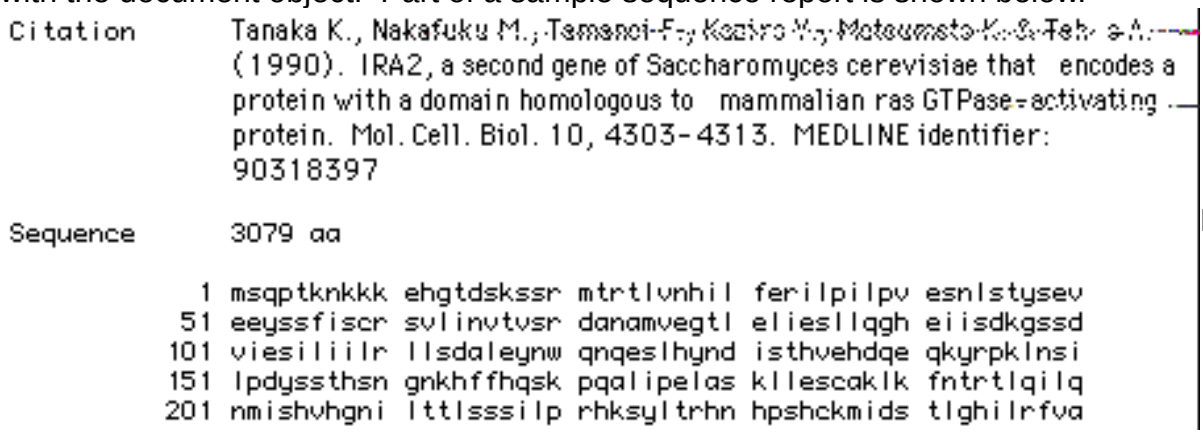
The release callback draws over and erases the last dotted rectangle.static void ReleaseProc (PaneL p, PoinT pt)

```
{
  InvertMode ();
  CommonFrameProc (p);
}
```

In this common function that actually does the drawing, the marquee rectangle is confined to the panel bounding rectangle by finding the intersection (overlap) of the two rectangles, calculated with the `SectRect` function.static void CommonFrameProc (PaneL p)

```
{
  RecT  dr;
  RecT  or;
  RecT  r;

  Dotted ();
  ObjectRect (p, &or);
  InsetRect (&or, 2, 2);
  LoadRect (&r, fstpnt.x, fstpnt.y, curpnt.x, curpnt.y);
  SectRect (&r, &or, &dr);
  FrameRect (&dr);
}
```

Document display panelThe document object is a (partial) general solution to the problem of displaying tabular text. While it is too complex to be described in full detail here, it is built with an autonomous panel, and it does allow "hook" callbacks for drawing and mouse events, so that it can be customized.The MEDLINE and sequence report viewers in *Entrez* are written with the document object. Part of a sample sequence report is shown below.



Documen

t types and functions are in the `<document.h>` header. A `DocumentPanel` can be placed in any window or group. The width and height parameters are in pixels, a 4 pixel border is added to each margin, and a vertical scroll bar is automatically created.DoC DocumentPanel (GrouP prnt, Int2 pixwidth, Int2 pixheight);The document object is most easily populated by repeated calls to `AppendText`. This takes the document, the text string, a paragraph format, a column format array, and a default font as parameters. Each column has width, font, justification, and word wrap parameters. Tabs embedded in the text control assignment of subtrings to particular columns. Existing paragraph items can be replaced, and new items can be inserted.void AppendText (DoC d, CharPtr text, ParPtr parFmt,

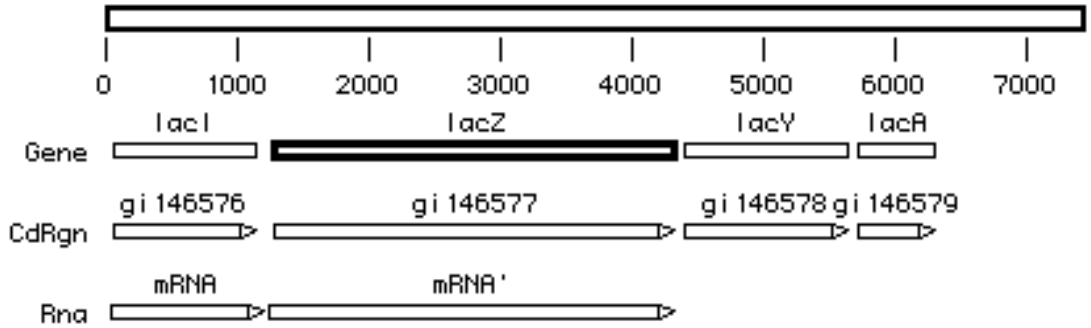
ColPtr colFmt, FonT font);void ReplaceText (DoC d, Int2 item, CharPtr text,

ParPtr parFmt, ColPtr colFmt, FonT font);void InsertText (DoC d, Int2 item, CharPtr text,

ParPtr parFmt, ColPtr colFmt, FonT font);

The above functions are special cases of `AppendItem`, `ReplaceItem`, and `InsertItem`, respectively. The `Text` functions take the paragraph text string as a parameter. The `Item` functions take a "print function" and a data pointer. The print function uses the data parameter and returns an allocated string containing the paragraph text.void AppendItem (DoC d, DocPrntProc proc, Pointer data,
Boolean docOwnsData, Int2 lines, ParPtr parFmt,
ColPtr colFmt, FonT font);Paragraph items can also be deleted dynamically.void DeleteItem (DoC d, Int2 item);The paragraph and column format shown below specify a blank space between paragraphs, word wrap on both columns, and a different font in the second column. Column pixel widths are frequently set dynamically.static ParData parFmt = {TRUE, FALSE, FALSE, FALSE, FALSE, 0, 0};static ColData colFmt [2] = {
{0, 0, 15, 0, NULL, 'l', TRUE, FALSE, FALSE, FALSE},
{0, 0, 63, 0, NULL, 'l', TRUE, FALSE, FALSE, TRUE}
};{
colFmt [0].pixWidth = 6 * stdCharWidth;
colFmt [1].pixWidth = 21 * stdCharWidth;
colFmt [1].font = ParseFont ("Geneva,12 | Helvetica,12");
AppendText (doc, "Citation\tTanaka K., Nakafuku M.,...\n",
&parFmt, colFmt, programFont);
}The mouse point can be mapped to a particular item, row, and column, the text in a given cell (or set of cells) can be obtained, and the program can force the document paragraph contents and its scroll bar to update after insertion, replacement, or deletion of paragraphs. The document can also be printed or saved to a file.void MapDocPoint (DoC d, PoinT pt, Int2Ptr item, Int2Ptr row,
Int2Ptr col, RectPtr rct);CharPtr GetDocText (DoC d, Int2 item, Int2 row, Int2 col);void UpdateDocument (DoC d, Int2 from, Int2 to);void AdjustDocScroll (DoC d);

Graphical viewer panelThe viewer object is implemented with an autonomous panel. It allows the creation of a picture, which can be composed of a hierarchy of "segments". The "primitive" items in segments can be lines or rectangles (which are measured in "world coordinates"), or annotation items such as text labels, tick marks, or symbols. An "attribute" item specifies changes to the color, line style, pixel shading, pen width, and drawing mode of subsequent primitives in a segment.The viewer was designed to map from world coordinates to "screen coordinates" by using integer arithmetic. Floating point calculations are not needed. Its performance is therefore suitable even on slower machines found in many molecular biology laboratories.A display of feature intervals on the *E. coli lac* operon, built with a viewer object using data read from the ASN.1 record on the *Entrez:* Sequences disc, is shown below. The *lacZ* gene has been selected with the mouse.



The scale was drawn using the "hook" callback, and the application can decide how fine a scale to display at any given magnification.Use of the viewer requires including the `<picture.h>` and `<viewer.h>` headers. A picture is created as a tree of segment objects. Picture segments are not Vibrant graphical objects, in that they cannot be placed into windows, groups, menus, etc., though their philosophy is very similar to the Vibrant group.SegmenT CreatePicture (void);SegmenT DeletePicture (SegmenT picture);A segment can be given an integer ID that can refer to the program object which it represents. It can also be told the maximum scale factor at which

it will be visible (0 means always visible). Given any segment, you can find its parent segment, and traverse up the hierarchy to the parent picture. The primitive items in segments can also contain integer IDs.SegmenT CreateSegment (SegmenT parent, Uint2 segID, Int4 maxScale);SegmenT ParentSegment (SegmenT segment);Rectangles and lines must specify their full dimensions in "world coordinates". These items will change size when the picture is scaled in the viewer. World coordinates are whatever is appropriate to the particular domain. For molecular sequences, they may be base pairs or residues.PrimitivE AddRectangle (SegmenT parent, Int4 left, Int4 top,
Int4 right, Int4 bottom, Int2 arrow,
Boolean fill, Uint2 primID);PrimitivE AddLine (SegmenT parent, Int4 pnt1X, Int4 pnt1Y,
Int4 pnt2X, Int4 pnt2Y, Boolean arrow,
Uint2 primID);Symbols, bitmaps, markers (tick marks), and text labels are annotation objects attached to a particular point in world coordinates. These items always remain the same size. The alignment parameter (e.g., `UPPER_LEFT`, `MIDDLE_CENTER`) determines which part of the annotation maps to the point. The orientation parameter (e.g., `HORIZ_LEFT`, `VERT_AB`OVE) determines a more limited orientation for marker lines.PrimitivE AddSymbol (SegmenT parent, Int4 pntX, Int4 pnty,
Int2 symbol, Boolean fill, Int2 align,
Uint2 primID)PrimitivE AddMarker (SegmenT parent, Int4 pntX, Int4 pntY,
Int2 length, Int2 orient, Uint2 primID);The symbols include a rectangle, a diamond, an oval, and four orientations of triangle, in empty and filled varieties. 
The attribute item changes the color, line style, shading, pen width and drawing mode for subsequent items within a given segment. The attributes do not affect subsequent items in any parent segments.PrimitivE AddAttribute (SegmenT parent, Uint1 flags, Uint1Ptr color,
Int1 linestyle, Int1 shading, Int1 penwidth,
Int1 mode);

A set of rectangles with various shadings are shown. The final one, with `EMPTY_SHADING`, is frames for visibility.  A viewer is created on a window or in a group. A picture is then attached to the viewer. The viewer is responsible for scaling (zooming) and panning (scrolling) the picture on the screen.ViewеR CreateViewer (GrouP prnt, Int2 width, Int2 height,

Boolean vscroll, Boolean hscroll);void ResetViewer (ViewеR viewer);ViewеR DeleteViewer (ViewеR viewer);void AttachPicture (ViewеR viewer, SegmenT picture, Int4 pntX,

Int4 pntY, Int2 align, Int4 scaleX,

Int4 scaleY, VwrDrawProc draw));During a click, drag, or release callback, `FindSegment` will return the ID of the deepest segment containing an object in which the mouse resides, the primitive ID if an item is under the mouse, and the index of the primitive.SegmenT FindSegment (ViewеR viewer, PoinT pt, Uint2Ptr segID,

Uint2Ptr primID, Uint2Ptr primCt);Given the primitive index, you can obtain a pointer to the primitive, and use that pointer to highlight the primitive.PrimitivE GetPrimitive (SegmenT segment, Uint2 primCt);void HighlightPrimitive (ViewеR viewer, SegmenT segment,

PrimitivE primitive, Int1 highlight);You can also show, hide, or highlight an entire segment.A pixel position on the screen (such as that returned by the mouse callbacks) can be interconverted with the position in world coordinates.void MapWorldToViewer (ViewеR viewer, PntInfo pnt, PointPtr pt);void MapViewerToWorld (ViewеR viewer, PoinT pt, PntPtr pnt);

Miscellaneous functionsA number of common portable code problems are addressed either by Vibrant or the CoreLib. These include accessing configuration files, obtaining file specifications, display of messages and progress monitors, and dynamic memory allocation.Configuration filesThe CoreLib provides a scheme for storing and modifying persistent system and application options or settings. It is modeled on services provided in the Microsoft Windows environment and has been extended to work on all of the platforms the NCBI toolkit supports. Configuration files are plain ASCII text files that may be edited by the user or modified by the program. They are divided into sections, each of which is headed by a section name enclosed in sequare brackets. Below each section heading is a series of key=value strings, somewhat analagous to the environment variables that are used on many platforms. The following is an example of the "`ncbi`" configuration file:[NCBI]
ROOT=SEQDATA:
ASNLOAD=genome:ENTREZ:asnload:
DATA=genome:ENTREZ:data:In this example, the ROOT entry refers to the path to the *Entrez:* Sequences CD-ROM, the ASNLOAD entry specifies the path to the ASN.1 parse tables (required by the AsnLib functions and all higher-level procedures that call them), and the DATA entry points to files containing information necessary to convert biomolecule sequence data into different alphabets (e.g., unpacking the 2-bit nucleotide code stored on the *Entrez* CD into standard IUPAC letters).The location and naming conventions of configuration files depends upon the platform. On the Macintosh, files are located in the System Folder:Preferences folder and have a "`.cnf`" suffix. Under Microsoft Windows they reside in the Windows directory, and have an "`.INI`" suffix. Configuration files are read and written by `GetAppParam` and `SetAppParam`.Int2 GetAppParam (CharPtr file, CharPtr section,
CharPtr type, CharPtr dflt,
CharPtr buf, Int2 buflen);Boolean SetAppParam (CharPtr file, CharPtr section,
CharPtr type, CharPtr value);

File specification dialog boxesIn traditional command-line programs, the user identifies files by typing the file name along with a path specification. This method is sensitive to typographical errors. In windowing systems, and in Vibrant, the user is presented a list of available files, and is able to traverse the directory hierarchy and specify files with file dialog boxes. `GetInputFileName` and `GetOutputFileName` return the full paths specified by the user.Boolean GetInputFileName (CharPtr fileName, size_t maxsize,
CharPtr extType, CharPtr macType);Boolean GetOutputFileName (CharPtr fileName, size_t maxsize,
CharPtr dfault);Messages and alertsAt certain times a program needs to send a message or warning to the user. The message may ask for certain responses. The `Message` function provides this service by displaying a small window on the screen.Int2 Message (Int2 key, char *format, ...);The key parameter choices include `MSG_ERROR`, `MSG_FATAL`, `MSG_OK`, `MSG_RC`, `MSG_ARI`, `MSG_YN`, `MSG_YNC`, `MSG_OKC`, `MSG_POST`, and `MSG_POSTERR`. (A message of type `MSG_POST` does not wait for a response, and leaves the message window visible at the bottom of the screen. This can be very useful for debugging program crashes.) The answer returned may be `ANS_NO`, `ANS_YES`, `ANS_OK`, `ANS_RETRY`, `ANS_CANCEL`, or `ANS_IGNORE`.The format and remaining arguments are passed to `vsprintf`. A typical message, which prints a string and an integer, is shown below.Message (MSG_OK, "The value of '%s' is %d", str, (int) val);The `Beep` function plays an audible tone to alert the user. (Messages of type `MSG_ERROR` or `MSG_FATAL` will also sound the beep.)void Beep ();Progress monitorsOne of the advantages of graphical interfaces is that users feel like they are in charge of the computer program. A consequence of this is that users worry (and sometimes "panic") if the program does not respond almost

instantaneously. Operations that take on the order of a few seconds (e.g., reading ASN.1 parse table files, connecting to network services) can be indicated by changing the mouse cursor from an arrow to a wait indicator.void ArrowCursor ();void WatchCursor ();For longer operations (e.g., copying large files), where the relative progress towards completion can be estimated, Vibrant provides graphical and text progress monitors. The `MonitorIntNew` function takes the range of values, and `MonitorIntValue` changes the graphical relative progress value. The program should update the progress indicator every few seconds.MonitorPtr MonitorIntNew (CharPtr title, Int4 n1, Int4 n2);MonitorPtr MonitorStrNew (CharPtr title, Int2 len);Boolean MonitorIntValue (MonitorPtr mon, Int4 ival);Boolean MonitorStrValue (MonitorPtr mon, CharPtr sval);MonitorPtr MonitorFree (MonitorPtr mon);Several object loader functions call `ProgMon` during time-consuming operations. The `SetProgMon` function allows these progress messages to be intercepted and passed to a callback.Boolean ProgMon (CharPtr str);void SetProgMon (ProgMonFunc func, VoidPtr data);Path to the executing programThe `ProgramPath` function provides the full path to the executing program. It is particularly useful for finding accessory files or directories that reside along with a program.void ProgramPath (CharPtr buf, size_t maxsize);Application timerThe `Metronome` procedure specifies an application procedure to be called 18 or 20 times per second (on the PC and the Macintosh, respectively) regardless of any action by the user.void Metronome (VoidProc actn);

Viewing keystrokesThe KeyboardView procedure specifies an application procedure to be called whenever the user presses a key on the keyboard, regardless of other action to be taken in response to that event.void KeyboardView (KeyProc key);Error interceptionUnder default conditions, errors detected by the NCBI software toolkit libraries will be reported to the user (via the Message function), and will result in program termination. A program can intercept errors, and prevent termination, through use of the ErrSetOpts function.void ErrSetOpts (short actopt, short logopt);The actopt parameter can be ERR_CONTINUE, ERR_IGNORE, ERR_ADVISE, ERR_ABORT, ERR_PROMPT, or ERR_TEE. The logopt parameter may be ERR_LOG_ON, ERR_LOG_OFF, or 0 to use the current error logging setting. The ErrShow function will cause intercepted errors to be displayed, without terminating program exectution.Memory managementAlthough ANSI C provides a malloc function, its properties and behavior differ among various platforms. The CoreLib memory functions use "far" pointers (VoidPtr). MemNew will always clear the memory block, and will post a fatal error (which can be intercepted) if the system is unable to allocate the requested memory, so checking for a NULL return value is normally not necessary. MemGet will return NULL upon failure, allowing the program to try again with a smaller request.VoidPtr MemNew (size_t size);VoidPtr MemGet (size_t size, Boolean clear_out);VoidPtr MemFree (VoidPtr ptr);There are also functions for MemCopy, MemMove, and MemFill, as well as functions that operate on "Handle" types (which reference relocatable memory on Macintosh and PC/Windows platforms).

Character and string functionsThe CoreLib string functions were written reluctantly, to avoid unpleasant behavior when standard routines were passed `NULL` arguments, and to overcome different pointer sizes on some PC platforms. The character macros (e.g., `IS_ALPHA`, `TO_UPPER`) are robust against "illegal" characters.The `StringSave` function copies a string to allocated memory.CharPtr StringSave (const char FAR *from);`StringRChr (fileName, DIRDELIMCHR)` will separate the file and path names returned by `GetInputFileName` or `GetOutputFileName`.Portable method of obtaining argumentsMany programs written for command-line driven systems (e.g., UNIX and DOS) need to be extensively rewritten in order to run on windowing machines. The `GetArgs` function allows arguments to be obtained in a portable manner, without regard to the presence or absence of a windowing system.Boolean GetArgs (CharPtr progname, Int2 numargs, ArgPtr args);The args array lists information on the parameters desired. The Arg structure includes the prompt, default value, range, whether the argument is optional, the command-line character tag, and the argument type. Arguments can be Booleans, integers, real numbers, strings, input or output files, or input or output data links (typed ASN.1 messages).On non-windowing systems, arguments are processed from the command line. On a windowing system, a dialog box is built, driven by the args array. `GetArgs` returns `FALSE` if not all essential arguments were supplied, if any were out of range, or if the user cancelled the argument dialog.The `GetArgs` and `Message` functions allow procedural code to be written without regard to environment. Existing programs can be quickly modified to run on a variety of platforms.

AcknowledgmentsI wish to thank a number of colleagues at the National Center for Biotechnology Information who have contributed critical ideas to this interface. Jim Ostell proposed the ideas of portability and of dealing with objects at a high level, as close to the level of the desired parameters as is possible. He also made suggestions that led to the concept of automatically positioning objects on the window and within groups. Warren Gish, Greg Schuler, Tim Clark, Peter Karp, and several others also made numerous useful suggestions and constructive criticisms. Early applications in Vibrant, written by or for John Spouge, Jill Shermer, Charles Beatty, Jonathan Epstein, Kenn Rudd and Jinghui Zhang, uncovered design limitations that were quickly remedied. Jill Shermer assisted in porting Vibrant to Motif.Vibrant will be published separately in the near future. You should reference J. Kans (manuscript in preparation). Details on the internal organization of Vibrant will be published as an NCBI technical report. Vibrant is supplied as-is, and is currently "not being supported". (This means that I do want to get bug reports, but won't address them with quite the same urgency as we would apply to the other parts of the toolkit, upon which much of our database and research efforts rely.) Questions or comments can be directed to toolbox@ncbi.nlm.nih.gov.TrademarksThe mention of trade names, commercial products, or organizations does not imply endorsement by the NCBI or the U.S. Government.Apple and Macintosh are registered trademarks of Apple Computer, Inc. Microsoft is a registered trademark and Windows is a trademark of Microsoft Corporation. Motif is a registered trademark of the Open Software Foundation.