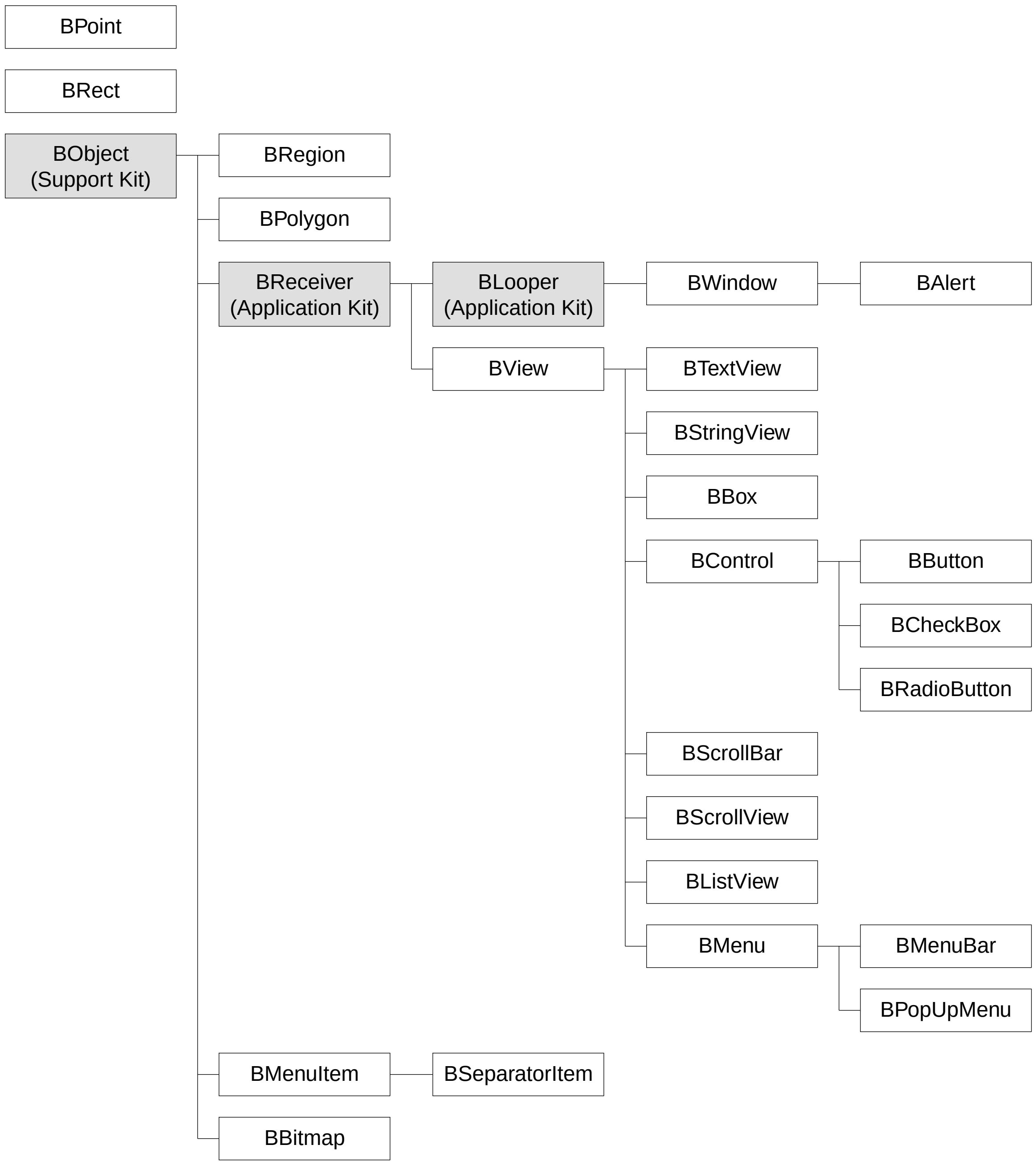
4 The Interface Kit

Interface Kit Inheritance Hierarchy



4 The Interface Kit

Most Be applications have an interactive and graphical user interface. When they start up, they present themselves to the user on-screen in one or more windows. The windows display areas where the user can do something—there may be menus to open, buttons to click, text fields to type in, images to drag, and so on. Each user action on the keyboard or mouse is an *interface event* that's reported to the application and that the application responds to. At least part of the response is always a change in what the window displays—so that users can see the results of their work.

To run this kind of user interface, an application has to do three things. It must:

- Manage a set of windows,
- Draw within the windows, and
- Respond to messages that report interface events.

The application, in effect, carries on a conversation with the user. It draws to present itself on-screen, the user does something with the keyboard or mouse, the event is reported to the application in a message, and the application draws in response, prompting more events and more messages.

The Interface Kit structures this interaction with the user. It defines a set of C++ classes that give applications the ability to manage windows, draw in them, and efficiently respond to the user's instructions. Taken together, these classes define a framework for interactive applications. By programming with the Kit, you'll be able to construct an application that effectively uses the capabilities of the Be machine.

This chapter first introduces the conceptual framework for the user interface, then describes all the classes, functions, types, and constants the Kit defines. The reference material that follows this introduction assumes the concepts and terminology presented here.

Framework for the User Interface

A graphical user interface is organized around windows. Each window has a particular role to play in an application and is more or less independent of other windows. While working on the computer, users think in terms of windows—what's in them and what can be done with them—perhaps more than in terms of applications.

The design of the software mirrors the way the user interface works: it's also organized around windows. Within an application, each window runs in its own thread and is represented by a separate BWindow object. The object is the application's interface to the window the system provides; the thread is where all the work that's centered on the window takes place.

Because every window has its own thread, the user can, for example, scroll the contents of one window while watching an animation in another, or start a time-consuming computation in an application and still be able to use the application's other windows. A window won't stop working when the user turns to another window.

Events that the user directs at a particular window initiate activity within that window's thread. When the user clicks a button within a window, for example, everything that happens in response to the click happens in the window thread (unless the application arranges for other threads to be involved). In its interaction with the user, each window acts on its own, independently of other windows.

Application Server Windows

In a multitasking environment, any number of applications might be running at the same time, each with its own set of windows on-screen. The windows of all running applications must cooperate in a common interface. For example, there can be only one active window at a time—not one per application, but one per machine. A window that comes to the front must jump over every other window, not just those belonging to the same application. When the active window is closed, the window behind it must become active, even if it belongs to a different application.

Because it would be difficult for each application to manage the interaction of its windows with every other application, windows are assigned, at the lowest level, to a separate entity, the Application Server. The Server's principal role in the user interface is to provide applications with the windows they require.

Everything a program or a user does is centered on the windows the Application Server provides. Users type into windows, click buttons in windows, drag images to windows, and so on; applications draw in windows to display the text users type, the buttons they can click, and the images they can drag.

The Application Server, therefore, is the conduit for an application's event input and drawing output:

- It monitors the user's actions on the keyboard and mouse and sends messages reporting interface events to the application.
- It receives drawing instructions from the application and interprets them to render images within windows.

The Server relieves applications of much of the burden of basic user-interface work. The Interface Kit organizes and further simplifies an application's interaction with the Server.

BWindow Objects

Every window in an application is represented by a separate BWindow object. Constructing the BWindow establishes a connection to the Application Server—one separate from, but initially dependent on, the connection previously established by the BApplication object. The Server creates a window for the new object and dedicates a separate thread to it.

The BWindow object is a kind of BLooper, so it spawns a thread for the window in the application's address space and begins running a message loop where it receives messages from the Server reporting interface events. The window thread in the application is directly connected to the dedicated thread in the Server.

The BWindow object, therefore, is in position to serve three crucial roles:

- It can act as the application's interface to a Server window. It has functions that the application can call to manipulate the window programmatically—move it, resize it, close it, and so on. It also declares the functions that the system calls to notify the application that the user manipulated the window.
- It can organize message-handling within the window thread. Since it runs the window's message loop, it gets to decide how each message should be handled. It's the focus and central distribution point for all messages that initiate activity in the thread.
- As the entity that holds rendered images, it can manage the objects that produce those images. (This is discussed under "BView Objects" below.)

All other Interface Kit objects play roles that depend on a BWindow. They draw in a window, respond to event messages received by a window, or act in support of other objects that draw and handle events.

BView Objects

For purposes of drawing and event-handling, a window can be divided up into smaller rectangular areas called *views*. Each view corresponds to one part of what the window displays—a scroll bar, a document, a list, a button, or some other more or less self-contained portion of the window's contents.

An application sets up a view by constructing a BView object and associating it with a particular BWindow. The BView object is responsible for drawing within the view rectangle, and for handling interface events directed at that area.

Drawing Agent

A window is a tablet that can retain and display rendered images, but it can't draw them; for that it needs a set of BViews. A BView is an agent for drawing, but it can't render the images it creates; for that it needs a BWindow. The two kinds of objects work hand in hand.

Each BView object is an autonomous graphics environment for drawing. Some aspects of the environment, such as the list of possible colors, are shared by all BViews and all applications. But within those broad limits, every BView maintains an independent graphics state. It has its own coordinate system, current colors, drawing mode, clipping region, pen position, and so on.

The BView class defines the functions that applications call to carry out elemental drawing tasks—such as stroking lines, filling shapes, drawing characters, and imaging bitmaps. These functions are typically used to implement another function—called **Draw()**—in a class derived from BView. This view-specific function draws the contents of the view rectangle.

The BWindow will call the BView's **Draw()** function whenever the window's contents (or at least the part that the BView has control over) need to be updated. A BWindow first asks its BViews to draw when the window is initially placed on-screen. Thereafter, they might be asked to refresh the contents of the window whenever the contents change or when they're revealed after being hidden or obscured. A BView might be called upon to draw at any time.

Because **Draw()** is called on the command of others, not the BView, it can be considered to draw *passively*. It presents the view as it currently appears. For example, the **Draw()** function of a BView that displays editable text would draw the characters that the user had inserted up to that point.

BViews also draw *actively* in response to messages reporting interface events. For example, text is highlighted as the user drags over it and is then replaced as the user types. Each change is the result of an event reported to the BView. For passive drawing, the BView implements a function (**Draw()**) that others may call. For active drawing, it calls the drawing functions itself (it may even call **Draw()**).

Message Receiver

The drawing that a BView does is often designed to prompt a user response of some kind—an empty text field with a blinking caret invites typed input, a menu item or a button invites a click, an icon looks like it can be dragged, and so on.

When the user acts, system messages that report the resulting events are sent to the BWindow object, which determines which BView elicited the events and should respond to them. For example, a BView that draws typed text can expect to respond to messages reporting the user's keystrokes. A BView that draws a button gets to handle the events that are generated when the button is clicked. The BView class derives from BReceiver, so BView objects are eligible to receive messages dispatched by the BWindow.

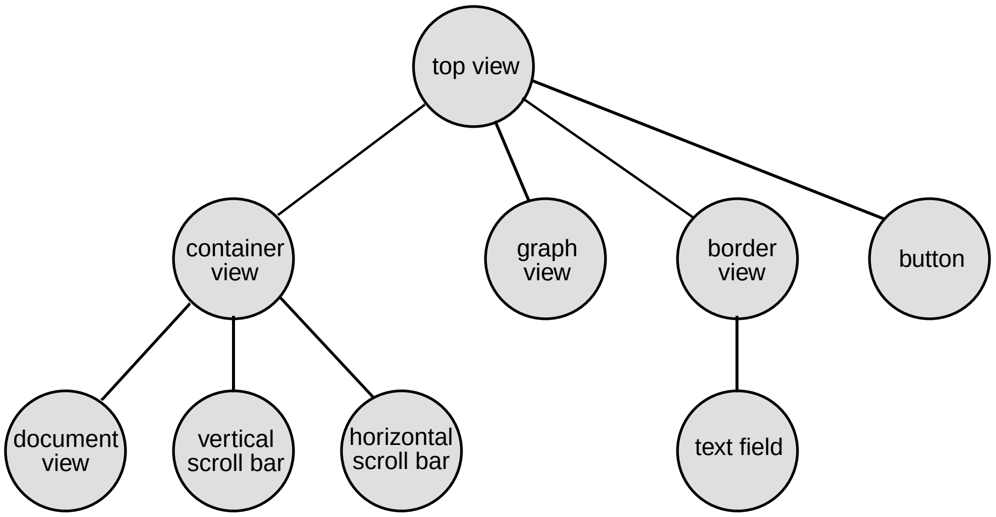
Just as classes derived from BView implement **Draw()** functions to draw within the view rectangle, they also implement the hook functions that respond to events. These functions are discussed later, under "Hook Functions for Interface Events" on page 42.

Largely because of its graphics role and its central role in handling events, BView is the biggest and most diverse class in the Interface Kit. Most other Interface Kit classes are derived from it.

The View Hierarchy

A window typically contains a number of different views—all arranged in a hierarchy beneath the *top view*, a view that's exactly the same size as the content area of the window. The top view is a companion of the window; it's created by the BWindow object when the BWindow is constructed. When the window is resized, the top view is resized to match. Unlike other views, the top view doesn't draw or handle events; it serves merely to connect the window to the views that the application creates and places in the hierarchy.

As illustrated in the diagram below, the view hierarchy can be represented as a branching tree structure with the top view at its root. All views in the hierarchy (except the top view) have one, and only one, parent view. Each view (including the top view) can have any number of child views.



In this diagram, the top view has four children, the container view has three, and the border view one. Child views are located within their parents, so the hierarchy is one of overlapping rectangles. The container view, for example, takes up some of the top view's area and divides its own area into a document view and two scroll bars.

When a new BView object is created, it isn't attached to a window and it has no parent. It's added to a window by making it a child of a view already in the view hierarchy. This is done with the **AddChild()** function. A view can be made a child of the window's top view by calling BWindow's version of **AddChild()**.

Until it's assigned to a window, a BView can't draw and won't receive reports of events. BViews know how to produce images, but it takes a window to display and retain the images they create.

Drawing and Event-Handling in the View Hierarchy

The view hierarchy determines what's displayed where on-screen, and also how user actions are associated with the responsible BView object:

- When the views in a window are called upon to draw, parents draw before their children; children draw in front of their ancestors.
- Mouse events (like the mouse-down and mouse-up events that result from a click) are associated with the view where the cursor is located. Since the cursor points to the frontmost view at any given location, it's likely to be pointing at a view close to the bottom of the hierarchy. It's those views—the ones that have no children—that are responsible for most of the drawing and event-handling for the window. Views farther up the hierarchy tend to contain and organize those at the bottom.

Overlapping Siblings

Although children wait for their parents when it comes time to draw and parents defer to their offspring when it comes to time to handle events, sibling views are not so well-behaved. Siblings don't draw in any predefined order. This doesn't matter, as long as the view rectangles of the siblings don't overlap. If they do overlap, it's indeterminate which view will draw last—that is, which one will draw on top of the other.

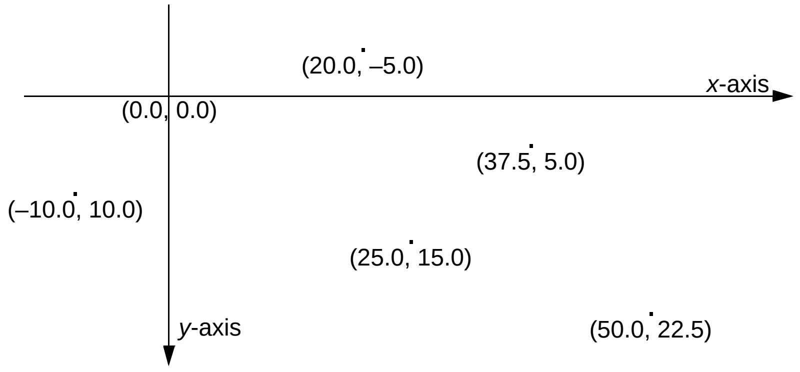
Similarly, it's indeterminate which view will be associated with mouse events in the area the siblings share. It may be one view or it may be the other, and it won't necessarily be the one that drew the image the user sees.

Therefore, it's strongly recommended that sibling views should be arranged so that they don't overlap.

The Coordinate Space

To locate windows and views, draw in them, and report where the cursor is positioned over them, it's necessary to have some conventional way of talking about the display surface. The same conventions are used whether the display device is a monitor that shows images on a screen or a printer that puts them on a page.

In Be software, the display surface is described by a standard two-dimensional coordinate system where the *y*-axis extends downward and the *x*-axis extends to the right, as illustrated below:



y coordinate values are greater towards the bottom of the display and smaller towards the top, *x* coordinate values are greater to the right and smaller to the left.

The axes define a continuous coordinate space where distances are measured by floating-point values (**float**s). All quantities in this space—including widths and heights, *x* and *y* coordinates, font sizes, angles, and the size of the pen—are floating point numbers.

Floating-point coordinates permit precisely stated measurements that can take advantage of display devices with higher resolutions than the screen. For example, a vertical line 0.4 units wide would be displayed using a single column of pixels on-screen, the same as a line 1.4 units wide. However, a 300 dpi printer would use two pixel columns to print the 0.4-unit line and six to print the 1.4-unit line.

A coordinate unit is 1/72 of an inch, roughly equal to a typographical point. However, all screens are considered to have a resolution of 72 pixels per inch (regardless of the actual dimension), so coordinate units count screen pixels. One unit is the distance between the centers of adjacent pixels on-screen.

Coordinate Systems

Specific coordinate systems are associated with the screen, with windows, and with the views inside windows. They differ only in where the two axes are located:

- The global or *screen coordinate system* has its origin, (0.0, 0.0), at the left top corner of the screen. It's used for positioning windows on-screen, < for arranging multiple screens connected to the same machine, > and for comparing coordinate values that weren't originally stated in a common coordinate system.

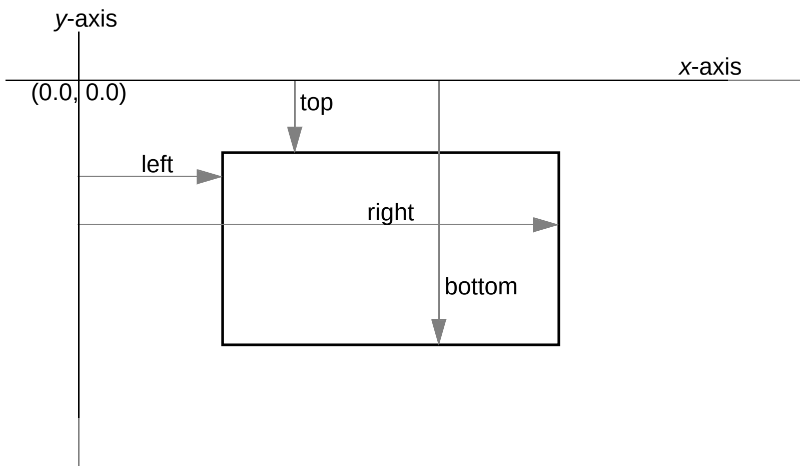
- A *window coordinate system* has its origin at the left top corner of the content area of a window. It's used principally for positioning views within the window. Each window has its own coordinate system so that locations within the window can be specified without regard to where the window happens to be on-screen.

- A *view coordinate system* has its default origin at the left top corner of the view rectangle. However, scrolling can shift view coordinates and move the origin. View-specific coordinates are used for all drawing operations and to report most events.

Coordinate Geometry

The Interface Kit defines a handful of basic classes for locating points and areas within a coordinate system:

- A BPoint object is the simplest way to specify a coordinate location. Each object stores two values—an *x* coordinate and a *y* coordinate—that together locate a specific point, (*x*, *y*), within a given coordinate system.
- A BRect object represents a rectangle; it's the simplest way to designate an area within a coordinate system. The BRect class defines a rectangle as a set of four coordinate values—corresponding to the rectangle's left, top, right, and bottom edges, as illustrated below:



The sides of the rectangle are therefore parallel to the coordinate axes. The left and right sides delimit the range of *x* coordinate values within the rectangle, and the top and bottom sides delimit the range of *y* coordinate values. For example, if a rectangle's left top corner is at (0.8, 2.7) and its right bottom corner is at (11.3, 49.5), all points having *x* coordinates ranging from 0.8 through 11.3 and *y* coordinates from 2.7 through 49.5 lie inside the rectangle.

If the top of a rectangle is the same as its bottom, or its left the same as its right, the rectangle defines a straight line. If the top and bottom are the same and also the left and right, it collapses to a single point. Such rectangles are still valid—they specify real locations within a coordinate system. However, if the top is greater than the bottom or the left greater than the right, the rectangle is invalid; it has no meaning.

- A BPolygon object represents a polygon, a closed figure with an arbitrary number of sides. The polygon is defined as an ordered set of points. It encloses the area that would be outlined by connecting the points in order, then connecting the first and last points to close the figure. Each point is therefore a potential vertex of the polygon.
- A BRegion object defines a set of points. A region can be any shape and even include discontinuous areas.

Mapping Coordinates to Pixels

The device-independent coordinate space described above must be mapped to the pixel grid of a particular display device—the screen, a printer, or some other piece of hardware that's capable of rendering an image. For example, to display a rectangle, it's necessary to find the pixel columns that correspond to its right and left sides and the pixel rows that correspond to its top and bottom.

This depends entirely on the resolution of the device. In essence, each device-independent coordinate value must be translated internally to a device-dependent value—an integer index to a particular column or row of pixels. In the coordinate space of the device, one unit equals one pixel.

This translation is easy for the screen, since, as mentioned above, there's a one-to-one correspondence between coordinate units and pixels. It reduces to rounding floating-point coordinates to integers. For other devices, however, the translation means first scaling the coordinate value to a device-specific value, then rounding. For example, the point (12.3, 40.8) would translate to (12, 41) on the screen, but to (51, 170) on a 300 dpi printer.

Screen Pixels

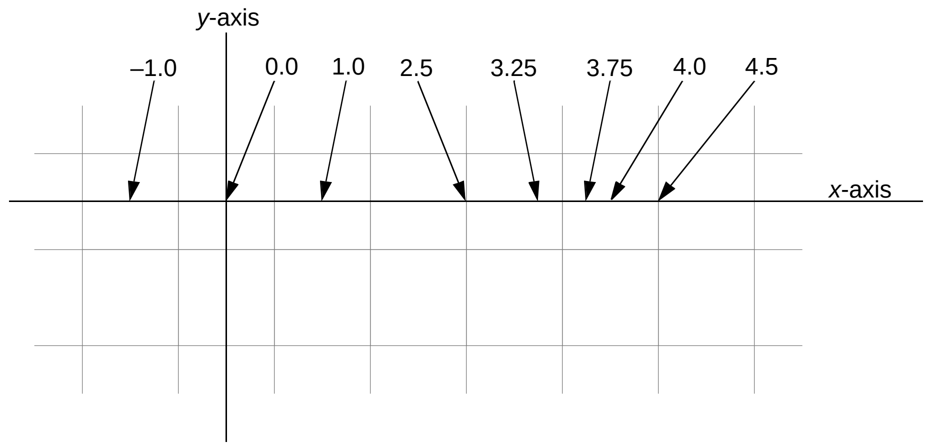
To map coordinate locations to device-specific pixels, you need to know only two things:

- The resolution of the device, and
- The location of the coordinate axes relative to pixel boundaries.

The axes are located in the same place for all devices: The *x*-axis runs left to right along the middle of a row of pixels and the *y*-axis runs down the middle of a pixel column. They meet at the very center of a pixel.

Because coordinate units match pixels on the screen, this means that all integral coordinate values (those without a fractional part) fall midway across a screen pixel.

The following illustration shows where various *x* coordinate values fall on the *x*-axis. The broken lines represent the division of the screen into a pixel grid:



As this illustration shows, it's possible to have coordinate values that lie on the boundary between two pixels. A later section, "Picking Pixels to Stroke and Fill" section on page 34, describes how these values are mapped to one pixel or the other.

Drawing

Drawing is done by BView objects. As discussed above, the views within a window are organized into a hierarchy—there can be views within views—but each view is an independent drawing agent and maintains a separate graphics environment. This section discusses the framework in which BViews draw, beginning with view coordinate systems. Detailed descriptions of the functions mentioned here can be found in the BView and BWindow class descriptions.

View Coordinate Systems

As a convenience, each view is assigned a coordinate system of its own. By default, the coordinate origin—(0.0, 0.0)—is located at the left top corner of the view rectangle. (For an overview of the coordinate systems assumed by the Interface Kit, see "The Coordinate Space" section on page 14 above.)

When a view is added as a child of another view, it's located within the coordinate system of its parent. A child is considered part of the contents of the parent view. If the parent moves, the child moves with it; if the parent view scrolls its contents, the child view is shifted along with everything else in the view.

Since each view retains its own internal coordinate system no matter who its parent is, where it's located within the parent, or where the parent is located, a BView's drawing and event-handling code doesn't need to be concerned about anything exterior to itself.

To do its work, a BView need look no farther than the boundaries of its own view rectangle.

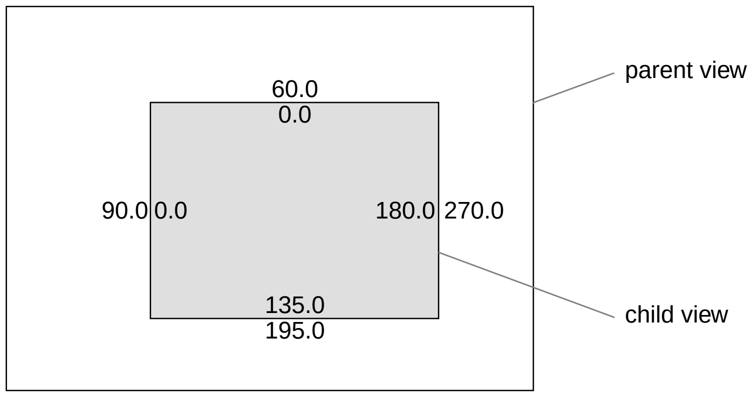
Frame and Bounds Rectangles

Although a BView doesn't have to look outside its own boundaries, it does have to know where those boundaries are. It can get this information in two forms:

- Since a view is located within the coordinate system of its parent, the view rectangle is initially defined in terms of the parent's coordinates. This defining rectangle for a view is known as its *frame rectangle*. (See the BView constructor and the **Frame()** function.)

- When translated from the parent's coordinates to the internal coordinates of the view itself, the same rectangle is known as the *bounds rectangle*. (See the **Bounds()** function.)

The illustration below shows a child view 181.0 units wide and 136.0 units high. When viewed from the outside, from the perspective of its parent's coordinate system, it has a frame rectangle with left, top, right, and bottom coordinates at 90.0, 60.0, 270.0, and 195.0, respectively. But when viewed from the inside, in the view's own coordinate system, it has a bounds rectangle with coordinates at 0.0, 0.0, 180.0, and 135.0:

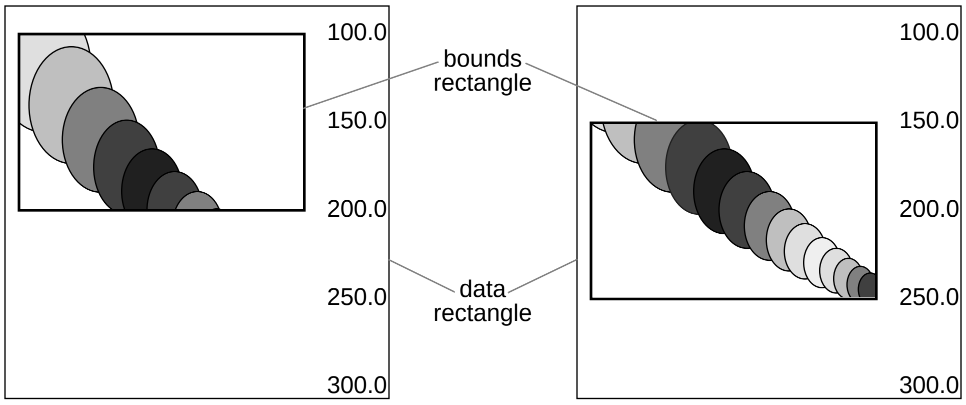


When a view moves to a new location in its parent, its frame rectangle changes but not its bounds rectangle. When a view scrolls its contents, its bounds rectangle changes, but not its frame. The frame rectangle positions the view in the world outside; the bounds rectangle positions the contents inside the view.

Since a BView does its work in its own coordinate system, it refers to the bounds rectangle more often than to the frame rectangle.

Scrolling

A BView scrolls its contents by shifting coordinate values within the view rectangle—that is, by altering the bounds rectangle. If, for example, the top of a view's bounds rectangle is at 100.0 and its bottom is at 200.0, scrolling downward 50.0 units would put the top at 150.0 and the bottom at 250.0. Contents of the view with *y* coordinate values of 150.0 to 200.0, originally displayed in the bottom half of the view, would be shifted to the top half. Contents with *y* coordinate values from 200.0 to 250.0, previously unseen, would become visible at the bottom of the view. This is illustrated below:



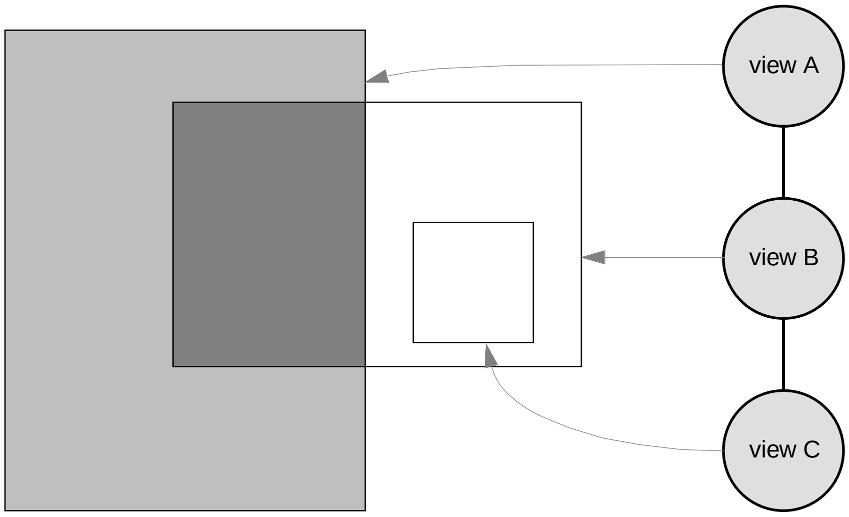
Scrolling doesn't move the view—it doesn't alter the frame rectangle—it moves only what's displayed inside the view. In the illustration above, a "data rectangle" encloses everything the BView is capable of drawing. For example, if the view is able to display an entire book, the data rectangle would be large enough to enclose all the lines and pages of the book laid end to end. However, since a BView can draw only within its bounds rectangle, everything in the data rectangle with coordinates that fall outside the bounds rectangle would be invisible. To make unseen data visible, the bounds rectangle must change the coordinates that it encompasses. Scrolling can be thought of as sliding the view's bounds rectangle to a new position on its data rectangle, as is shown in the illustration above. However, as it appears to the user, it's moving the data rectangle under the bounds rectangle. The view doesn't move; the data does.

Clipping Region

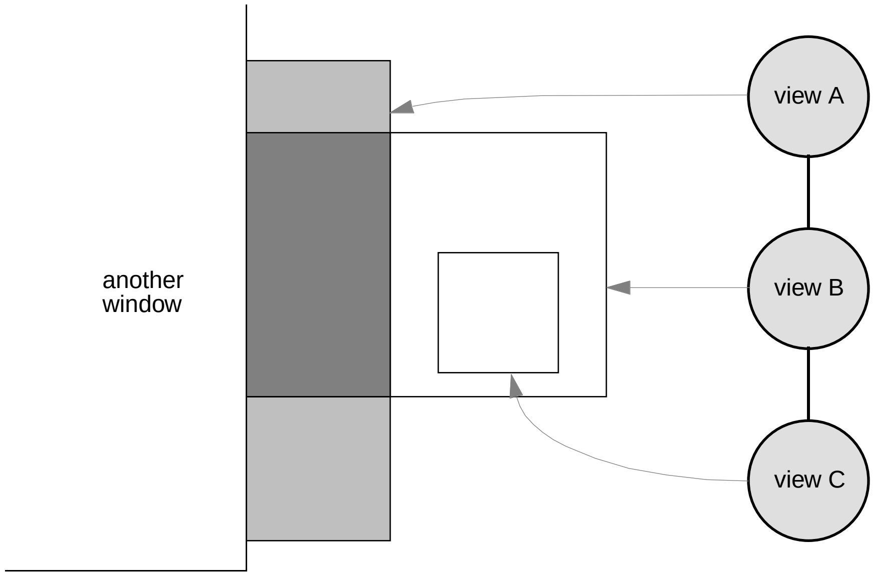
The Application Server clips the images that a BView produces to the region where it's permitted to draw.

This region is never any larger than the view's bounds rectangle; a view cannot draw outside its bounds. Furthermore, since a child is considered part of its parent, a view can't draw outside the bounds rectangle of its parent either—or, for that matter, outside the bounds rectangle of any ancestor view. In addition, since child views draw after, and therefore logically in front of, their parents, a view concedes some of its territory to its children.

Thus, the *visible region* of a view is the part of its bounds rectangle that's inside the bounds rectangles of all its ancestors, minus the frame rectangles of its children. This is illustrated in the figure below. It shows a hierarchy of three views. The area filled with a crosshatch pattern is the visible region of view *A*; it omits the area occupied by its child, view *B*. The visible region of view *B* is colored dark gray; it omits the part of the view that lies outside its parent. View *C* has no visible region, for it lies outside the bounds rectangle of its ancestor, view *A*:



The visible region of a view might be further restricted if its window is obscured by another window or if the window it's in lies partially off-screen. The visible region includes only those areas that are actually visible to the user. For example, if the three views in the illustration above were in a window that was partially blocked by another window, their visible regions might be considerably smaller. This is illustrated below:



Note that in this case, view *A* has a discontinuous visible region.

The Application Server clips the drawing that a view does to a region that's never any larger than the visible region. On occasion, it may be smaller. For the sake of efficiency, while a view is being automatically updated, the *clipping region* excludes portions of the visible region that don't need to be redrawn:

- When a view is scrolled, the Application Server may be able to shift some of its contents from one portion of the visible region to another. The clipping region excludes any part of the visible region that the Server was able to update on its own; it includes only the part where the BView must produce images that were not previously visible.

- If a view is resized larger, the clipping region may include only the new areas that were added to the visible region. (But see the *flags* argument for the BView constructor.)

- If only part of a view is invalidated (by the **Invalidate()** function), the clipping region is the intersection of the visible region and the invalid rectangle.

An application can also limit the clipping region for a view by passing a BRegion object to **SetClippingRegion()**. The clipping region won't include any areas that aren't in the region passed. The Application Server calculates the clipping region as it normally would, but intersects it with the specified region.

You can obtain the current clipping region for a view by calling **GetClippingRegion()**. (See also the BRegion class description.)

The View Color

Every view has a basic, underlying color. It's the color that fills the view rectangle before the BView does any drawing. The user may catch a glimpse of this color when the view is first shown on-screen, when it's resized larger, and when it's erased in preparation for an update. It will also be seen wherever the BView fails to draw in the visible region.

In a sense, the view color is the canvas on which the BView draws. It doesn't enter into any of the object's drawing operations except to provide a background; it's not one of the BView's graphics parameters.

By default, the view color is white. You can assign a different color to a view by calling BView's **SetViewColor()** function. Every view can have its own color.

The Mechanics of Drawing

Views draw through a set of primitive functions such as:

- **DrawString()**, which draws a string of characters,

- **DrawBitmap()**, which produces an image from a bitmap,

- **StrokeLine()**, **StrokeArc()**, and other **Stroke...()** functions, which stroke lines along defined paths, and
- **FillEllipse()**, **FillRect()**, and other **Fill...()** functions, which fill closed shapes.

The way these functions work depends not only on the values that they're passed—the particular string, bitmap, arc, or ellipse that's to be drawn—but on previously set values in the BView's graphics environment.

Graphics Environment

Each BView object maintains its own graphics environment for drawing. The coordinate system and the clipping region are two fundamental parts of that environment, but not the only parts. It also includes a number of parameters that can be set and reset at will to affect the next image drawn. These parameters are:

- Font attributes that determine the appearance of text the BView draws. (See **SetFontName()** and its companion functions.)
- Two pen parameters—a location and a size. The pen location determines where the next drawing will occur and the pen size determines the thickness of stroked lines. (See **MovePenBy()** and **SetPenSize()**.)
- Two current colors—a *front color* and a *background color*—that can be used either alone or in combination to form a pattern or halftone. The front color is used for most drawing. The background color is sometimes set to the underlying view color so that it can be used to erase other drawing or, because it matches the view background, make it appear that drawing has not touched certain pixels. (See the **SetFrontColor()** and **SetBackColor()** functions and the "Patterns" section below.)
- A drawing mode that determines how the next image is to be rendered. (See the "Drawing Modes" section below and the **SetDrawingMode()** function.)

By default, a BView's graphics parameters are set to the following values:

Font	"monaco" (9-point bitmap font, no rotation, 90° shear)
Pen position	(0.0, 0.0)
Pen size	1.0 coordinate units
Front color	Black (red, green, and blue components all equal to 0)
Background color	White (red, green, and blue components all equal to 255)
Drawing mode	Copy mode (**OP_COPY**)
Clipping region	The visible region of the view
Coordinate system	Origin at the left top corner of the bounds rectangle

However, as the next section, "Views and the Server" on page 30, explains, these values take effect only when the BView is assigned to a window.

The Pen

The pen is a fiction that encompasses two properties of a view's graphics environment: the current drawing location and the thickness of stroked lines.

The pen location determines where the next image will be drawn—but only for the few functions that don't fully specify the drawing coordinates. Some drawing functions alter the pen location—as if the pen actually moves as it does the drawing—but usually it's set by calling **MovePenBy()** or **MovePenTo()**.

The pen that draws lines (through the various **Stroke...()** functions) has a square tip. The larger the square, the thicker the line that it draws. The size of the tip—the length of one side of the square—is fixed by **SetPenSize()**.

The pen size is expressed in coordinate units, which must be translated to a particular number of pixels for the display device. This is done by scaling the pen size to a device-specific value and rounding to the closest integer. For example, pen sizes of 2.6 and 3.3 would both translate to 3 pixels on-screen, but to 7 and 10 pixels respectively on a 300 dpi printer.

The size is never rounded to 0; no matter how small the pen may be, the line never disappears. If the pen size is set to 0.0, the line will be as thin as possible—it will be drawn using the fewest possible pixels on the display device. (In other words, it will be rounded to 1 for all devices.)

If the size translates to a square with more than one pixel on a side, the pixel in the left top corner follows the path of the line. The pen extends to the right and below the path that it strokes.

A later section, "Picking Pixels to Stroke and Fill" on page 34, illustrates how pens of different sizes choose the pixels to be colored.

Colors

The front and background colors are specified as **rgb_color** values—full 24-bit values with separate red, green, and blue components. Although there may be limitations on the colors that can be rendered on-screen, there are none on the colors that can be specified.

The way colors are specified for a bitmap depends on the color space in which they're interpreted. The color space determines the *depth* of the bitmap data (how many bits of information are stored for each pixel) and its *interpretation* (whether the data represents shades of gray or true colors, whether it's segmented into color components, what the components are, and so on). Four possible color spaces are recognized:

MONOCHROME_1_BIT	One bit of data per pixel, where 1 is black and 0 is white.

GRAYSCALE_8_BIT	Eight bits of data per pixel, where a value of 255 is black and 0 is white. <This color space is currently not implemented. >
COLOR_8_BIT	Eight bits of data per pixel, interpreted as an index into a list of 256 colors. The list is part of the system color map, and is the same for all applications.
RGB_24_BIT	Four components of data per pixel—red, green, blue, and alpha, arranged in that order—with eight bits per component. A component value of 255 yields the maximum amount of red, green, or blue, and a value of 0 indicates the absence of that color. < The alpha component is currently ignored. It will specify the coverage of the color—how transparent or opaque it is. >

The components of an **RGB_24_BIT** color are meshed rather than separated into distinct planes; all four components are specified for the first pixel before the four components for the second pixel, and so on.

The format of a **rgb_color** value exactly matches that of the **RGB_24_BIT** color space—in other words, the front and background colors are specified as **RGB_24_BIT** colors. However, on-screen, all colors are rendered in the **COLOR_8_BIT** color space. Specified 24-bit colors are converted to the closest 8-bit color in the color list. (See the BBitmap class and the **system_colors()** global function.)

Patterns

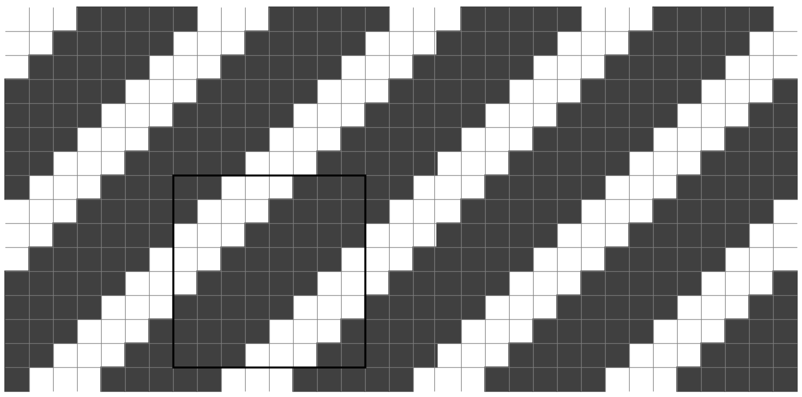
Functions that stroke a line or fill a closed shape don't draw directly in either the front or the background color. Rather they take a *pattern*, an arrangement of one or both colors that's repeated over the entire surface being drawn.

By combining the background color with the front color, patterns can produce dithered colors that lie somewhere between two hues in the **COLOR_8_BIT** color space. Patterns also permit drawing with less than the solid front color (for intermittent or broken lines, for example) and can take advantage of drawing modes that treat the background color as if it were transparent, as discussed below.

A pattern is defined as an 8-pixel by 8-pixel square. The **pattern** type is 8 bytes long, with one byte per row and one bit per pixel. Rows are specified from top to bottom and pixels from left to right. Bits marked 1 designate the front color; those marked 0 designate the background color. For example, a pattern of wide diagonal stripes could be defined as follows:

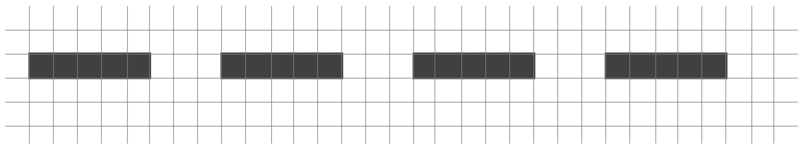
```
pattern stripes = { 0xc7, 0x8f, 0x1f, 0x3e,
                    0x7c, 0xf8, 0xf1, 0xe3 };
```

Patterns repeat themselves across the screen, like tiles that are laid side by side. The pattern defined above looks like this:



The dotted lines in this illustration show the separation of the screen into pixels. The thicker black line outlines one 8-by-8 square that the pattern defines.

The outline of the shape being filled or the width of the line being stroked determines where the pattern is revealed. It's as if the screen was covered with the pattern just below the surface, and stroking or filling allowed some of it to show through. For example, stroking a one-pixel wide horizontal path in the pattern illustrated above would result in a dotted line, with the dashes (in the front color) slightly longer than the spaces between (in the background color):



When stroking a line or filling a shape, the pattern serves as the source image for the current drawing mode, as explained under "Drawing Modes" below. The nature of the mode determines how the pattern interacts with the destination image, the image already in place.

The Interface Kit defines three patterns:

- **solid_front** consists only of the front color,
- **solid_back** has only the background color, and
- **mixed_colors** mixes the two colors evenly, like the pattern on a checkerboard.

solid_front is the default pattern for all drawing functions. Applications can define as many other patterns as they need.

Drawing Modes

When a BView draws, it in effect transfers an image to a target location somewhere in the view rectangle. The drawing mode determines how the image being transferred interacts with the image already in place at that location. The image being transferred is known as the *source image*; it might be a bitmap or a pattern of some kind. The image already in place is known as the *destination image*.

In the simplest and most straightforward kind of drawing, the source image is simply painted on top of the destination; the source replaces the destination. However, there are other possibilities. There are nine different drawing modes, nine distinct ways of combining the source and destination images. The modes are designated by **drawing_mode** constants that can be passed to **SetDrawingMode()**:

OP_COPY	**OP_MIN**	**OP_ADD**
OP_OVER	**OP_MAX**	**OP_SUBTRACT**
OP_ERASE	**OP_INVERT**	**OP_BLEND**

OP_COPY is the default mode and the simplest. It transfers the source image to the destination, replacing whatever was there before. The destination is ignored.

In the other modes, however, some of the destination might be preserved, or the source and destination might be combined to form a result that's different from either of them. For these modes, it's convenient to think of the source image as an image that exists somewhere independent of the destination location, even though it's not actually visible. It's the image that would be rendered at the destination in **OP_COPY** mode.

The modes work for all BView drawing functions—including those that stroke lines and fill shapes, those that draw characters, and those that image bitmaps. The way they work depends foremost on the nature of the source image—whether it's a *pattern* or a *bitmap*. For the **Fill**... and **Stroke**... functions, the source image is a pattern that has the same shape as the area being filled or the area the pen touches as it strokes a line. For **DrawBitmap()**, the source image is a rectangular bitmap.

- Only a source pattern has designated "front" and "background" colors. Even if a source bitmap has colors that match the current front and background, they're not handled like the colors in a pattern; they're treated just like any other color in the bitmap.

- On the other hand, only a source bitmap can have transparent pixels. In the **COLOR_8_BIT** color space, a pixel is made transparent by assigning it the **TRANSPARENT_8_BIT** value. In the **RGB_24_BIT** color space, a pixel assigned the **TRANSPARENT_24_BIT** value is considered transparent. These values have meaning only for source bitmaps, not for source patterns. If the current front or background color in a pattern happens to have a transparent value, it's still treated as the front or background color, not like transparency in a bitmap.

The way the drawing modes work also depends on the color space of the source image and the color space of the destination. The following discussion concentrates on drawing where the source and destination both contain colors. This is the most common case, and also the one that's most general.

When applied to colors, the nine drawing modes fall naturally into four groups:

- The **OP_COPY** mode, which copies the source image to the destination.
- The **OP_OVER**, **OP_ERASE**, and **OP_INVERT** modes, which—despite their differences—all treat the background color in a pattern as if it were transparent.
- The **OP_ADD**, **OP_SUBTRACT**, and **OP_BLEND** modes, which combine colors in the source and destination images.
- The **OP_MIN** and **OP_MAX** modes, which choose between the source and destination colors.

The following paragraphs describe each of these groups in turn.

OP_COPY. In **OP_COPY** mode, the source image replaces the destination. This is the default drawing mode and the one most commonly used. Because this mode doesn't have to test for particular color values in the source image, look at the colors in the destination, or compute colors in the result, it's also the fastest of the modes.

If the source image contains transparent pixels, their transparency will be retained in the result; the transparent value is copied just like any other color. However, the appearance of a transparent pixel when shown on-screen is indeterminate. If a source image has transparent portions, it's best to transfer it to the screen in **OP_OVER** or another mode. In all modes other than **OP_COPY**, a transparent pixel in a source bitmap preserves the color of the corresponding destination pixel.

OP_OVER, OP_ERASE, and OP_INVERT. These three drawing modes are designed specifically to make use of transparency in the source image; they're able to preserve some of the destination image. In these modes (and only these modes) the background color in a source pattern acts just like transparency in a source bitmap.

- The **OP_OVER** mode places the source image "over" the destination; the source provides the foreground and the destination the background. In this mode, the source image replaces the destination image (just as in the **OP_COPY** mode)—except where a source bitmap has transparent pixels and a source pattern has the background color. Transparency in a bitmap and the background color in a pattern retain the destination image in the result.

  By masking out the unwanted parts of a rectangular bitmap with transparent pixels, this mode can place an irregularly shaped source image on top of a background image. Transparency in the source foreground lets the destination background show through. The versatility of **OP_OVER** makes it the second most commonly used mode, after **OP_COPY**.

- The **OP_ERASE** mode doesn't draw the source image at all. Instead, it erases the destination image. Like **OP_OVER**, it preserves the destination image wherever a source bitmap is transparent or a source pattern has the background color. But everywhere else—where the source bitmap isn't transparent and the source pattern has the front color—it removes the destination image, replacing it with the background color.

Although this mode can be used for selective erasing, it's simpler to erase by filling an area with the **solid_back** pattern in **OP_COPY** mode.

- The **OP_INVERT** mode, like **OP_ERASE**, doesn't draw the source image. Instead, it inverts the colors in the destination image. As in the case of the **OP_OVER** and **OP_ERASE** modes, where a source bitmap is transparent or a source pattern has the background color, the destination image remains unchanged in the result. Everywhere else, the color of the destination image is inverted.

These three modes also work for monochrome images. If the source image is monochrome, the distinction between source bitmaps and source patterns breaks down. Two rules apply:

- If the source image is a monochrome bitmap, it acts just like a pattern. A value of 1 in the bitmap designates the current front color and a value of 0 designates the current background color. Thus, 0, rather than **TRANSPARENT_24_BIT** or **TRANSPARENT_8_BIT**, becomes the transparent value.

- If the source and destination are both monochrome, the front color is necessarily black (1) and the background color is necessarily white (0)—but otherwise the drawing modes work as described. With the possible colors this severely restricted, the three modes are reduced to boolean operations: **OP_OVER** is the same as a logical '*OR*', **OP_INVERT** the same as logical '*exclusive OR*', and **OP_ERASE** the same as an inversion of logical '*AND*'.

OP_ADD, OP_SUBTRACT, and OP_BLEND. These three drawing modes combine the source and destination images, pixel by pixel, and color component by color component. As in most of the other modes, transparency in a source bitmap preserves the destination image in the result. Elsewhere, the result is a combination of the source and destination. The front and background colors of a source pattern aren't treated in any special way; they're handled just like other colors.

- **OP_ADD** adds each component of the source color to the corresponding component of the destination color, with a component value of 255 as the limit. Colors become brighter, closer to white.

  By adding a uniform gray to each pixel in the destination, for example, the whole destination image can be brightened by a constant amount.

- **OP_SUBTRACT** subtracts each component of the source color from the corresponding component of the destination color, with a component value of 0 as the limit. Colors become darker, closer to black.

  For example, by subtracting a uniform amount from the red component of each pixel in the destination, the whole image can be made less red.

- **OP_BLEND** averages each component of the source and destination colors (adds the source and destination components and divides by 2). The two images are merged into one.

These modes work only for color images, not for monochrome ones. If the source or destination is specified in the **COLOR_8_BIT** color space, the color will be expanded to a full **COLOR_24_BIT** value to compute the result; the result is then contracted to the closest color in the **COLOR_8_BIT** color space.

OP_MIN and OP_MAX. These two drawing modes compare each pixel in the source image to the corresponding pixel in the destination image and select one to keep in the result. If the source pixel is transparent, both modes select the destination pixel. Otherwise, **OP_MIN** selects the darker of the two colors and **OP_MAX** selects the brighter of the two. If the source image is a uniform shade of gray, for example, **OP_MAX** would substitute that shade for every pixel in the destination image that was darker than the gray.

Like **OP_ADD**, **OP_SUBTRACT**, and **OP_BLEND**, **OP_MIN** and **OP_MAX** work only for color images.

Views and the Server

Just as windows lead a dual life—as on-screen entities provided by the Application Server and as BWindow objects in the application—so too do views. Each BView object has a shadow counterpart in the Server. The Server knows the view's location, its place in the window's hierarchy, its visible area, and the current state of its graphics parameters. Because it has this information, the Server can more efficiently associate a user action with a particular view and interpret the BView's drawing instructions.

BWindows become known to the Application Server when they're constructed. Creating a BWindow object causes the Server to produce the window that the user will eventually see on-screen. A BView, on the other hand, has no effect on the Server when it's constructed. It becomes known to the Server only when it's attached to a BWindow. The Server must look through the application's windows to see what views it has.

A BView that's not attached to a window therefore lacks a counterpart in the Server. This means that some functions can't operate on unattached BViews. Three kinds of functions are included in this group:

- Most obvious among them are the drawing functions—**DrawBitmap()**, **FillRect()**, **StrokeLine()**, and so on. A BView can't draw unless it's in a window.

- Also included are functions that set and return graphics parameters—such as **DrawingMode()**, **SetFontSize()**, **ScrollTo()**, and **SetFrontColor()**. A view's graphic state is kept within the Server (where it's needed to carry out drawing instructions). BViews that the Server doesn't know about don't have a valid graphics state. It won't work, for example, to create a BView, set its background color, and then attach it to a window. The background color can be set only after the BView belongs to the window.

- The group similarly includes functions that indirectly depend on a BView's graphics parameters—such as **GetMouse()**, which reports the cursor location in the BView's coordinates, and **StringWidth()**, which returns how much room a string would take up in the BView's font. These functions require information that an unattached BView can't provide.

Because of these restrictions, you may find it impossible to complete the initialization of a BView at the time it's constructed. Instead, you may need to wait until the BView receives an **AttachedToWindow()** notification informing it that it has been added to a window's view hierarchy. **AttachedToWindow()** can be implemented to set graphics parameters and to take care of any other final initialization that's required.

When a BView is removed from a window, it loses its graphics environment. Thus if a BView is moved to a different window or it changes its position in the view hierarchy of the same window, its graphics parameters must be reset. **AttachedToWindow()** is called to reset them.

The Update Mechanism

The Application Server sends a message to a BWindow whenever any of the views within the window need to be updated. The BWindow then calls the **Draw()** function of each out-of-date BView so that it can redraw the contents of its on-screen display.

Update messages can arrive at any time. A BWindow receives one whenever:

- The window is first placed on-screen, or is shown again after having been hidden.
- Any part of the window becomes visible after being obscured.
- The views in the window are rearranged—for example, if a view is resized or a child is added or removed from the hierarchy.
- Something happens to alter what a particular view displays. For example, if the contents of a view are scrolled, the BView must draw any new images that scrolling makes visible. If one of its children moves, it must fill in the area the child view vacated.
- The application forces an update by "invalidating" a view, or a portion of a view.

Update messages take precedence over other kinds of messages. To keep the on-screen display as closely synchronized with event handling as possible, the window acts on update messages as soon as they arrive. They don't need to wait their turn in the message queue.

(Update messages do their work quietly and behind the scenes. You won't find them in the BWindow's message queue, they aren't handled by BWindow's **DispatchMessage()** function, and they aren't returned by BLooper's **CurrentMessage()**.)

Forcing an Update

When a user action or a BView function alters a view in a window—for example, when a view is resized or it's contents are scrolled—the Application Server knows about it. It makes sure that an update message is sent to the window so the view can be redrawn.

However, if code that's specific to your application alters a view, you'll need to inform the Server that the view needs updating. This is done by calling the **Invalidate()** function. For example, if you write a function that changes the number of elements a view displays, you might invalidate the view after making the change, as follows:

```
void MyView::SetNumElements(long count)
{
    if ( numElements == count )
        return;
    numElements = count;
    Invalidate();
}
```

Invalidate() ensures that the view's **Draw()** function—which presumably looks at the new value of the **numElements** data member—will be called automatically.

At times, the update mechanism may be too slow for your application. Update messages arrive just like other messages sent to a window thread, including messages that report events. Although they take precedence over other messages, update messages must wait their turn. The window thread can respond to only one message at a time; it will get the update message only after it finishes with the current one.

Therefore, if your application alters a view and calls **Invalidate()** while responding to an event message, the view won't be updated until the response to the event is finished and the window thread is free to turn to the next message. Usually, this is soon enough. But if it's not, if the response to the event message includes some time-consuming operations, the application can request an immediate update by calling BWindow's **UpdateIfNeeded()** function.

Erasing the Clipping Region

Just before sending an update message, the Application Server prepares the clipping region of each BView that is about to draw by erasing it to the view color. Note that only the clipping region is erased, not the entire view, and perhaps not the entire area where the BView will, in fact, draw.

Drawing during an Update

While drawing, a BView may set and reset its graphics parameters any number of times—for example, the pen position and front color might be repeatedly reset so that whatever is drawn next is in the right place and has the right color. These settings are temporary. When the update is over, all graphics parameters are reset to their initial values.

If, for example, **Draw()** sets the front color to a shade of light blue, as shown below,

```
    SetFrontColor(152, 203, 255);
```

it doesn't mean that the front color will be blue when **Draw()** is called next. If this line of code is executed during an update, light blue would remain the front color only until the update ends or **SetFrontColor()** is called again, whichever comes first. When the update ends, the previous graphics state, including the previous front color, is restored.

Although you can change most graphics parameters during an update—move the pen around, reset the font, change the front color, and so on—the coordinate system can't be touched; a view can't be scrolled while it's being updated. Since scrolling causes a view to be updated, scrolling during an update would, in effect, be an attempt to nest one update in another, something that can't logically be done (since updates happen sequentially through messages). If the view's coordinate system were to change, it would alter the current clipping region and confuse the update mechanism.

Drawing outside of an Update

Graphics parameters that are set outside the context of an update are not limited; they remain in effect until they're explicitly changed. For example, if application code calls **Draw()**, perhaps in response to an event, the parameter values that **Draw()** last sets would persist even after the function returns. They would become the default values for the view and would be assumed the next time **Draw()** is called.

Default graphics parameters are typically set as part of initializing the BView once it's attached to a window—in an **AttachedToWindow()** function. If you want a **Draw()** function to assume the values set by **AttachedToWindow()**, it's important to restore those values after any drawing the BView does that's not the result of an update. For example, if a BView invokes **SetFrontColor()** while drawing in response to an event message, it will need to restore the default front color when done.

If **Draw()** is called outside of an update, it can't assume that the clipping region will have been erased to the view color, nor can it assume that default graphics parameters will be restored when it's finished.

Picking Pixels to Stroke and Fill

This section discusses how the various BView **Stroke...()** and **Fill...()** functions pick specific pixels to color. Pixels are chosen after the pen size and all coordinate values have been translated to device-specific units. The device-specific value measures distances by counting pixels; one unit equals one pixel on the device.

A device-specific value can be derived from a coordinate value using a formula like this, which takes the size of a coordinate unit and the resolution of the device into account:

```
    device_value = coordinate_value × ( dpi / 72 )
```

dpi is the resolution of the device in dots (pixels) per inch, 72 is the number of coordinate units in an inch, and *device_value* is rounded to the closest integer.

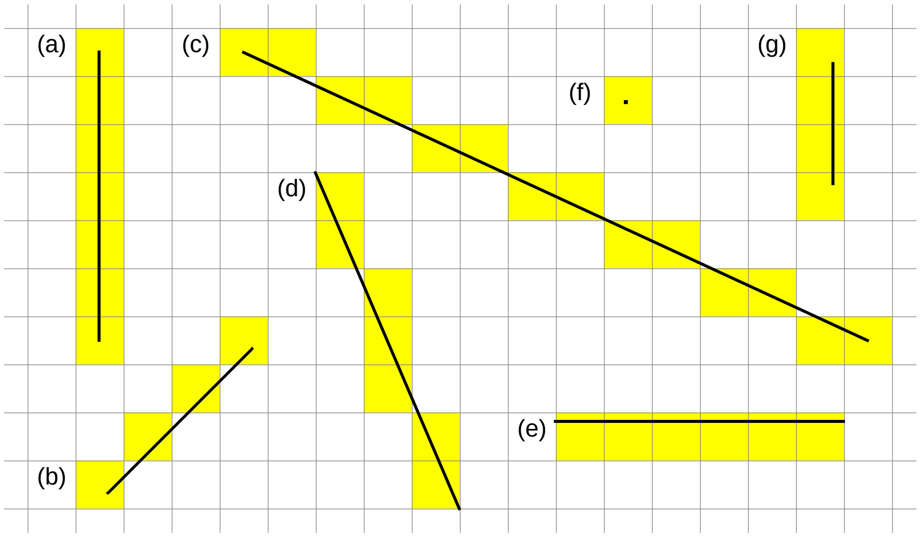
To describe where lines and shapes fall on the pixel grid, this section mostly talks about pixel units rather than coordinate units. The accompanying illustrations magnify the grid so that pixel boundaries are clear. As a consequence, they can show only very short lines and small shapes; they therefore exaggerate the phenomena they illustrate.

Stroking Thin Lines

The thinnest possible line is drawn when the pen size translates to 1 pixel on the device. Setting the size to 0.0 coordinate units guarantees that the pen will be a one-pixel square on all devices.

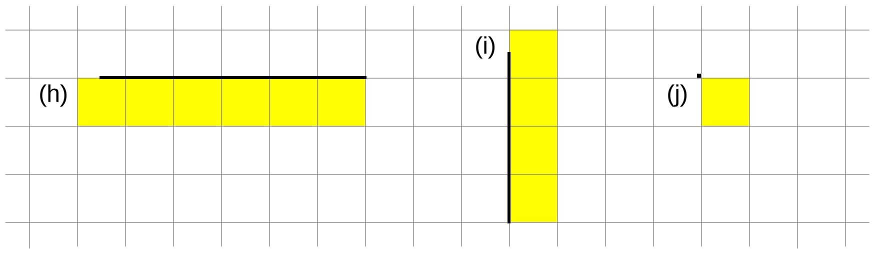
A one-pixel pen follows the path of the line it strokes and makes the line exactly one pixel thick. If the line is more vertical than horizontal, only one pixel in each row is used to render the line. If the line is more horizontal than vertical, only one pixel in each column is used.

Some illustrations of one-pixel thick lines are given below. The broken lines show the separation of the display surface into pixels:



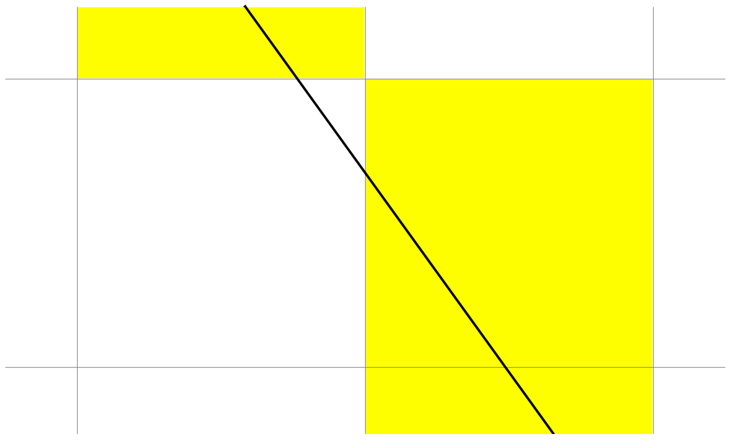
The first thing to notice about this illustration is that only pixels that the line path actually passes through are colored to display the line. If a path begins or ends on a pixel boundary, as it does for lines (d) and (e), for example, the pixels at the boundary aren't colored unless the path crosses into the pixel. The pen touches the fewest possible number of pixels.

It's possible for a line path not to enter any pixels, but to lie entirely on the boundaries between pixels. Such a line is not invisible. A horizontal path between pixels colors the pixel row beneath it. A vertical path between pixels colors the pixel column to its right. A line path that reduces to a single point lying on the corner of four pixels colors the pixel at its lower right. The orientation of the pen is always toward the bottom and the right.

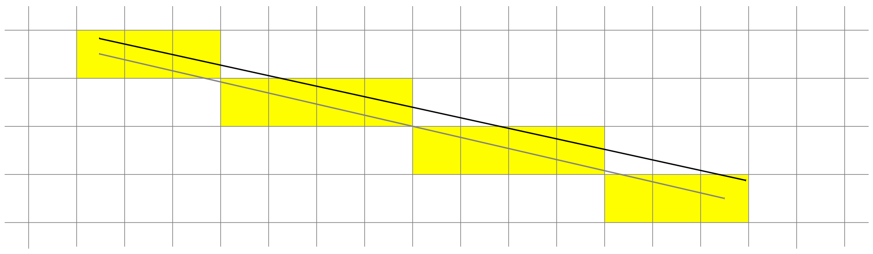


< However, for the current release, it's indeterminate which column or row of adjacent pixels would be used to display vertical and horizontal lines like (h) and (i) above. Point (j) would not be visible. >

Although a one-pixel pen touches only pixels that lie on the path it strokes, it won't touch every pixel that the path crosses if that would mean making the line thicker than specified. When the path cuts though two pixels in a column or row, but only one of those pixels can be colored, the one that contains more of the path (the one that contains the midpoint of the segment cut by the column or row) is chosen. This is illustrated in the close-up below, which shows where a mostly vertical line crosses one row of pixels:



However, before a choice is made as to which pixel in a row or column to color, the line path is normalized for the device. For example, if a line is defined by two endpoints, it's first determined which pixels correspond to those endpoints. The line path is then treated as if it connected the centers of those pixels. This may alter which pixels get colored, as is illustrated below. In this illustration, the solid black line is the line path as originally specified and the broken line is its normalized version:



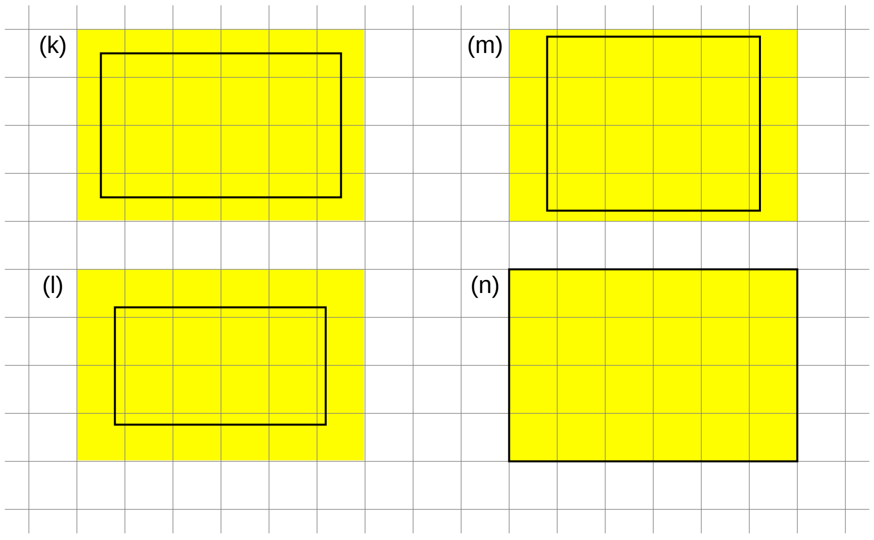
This normalization is nothing more than the natural consequence of the rounding that occurs when coordinate values are translated to device-specific pixel values.

Stroking Curved Lines

Although all the diagrams above show straight lines, the principles they illustrate apply equally to curved line paths. A curved path can be treated as if it were made up of a large number of short straight segments.

Filling and Stroking Rectangles

The following illustration shows how some rectangles, represented by the solid black line, would be filled with a solid color.

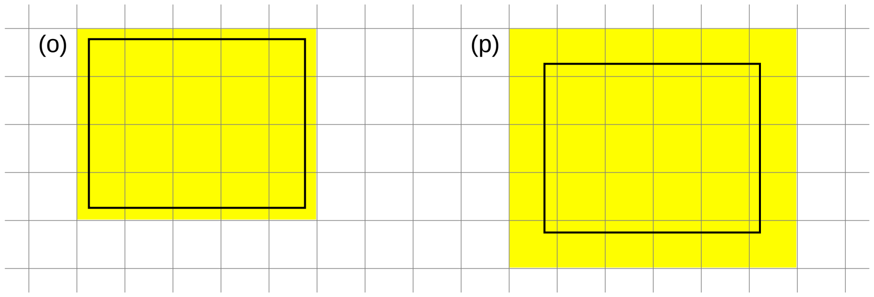


A rectangle includes every pixel that it encloses and every pixel that its sides pass through. However, as rectangle (n) illustrates, it doesn't include pixels that its sides merely touch at the boundary.

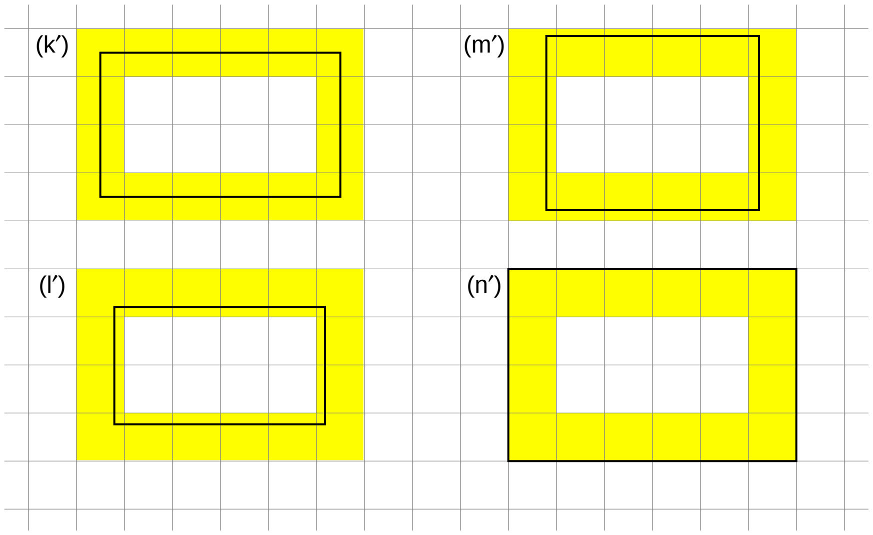
If the pixel grid in this illustration represented the screen, rectangle (n) would have left, top, right, and bottom coordinates with fractional values of .5. Rectangle (k), on the other hand, would have coordinates without any fractional parts. Nonfractional coordinates lie at the center of screen pixels.

Rectangle (k), in fact, is the normalized version of all four of the illustrated rectangles. It shows how the sides of the four rectangles would be translated to pixel values. Note that for a rectangle like (n), with edges that fall on pixel boundaries, normalization means rounding the left and top sides upward and rounding the right and bottom sides downward. This follows from the principal that the fewest possible number of pixels should be colored.

Although the four rectangles above differ in size and shape, when filled they all cover a 6 × 4 pixel area. You can't predict this area from the dimensions of the rectangle. Because the coordinate space is continuous and *x* and *y* values can be located anywhere, rectangles with different dimensions might have the same rendered size, as shown above, and rectangles with the same dimensions might have different rendered sizes, as shown below:



If a one-pixel pen strokes a rectangular path, it touches only pixels that would be included if the rectangle were filled. The illustration below shows the same rectangles that were presented above, but strokes them rather than fills them:

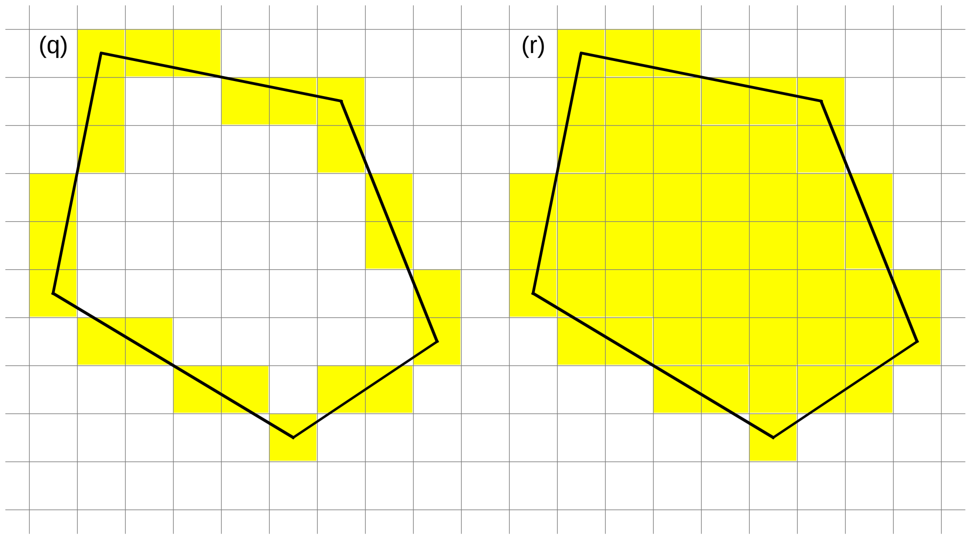


Each of the rectangles still covers a 6 × 4 pixel area. Note that even though the path of rectangle (n′) lies entirely on pixel boundaries, pixels below it and to its right are not touched by the pen. The pen touches only pixels that lie within the rectangle.

If a rectangle collapses to a straight line or to a single point, it no longer contains any area. Filling such a rectangle is equivalent to stroking the line path with a one-pixel pen, as was discussed in the previous section. Stroking such a rectangle is equivalent to stroking the line.

Filling and Stroking Polygons

The figure below shows a polygon as it would be stroked by a one-pixel pen and as it would be filled:



The same rules apply when stroking each segment of a polygon as would apply if that segment were an independent line. Therefore, the pen may not touch every pixel the segment passes through.

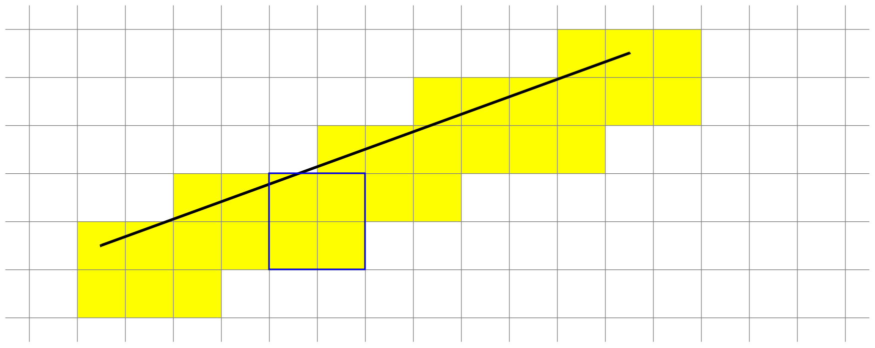
When the polygon is filled, no additional pixels around its border are colored. As is the case for a rectangle, the displayed shape of filled polygon is identical to the shape of the polygon when stroked with a one-pixel pen. The pen doesn't touch any pixels when stroking the polygon that aren't colored when the polygon is filled. Conversely, filling doesn't color any pixels at the border of the polygon that aren't touched by a one-pixel pen.

Stroking Thick Lines

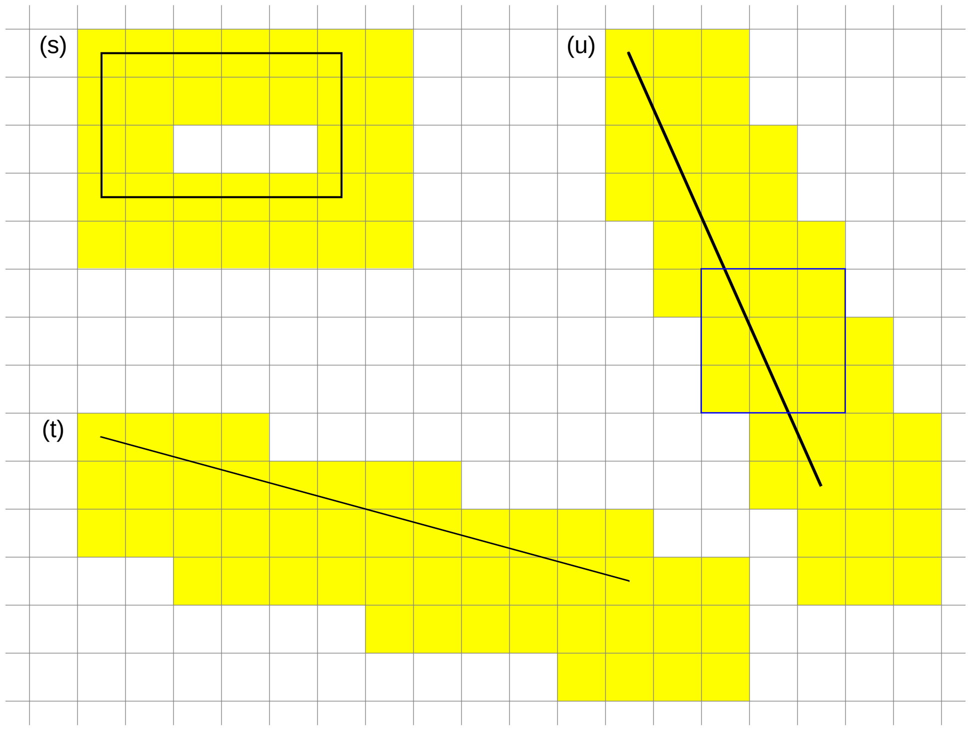
A pen that's thicker than one pixel touches the same pixels that a one-pixel pen does, but it adds extra columns to the right of the path and extra rows beneath it.

It’s as if the pen is a square that moves along the line path and stops on selected pixels. When it stops, it colors every pixel that it covers. If the pen is a one-pixel square, it covers only pixels that lie on the path. If it’s a multiple-pixel square, its left top corner follows the path so that it also covers pixels below the path and to the right.

The following diagram outlines the position of a two-pixel pen at one stop along a line path:



In the following example, rectangle (s) is stroked by a two-pixel pen and lines (t) and (u) by a three-pixel pen. One of the positions of the pen on line (u) is outlined.



No matter what its size, the pen always stops at the same pixels on the line path.

Handling Events

The BWindow and BView classes together define a structure for responding to user actions on the keyboard and mouse. These actions generate interface events that are reported to a BWindow object and that the BWindow distributes to other objects, typically BViews.

This section describes interface events, the messages that report them, and the way that BWindow and BView objects are structured to respond to them.

Interface Events

In most cases, an interface event merely reports what the user did. However, in some cases, it may reflect the way the Application Server interpreted or handled a user action. The Server might respond directly to the user action and pass along an event that reflects what it did—moved a window or changed a value, for example. In a few cases, the event may even reflect what the application thinks the user intended—that is, an application might interpret one or more generic user actions as a more specific event.

Seventeen interface events are currently defined. The following five capture atomic user actions on the keyboard and mouse:

- A *key-down* event occurs when the user presses a character key on the keyboard. After the initial event (and a brief threshold), most keys generate repeated key-down events—as long as the user continues to hold the key down and doesn't press another key. Only character keys produce keyboard events. The modifier keys—Shift, Control, Caps Lock, and so on—don't generate events of any kind but may affect the character that's reported for another key.

- A *key-up* event occurs when the user releases the character key. < This event isn't implemented for the current release. >

- A *mouse-down* event occurs when the user presses one of the mouse buttons while the cursor is over the content area of a window. < The event is generated only for the first button the user presses—that is, only if no other mouse buttons are down at the time. >

- A *mouse-up* event occurs when the user releases the mouse button. < The event is generated only for the last button the user releases—that is, only if no other mouse button remains down. >

- A *mouse-moved* event captures some small portion of the cursor's movement into, within, or out of a window. If the cursor isn't over a window, it's movement doesn't generate mouse-moved events. (All interface events are associated with windows.) Repeated mouse-moved events are generated as the user moves the mouse.

A closely related event announces the arrival of a package of information:

- A *message-dropped* event occurs when the user releases the mouse button after dragging an image from one view to another. The image represents information bundled in a BMessage object. The message is “dropped” on the view where the cursor is located when the mouse button goes up.

The six events above are all directed at particular views. Three others also concern views:

- A *view-moved* event occurs when a view is moved within its parent’s coordinate system. This can be a consequence of a programmatic action or of the parent view being automatically resized. If the parent view is being continuously resized because the user is resizing the window, repeated mouse-moved events may be recorded.

- A *view-resized* event occurs when a view is resized, perhaps because the program resized it or possibly as an automatic consequence of the window being resized. If the resizing is continuous, because the user is resizing the window, repeated view-resized events are reported.

- A *value-changed* event occurs when the Application Server changes a value associated with an object. Currently, the event is generated only for BScrollBar objects. It’s generated repeatedly as the user manipulates a scroll bar.

A few events affect the window itself:

- An *activation* event happens when a window becomes the active window, and when it gives up that status. The single action of clicking a window to make it active might result in two activation events—one for the window that gains active-window status and one for the window that relinquishes it—plus a mouse-down and a mouse-up event.

- A *quit-requested* event occurs when the user clicks a window’s close box, or when the system perceives some other reason to request the window to quit.

- A *window-moved* event records the new location of a window that has been moved, either programmatically or by the user. When the user drags a window, repeated events occur, each one capturing a small portion of the window’s continuous movement. Only one event occurs when the program moves a window.

- A *window-resized* event occurs when the window is resized, again either programmatically or by the user. The event is generated repeatedly as the user resizes the window, but only once each time the application resizes it.

- A *screen-changed* event occurs when the configuration of the screen—the size of the pixel grid it displays < or the color space of the frame buffer >—changes. Such changes may require the window to take compensatory measures.

Two events are produced by the save panel:

- A *save-requested* event occurs when the user operates the panel to request that a document be saved.

- A *panel-closed* event occurs when the application or the user closes the panel.

Finally, there's one event that doesn't derive from a user action:

- Periodic *pulse* events occur at regularly spaced intervals, like a steady heartbeat. Pulses don't involve any communication between the application and the Server. They're generated as long as no other events are pending, but only if the application asks for them.

An application doesn't have to wait for an event to discover what the user is doing on the keyboard and mouse. Two BView functions, **GetKeys()** and **GetMouse()**, can provide an immediate check on the state of these devices.

Hook Functions for Interface Events

Events are reported to an application as they occur. The Application Server determines which window an event affects and notifies the appropriate window thread. Keyboard events are reported to the current active window, mouse events to the window where the cursor is located.

Reports of events are delivered as BMessage objects. When a message arrives, the BWindow dispatches it to initiate action within the window thread. Typically, one of the BViews associated with the window is asked to respond to the message—usually the BView that drew the image that elicited the user action. But some messages are handled by the BWindow itself.

Interface events are dispatched by calling a virtual function that's matched to the event. For example, the BView where a mouse-down event occurs is notified with a **MouseDown()** function call. When the user clicks the close box of a window, generating a quit-requested event, the BWindow's **QuitRequested()** function is called.

The chart below lists the virtual functions that are called to initiate the application's response to interface events, and the base classes where the functions are declared. Each application can implement these event-specific functions in a way that's appropriate to its purposes.

Event type	Virtual function	Class
Key-down	**KeyDown()**	BView
Key-up	*none*	
Mouse-down	**MouseDown()**	BView

Mouse-up	*none*	
Mouse-moved	**MouseMoved()**	BView
Message-dropped	**MessageDropped()**	BView
View-moved	**FrameMoved()**	BView
View-resized	**FrameResized()**	BView
Value-changed	**ValueChanged()**	BScrollBar
Window-activated	**WindowActivated()**	BWindow and BView
Quit-requested	**QuitRequested()**	BLooper, inherited by BWindow
Window-moved	**FrameMoved()**	BWindow
Window-resized	**FrameResized()**	BWindow
Screen-changed	**ScreenChanged()**	BWindow
Save-requested	**SaveRequested()**	BWindow
Panel-closed()	**SavePanelClosed()**	BWindow
Pulse	**Pulse()**	BView

< Key-up events are currently not reported. > Mouse-up events are reported to the application, but they aren't dispatched by calling a virtual function. A BView can determine when a mouse button goes up by calling **GetMouse()** from within its **MouseDown()** function. As it reports information about the location of the cursor and the state of the mouse buttons, **GetMouse()** removes mouse-moved and mouse-up messages from the BWindow's message queue, so the same information won't be reported twice.

Dispatching

Notice, from the chart above, that the BWindow class declares the functions that handle events directed at the window itself. **FrameMoved()** is called when the user moves the window, **FrameResized()** when the user resizes it, **WindowActivated()** when it becomes, or ceases to be, the active window, and so on.

Although the BWindow handles some interface events, most are handled by BViews. When the BWindow receives an event message, it must decide which view is responsible.

This decision is relatively easy for mouse events. The cursor points to the affected view. For example, when the user presses a mouse button, the BWindow calls the **MouseDown()** virtual function of the view under the cursor. When the user moves the mouse, it calls the **MouseMoved()** function of each view the cursor travels through. When the user drags a message to a window and drops it there, it calls the **MessageDropped()** function of the view the cursor points to.

However, there's no cursor attached to the keyboard, so the BWindow object must keep track of the view that's responsible for key-down events. That view is known as the *focus view.*

The Focus View

The focus view is whatever view happens to be displaying the current selection (possibly an insertion point) within the window, or whatever check box or other gadget is marked to show that it can be operated from the keyboard.

The focus view is expected to handle keyboard events when the window is the active window and to handle data pasted from the clipboard. When the user presses a key on the keyboard, the BWindow calls the focus view's **KeyDown()** virtual function. When the user pastes material from the clipboard, the application should arrange for the focus view to respond.

The focus doesn't have to stay on one view all the time; it can shift from view to view. It may change as the user changes the current selection in the window—from text field to text field, for example. Only one view in the window can be in focus at a time.

Views put themselves in focus when they're selected by a user action of some kind. For example, when a BView's **MouseDown()** or **MessageDropped()** function is called, notifying it that the user has selected the view, it can grab the focus by calling **MakeFocus()**. When a BView makes itself the focus view, the previous focus view is notified that it has lost that status.

A view should become the focus view if:

- It has a **KeyDown()** function so that the user can operate it from the keyboard,
- It has a **KeyDown()** function to display typed characters, or
- It can show the current selection, whether or not it displays what the user types.

A view should highlight the current selection only while it's in focus.

BViews make themselves the focus view (with the **MakeFocus()** function), but BWindows report which view is currently in focus (with the **CurrentFocus()** function).

Filtering Events

A BWindow can scrutinize mouse and keyboard events before it gives the target BView a chance to respond. The BWindow class declares four hook functions that preview events before a BView is notified:

> **FilterKeyDown()**,
> **FilterMouseDown()**,
> **FilterMouseMoved()**, and
> **FilterMessageDropped()**

These functions give BWindows an opportunity to modify aspect of the event or even change the BView that will be expected to respond. Unless the BWindow completely intercepts the event, the responsible BView is notified through its **KeyDown()**, **MouseDown()**, **MouseMoved()**, or **MessageDropped()** function.

The filter functions are rarely implemented to prevent the BView functions from being called. Since the response to an event depends on what prompted it—for example, a click would mean one thing to a button and quite another to a text field—the principal event-handling code must be located within BViews, not at the BWindow level.

Message Protocols

As noted above, reports of events are delivered to the window thread as BMessage objects. The object's **what** data member is always a constant that names what kind of event it is. The constants for interface events are:

KEY_DOWN	**WINDOW_ACTIVATED**
KEY_UP	**QUIT_REQUESTED**
MOUSE_DOWN	**WINDOW_MOVED**
MOUSE_UP	**WINDOW_RESIZED**
MOUSE_MOVED	**SCREEN_CHANGED**
MESSAGE_DROPPED	**SAVE_REQUESTED**
	PANEL_CLOSED
VIEW_MOVED	
VIEW_RESIZED	**PULSE**
VALUE_CHANGED	

Typically, the BMessage object also carries various kinds of data describing the event. In some cases, it may contain more information about the event than is passed to the function that starts the application's response. For example, a **MouseDown()** function is passed the point where the cursor was located when the user pressed the mouse button. But a **MOUSE_DOWN** BMessage also includes information about when the event occurred, what modifier keys the user was holding down at the time, < which mouse button was pressed, whether the event counts as a solitary mouse-down, the second of a double-click, or the third of a triple-click, and so on. >

A **MouseDown()** function can get this information by taking it directly from the BMessage. The BMessage that the window thread is currently responding to can be obtained by calling **CurrentMessage()**, which the BWindow inherits from BLooper. For example, a **MouseDown()** function might get the time of the mouse-down event as follows:

```
void MyView::MouseDown(BPoint point)
{
     . . .
   long time = Window()->CurrentMessage()->FindLong("when");
     . . .
}
```

With the exception of **SAVE_REQUESTED** and **PANEL_CLOSED** messages, all BMessages that report interface events record when the event occurred. This information is placed in the BMessage under the name "when" as a **long** integer.

For pulse and quit-requested events, "when" is the only data the BMessage contains. There's nothing else to know about the event. However, for most interface events, the BMessage carries additional information describing the content of the event—which key was pressed, where the cursor was pointing, what the new value is, and so on.

The following sections list the data that's available within the BMessage objects that report interface events:

Key-Down Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the key went down, as measured in milliseconds from the time the machine was last booted.
"key"	**LONG_TYPE**	The code for the key that was pressed.
"modifiers"	**LONG_TYPE**	A mask that identifies which modifier keys the user was holding down and which keyboard locks were on at the time of the event.
"char"	**LONG_TYPE**	The character that's generated by the combination of the key and modifiers.
"states"	**UCHAR_TYPE**	A bit field that records the state of all keys and keyboard locks at the time of the event. Although declared as **UCHAR_TYPE**, this is actually an array of 16 bytes.

For most applications, the "char" code is sufficient to distinguish one sort of user action on the keyboard from another. It reflects both the key that was pressed and the effect that the modifiers have on the resulting character. For example, if the Shift key is down when the user presses the *A* key, or if Caps Lock is on, the "char" produced will be uppercase 'A' rather than lowercase 'a'. If the Control key is down, it will be the **HOME** character. A later section, "Keyboard Information" on page 53, discusses the mapping of keys to characters in more detail.

The "modifiers" mask explicitly identifies which modifier keys the user is holding down and which keyboard locks are on at the time of the event. It's described under "Modifier Keys" on page 57 below.

The "key" code is an arbitrarily assigned number that identifies which character key the user pressed. All keys on the keyboard, including modifier keys, have key codes (but only character keys produce key-down events). The codes for the keys on a standard keyboard are shown in the "Key Codes" section on page 53.

The "states" bit field has one bit assigned to each key. For most keys, the bit is set to 1 if the key is down, and to 0 if the key is up. However, the bits corresponding to keys that toggle keyboard locks (the Caps Lock, Num Lock, and Scroll Lock keys) are set to 1 if the lock is on, and to 0 if the lock is off. See "Key States" on page 61 for details on how to read information from the "states" array.

Key-Up Events

< Key-up events are not currently reported. >

Mouse-Down Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the mouse button went down, as measured in milliseconds from the time the machine was last booted.
"where"	**POINT_TYPE**	Where the cursor was located when the user pressed the mouse button, expressed in the coordinate system of the target BView—the view where the cursor was located at the time of the event.
"modifiers"	**LONG_TYPE**	A mask that identifies which modifier keys were down and which keyboard locks were on when the user pressed the mouse button.

The "modifiers" mask is the same as for key-down events and is described under "Modifier Keys" on page 57.

Mouse-Up Events

Data name	Type code	Description
“when”	**LONG_TYPE**	When the mouse button went up again, as measured in milliseconds from the time the machine was last booted.
“where”	**POINT_TYPE**	Where the cursor was located when the user released the mouse button, expressed in the coordinate system of the target BView—the view where the cursor was located when the button went up.
“modifiers”	**LONG_TYPE**	A mask that identifies which of the modifier keys were down and which keyboard locks were in effect when the user released the mouse button.

The “modifiers” mask is the same as for key-down events and is described under “Modifier Keys” on page 57.

Mouse-Moved Events

Data name	Type code	Description
“when”	**LONG_TYPE**	When the event occurred, as measured in milliseconds from the time the machine was last booted.
“where”	**POINT_TYPE**	The new location of the cursor, where it has moved to, expressed in window coordinates.
“area”	**LONG_TYPE**	The area of the window where the cursor is now located.
“buttons”	**LONG_TYPE**	Which mouse buttons, if any, are down.
“dragging”	**OBJECT_TYPE**	A pointer to a BMessage object that the user is dragging, or **NULL** if nothing is being dragged.

The “area” constant records which part of the window the cursor is over. It can be:

CONTENT_AREA	The cursor is over the content area of the window.
CLOSE_BOX	The cursor is over the close box in the title bar.
TITLE_BAR	The cursor is inside the title tab, but not over the close box.
RESIZE_AREA	The cursor is over the area where the window can be resized.
UNKNOWN_AREA	It’s not known where the cursor is.

If the location of the cursor is unknown, it's probably because it just left the window.

< Currently, no distinction is made between the different buttons on a mouse. The "buttons" variable will be 0 when the user moves the mouse without holding down any of its buttons, and something other than 0 when a button is held down. >

Message-Dropped Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the message was dropped, as measured in milliseconds from the time the machine was last booted.
"where"	**POINT_TYPE**	Where the cursor was located when the user released the mouse button to drop the dragged message. The point is expressed in window coordinates.

A **MESSAGE_DROPPED** BMessage simply informs the window that another BMessage has been dragged to it and dropped on one of its views. The dropped BMessage is passed to the BView as an argument in a **MessageDropped()** function call; it's not recorded as part of the message-dropped event.

View-Moved Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the view moved, as measured in milliseconds from the time the machine was last booted.
"where"	**POINT_TYPE**	The new location of the left top corner of the view's frame rectangle, expressed in the coordinate system of its parent.

View-Resized Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the view was resized, as measured in milliseconds from the time the machine was last booted.
"width"	**LONG_TYPE**	The new width of the view's frame rectangle.
"height"	**LONG_TYPE**	The new height of the view's frame rectangle.

"where"	**POINT_TYPE**	The new location of the left top corner of the view's frame rectangle, expressed in the coordinate system of its parent. (A "where" entry is present only if the view was moved while being resized.)

A **VIEW_RESIZED** BMessage has a "where" entry only if resizing the view also served to move it. The new location of the view would first be reported in a **VIEW_MOVED** BMessage.

Value-Changed Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the value changed, as measured in milliseconds from the time the machine was last booted.
"value"	**LONG_TYPE**	The new value of the object.

Window-Activated Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the window's status changed, as measured in milliseconds from the time the machine was last booted.
"active"	**BOOL_TYPE**	A flag that records the new status of the window. It's **TRUE** if the window has become the active window, and **FALSE** if it is giving up that status.

Quit-Requested Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the event occurred, as measured in milliseconds from the time the machine was last booted.

This data entry is added by the Application Server whenever it posts a **QUIT_REQUESTED** message—for example, when the user clicks the window's close box. However, it's not crucial to the interpretation of the event. You don't need to add it to messages that are posted in application code.

Window-Moved Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the window moved, as measured in milliseconds from the time the machine was last booted.
"where"	**POINT_TYPE**	The new location of the left top corner of the window's content area, expressed in screen coordinates.

Window-Resized Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the window was resized, as measured in milliseconds from the time the machine was last booted.
"width"	**LONG_TYPE**	The new width of the window's content area.
"height"	**LONG_TYPE**	The new height of the window's content area.

Screen-Changed Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the screen changed, as measured in milliseconds from the time the machine was last booted.
"frame"	**RECT_TYPE**	A rectangle with the same dimensions as the pixel grid the screen displays.
"mode"	**LONG_TYPE**	The color space of the screen. < This will always be **COLOR_8_BIT**. >

Save-Requested Events

Data name	Type code	Description
"directory"	**REF_TYPE**	A **record_ref** reference to the directory where the document should be saved.
"name"	**STRING_TYPE**	The name of the file in which the document should be saved.

These entries are added to all messages reporting save-requested events. Generally, the message has **SAVE_REQUESTED** as its **what** data member. However, you can define a custom message to report the event, one with another constant and additional data entries. See **RunSavePanel()** in the BWindow class.

Panel-Closed Events

Data name	Type code	Description
"frame"	**RECT_TYPE**	The frame rectangle of the save panel in screen coordinates at the time the panel was closed. (The user may have resized it and relocated it on-screen before it was closed.)
"directory"	**REF_TYPE**	A **record_ref** reference to the last directory displayed in the panel.
"canceled"	< **LONG_TYPE** >	An indication of whether or not the panel was closed by user. It's **TRUE** if the user closed the panel by operating the "Cancel" button < and **FALSE** otherwise. > < Currently, this entry is present only if the user canceled the panel. >

Pulse Events

Data name	Type code	Description
"when"	**LONG_TYPE**	When the event occurred, as measured in milliseconds from the time the machine was last booted.

Keyboard Information

Most information about what the user is doing on the keyboard comes to applications by way of key-down events. The application can usually determine what the user's intent was in pressing a key by looking at the character reported in the event. But, as discussed under "Key-Down Events" on page 46 above, the event carries other keyboard information in addition to the character—the key the user pressed, the modifier states that were in effect at the time, and the current state of all keys on the keyboard.

Some of this information can be obtained in the absence of key-down events:

- The BWindow, BView, and BApplication classes have **Modifiers()** functions that return the current modifier states, and
- The BView class has a **GetKeys()** function that can provide the current state of all the keys and modifiers on the keyboard.

This section discusses in detail the kinds of information that you can get about the keyboard through key-down events and these functions.

Key Codes

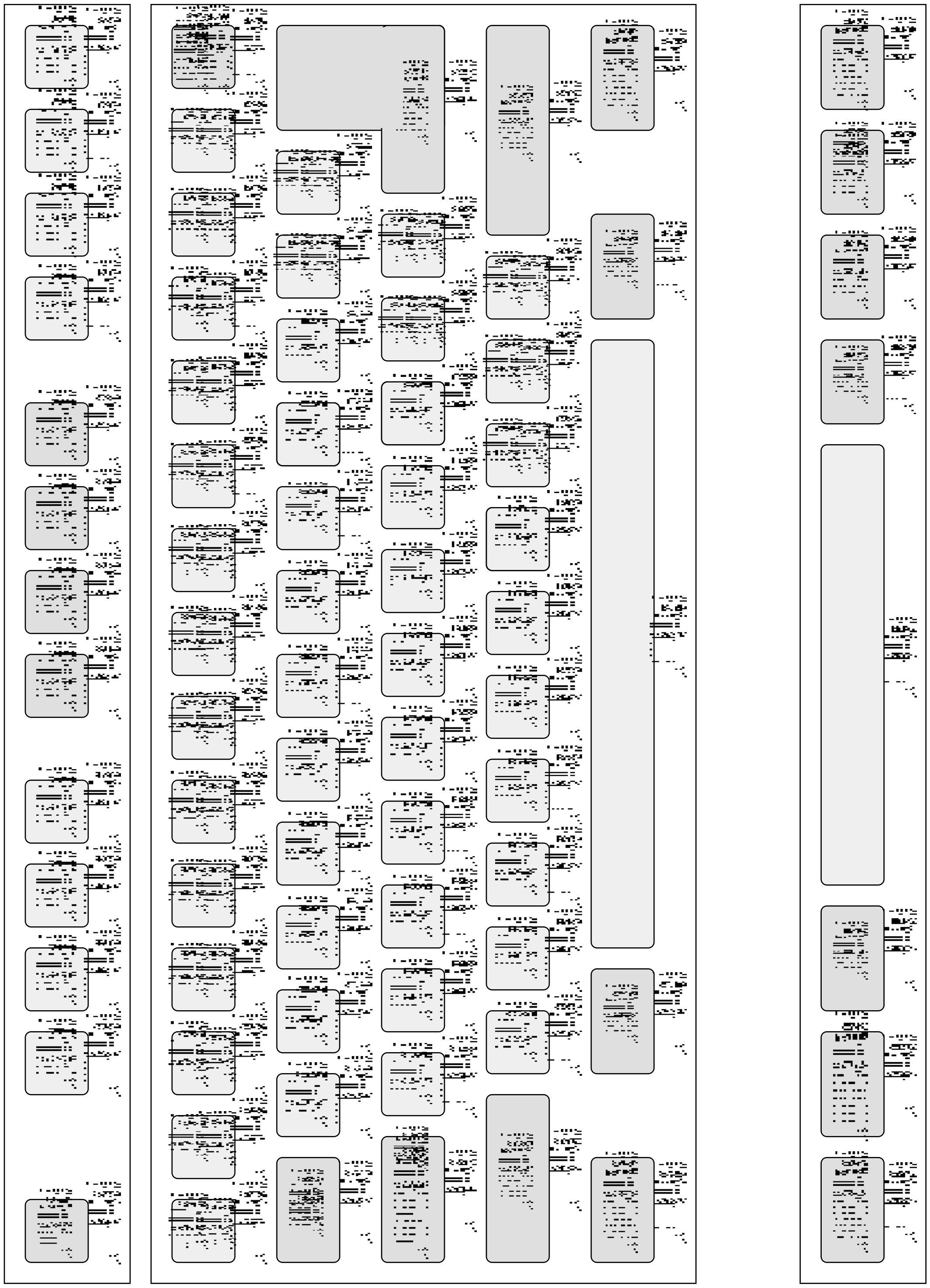
To talk about the keys on the keyboard, it's necessary first to have a standard way of identifying them. For this purpose, each key is arbitrarily assigned a numerical code.

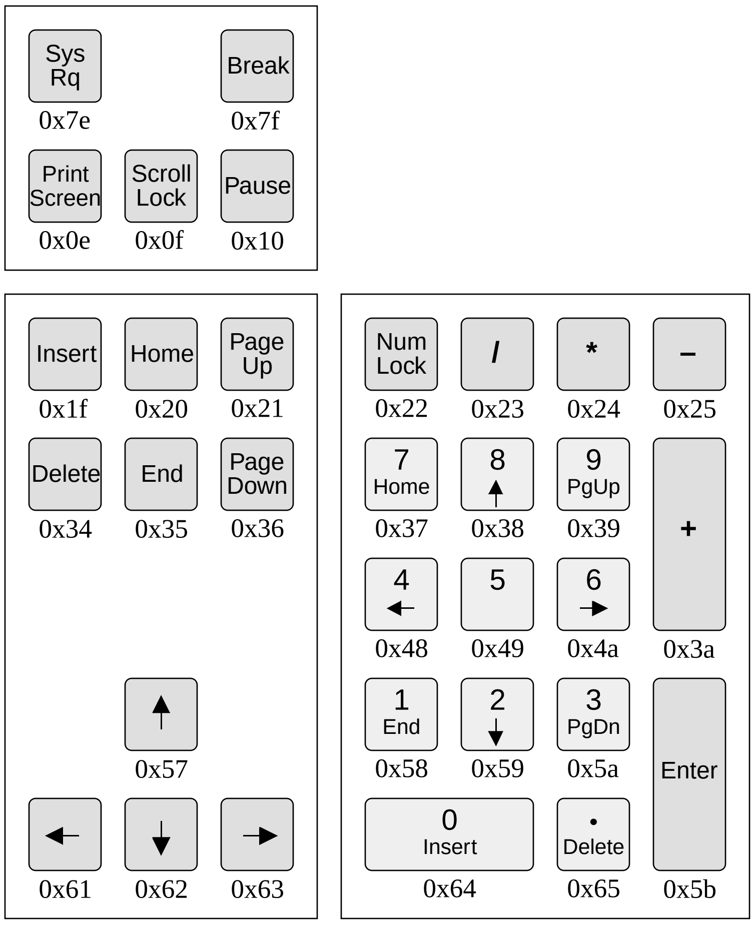
The illustrations on the next two pages show the key identifiers for a typical keyboard. The codes for the main keyboard are shown on page 54. This diagram shows a standard 101-key keyboard and an alternate version of the bottom row of keys—one that adds a Menu key and left and right Command keys.

The codes for the numerical keypad and for the keys between it and the main keyboard are shown on page 55.

Different keyboards locate keys in slightly different positions. The function keys may be to the left of the main keyboard, for example, rather than along the top. The backslash key (0x33) shows up in various places—sometimes above the Enter key, sometimes next to Shift, and sometimes in the top row (as shown here). No matter where these keys are located, they have the codes indicated in the illustrations.

The BMessage that reports a key-down event contains an entry named "key" for the code of the key that was pressed.





Kinds of Keys

Keys on the keyboard can be distinguished by the way they behave and by the kinds of information they provide. A principal distinction is between *character keys* and *modifier keys*:

- *Character keys* are mapped to particular characters; they generate key-down events when pressed. Keys not mapped to characters don't generate events.
- *Modifier keys* set states that can be discerned independently of key-down events (through the various **Modifiers()** functions). Some modifier keys—like Caps Lock and Num Lock—toggle in and out of a locked modifier state. Others—like Shift and Control—set the state only while the key is being held down.

If a key doesn't fall into one of these categories or the other, there's nothing for it to do; it has no role to play in the interface. For most keys, the categories are mutually exclusive. Modifier keys are typically not mapped to characters, and character keys don't set modifier states. However, the Scroll Lock key is an exception. It both sets a modifier state and generates a character.

Keys can be distinguished on two other grounds as well:

- *Repeating keys* produce a continuous series of key-down events, as long as the user holds the key down and doesn't press another key. After the initial event, there's a slight delay before the key begins repeating, but then events are generated in rapid succession.

  All keys are repeating keys except for Pause, Break, and the three that set locks (Caps Lock, Num Lock, and Scroll Lock). Even modifier keys like Shift and Control would repeat if they were mapped to characters (but, since they're not, they don't produce any key-down events at all).

- *Dead keys* are keys that don't produce characters until the user strikes another key (or the key repeats). If the key the user strikes after the dead key belongs to a particular set, the two keys together produce one character (one key-down event). If not, each produces a separate character. The key-down event for the dead key is delayed until it can be determined whether it will be combined with another key to produce just one event.

  Dead keys are dead only when the Option key is held down. They're most appropriate for situations where the user can imagine a character being composed of two distinguishable parts—such as 'a' and 'e' combining to form 'æ'.

  The system permits up to five dead keys. By default, they're reserved for combining diacritical marks with other characters. The diacritical marks are the acute (´) and grave (`) accents, dieresis (¨), circumflex (ˆ), and tilde (˜).

There's a system key map that determines the role that each key plays—whether it's a character key or a modifier key, which modifier states it sets, which characters it produces, whether it's dead or not, how it combines with other keys, and so on. The map is shared by all applications.

Users can modify the key map with the Keyboard utility. Applications can look at it (and perhaps modify it) by calling the **system_key_map()** global function. See that function on page 277 for details on the structure of the map. The discussion here assumes the default key map that comes with the computer.

Modifier Keys

The role of a modifier key is to set a temporary, modal state. There are eight modifier states—eight different kinds of modifier key—defined functionally. Three of them affect the character that's reported in a key-down event:

- The *Shift key* maps alphabetic keys to the uppercase version of the character, and other keys to alternative symbols.
- The *Control key* maps alphabetic keys to Control characters—those with ASCII values (character codes) below 0x20.
- The *Option key* maps keys to alternative characters, typically characters in an extended set—those with ASCII values above 0x7f.

Two modifier keys permit users to give the application instructions from the keyboard:

- When the *Command key* is held down, the character keys perform keyboard shortcuts.
- The *Menu key* initiates keyboard navigation of menus. Pressing and releasing a Command key (without touching another key) accomplishes the same thing.

Three modifiers toggle in and out of locked states:

- The *Caps Lock* key reverses the effect of the Shift key for alphabetic characters. With Caps Lock on, the uppercase version of the character is produced without the Shift key, and the lowercase version with the Shift key.
- The *Num Lock* key similarly reverses the effect of the Shift key for keys on the numeric keypad.
- The *Scroll Lock* key temporarily prevents the display from updating. (It's up to applications to implement this behavior.)

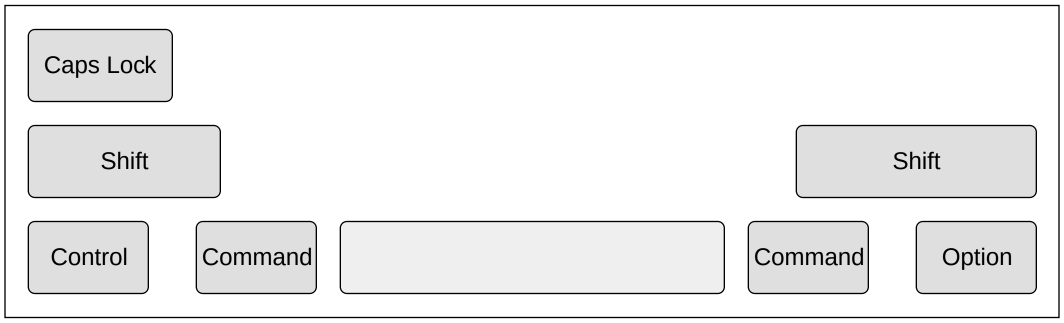
There are two things to note about these eight modifier states. First, since applications can read the modifiers directly from the messages that report key-down events and obtain them at other times by calling the **Modifiers()** and **GetKeys()** functions, they are free to interpret the modifier states in any way they desire. They're not tied to the narrow interpretation of, say, the Control key given above. Control, Option, and Shift, for example, often modify the meaning of a mouse event or are used to set other temporary modes of behavior.

Second, the set of modifier states listed above doesn't quite match the keys that are marked on a typical keyboard. A standard 101-key keyboard has left and right "Alt(ernate)" keys, but lacks those labeled "Command," "Option," or "Menu."

The key map must, therefore, bend the standard keyboard to the required modifier states. The default key map does this in three ways:

- Because the "Alt(ernate)" keys are close to the space bar and are easily accessible, the default key map assigns them the role of Command keys.
- It turns the right "Control" key into an Option key. Therefore, there's just one functional Control key (on the left) and one Option key (on the right).
- It leaves the Menu key unmapped. It relies on the Command key as an adequate alternative for initiating keyboard navigation of menus.

The illustration below shows the modifier keys on the main keyboard, with labels that match their functional roles. Users can, of course, remap these keys with the Keyboard utility. Applications can remap them by calling **set_modifier_key()** or **system_key_map()**.



Current modifier states are reported in a mask that can be tested against these constants:

SHIFT_KEY	**COMMAND_KEY**	**CAPS_LOCK**
CONTROL_KEY	**MENU_KEY**	**NUM_LOCK**
OPTION_KEY		**SCROLL_LOCK**

The ...**_KEY** modifiers are set if the user is holding the key down. The ...**_LOCK** modifiers are set only if the lock is on—regardless of whether the key that sets the lock happens to be up or down at the time.

If it's important to know which physical key the user is holding down, the one on the right or the one on the left, the mask can be more specifically tested against these constants:

LEFT_SHIFT_KEY	**RIGHT_SHIFT_KEY**
LEFT_CONTROL_KEY	**RIGHT_CONTROL_KEY**
LEFT_OPTION_KEY	**RIGHT_OPTION_KEY**
LEFT_COMMAND_KEY	**RIGHT_COMMAND_KEY**

If no keyboard locks are on and the user isn't holding a modifier key down, the modifiers mask will be 0.

The modifiers mask is returned by the various **Modifiers()** functions (defined by the BApplication class in the Application Kit and by BWindow and BView in the Interface

Kit). It's returned, along with other information, by BView's **GetKeys()** function. And it's also included as a "modifiers" entry in every BMessage that reports a keyboard or mouse event.

Character Mapping

Most keys are mapped to more than one character. The precise character that the key produces depends on which modifier keys are being held down and which lock states the keyboard is in at the time the key is pressed.

A few examples are given in the table below:

Key Code	Without Modifiers	With Shift	With Option	With Shift & Option	With Control
0x15	'4'	'$'	'¢'		'4'
0x18	'7'	'&'	'¶'	'§'	'7'
0x26	**TAB**	**TAB**	**TAB**	**TAB**	**TAB**
0x2e	'i'	'I'			**TAB**
0x38	**UP_ARROW**	'8'	**UP_ARROW**	'8'	**UP_ARROW**
0x40	'g'	'G'	'©'		0x1a
0x44	'l'	'L'	'æ'	'Æ'	**PAGE_DOWN**
0x51	'n'	'N'	'ñ'	'Ñ'	0x0e
0x55	'/'	'?'	'÷'	'¿'	'/'

The mapping follows some fixed rules, including these:

- If a Command key is held down, the Control keys are ignored. Command trumps Control. Otherwise, Command doesn't affect the character that's reported for the key. If only Command is held down, the character that's reported is the same as if no modifiers were down; if Command and Option are held down, the character that's reported is the same as for Option alone; and so on.
- If a Control key is held down (without a Command key), Shift, Option, and all keyboard locks are ignored. Control trumps the other modifiers (except for Command).
- Num Lock applies only to keys on the numerical keypad. While this lock is on, the effect of the Shift key is inverted. Num Lock alone yields the same character that's produced when a Shift key is down (and Num Lock is off). Num Lock plus Shift yields the same character that's produced without either Shift or the lock.
- Menu and Scroll Lock play no role in determining how keys are mapped to characters.

The default key map also follows the conventional rules for Caps Lock and Control:

- Caps Lock applies only to the 26 alphabetic keys on the main keyboard. It serves to map the key to the same character as Shift. Using Shift while the lock is on undoes the effect of the lock; the character that's reported is the same as if neither Shift nor Caps Lock applied. For example, Shift-*G* and Caps Lock-*G* both are mapped to uppercase 'G', but Shift-Caps Lock-*G* is mapped to lowercase 'g'.

  However, if the lock doesn't affect the character, Shift plus the lock is the same as Shift alone. For example, Caps Lock-*7* produces '7' (the lock is ignored) and Shift-*7* produces '&' (Shift has an effect), so Shift-Caps Lock-*7* also produces '&' (only Shift has an effect).

- When Control is used with a key that otherwise produces an alphabetic character, the character that's reported has an ASCII value 0x40 less than the value of the uppercase version of the character (0x60 less than the lowercase version of the character). This often results in a character that is produced independently by another key. For example, Control-*I* produces the **TAB** character and Control-*L* produces **PAGE_DOWN**.

  When Control is used with a key that doesn't produce an alphabetic character, the character that's reported is the same as if no modifiers were on. For example, Control-*7* produces a '7'.

The Interface Kit defines constants for characters that aren't normally represented by a visible symbol. This includes the usual space and backspace characters, but most invisible characters are produced by the function keys and the navigation keys located between the main keyboard and the numeric keypad. The character values associated with these keys are more or less arbitrary, so you should always use the constant in your code rather than the actual character value. Many of these characters are also produced by alphabetic keys when a Control key is held down.

The table below lists all the character constants defined in the Kit and the keys they're associated with.

Key Label	Key Code	Character Reported
Backspace	0x1e	**BACKSPACE**
Tab	0x26	**TAB**
Enter	0x47	**ENTER**
(space bar)	0x5e	**SPACE**
Escape	0x01	**ESCAPE**
F1 – F12	0x02 through 0x0d	**FUNCTION_KEY**
Print Screen	0x0e	**FUNCTION_KEY**
Scroll Lock	0x0f	**FUNCTION_KEY**
Pause	0x10	**FUNCTION_KEY**
System Request	0x7e	0xc8
Break	0x7f	0xca

Key Label	Key Code	Character Reported
Insert	0x1f	**INSERT**
Home	0x20	**HOME**
Page Up	0x21	**PAGE_UP**
Delete	0x34	**DELETE**
End	0x35	**END**
Page Down	0x36	**PAGE_DOWN**
(up arrow)	0x57	**UP_ARROW**
(left arrow)	0x61	**LEFT_ARROW**
(down arrow)	0x62	**DOWN_ARROW**
(right arrow)	0x63	**RIGHT_ARROW**

Several keys are mapped to the **FUNCTION_KEY** character. An application can determine which function key was pressed to produce the character by testing the key code against these constants:

F1_KEY	**F6_KEY**	**F11_KEY**
F2_KEY	**F7_KEY**	**F12_KEY**
F3_KEY	**F8_KEY**	**PRINT_KEY** (the "Print Screen" key)
F4_KEY	**F9_KEY**	**SCROLL_KEY** (the "Scroll Lock" key)
F5_KEY	**F10_KEY**	**PAUSE_KEY**

Note that key 0x30 (*P*) is also mapped to **FUNCTION_KEY** when the Control key is held down.

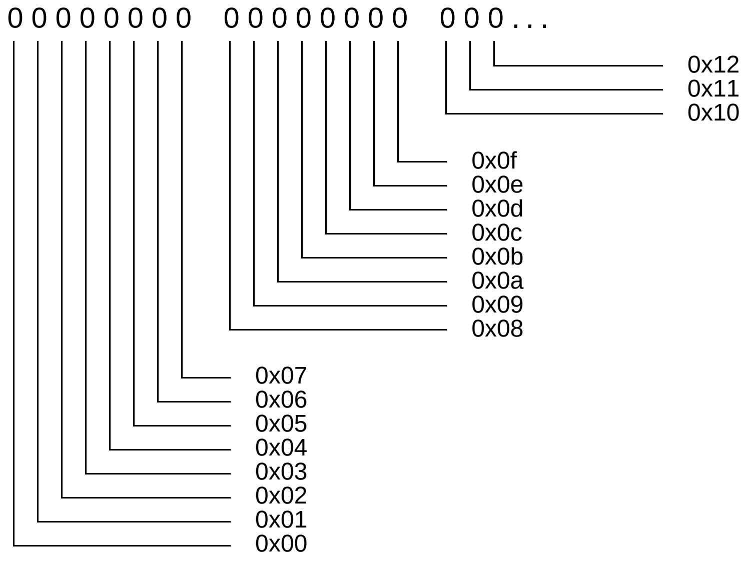
Key States

The "states" bit field that's reported in a key-down message captures the state of all keys and keyboard locks at the time of the event. At other times, you can obtain the same information through BView's **GetKeys()** function.

Although the "states" bit field is declared as **UCHAR_TYPE**, it's not just a single **uchar**. It's really an array of 16 bytes,

```
uchar states[16];
```

with one bit standing for each key on the keyboard. Bits are numbered from left to right, beginning with the first byte in the array, as illustrated on the next page.



Bit numbers start with 0 and match key codes. For example, bit 0x3c corresponds to the *A* key, 0x3d to the *S* key, 0x3e to the *D* key, and so on. The first bit is 0x00, which doesn't correspond to any key. The first meaningful bit is 0x01, which corresponds to the Escape key.

When a key is down, the bit corresponding to its key code is set to 1. Otherwise, the bit is set to 0. However, for the three keys that toggle keyboard locks—Caps Lock (key 0x3b), Num Lock (key 0x22), and Scroll Lock (key 0x0f)—the bit is set to 1 if the lock is on and set to 0 if the lock is off, regardless of the state of the key itself.

To test the "states" mask against a particular key,

- Select the byte in the "states" array that contains the bit for that key,
- Form a mask for the key that can be compared to that byte, and
- Compare the byte to the mask.

For example:

```
if ( states[keyCode>>3] & (1 << (7 - (keyCode%8))) )
     . . .
```

Here, the key code is divided by 8 to obtain an index into the *states* array. This selects the byte (the **uchar**) in the array that contains the bit for that key. Then, the part of the key code that remains after dividing by 8 is used to calculate how far a bit needs to be shifted to the left so that it's in the same position as the bit corresponding to the key. This mask is compared to the *states* byte with the bitwise **&** operator.

Guide to the Classes

The classes in the Interface Kit work together to define a program structure for drawing and responding to events. The two classes at the core of the structure—BWindow and BView—have been discussed extensively above. Other Kit classes either derive from BWindow and BView or support the work of those that do. The Kit defines several different kinds of BViews that you can use in your application, but each application must also invent some BViews of its own, to do the drawing and event handling that's unique to it.

To learn about the Interface Kit for the first time, it's recommended that you first read this introduction, then look at the class descriptions in roughly the following order:

1	BWindow	Windows are at the center of the user interface. They're where applications present themselves to the user and where users do their work. All other Interface Kit objects are associated with BWindows in one way or another.
2	BView	BView objects draw within windows and handle most user actions on the keyboard and mouse. Each object corresponds to a particular *view*, one part of the window's display. Several of the other classes in the Interface Kit inherit from BView and implement particular kinds of views—such as buttons, text displays, and scroll bars. In conjunction with the BWindow class, BView defines the Kit's mechanisms for drawing and handling events.
3	BPoint and BRect	These two classes define the basic data types for coordinate geometry. They're ubiquitous throughout the kit.
4	BRegion and BPolygon	Like BRect, these two classes define objects that describe areas and shapes within a coordinate system. They're used by functions in the BView class.
5	BBitmap	This class defines objects that store bitmap data. BBitmaps are passed to BView functions, which place the bitmap images on-screen.
6	BScrollBar and BScrollView	BScrollBar objects provide scroll bars for an application, and a BScrollView sets up the scroll bars for a target view. Scrolling is explained in the BView and BScrollBar class descriptions.
7	BMenu, BMenuItem, BMenuBar, and	These classes implement the Be menu system. A BMenu object represents a menu list, and a

	BPopUpMenu	BMenuItem represents a single item in the list. An item can control a submenu—another BMenu object—so menus can be hierarchically arranged. A BMenuBar is the visible menu at the root of the hierarchy.
8	BTextView	A BTextView object displays text on-screen and implements the user interface for editing and selecting text.
9	BControl, BCheckBox, BRadioButton, and BButton	The BControl class is the base class for objects that implement control devices. The other three classes are derived from BControl.
10	BListView	A BListView is similar to the control classes. It displays a list of items that the user can select and invoke. This class is based on the BList class of the Storage Kit.
11	BAlert	A BAlert runs a modal window that alerts the user to something and asks for a response. It's a convenience for putting warnings and dialogs on-screen.
12	BStringView and BBox	These are simple views that don't respond to events. A BStringView draws a string (such as a label). A BBox draws a labeled box around other views.

The class overview should help you determine which specific functions you need to turn to in order to get more information about a class. The class constructor is often a good place to start, as it contains general information on how instances of the class are initialized.

If you haven't already read about the BApplication object and messaging classes in the Application Kit, be sure to do so. A program must have a BApplication object before it can use the Interface Kit.

A reference to the Interface Kit follows. The classes are presented in alphabetical order, beginning with BAlert.

BAlert

Derived from:	public BWindow
Declared in:	<interface/Alert.h>

Overview

A BAlert places a modal window on-screen in front of other windows and keeps it there until the user dismisses it. The window has a message for the user to read and one or more buttons along the bottom that present various options for the user to choose among. Clicking a button selects a course of action and dismisses the window (closes it). The message might warn the user of something or convey some information that the application doesn't want the user to overlook. Typically, it asks a question that the user must answer (by clicking the appropriate button).

The window stays on-screen only temporarily, until the user operates one of the buttons. As long as it's on-screen, other parts of the application's user interface are disabled. < However, the user can continue to move windows around and work in other applications. >

It's possible to design such a window using a BWindow object, some BButtons, and other views. However, the BAlert class provides a simple way to do it. There's no need to construct views and arrange them, or call functions to show the window and then get rid of it. All you do is:

- Construct the object,
- Call **SetShortcut()** if you want the user to be able to operate any buttons (other than the default button) from the keyboard, and
- Call **Go()** to put the window on-screen.

For example:

```
BAlert *alert;
long result;

alert = new BAlert("", "Time’s up!  Do you want to continue?",
                      "Cancel", "Continue");
alert->SetShortcut("Cancel", ESCAPE);
result = alert->Go();
```

Go() doesn't return until the user dismisses the window. When it returns, the window will have been closed, the window thread will have been killed, and the BAlert object will have been deleted.

The value **Go()** returns indicates which button dismissed the window. If the user clicked the "Cancel" button in this example or pressed the Escape key, the return result would be 1. If the user clicked "Continue", the result would be 2. Since the BAlert sets up the rightmost button as the default button for the window, the user could also operate the "Continue" button by pressing the Enter key.

Constructor and Destructor

BAlert()

```
BAlert(const char *title, const char *information,
                         const char *firstButton,
                         const char *secondButton = NULL,
                         const char *thirdButton = NULL)
```

Creates a modal window and shows it on-screen. The window displays some textual *information* for the user to read, and can have up to three buttons. There must be at least a *firstButton*; the others are optional. The window must also have a *title* (which can be an empty string, but not **NULL**), even though it won't be displayed to the user. Modal windows lack a title tab.

The buttons are arranged in a row at the bottom of the window so that one is always in the right bottom corner. They're placed from left to right in the order specified to the constructor. If labels for three buttons are provided, *firstButton* will be on the left, *secondButton* in the middle, and *thirdButton* on the right. If only two labels are provided, *firstButton* will come first and *secondButton* will be in the right bottom corner. If there's just one label (*firstButton*), it will be at the right bottom location.

By default, the user can operate the rightmost button by pressing the Enter key.

After the BAlert is constructed, **Go()** must be called to place it on-screen.

See also: Go()

Member Functions

FilterKeyDown()

virtual bool **FilterKeyDown(**ulong **aChar*, BView ***target***)**

Permits keyboard shortcuts to operate the buttons and dismiss the window. There's no need for your application to call or override this function. Call **SetShortcut()** to assign shortcut characters to buttons.

See also: SetShortcut()

Go()

long **Go(**void**)**

Calls the **Show()** virtual function to place the window on-screen, sets the modal loop for the BAlert in motion, and returns when the loop has quit and the window has been closed. The value returned is the number of the button that the user operated (either by clicking it or using its keyboard shortcut) to dismiss the window. Buttons are numbered from left to right, beginning with 1.

To put an alert panel on-screen, simply construct a BAlert object, set its keyboard shortcuts, if any, and call this function. See the example code in the "Overview" section above.

See also: the BAlert constructor

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Closes the window in response to messages posted from the window's buttons. There's no need for your application to call or override this function.

SetShortcut()

void **SetShortcut(**const char **button*, char *shortcut***)**

Sets a *shortcut* character that the user can type to operate the *button*. The *button* argument must match one of the button labels passed to the constructor. By default, **ENTER** is the shortcut for the rightmost button.

The shortcut doesn't require the user to hold down a Command key or other modifier (except for any modifiers that would normally be required to produce the *shortcut* character).

The shortcut is valid only while the window is on-screen.

BBitmap

Derived from:	public BObject
Declared in:	<interface/Bitmap.h>

Overview

A BBitmap object is a container for an image bitmap; it stores pixel data—data that describes an image pixel by pixel. The class provides a way of specifying a bitmap from raw data, and also a way of creating the data from scratch using the Interface Kit graphics mechanism.

BBitmap functions manage the bitmap data and provide information about it. However, they don't do anything with the data. Placing the image somewhere so that it can be seen is the province of BView functions—such as **DrawBitmap()** and **DragMessage()**—not this class.

Bitmap Data

An image bitmap records the color values of pixels within a rectangular area. The pixels in the rectangle, as on the screen, are arranged in rows and columns. The data is specified in rows, beginning with the top row of pixels in the image and working downward to the bottom row. Each row of data is aligned on a long word boundary and is read from left to right.

New BBitmap objects are constructed with two pieces of information that prepare them to store bitmap data—a bounds rectangle and a color space. For example, this code

```
BRect rect(0.0, 0.0, 39.0, 79.0);
BBitmap *image = new BBitmap(rect, COLOR_8_BIT);
```

constructs a bitmap of 40 rows and 80 pixels per row. Each pixel is specified by an 8-bit color value.

The Bounds Rectangle

A BBitmap's bounds rectangle serves two purposes:

- It sets the size of the image. A bitmap covers as many pixels as its bounds rectangle encloses—under the assumption that one coordinate unit equals one pixel, as it does when the display device is the screen.

  Since a bitmap can't contain a fraction of a pixel, the bounds rectangle shouldn't contain any fractional coordinates. Without fractional coordinates, each side of the bounds rectangle will be aligned with a column or a row of pixels. The pixels around the edge of the rectangle are included in the image, so the bitmap will contain one more column of pixels than the width of the rectangle and one more row than the rectangle's height. (See the BRect class "Overview" on page 151 for an illustration.)

- It establishes a coordinate system that can be used later by drawing functions, such as **DrawBitmap()** and **DragMessage()**, to designate particular points or portions of the image.

  For example, if one BBitmap was constructed with this bounds rectangle,

  ```
  BRect firstRect (0.0, 0.0, 60.0, 100.0);
  ```

  and another with this rectangle,

  ```
  BRect secondRect(60.0, 100.0, 120.0, 200.0);
  ```

  they would both have the same size and shape. However, the coordinates (60.0, 100.0) would designate the right bottom corner of the first bitmap, but the left top corner of the second.

< If a BBitmap object enlists BViews to create the bitmap data, it must have a bounds rectangle with (0.0, 0.0) at the left top corner. >

The Color Space

The color space of a bitmap determines its depth (how many bits of information are stored for each pixel) and its interpretation (what the data values mean). These four color spaces are currently defined:

MONOCHROME_1_BIT
GRAYSCALE_8_BIT
COLOR_8_BIT
RGB_24_BIT

In the **RGB_24_BIT** color space, the color of each pixel is specified as an **rgb_color** value. In the **COLOR_8_BIT** color space, colors are specified as indices into the color map. In the **MONOCHROME_1_BIT** color space, a value of 1 means black and 0 means white. (A more complete description of the four color spaces can be found under “Colors” on page 24 of the introduction to this chapter.)

< Currently, bitmap data is stored only in the **COLOR_8_BIT** and **MONOCHROME_1_BIT** color spaces, though it can also be specified in the **RGB_24_BIT** format. The **GRAYSCALE_8_BIT** color space is not used at the present time. >

Specifying the Image

BBitmap objects begin life empty. When constructed, they allocate sufficient memory to store an image of the size and color space specified. However, the memory isn’t initialized. The actual image must be set after construction. This can be done by explicitly assigning pixel values with the **SetBits()** function:

```
     image->SetBits(rawData, numBytes, 0, COLOR_8_BIT);
```

In addition to this function, BView objects can be enlisted to produce the bitmap. Views are assigned to a BBitmap object just as they are to a BWindow (by calling the **AddChild()** function). In reality, the BBitmap sets up a private, off-screen window for the views. When the views draw, the window renders their output into the bitmap buffer. The rendered image has the same format as the data captured by the **SetBits()** function. **SetBits()** and BViews can be used in combination to create a bitmap.

The BViews that construct a bitmap behave a bit differently than the BViews that draw in regular windows:

- In contrast to BViews attached to an ordinary window, the BViews assigned to a BBitmap can create an image off-screen. When an ordinary window is hidden, it doesn’t render images; its BViews may draw, but they don’t produce image data. However, the BViews assigned to a BBitmap produce an off-screen bitmap.

- Because they never appear on-screen, the BViews that produce a bitmap image never handle events and never get update messages telling them to draw. You must call their drawing functions directly in your own code.

  This is typically done just once, to create the bitmap. After that, the BViews can be discarded; they’ll never be called upon to update the image. However, if the bitmap will change—perhaps to reflect decisions the user makes as the program runs—the BViews can be retained to make the changes.

- A BBitmap has no background color against which images are drawn. Your code must color every pixel within the bounds rectangle.

So that you can manage the BViews that are assigned to a BBitmap, the BBitmap class duplicates a number of BWindow functions—such as **AddChild()**, **FindView()**, and **ChildAt()**.

A BBitmap that enlists views to produce the bitmap consumes more system resources than one that relies solely on **SetBits()**. Therefore, by default, BBitmaps refuse to accept

BViews. If BViews will be used to create bitmap data, the BBitmap constructor must be informed so that it can set up the off-screen window and prepare the rendering mechanism.

Transparency

Color bitmaps can have transparent pixels. When the bitmap is imaged in a drawing mode other than **OP_COPY**, its transparent pixels won't be transferred to the destination view. The destination image will show through wherever the bitmap is transparent.

To introduce transparency into a **COLOR_8_BIT** bitmap, a pixel can be assigned a value of **TRANSPARENT_8_BIT**. In a **RGB_24_BIT** bitmap, a pixel can be assigned the special value of **TRANSPARENT_24_BIT**. (Or **TRANSPARENT_24_BIT** can be made the front or background color of the BView drawing the bitmap.)

Transparency is covered in more detail under "Drawing Modes" on page 27 of the chapter introduction.

See also: system_colors() global function

Constructor and Destructor

BBitmap()

BBitmap(BRect *bounds*, color_space *mode*, bool *acceptsViews* = **FALSE)**

Initializes the BBitmap to the size and internal coordinate system implied by the *bounds* rectangle and to the depth and color interpretation specified by the *mode* color space.

This function allocates enough memory to store data for an image the size of *bounds* at the depth required by *mode*, but does not initialize any of it. All pixel data should be explicitly set using the **SetBits()** function, or by enlisting BViews to produce the bitmap. If BViews are to be used, the constructor must be informed by setting the *acceptsViews* flag to **TRUE**. This permits it to set up the mechanisms for rendering the image, including an off-screen window to contain the views.

< If the BBitmap accepts BViews, the left and top sides of its *bounds* rectangle must be located at 0.0. >

~BBitmap()

virtual **~BBitmap(**void**)**

Frees all memory allocated to hold image data, deletes any BViews used to create the image, gets rid of the off-screen window that held the views, and severs the BBitmaps's connection to the Application Server.

Member Functions

AddChild()

virtual void **AddChild(**BView **aView***)**

Adds *aView* to the hierarchy of views associated with the BBitmap, attaching it to an off-screen window (one created by the BBitmap for just this purpose) by making it a child of the window's top view. If *aView* already has a parent, it's removed from that view hierarchy and adopted into this one. A view can serve only one window at a time.

Like **AddChild()** in the BWindow class, this function calls the BView's **AttachedToWindow()** function to inform it that it now belongs to a view hierarchy. Every view that descends from *aView* also becomes attached to the BBitmap's off-screen window and receives its own **AttachedToWindow()** notification.

AddChild() fails if the BBitmap was not constructed to accept views.

See also: AddChild() in the BWindow class, **AttachedToWindow()** in the BView class, **RemoveChild()**, the BBitmap constructor

Bits()

inline void ***Bits(**void**)** const

Returns a pointer to the bitmap data. The data lies in memory shared by the application and the Application Server. The length of the data can be obtained by calling **BitsLength()**—or it can be calculated from the height of the bitmap (the number of rows) and the number of bytes per row.

See also: Bounds(), **BytesPerRow()**, **BitsLength()**

BitsLength()

inline long **BitsLength(**void**)** const

Returns the number of bytes that were allocated to store the bitmap data.

See also: Bits(), **BytesPerRow()**

Bounds()

inline BRect **Bounds(**void**)** const

Returns the bounds rectangle that defines the size and coordinate system of the bitmap. This should be identical to the rectangle used in constructing the object.

See also: the BBitmap constructor

BytesPerRow()

inline long **BytesPerRow(**void**)** const

Returns how many bytes of data are required to specify a row of pixels. For example, a monochrome bitmap (one bit per pixel) 80 pixels wide would require twelve bytes per row (96 bits). The extra sixteen bits at the end of the twelve bytes are ignored. Every row of bitmap data is aligned on a long word boundary.

ChildAt(), CountChildren()

BView ***ChildAt(**long *index***)** const

long **CountChildren(**void**)** const

ChildAt() returns the child BView at *index*, or **NULL** if there's no child at *index*. Indices begin at 0 and count only BViews that were added to the BBitmap (added as children of the top view of the BBitmap's off-screen window) and not subsequently removed.

CountChildren() returns the number of BViews the BBitmap currently has. (It counts only BViews that were added directly to the BBitmap, not BViews farther down the view hierarchy.)

< Do not rely on these functions as they may not remain in the API. >

These functions fail if the BBitmap wasn't constructed to accept views.

See also: **ChildAt()** in the BWindow class

ColorSpace()

inline color_space **ColorSpace(**void**)** const

Returns the color space of the data being stored (not necessarily the color space of the data passed to the **SetBits()** function). Once set by the BBitmap constructor, the color space doesn't change.

The **color_space** data type is defined in **interface/InterfaceDefs.h** and is explained on page 24 and in the overview above.

See also: the BBitmap constructor

CountChildren() see **ChildAt()**

FindView()

> BView ***FindView(**BPoint *point***)** const
> BView ***FindView(**const char **name***)** const

Returns the BView located at *point* within the bitmap, or the BView tagged with *name*. The point must be somewhere within the BBitmap's bounds rectangle, which must have the coordinate origin, (0.0, 0.0), at its left top corner.

If the BBitmap doesn't accept views, this function fails. If no view draws at the *point* given, or no view associated with the BBitmap has the *name* given, it returns **NULL**.

See also: **FindView()** in the BView class

Lock(), Unlock()

> bool **Lock(**void**)**
>
> void **Unlock(**void**)**

These functions lock and unlock the off-screen window where BViews associated with the BBitmap draw. Locking works for this window and its views just as it does for ordinary on-screen windows.

Lock() returns **FALSE** if the BBitmap doesn't accept views or if its off-screen window is unlockable (and therefore unusable) for some reason. Otherwise, it doesn't return until it has the window locked and can return **TRUE**.

See also: **Lock()** in the BWindow class

RemoveChild()

> virtual bool **RemoveChild(**BView **aView***)**

Removes *aView* from the hierarchy of views associated with the BBitmap, but only if *aView* was added to the hierarchy by calling BBitmaps's version of the **AddChild()** function.

If *aView* is successfully removed, **RemoveChild()** returns **TRUE**. If not, it returns **FALSE**.

See also: **AddChild()**

SetBits()

void **SetBits(**const void **data*, long *length*, long *offset*, color_space *mode***)**

Assigns *length* bytes of *data* to the BBitmap. The new data is copied into the bitmap beginning *offset* bytes from the start of allocated memory. To set data beginning with the first (left top) pixel in the image, the *offset* should be 0.

The data is specified in the *mode* color space, which may or may not be the same as the color space that the BBitmap uses to store the data. If not, a conversion is automatically made. < Currently, only **RGB_24_BIT** data is converted, to **COLOR_8_BIT** data. In the conversion, colors are dithered, so that the resulting image will match the original as closely as possible, despite the lost information. **SetBits()** rejects data in **GRAYSCALE_8_BIT** mode. >

This function works for all BBitmaps, whether or not BViews are also enlisted to produce the image.

BBox

Derived from:	public BView
Declared in:	<interface/Box.h>

Overview

A BBox draws a labeled border around other views. It serves only to label those views and organize them visually. It doesn't respond to events.

The border is drawn around the edge of the view's frame rectangle. If the BBox has a label, the border at the top of box is broken where the label appears (and the border is inset from the top somewhat to make room for the label).

The current pen size of the view determines the width of the border, which by default is 1 coordinate unit. The label is drawn in the current font, which **AttachedToWindow()** sets to a 9-point "geneva." Both the border and the label are drawn in the current front color; the default front color is black.

The views that the box encloses should be made children of the BBox object.

Constructor and Destructor

BBox()

BBox(BRect *frame*, const char **name* = **NULL**,
ulong *resizingMode* = **FOLLOW_LEFT_TOP**,
ulong *flags* = **WILL_DRAW)**

Initializes the BBox by passing all arguments to the BView constructor. The new object doesn't have a label; call **SetLabel()** to assign it one.

See also: SetLabel()

~BBox()

virtual **~BBox(**void**)**

Frees the label, if the BBox has one.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Sets the default font for drawing the label to the 9-point "geneva" bitmap font.

This function is called by the Interface Kit; you shouldn't call it yourself. However, you can reimplement it to set a different font and other graphics parameters—such as the front color and pen size that will be used to draw the box.

See also: **AttachedToWindow()** in the BView class

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the box and its label. This function is called automatically in response to update messages.

See also: **Draw()** in the BView class

SetLabel(), Label()

void **SetLabel(**const char **string***)**

const char ***Label(**void**)** const

These functions set and return the label that's displayed along the top edge of the box. **SetLabel()** copies *string* and makes it the BBox's label, freeing the previous label, if any. If *string* is **NULL**, it removes the current label and frees it.

Label() returns a pointer to the BBox's current label, or **NULL** if it doesn't have one.

BButton

Derived from:	public BControl
Declared in:	<interface/Button.h>

Overview

A BButton object draws a labeled button on-screen and responds when the button is clicked or when it's operated from the keyboard. If the BButton is the *default button* for its window and the window is the active window, the user can operate it by pressing the Enter key.

BButtons have a single state. Unlike check boxes and radio buttons, the user can't toggle a button on and off. However, the button's value changes while it's being operated. During a click (while the user holds the mouse button down and the cursor points to the button on-screen), the BButton's value is set to 1. Otherwise, the value is 0.

This class, like BCheckBox and BRadioButton, depends on the control framework defined in the BControl class. In particular, it calls these BControl functions:

- **SetValue()** to make each change in the BControl's value. This is a hook function that you can override to take collateral action when the value changes.
- **Invoke()** to post a message each time the button is clicked or operated from the keyboard. You can designate the object that should receive the message by calling BControl's **SetTarget()** function. A model for the message is set by the BButton constructor (or by BControl's **SetMessage()** function).
- **IsEnabled()** to determine how the button should be drawn and whether it's enabled to post a message. You can call BControl's **SetEnabled()** to enable and disable the button.

A BButton is an appropriate control device for initiating an action. Use a BCheckBox or BRadioButtons to set a state.

Hook Functions

MakeDefault()	Makes the BButton the default button for its window or removes that status; can be augmented by derived classes to take note when the status of the button changes.

Constructor

BButton()

> **BButton(**BRect *frame*, const char **name*,
> const char **label*,
> BMessage **message*,
> ulong *resizingMode* = **FOLLOW_LEFT_TOP**,
> ulong *flags* = **WILL_DRAW)**

Initializes the BButton by passing all arguments to the BControl constructor. BControl initializes the button's *label* and assigns it a model *message* that identifies the action that should be carried out when the button is invoked.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed up the inheritance hierarchy to the BView constructor without change.

See also: the BControl and BView constructors, **Invoke()** in the BControl class

Member Functions

AttachedToWindow()

> virtual void **AttachedToWindow(**void**)**

Augments the BControl version of this function to make sure that the BButton does not consider itself the default button for the window to which it has just become attached—even if it may have been the default button for the window to which it was previously attached.

This version of **AttachedToWindow()** incorporates the BControl version.

See also: **AttachedToWindow()** in the BControl and BView classes, **MakeDefault()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the button and labels it. If the BButton's value is anything but 0, the button is highlighted. If it's disabled, it drawn in muted shades of gray. Otherwise, it's drawn in its ordinary, enabled, unhighlighted state.

See also: Draw() in the BView class

IsDefault() see MakeDefault

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Responds to a key-down event that reports that the user pressed the Enter key by:

- Momentarily highlighting the button and changing its value, and
- Posting a copy of the model BMessage to the target receiver.

This function is called if:

- The window the button is in is the active window,
- The BButton is the default button for the window, and
- *aChar* is **ENTER**.

It might also be called if the BButton object is the focus view for the active window, but BButtons normally don't make themselves the focus for keyboard events.

See also: Invoke() in the BControl class, **MakeDefault()**

MakeDefault(), IsDefault()

virtual void **MakeDefault(**bool *flag***)**

bool **IsDefault(**void**)** const

MakeDefault() makes the BButton the default button for its window when *flag* is **TRUE**, and removes that status when *flag* is **FALSE**. The default button is the button the user can operate by striking the Enter key when the window is the active window. **IsDefault()** returns whether the BButton is currently the default button.

A window can have only one default button at a time. Setting a new default button, therefore, may deprive another button of that status. When **MakeDefault()** is called with an argument of **TRUE**, it generates a **MakeDefault()** call with an argument of **FALSE** for previous default button. Both buttons are redisplayed so that the user can see which one is currently the default.

The default button can also be set by calling BWindow's **SetDefaultButton()** function. That function makes sure that the button that's forced to give up default status and the button that obtains it are both notified through **MakeDefault()** function calls.

MakeDefault() is therefore a hook function that can be augmented to take note each time the default status of the button changes. It's called once for each change in status, no matter which function initiated the change.

See also: **SetDefault()** in the BWindow class

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event in the button by tracking the cursor while the user holds the mouse button down. As the cursor moves in and out of the button, the BButton's value is reset accordingly. The **SetValue()** virtual function is called to make the change each time.

If the cursor is inside the BButton's bounds rectangle when the user releases the mouse button, this function posts a copy of the model message so that it will be dispatched to the target receiver.

See also: **MessageReceived()** in the BReceiver class, **Invoke()** and **SetTarget()** in the BControl class

BCheckBox

Derived from:	public BControl
Declared in:	<interface/CheckBox.h>

Overview

A BCheckBox object draws a labeled check box on-screen and responds to a click by changing the state of the device. A check box has two states: An “X” is displayed in the box when the object’s value is 1 (on), and is absent when the value is 0 (off). The BCheckBox is invoked (it posts a message to the target receiver) whenever its value changes in either direction—when it’s turned on *and* when it’s turned off.

A check box is an appropriate control device for setting a state—turning a value on and off. Use menu items or buttons to initiate actions within the application.

Constructor

BCheckBox()

BCheckBox(BRect *frame*, const char **name*, const char **label*,
BMessage **message*,
ulong *resizingMode* = **FOLLOW_LEFT_TOP**,
ulong *flags* = **WILL_DRAW)**

Initializes the BCheckBox by passing all arguments to the BControl constructor. BControl initializes the *label* of the check box and assigns it a model *message* that encapsulates the action that should be taken when the state of the check box changes.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed unchanged to the BView constructor.

The frame rectangle of a BCheckBox should be at least 11.0 units high to accommodate the check box and the label in the default font. The object draws at the bottom of its frame rectangle beginning at the left side; it doesn’t use any extra space there may happen to be at the top or on the right. (However, the user can click anywhere within the frame rectangle to operate the check box).

See also: the BControl and BView constructors

Member Functions

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the check box and its label. If the current value of the BCheckBox is 1, it's marked with an "X". If the value is 0, it's empty.

See also: **Draw()** in the BView class

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event within the check box by tracking the cursor while the user holds the mouse button down. If the cursor is inside the bounds rectangle when the user releases the mouse button, this function toggles the value of the BCheckBox and calls **Draw()** to redisplay it. If the box was empty before the mouse-down event, it will be marked afterward; if marked before, it will be empty afterwards.

When the value of the BCheckBox changes, a copy of the model BMessage is posted so that it can be delivered to the object's target receiver. See BControl's **Invoke()** and **SetTarget()** functions for more information. The message is dispatched by calling the target's **MessageReceived()** virtual function.

The receiver can get a pointer to the BCheckBox from the message, and use it to discover the object's new value. For example:

```
void MyReceiver::MessageReceived(BMessage *msg)
{
    . . .
    BCheckBox *box = (BCheckBox *)msg->FindObject("source");
    if ( message->Error() == NO_ERROR ) {
        long value = box->Value();
        . . .
    }
    . . .
}
```

See also: **Invoke()**, **SetTarget()**, and **SetValue()** in the BControl class

BControl

Derived from:	public BView
Declared in:	<interface/Control.h>

Overview

BControl is an abstract class for views that draw control devices on the screen. Objects that inherit from BControl emulate, in software, real-world control devices—like the switches and levers on a machine, the check lists and blank lines on a form to fill out, or the dials and knobs on a home appliance.

Controls turn the messages that report generic mouse and keyboard events into other messages with more specific instructions for the application. Just as a switch that you might buy in a hardware store can be hooked up to do various kinds of work, a BControl object can be customized by setting the message it posts when invoked and the target receiver that should handle the message.

The Interface Kit currently includes three classes derived from BControl—BButton, BRadioButton, and BCheckBox. In addition, it has two classes—BListView and BMenuItem—that implement control devices but are not derived from this class. BListView shares an interface with the BList class (of the Support Kit) and BMenuItem is designed to work with the other classes in the menu system.

As BListView and BMenuItem demonstrate, it's possible to implement a control device that's not a BControl. However, it's simpler to take advantage of the code that's already provided by the BControl class. That way you can keep a simple programming interface and avoid reimplementing functions that BControl has defined for you. If your application defines its own control devices—dials, sliders, selection lists, text fields, and the like—they should be derived from BControl.

Hook Functions

SetEnabled()	Enables and disables the control device; can be augmented by derived classes to note when the state of the object has changed.
SetValue()	Changes the value of the control device; can be augmented to take collateral action when the change is made.

Constructor and Destructor

BControl()

> **BControl(**BRect *frame*, const char **name*, const char **label*,
> BMessage **message*, ulong *resizingMode*, ulong *flags***)**

Initializes the BControl by setting its initial value to 0, assigning it a *label*, which can be **NULL**, and registering a model *message* that captures what the control does—the command it gives when it's invoked and the information that accompanies the command.

The *label* is copied, but the *message* is not. The BMessage object becomes the property of the BControl; it should not be deleted, posted, assigned to another object, or otherwise used in application code. The label and message can be altered after construction with the **SetLabel()** and **SetMessage()** functions.

The BControl class doesn't define **Draw()**, **MouseDown()**, or **KeyDown()** functions. It's up to derived classes to determine how the *label* is drawn and how the *message* is to be used. Typically, when a BControl object needs to take action (in response to a click, for example), it calls the **Invoke()** function, which copies the model message and posts the copy so that it will be received by the designated target. By default, the target is the window where the control is located, but **SetTarget()** can designate another receiver.

Before posting a copy of the model message, **Invoke()** adds two data entries to it, under the names "when" and "source". These names should not be used for data items in the model.

The *frame*, *name*, *resizingMode*, and *flags* arguments are identical to those declared for the BView class and are passed unchanged to the BView constructor.

See also: the BView constructor, **PostMessage()** in the BLooper class of the Application Kit, **SetLabel()**, **SetMessage()**, **SetTarget()**, **Invoke()**

~BControl()

> virtual **~BControl(**void**)**

Frees the model message and all memory allocated by the BControl.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Overrides BView's version of this function to set the default font for all control devices to 9-point "chicago". It also makes the BWindow to which the BControl has become attached the default target for the **Invoke()** function, provided that another target hasn't already been set. To make the font change, it calls BView's **SetFontName()**; to designate the target, it calls **SetTarget()**. Both are virtual functions.

AttachedToWindow() is called for you when the BControl becomes a child of a view already associated with the window.

See also: AttachedToWindow() and **SetFontName()** in the BView class, **Invoke()**, **SetTarget()**

Command() see SetMessage()

Invoke()

protected:
void **Invoke(**void**)**

Copies the BControl's model BMessage and posts the copy so that it will be dispatched to the designated target. The following two pieces of information are added to the copy before it's posted:

Data name	Type code	Description
"when"	**LONG_TYPE**	When the control was invoked, as measured in milliseconds from the time the machine was last booted.
"source"	**OBJECT_TYPE**	A pointer to the BControl object. This permits the message receiver to request more information from the source of the message.

These two names shouldn't be used for data entries in the model.

If the control doesn't have a designated target, but it does have a designated BLooper where it can post the message, it will ask the BLooper for its preferred receiver and name it as the target. Since the preferred receiver for a BWindow object is the current focus view, this option allows control devices to be targeted to whatever view happens to be in focus at the time. See the **SetTarget()** function for information on how to designate a target BReceiver and BLooper for the control.

Invoke() is designed to be called from the **MouseDown()** and **KeyDown()** functions defined for derived classes; it's not called for you in BControl code. It's up to each derived class to define what user actions trigger the call to **Invoke()**—what activity constitutes "invoking" the control.

This function doesn't check to make sure the BControl is currently enabled. Derived classes should make that determination before calling **Invoke()**.

See also: SetTarget(), **SetMessage()**, **SetEnabled()**

IsEnabled() see SetEnabled()

Label() see SetLabel()

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *flag***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BControl if *flag* is **TRUE**, and disables it if *flag* is **FALSE**. **IsEnabled()** returns whether or not the object is currently enabled. BControls are enabled by default.

While disabled, a BControl typically won't post messages and won't respond visually to mouse and keyboard manipulation. To indicate this nonfunctional state, the control device is displayed on-screen in subdued colors.

However, it's left to each derived class to carry out this strategy in a way that's appropriate for the kind of control it implements. The BControl class merely marks an object as being enabled or disabled; none of its functions take the enabled state of the device into account.

Derived classes can augment **SetEnabled()** (override it) to take action when the control device becomes enabled or disabled. To be sure that **SetEnabled()** has been called to actually make a change, its current state should be checked before calling the inherited version of the function. For example:

```
void MyControl::SetEnabled(bool flag)
{
    if ( flag == IsEnabled() )
        return;
    BControl::SetEnabled(flag);
    /* Code that responds to the change in state goes here. */
}
```

Note, however, that you don't have to override **SetEnabled()** just to update the on-screen display when the control becomes enabled or disabled. If the BControl is attached to a window, the Kit's version of **SetEnabled()** always calls the **Draw()** function. Therefore, the device on-screen will be updated automatically—as long as **Draw()** has been implemented to take the enabled state into account.

See also: the BControl constructor

SetLabel(), Label()

virtual void **SetLabel(**const char **string***)**

const char ***Label(**void**)** const

These functions set and return the label on a control device—the text that's displayed, for example, on top of a button or alongside a check box or radio button. The label is a null-terminated string.

SetLabel() makes a copy of *string*, replaces the current label with it, frees the old label, and updates the control on-screen so the new label will be displayed to the user. The label is first set by the constructor and can be modified thereafter by this function.

Label() returns the current label. The string it returns belongs to the BControl and may be altered or freed without notice.

See also: the BControl constructor, **AttachedToWindow()**, **SetFontName()** in the BView class

SetMessage(), Message(), Command()

virtual void **SetMessage(**BMessage **message***)**

BMessage ***Message(**void**)** const

ulong **Command(**void**)** const

SetMessage() sets the model BMessage that defines what the BControl does, and frees the message that was previously set. **Message()** returns a pointer to the BMessage that's the current model, and **Command()** returns its **what** data member. The message is first set by the BControl constructor.

Because **Invoke()** adds "when" and "source" entries to the messages it posts, these two names shouldn't be used for any data entries in the model BMessage.

The model message passed to **SetMessage()** and returned by **Message()** belongs to the BControl object; it can be modified in application code, but it shouldn't be deleted (except by passing **NULL** to **SetMessage()**), posted, or put to any other use.

See also: the BControl constructor, **Invoke()**, **SetTarget()**

SetTarget(), Target()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper* = **NULL)**

BReceiver ***Target(**BLooper ***looper* = **NULL)** const

These functions set and return the object that's targeted to receive the messages the BControl posts (through its **Invoke()** function).

SetTarget() sets the *target* BReceiver, but is successful only if it can also discern a BLooper object where **Invoke()** can post messages to that target. **Invoke()** calls the BLooper's **PostMessage()** function and names the *target* as the object that should receive the message:

```
looper->PostMessage(theMessage, target);
```

If the *target* receiver passed to **SetTarget()** is itself a BLooper object (such as a BWindow) or if it's associated with a BLooper object (as BViews are associated with BWindows), the *looper* argument can be **NULL**. **SetTarget()** can discover the BLooper from the *target* (by calling the *target*'s **Looper()** function).

However, if the *target* can't supply a BLooper object, a specific *looper* must be named as an argument. If a *looper* isn't named and the *target* can't supply one, the function fails and returns **BAD_VALUE** to indicate that the *target* alone is inadequate.

Moreover, **SetTarget()** also fails if a specific *looper* is named but the *target* is associated with some other BLooper object. In this case, **MISMATCHED_VALUES** is returned to indicate that there's a conflict between the two arguments.

It's also possible to name a specific *looper*, but a **NULL** *target.* In this case, messages will be targeted to the *looper*'s preferred receiver (the object returned by its **PreferredReceiver()** function). For a BWindow, the preferred receiver is the current focus view. Therefore, by passing a **NULL** *target* and a BWindow *looper* to **SetTarget()**,

```
myControl->SetTarget(NULL, myControl->Window());
```

the control device can be targeted to whatever BView happens to be in focus at the time the control is invoked. This is useful for controls that act on the current selection. (Note, however, that if the **PreferredReceiver()** is **NULL**, the *looper* itself becomes the target.)

When successful, **SetTarget()** returns **NO_ERROR**.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where **Invoke()** will post messages. By default (established by **AttachedToWindow()**), both roles are filled by the BWindow where the control device is located.

See also: **Looper()** in the BReceiver and BView classes, **PreferredReceiver()** in the BLooper and BWindow classes, **Invoke()**, **AttachedToWindow()**

SetValue(), Value()

virtual void **SetValue(**long *value***)**

long **Value(**void**)** const

These functions set and return the value of the BControl object.

SetValue() assigns the object a new value. If the *value* passed is in fact different from the BControl's current value, this function calls the object's **Draw()** function so that the new value will be reflected in what the user sees on-screen; otherwise it does nothing.

Value() returns the current value.

Each class that's derived from BControl should call **SetValue()** in its **MouseDown()** and **KeyDown()** functions to change the value of the control device in response to user actions. The derived classes defined in the Be software kits change values only by calling this function.

Since **SetValue()** is a virtual function, you can override it to take note whenever a control's value changes. However, if you want your code to act only when the value actually changes, you must check to be sure the new value doesn't match the old before calling the inherited version of the function. For example:

```
void MyControl::SetValue(long value)
{
    if ( value != Value() ) {
        BControl::SetValue(value);
        /* MyControl's additions to SetValue() go here */
    }
}
```

Remember that the BControl version of **SetValue()** does nothing unless the new value differs from the old.

Target() see SetTarget()

Value() see SetValue()

BListView

Derived from:	public BView
Declared in:	<interface/ListView.h>

Overview

A BListView is a view that displays a list of items the user can select and invoke. This class is based on the BList class of the Support Kit. Every member function of the BList class is replicated by BListView, so you can treat a BListView object just like a BList. BListView simply makes the list visible.

Displaying the List

In both classes, the list keeps track of data pointers. Adding an item to the list adds only the pointer; the data itself isn't copied. Neither class imposes a type restriction on the data (both declare items to be type **void ***). However, by default, BListView assumes they're pointers to strings (type **char ***). Its functions can display the strings, highlight them when selected, and so on. As long as only string pointers are placed in the list, a BListView object can be used as is. However, if the list is to contain another kind of data, it's necessary to derive a class from BListView and reimplement some of its hook functions.

When the contents of the list change, the BListView makes sure the visible list on-screen is updated. However, it can know that something changed only when a data pointer changes, since pointers are all that the list records. If any pointed-to data is altered, but the pointer remains the same, you must force the list to be redrawn (by calling the **InvalidateItem()** function or BView's **Invalidate()**).

Selecting and Invoking Items

The user can click an item in the list to select it and double-click an item to both select and invoke it. The user can also select and invoke items from the keyboard. The navigation keys (such as Down Arrow, Home, and Page Up) select items; Enter invokes the item that's currently selected.

The BListView highlights the selected item, but otherwise it doesn't define what, if anything, should take place when an item is selected. You can determine that yourself by registering a “selection message” (a BMessage object) that should be delivered to a target receiver whenever the user selects an item.

Similarly, the BListView doesn’t define what it means to “invoke” an item. You can register a separate “invocation message” that’s posted whenever the user double-clicks an item or presses Enter while an item is selected. For example, if the user double-clicks an item in a list of file names, a message might be posted telling the BApplication object to open that file.

A BListView doesn’t have a default selection message or invocation message. Messages are posted only if registered with the **SetSelectionMessage()** and **SetInvocationMessage()** functions. The registered message is only a model. When an item is selected or invoked, the BListView makes a copy of the model, adds information to the copy about itself and the item, then posts the copy. See the function descriptions for information on the data that automatically gets added to the message.

See also: the BList class in the Support Kit

Hook Functions

DrawItem()	Draws the character string that the item points to; can be reimplemented to draw from another kind of data.
HighlightItem()	Highlights the item by inverting all the colors in its frame rectangle; can be reimplemented to highlight in a different way.
Invoke()	Posts the invocation message, if one has been registered for the BListView; can be augmented to do whatever else may be necessary when a item is invoked.
ItemHeight()	Returns the height of a single item, assuming that it’s a character string and is to be drawn in the current font; can be reimplemented to return the height required to draw a different kind of item. All items are taken to have the same height.
Select()	Highlights the selected item and posts the selection message, if one has been registered for the BListView; can be augmented to take any collateral action that may be required when the selection changes.

Constructor and Destructor

BListView()

> **BListView(**BRect *frame*, const char **name*,
> ulong *resizingMode* = **FOLLOW_LEFT_TOP**,
> ulong *flags* = **WILL_DRAW** | **FRAME_EVENTS)**

Initializes the new BListView. The *frame*, *name*, *resizingMode*, and *flags* arguments are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The list begins life empty. Call **AddItem()** or **AddList()** (documented for the BList class) to put items in the list. Call **Select()** (documented below) to select one of the items so that it's highlighted when the list is initially displayed to the user.

See also: the BView constructor, **AddItem()** in the BList class

~BListView()

> virtual **~BListView(**void**)**

Frees the model messages, if any, and all memory allocated to hold the list of items.

Member Functions

The BListView class reimplements *all* of the member functions of the BList class in the Support Kit. BListView's versions of these functions work identically to the BList versions, except that a BListView makes sure that the on-screen display is properly updated whenever the list changes.

Consequently, this section excludes all functions that BList and BListView have in common. It concentrates instead on those member functions that deal with the BListView's behavior as a view, not as a list. See the BList class for information on the functions that you can use to manipulate the BListView's list.

AttachedToWindow()

> virtual void **AttachedToWindow(**void**)**

Sets up the BListView so that it's prepared to draw character strings for items, and makes the BWindow to which the object has become attached the target for messages posted by the **Select()** and **Invoke()** functions—provided another target hasn't already been set.

This function is called for you when the BListView becomes part of a window's view hierarchy.

See also: AttachedToWindow() in the BView class, **SetTarget()**

BaselineOffset()

protected:

  float **BaselineOffset(**void**)**

Returns the distance from the bottom of an item's frame rectangle to the baseline where the item, assuming it is a character string, is drawn. The string is drawn beginning at a point that's offset 2.0 coordinate units from the left of the frame rectangle and **BaselineOffset()** units from the bottom. The offsets are the same for all items.

This function will give unreliable results unless the BListView is attached to a window.

CurrentSelection()

  inline long **CurrentSelection(**void**)** const

Returns the index of the currently selected item, or a negative number if no item is selected.

See also: Select()

Draw()

  virtual void **Draw(**BRect *updateRect***)**

Calls the **DrawItem()** hook function to draw each visible item in the *updateRect* area of the view and highlights the currently selected item by calling the **HighlightItem()** hook function.

Draw() is called for you whenever the list view is to be updated or redisplayed; you don't need to call it yourself. You also don't need to reimplement it, even if you're defining a list that displays something other than character strings. You should implement data-specific versions of **DrawItem()** and **HighlightItem()** instead.

See also: Draw() in the BView class, **DrawItem()**, **HighlightItem()**

DrawItem()

protected:

virtual void **DrawItem(**BRect *updateRect*, long *index***)**

Draws the item at *index*. The default version of this function assumes that the item is a character string. It can be reimplemented by derived classes to draw differently, based on other kinds of data.

The *updateRect* rectangle is stated in the BListView's coordinate system. It's the portion of the item's frame rectangle that needs to be updated. The full frame rectangle of the item is returned by the **ItemFrame()** function.

The **Draw()** function determines which items in the BListView need to be updated and calls **DrawItem()** for each one.

See also: ItemHeight(), **ItemFrame()**, **HighlightItem()**, **BaselineOffset()**

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Updates the on-screen display in response to a notification that the BListView's frame rectangle has been resized. In particular, this function looks for a vertical scroll bar that's a sibling of the BListView. It adjusts this scroll bar to reflect the way the list view was resized, under the assumption that it must have the BListView as its target.

FrameResized() is called automatically at the appropriate times; you shouldn't call it yourself.

See also: FrameResized() in the BView class

HighlightItem()

protected:

virtual void **HighlightItem(**bool *flag*, long *index***)**

Highlights the item at *index* if *flag* is **TRUE**, and removes the highlighting if *flag* is **FALSE**. Items are highlighted by inverting all colors in their frame rectangles.

This function is called (by **Draw()**) to highlight the selected item and (by **Select()**) to change the item that's highlighted whenever the selection changes. It can be reimplemented in a derived class to highlight in a different way.

See also: Select(), **Draw()**

InvalidateItem()

void **InvalidateItem(**long *index***)**

Invalidates the item at *index* so that an update message will be sent forcing the BListView to redraw it.

See also: Invalidate() in the BView class

Invoke()

virtual void **Invoke(**long *index***)**

Invokes the item at *index*, provided that the *index* isn't out-of-range.

This function is called whenever the user double-clicks an item in the list, or presses the Enter key while the BListView is the current focus view for the window and there's a selected item. It can also be called from application code to invoke a particular item; usually **Select()** would first be called to select the item.

To invoke an item that's identified by a pointer, first call **IndexOf()** to find where it's located in the list:

```
long i = myList->IndexOf(someItem);
myList->Select(i);
myList->Invoke(i);
```

If a model "invocation message" has been registered with the BListView (through **SetInvocationMessage()**), **Invoke()** makes a copy of the message, adds information to the copy identifying the BListView and the invoked item, and posts the copy so that it will be received by the designated target. The default target (established by **AttachedToWindow()**) is the BWindow where the BListView is located. If **SetTarget()** was called to name a particular BLooper where the message should be posted, but to set a **NULL** target, the target will be the BLooper's preferred receiver.

What it means to "invoke" an item depends entirely on the BMessage that's posted and the receiver's response when it gets the message. This function does nothing but post the message.

See also: Select(), **SetInvocationMessage()**, **SetTarget()**

IsItemSelected()

inline bool **IsItemSelected(**long *index***)** const

Returns **TRUE** if the item at *index* is currently selected, and **FALSE** if it's not.

See also: CurrentSelection()

ItemFrame()

protected:

BRect **ItemFrame(**long *index***)** const

Returns the frame rectangle of the item at *index*. The rectangle defines the area where the item is drawn; it's stated in the coordinate system of the BListView. The rectangle is calculated from the ordinal position of the item in the list and the value returned by **ItemHeight()**.

It's expected that you'd need to find an item's frame rectangle only if you're implementing a **DrawItem()** function.

< This function currently doesn't check to be sure that the index is in range. >

See also: DrawItem()

ItemHeight()

protected:

virtual float **ItemHeight(**void**)** const

Returns how much vertical room is required to draw a single item in the list—how high each item's frame rectangle should be. The BListView calls **ItemHeight()** extensively to determine where items are located and where to draw them. By default, it returns a height sufficient to draw a character string in the current font.

A derived class that draws items other than character strings should reimplement **ItemHeight()** so that it returns the height required to draw one of its items.

See also: DrawItem()

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Permits the user to operate the list using the following keys:

Keys	Perform Action
Up Arrow and Down Arrow	Select the items that are immediately before and immediately after the currently selected item.
Page Up and Page Down	Select the items that are one viewful above and below the currently selected item—or the first and last items if there's no item a viewful away.
Home and End	Select the first and last items in the list.
Enter	Invokes the currently selected item.

This function is called to notify the BListView of key-down events whenever it's the focus view in the active window; you shouldn't call it yourself.

See also: **KeyDown()** in the BView class, **Select()**, **Invoke()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Determines which item is located at *point* and calls **Select()** to select it (for a single-click or the first event in a series) and **Invoke()** to invoke it (for a double-click or the second in a series).

This function also makes the BListView the focus view so the user can operate the list from the keyboard.

MouseDown() is called to notify the BListView of a mouse-down event; you don't need to call it yourself.

See also: **MouseDown()** in the BView class, **Select()**, **Invoke()**

Select()

virtual void **Select(**long *index***)**

Selects the item located at *index*, provided that the *index* isn't out-of-range. This function removes the highlighting from the previously selected item and highlights the new selection, scrolling the list so the item is visible if necessary. Selecting an item also marks it as the item that **CurrentSelection()** returns and that the Enter key can invoke.

Select() is called whenever the user selects an item, using either the keyboard or the mouse. It can also be called from application code to set an initial selection in the list or change the current selection.

If a model "selection message" has been registered with the BListView, **Select()** copies the message, adds information to the copy identifying the list and the item that was selected, and posts the copy so that it will be dispatched to the target BReceiver. If a message hasn't been registered, "selecting" an item simply means to highlight it and mark is as the selected item.

Typically, BListViews are set up to post a message when an item is invoked, but not when one is selected.

See also: **SetSelectionMessage()**, **Invoke()**

SetFontName(), SetFontSize(), SetFontRotation(), SetFontShear(), SetFontSymbolSet()

virtual void **SetFontName(**const char **name***)**

virtual void **SetFontSize(**float *points***)**

virtual void **SetFontRotation(**float *degrees***)**

virtual void **SetFontShear(**float *angle***)**

virtual void **SetFontSymbolSet(**const char **name***)**

These functions augment their BView counterparts to update the BListView so that all visible items are redisplayed on-screen in the new font. However, **SetFontRotation()** is disabled; a rotated font is incompatible with a list horizontal items.

See also: SetFontName() in the BView class

SetInvocationMessage(), InvocationMessage(), InvocationCommand()

virtual void **SetInvocationMessage(**BMessage **message***)**

BMessage ***InvocationMessage(**void**)** const

ulong **InvocationCommand(**void**)** const

These functions set, and return information about, the BMessage that the BListView posts when an item is invoked.

SetInvocationMessage() assigns *message* to the BListView, freeing any message previously assigned. The message becomes the responsibility of the BListView object and will be freed only when it's replaced by another message or the BListView is freed; you shouldn't free it yourself. Passing a **NULL** pointer to this function deletes the current message without replacing it.

The BListView treats the BMessage as its "invocation message," a model for the message it posts when an item in the list is invoked. The **Invoke()** function makes a copy of the model and adds two pieces of relevant information. It then posts the copy, not the original.

The added information identifies the BListView and the invoked item:

Data name	Type code	Description
"source"	**OBJECT_TYPE**	A pointer to the BListView object.
"index"	**LONG_TYPE**	The index of the item that was invoked.

These names should not be used for any data that you add to the model *message*.

Given this information, the message receiver can get a pointer to item data. For example:

```
void myWindow::MessageReceived(BMessage *message)
{
    BListView *theList;
    long theIndex;
    char *theItem;
    . . .
    theList = (BListView *)message->FindObject("source");
    if ( message->Error() == NO_ERROR ) {
        theIndex = message->FindLong("index");
        if ( message->Error() == NO_ERROR ) {
            theItem = (char *)theList->ItemAt(theIndex);
            . . .
        }
    }
    . . .
 }
```

(Although not shown in this example, you might also want to use the **cast_as()** macro to make sure that it's safe to cast the "source" object pointer to the BListView class.)

InvocationMessage() returns a pointer to the model BMessage and **InvocationCommand()** returns its **what** data member. The message belongs to the BListView; it can be altered by adding or removing data, but it shouldn't be deleted. Nor should it be posted or sent anywhere, since that would eventually free it. To get rid of the current message, pass a **NULL** pointer to **SetInvocationMessage()**.

See also: **Invoke()**, the BMessage class

SetSelectionMessage(), SelectionMessage(), SelectionCommand()

virtual void **SetSelectionMessage(**BMessage **message***)**

BMessage ***SelectionMessage(**void**)** const

ulong **SelectionCommand(**void**)** const

These functions set, and return information about, the message that a BListView posts whenever one of its items is selected. They're exact counterparts to the invocation message functions described above under **SetInvocationMessage()**, except that the "selection message" is posted whenever an item in the list is selected, rather than when invoked. It's more common to take action (to post a message) on invoking an item than on selecting one.

The *message* that **SetSelectionMessage()** assigns to the BListView is a model for the messages that the **Select()** function posts. **Select()** copies the model and posts the copy.

It adds the same two pieces of information to the copy as are added to the invocation message:

Data name	Type code	Description
"source"	**OBJECT_TYPE**	A pointer to the BListView object.
"index"	**LONG_TYPE**	The index of the item that was selected.

You should not use these names for data you add to the model *message*.

See also: Select(), **SetInvocationMessage()**, the BMessage class

SetTarget(), Target()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper* = **NULL)**

BReceiver ***Target(**BLooper ***looper* = **NULL)** const

SetTarget() sets the *target* BReceiver that's expected to handle messages the BListView posts (though its **Select()** and **Invoke()** functions). It's successful only if it can also learn about a BLooper object where messages can be posted to the target. To post a message, the BListView calls the BLooper's **PostMessage()** function and names the *target* as the object that should receive the message:

```
looper->PostMessage(theMessage, target);
```

If the *target* receiver passed to **SetTarget()** is itself a BLooper object (such as a BWindow) or if it's associated with a BLooper object (as BViews are associated with BWindows), the *looper* argument can be **NULL**. **SetTarget()** can discover the BLooper from the *target* (by calling the *target*'s **Looper()** function).

However, if the *target* can't supply a BLooper object, a specific *looper* must be named as an argument. If a *looper* isn't named and can't be discovered from the *target*, the function fails and **BAD_VALUE** is returned to indicate that the target alone is insufficient.

Moreover, **SetTarget()** also fails if a specific *looper* is named but the *target* is associated with some other BLooper object. In this case, **MISMATCHED_VALUES** is returned to indicate that there's a conflict between the two arguments.

It's also possible to specify a **NULL** *target*. In this case, the message will be targeted to the *looper*'s preferred receiver (the object returned by its **PreferredReceiver()** function). For a BWindow, the preferred receiver is the current focus view. Therefore, by passing a **NULL** *target* and a BWindow *looper* to **SetTarget()**,

```
myList->SetTarget(NULL, myList->Window());
```

the BListView can be targeted to whatever BView happens to be in focus at the time an item is invoked.

Note, however, that if the *looper* doesn't have a preferred receiver (as a BLooper doesn't by default, and a BWindow won't if none of its views are currently in focus), the message will be targeted to the *looper* itself.

If both *target* and *looper* are **NULL**, the function fails and **BAD_VALUE** is returned. When successful, **SetTarget()** returns **NO_ERROR**.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where **Invoke()** will post messages. By default (established by **AttachedToWindow()**), both roles are filled by the BWindow where the list is displayed. If the BListView isn't attached to a window and a target hasn't been set, **Target()** returns **NULL**.

See also: **Looper()** in the BReceiver and BView classes, **PreferredReceiver()** in the BLooper and BWindow classes, **Invoke()**, **AttachedToWindow()**

BMenu

Derived from:	public BView
Declared in:	<interface/Menu.h>

Overview

A BMenu object displays a pull-down or pop-up list of menu items. Menus organize the features of an application—the common ones as well as the more obscure—and provide users with points of entry for most everything the application can do.

Menus categorize the features of the application—all formatting possibilities might be grouped in one menu, a list of documents in another, graphics choices in a third, and so on. The arrangement of menus presents an outline of how the various parts of the application fit together.

Menu Hierarchy

Menus are hierarchically arranged; an item in one menu can control another menu. The controlled menu is a *submenu*; the menu that contains the item that controls it is its *supermenu*. A submenu remains hidden until the user operates the item that controls it; it becomes hidden again when the user is finished with it. A submenu can have its own submenus, and those submenus can have submenus of their own, and so on—although it becomes hard for users to find their way around in a menu hierarchy that becomes too deep.

The menu at the root of the hierarchy is displayed in a window as a list—perhaps a list of just one item. Since it, unlike other menus, doesn't have a controlling item, it must remain visible. A root menu is therefore a special kind of menu in that it behaves more like an ordinary view than do other menus, which stay hidden. Root menus should belong to the BMenuBar class, which is derived from BMenu. The typical root menu is a menu bar displayed across the top of a window (hence the name of the class).

Menu Items

Each item in a menu is a kind of BMenuItem object. An item can be marked (displayed with a check mark to its left), assigned a keyboard shortcut, enabled and disabled, and given a "trigger" character that the user can type to invoke the item when its menu is open on-screen.

Every item has a particular job to do. If an item controls a submenu, its job is to show the submenu on-screen and hide it again. All other items give instructions to the application. When invoked by the user, they post a BMessage object to a target BReceiver. What the item does depends on the content of the BMessage and the BReceiver's response to it.

The BMenu and BMenuItem classes share some functions that accomplish the same thing when called for a submenu or for the supermenu item that controls the submenu. For example, setting the target for a BMenu (**SetTarget()**) sets the target for each of its items. Disabling a submenu (**SetEnabled()**) is the same as disabling the item that controls it; the user will be able to bring the submenu to the screen, but none of its items will work. This, in effect, disables all items and menus in the branch of the menu hierarchy under the superitem.

Hook Functions

ScreenLocation()	Can be implemented to have the menu appear on-screen at some location other than the default.

Constructor and Destructor

BMenu()

public:

BMenu(const char **name*, menu_layout *layout* = **ITEMS_IN_COLUMN)**
BMenu(const char **name*, float *width*, float *height***)**

protected:

BMenu(BRect *frame*, const char **name*, ulong *resizingMode*, ulong *flags*,
menu_layout *layout*, bool *resizeToFit***)**

Initializes the BMenu object. The *name* of the object becomes the initial label of the supermenu item that controls the menu and brings it to the screen. (It's also the view name that can be passed to BView's **FindView()** function.)

A new BMenu object doesn't contain any items; you need to call **AddItem()** to set up its contents.

A menu can arrange its items in any of three ways:

ITEMS_IN_COLUMN	The items are stacked vertically in a column, one on top of the other, as in a typical menu.
ITEMS_IN_ROW	The items are laid out horizontally in a row, from end to end, as in a typical menu bar.
ITEMS_IN_MATRIX	The items are arranged in a custom fashion, such as a matrix.

Either **ITEMS_IN_ROW** or the default **ITEMS_IN_COLUMN** can be passed as the *layout* argument to the public constructor. (A column is the default for ordinary menus; a row is the default for BMenuBars.) This version of the constructor isn't designed for **ITEMS_IN_MATRIX** layouts.

A BMenu object can arrange items that are laid out in a column or a row entirely on its own. The menu will be resized to exactly fit the items that are added to it.

However, when items are laid out in a custom matrix, the menu needs more help. First, the constructor must be informed of the exact *width* and *height* of the menu rectangle. The version of the constructor that takes these two parameters is designed just for matrix menus—it sets the layout to **ITEMS_IN_MATRIX**. Then, when items are added to the menu, the BMenu object expects to be informed of their precise positions within the specified area. The menu is *not* resized to fit the items that are added. Finally, when items in the matrix change, you must take care of any required adjustments in the layout yourself.

The protected version of the constructor is supplied for derived classes that don't simply devise different sorts of menu items or arrange them in a different way, but invent a different kind of menu. If the *resizeToFit* flag is **TRUE**, it's expected that the *layout* will be **ITEMS_IN_COLUMN** or **ITEMS_IN_ROW**. The menu will resize itself to fit the items that are added to it. If the layout is **ITEMS_IN_MATRIX**, the *resizeToFit* flag should be **FALSE**.

~BMenu()

virtual **~BMenu(**void**)**

Deletes all the items that were added to the menu and frees all memory allocated by the BMenu object. Deleting the items serves also to delete any submenus those items control and, thus, the whole branch of the menu hierarchy.

Member Functions

AddItem()

> bool **AddItem(**BMenuItem **item***)**
> bool **AddItem(**BMenuItem **item*, long *index***)**
> bool **AddItem(**BMenuItem **item*, BRect *frame***)**
> bool **AddItem(**BMenu **submenu***)**
> bool **AddItem(**BMenu **submenu*, long *index***)**
> bool **AddItem(**BMenu **submenu*, BRect *frame***)**

Adds an item to the menu list at *index*—or, if no *index* is mentioned, to the end of the list. If items are arranged in a matrix rather than a list, it's necessary to specify the item's *frame* rectangle—the exact position where it should be located in the menu view. Assume a coordinate system for the menu that has the origin, (0.0, 0.0), at the left top corner of the view rectangle. The rectangle will have the width and height that were specified when the menu was constructed.

The versions of this function that take an *index* (even an implicit one) can be used only if the menu arranges items in a column or row (**ITEMS_IN_COLUMN** or **ITEMS_IN_ROW**); it's an error to use them for items arranged in a matrix. Conversely, the versions of this function that take a *frame* rectangle can be used only if the menu arranges items in a matrix (**ITEMS_IN_MATRIX**); it's an error to use them for items arranged in a list.

If a *submenu* is specified rather than an *item*, **AddItem()** constructs a controlling BMenuItem for the submenu and adds the item to the menu.

If it's unable to add the item to the menu—for example, if the *index* is out-of-range or the wrong version of the function has been called—**AddItem()** returns **FALSE**. If successful, it returns **TRUE**.

See also: the BMenu constructor, the BMenuItem class, **RemoveItem()**

AddSeparatorItem()

> bool **AddSeparatorItem(**void**)**

Creates an instance of the BSeparatorItem class and adds it to the end of the menu list, returning **TRUE** if successful and **FALSE** if not (a very unlikely possibility). This function is a shorthand for:

```
BSeparatorItem *separator = new BSeparatorItem;
AddItem(separator);
```

A separator serves only to separate other items in the list. It counts as an item and has an indexed position in the list, but it doesn't do anything. It's drawn as a horizontal line across the menu. Therefore, it's appropriately added only to menus where the items are laid out in a column.

See also: **AddItem()**, the BSeparatorItem class

AreTriggersEnabled() see SetTriggersEnabled()

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Finishes initializing the BMenu object by setting graphics parameters and laying out items. This function is called for you each time the BMenu is assigned to a window. For a submenu, that means each time the menu is shown on-screen.

See also: **AttachedToWindow()** in the BView class

CountItems()

long **CountItems(**void**)** const

Returns the total number of items in the menu, including separator items.

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the menu. This function is called for you whenever the menu is placed on-screen or is updated while on-screen. It's not a function you need to call yourself.

See also: **Draw()** in the BView class

FindItem()

BMenuItem ***FindItem(**const char **label***)** const
BMenuItem ***FindItem(**ulong *command***)** const

Returns the item with the specified *label*—or the one that posts a message with the specified *command*. If there's more than one item in the menu with that particular *label* or associated with that particular *command*, this function returns the first one it finds (the one with the lowest index). If none of the items in the menu meet the criterion, it returns **NULL**.

FindMarked()

BMenuItem ***FindMarked(**void**)**

Returns the first marked item in the menu list (the one with the lowest index), or **NULL** if no item is marked.

See also: SetMarked() in the BMenuItem class, **SetRadioMode()**

Hide(), Show()

protected:

void **Hide(**void**)**

void **Show(**bool *selectFirst* = **FALSE)**

These functions hide the menu (remove the BMenu view from the window it's in and remove the window from the screen) and show it (attach the BMenu to a window and place the window on-screen). If the *selectFirst* flag passed to **Show()** is **TRUE**, the first item in the menu will be selected when it's shown.

These functions are not ones that you'd ordinarily call, even when implementing a derived class. You'd need them only if you're implementing a nonstandard menu of some kind and want to control when the menu appears on-screen.

See also: Track()

IndexOf()

long **IndexOf(**BMenuItem **item***)** const
long **IndexOf(**BMenu **submenu***)** const

Returns the index of the specified menu *item*—or the item that controls the specified *submenu*. Indices record the position of the item in the menu list. They begin at 0 for the item at the top of a column or at the left of a row and include separator items.

If the menu doesn't contain the specified *item*, or the item that controls *submenu*, the return value will be **SYS_ERROR**.

See also: AddItem()

InvalidateLayout()

void **InvalidateLayout(**void**)**

Forces the BMenu to recalculate the layout of all menu items and, consequently, its own size. It can do this only if the items are arranged in a row or a column. If the items are arranged in a matrix, it's up to you to keep their layout up-to-date.

All BMenu and BMenuItem functions that change an item in a way that might affect the overall menu automatically invalidate the menu's layout so it will be recalculated. For example, changing the label of an item might cause the menu to become wider (if it needs more room to accommodate the longer label) or narrower (if it no longer needs as much room as before).

Therefore, you don't need to call **InvalidateLayout()** after using a Kit function to change a menu or menu item; it's called for you. You'd call it only when making some other change to a menu.

See also: the BMenu constructor

IsEnabled() see SetEnabled()

IsLabelFromMarked() see SetLabelFromMarked()

IsRadioMode() see SetRadioMode()

ItemAt(), SubmenuAt()

BMenuItem ***ItemAt(**long *index***)** const

BMenu ***SubmenuAt(**long *index***)** const

These functions return the item at *index*—or the submenu controlled by the item at *index*. If there's no item at the index, they return **NULL**. **SubmenuAt()** is a shorthand for:

```
ItemAt(index)->Submenu()
```

It returns **NULL** if the item at *index* doesn't control a submenu.

See also: **AddItem()**

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Handles keyboard navigation through the menu. This function is called as the result of key-down events. It should not be called from application code.

See also: **KeyDown()** in the BView class

Layout()

protected:

  menu_layout **Layout(**void**)** const

Returns **ITEMS_IN_COLUMN** if the items in the menu are stacked in a column from top to bottom, **ITEMS_IN_ROW** if they're stretched out in a row from left to right, or **ITEMS_IN_MATRIX** if they're arranged in some custom fashion. By default BMenu items are arranged in a column and BMenuBar items in a row.

The layout is established by the constructor.

See also: the BMenu and BMenuBar constructors

RemoveItem()

  BMenuItem ***RemoveItem(**long *index***)**
  bool **RemoveItem(**BMenuItem **item***)**
  bool **RemoveItem(**BMenu **submenu***)**

Removes the item at *index*, or the specified *item*, or the item that controls the specified *submenu*. Removing the item doesn't free it.

- If passed an *index*, this function returns a pointer to the item so you can free it. It returns a **NULL** pointer if the item couldn't be removed (for example, if the *index* is out-of-range).
- If passed an *item*, it returns **TRUE** if the item was in the list and could be removed, and **FALSE** if not.
- If passed a *submenu*, it returns **TRUE** if the submenu is controlled by an item in the menu and that item could be removed, and **FALSE** otherwise.

When an item is removed from a menu, it loses its target; the cached value is set to **NULL**. If the item controls a submenu, it remains attached to the submenu even after being removed.

See also: **AddItem()**

ScreenLocation()

protected:

  virtual BPoint **ScreenLocation(**void**)**

Returns the point where the left top corner of the menu should appear when the menu is shown on-screen. The point is specified in the screen coordinate system.

This function is called each time a hidden menu (a submenu of another menu) is brought to the screen. It can be overridden in a derived class to change where the menu appears.

For example, the BPopUpMenu class overrides it so that a pop-up menu pops up over the controlling item.

See also: the BPopUpMenu class

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *flag***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenu if *flag* is **TRUE**, and disables it if *flag* is **FALSE**. If the menu is a submenu, this enables or disables its controlling item, just as if **SetEnabled()** were called for that item. The controlling item is updated so that it displays its new state, if it happens to be visible on-screen.

Disabling a menu disables its entire branch of the menu hierarchy. All items in the menu, including those that control other menus, are disabled.

IsEnabled() returns **TRUE** if the BMenu, and every BMenu above it in the menu hierarchy, is enabled. It returns **FALSE** if the BMenu, or any BMenu above it in the menu hierarchy, is disabled.

See also: **SetEnabled()** in the BMenuItem class

SetLabelFromMarked(), IsLabelFromMarked()

protected:

void **SetLabelFromMarked(**bool *flag***)**

bool **IsLabelFromMarked(**void**)**

SetLabelFromMarked() determines whether the label of the item that controls the menu (the label of the superitem) should be taken from the currently marked item within the menu. If *flag* is **TRUE**, the menu is placed in radio mode and the superitem's label is reset each time the user selects a different item. If *flag* is **FALSE**, the setting for radio mode doesn't change and the label of the superitem isn't automatically reset.

IsLabelFromMarked() returns whether the superitem's label is taken from the marked item (but not necessarily whether the BMenu is in radio mode).

See also: **SetRadioMode()**

SetRadioMode(), IsRadioMode()

virtual void **SetRadioMode(**bool *flag***)**

bool **IsRadioMode(**void**)**

SetRadioMode() puts the BMenu in radio mode if *flag* is **TRUE** and takes it out of radio mode if *flag* is **FALSE**. In radio mode, only one item in the menu can be marked at a time. If the user selects an item, a check mark is placed in front of it automatically (you don't need to call BMenuItem's **SetMarked()** function; it's called for you). If another item was marked at the time, its mark is removed. Selecting a currently marked item retains the mark.

IsRadioMode() returns whether the BMenu is currently in radio mode. The default radio mode is **FALSE** for ordinary BMenus, but **TRUE** for BPopUpMenus.

SetRadioMode() doesn't change any of the items in the menu. If you want an initial item to be marked when the menu is put into radio mode, you must mark it yourself.

When **SetRadioMode()** turns radio mode off, it calls **SetLabelFromMarked()** and passes it an argument of **FALSE**—turning off the feature that changes the label of the menu's superitem each time the marked item changes. Similarly, when **SetLabelFromMarked()** turns on this feature, it calls **SetRadioMode()** and passes it an argument of **TRUE**—turning on radio mode.

See also: **SetMarked()** in the BMenuItem class, **SetLabelFromMarked()**

SetTarget()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper* = **NULL)**

This function is a convenience for assigning the same *target* and *looper* to all items in the menu. It works through the list of items in order, calling BMenuItem's **SetTarget()** virtual function for each one. However, if it's unable to set the target of any item, it aborts and returns the error it encountered. If successful in setting the *target* (and *looper*) of all items, it returns **NO_ERROR**. See BMenuItem's **SetTarget()** for information on acceptable *target* and *looper* values.

This function doesn't work recursively; it acts only on items added to the BMenu, not on items added to submenus of the BMenu.

See also: **SetTarget()** in the BMenuItem class

SetTriggersEnabled(), AreTriggersEnabled()

virtual void **SetTriggersEnabled(**bool *flag***)**

bool **AreTriggersEnabled(**void**)** const

SetTriggersEnabled() enables the triggers for all items in the menu if *flag* is **TRUE** and disables them if *flag* is **FALSE**. **AreTriggersEnabled()** returns whether the triggers are currently enabled or disabled. They're enabled by default.

Triggers are displayed to the user only if they're enabled, and only when keyboard actions can operate the menu.

Triggers are appropriate for some menus, but not for others. **SetTriggersEnabled()** is typically called to initialize the BMenu when it's constructed, not to enable and disable triggers as the application is running. If triggers are ever enabled for a menu, they should always be enabled; if they're ever disabled, they should always be disabled.

See also: SetTrigger() in the BMenuItem class

Show() see Hide()

SubmenuAt() see ItemAt()

Superitem(), Supermenu()

BMenuItem ***Superitem(**void**)** const

BMenu ***Supermenu(**void**)** const

These functions return the supermenu item that controls the BMenu and the supermenu where that item is located. The supermenu could be a BMenuBar object. If the BMenu hasn't been made the submenu of another menu, both functions return **NULL**.

See also: AddItem()

Track()

protected:

BMenuItem ***Track(**void**)**

Initiates tracking of the cursor within the menu. This function passes tracking control to submenus (and submenus of submenus) depending on where the user moves the mouse. If the user ends tracking by invoking an item, **Track()** returns the item. If the user didn't invoke any item, it returns **NULL**. The item doesn't have to be located in the BMenu; it could, for example, belong to a submenu of the BMenu.

Track() is called by the BMenu to initiate tracking in the menu hierarchy. You would need to call it yourself only if you're implementing a different kind of menu that starts to track the cursor under nonstandard circumstances.

BMenuBar

Derived from:	public BMenu
Declared in:	<interface/MenuBar.h>

Overview

A BMenuBar is a menu that can stand at the root of a menu hierarchy. Rather than appear on-screen when commanded to do so by a user action, a BMenuBar object has a settled location in a window's view hierarchy, just like other views. Typically, the root menu is the menu bar that's drawn across the top of the window. It's from this use that the class gets its name.

However, instances of this class can also be used in other ways. A BMenuBar might simply display a list of items arranged in a column somewhere in a window. Or it might contain just one item, where that item controls a pop-up menu (a BPopUpMenu object). Rather than look like a "menu bar," the BMenuBar object would look something like a button.

The "Main" Menu Bar

The "real" menu bar at the top of the window usually represents an extensive menu hierarchy; each of its items typically controls a submenu.

The user should be able to operate this menu bar from the keyboard (using the arrow keys and Enter). There are two ways that the user can put the BMenuBar and its hierarchy in focus for keyboard events:

- Clicking an item in a menu bar. This opens the submenu the item controls so that it stays visible on-screen and puts the submenu in focus.
- Pressing the Menu key, or pressing and releasing a Command key. This puts the BMenuBar in focus and selects its first item.

Either method opens the entire menu hierarchy to keyboard navigation.

If there's only one BMenuBar in the window's view hierarchy, the Menu key (or Command) will put it in focus. But if there's more than one BMenuBar object, the Menu key must choose one of them. By default, it selects the last one added to the window. However, the **SetMainMenuBar()** function defined in the BWindow class can be called to designate a different BMenuBar object as the "main" menu bar for the window.

A Kind of BMenu

BMenuBar inherits most of its functions from the BMenu class. It reimplements the **AttachedToWindow()**, **Draw()**, and **MouseDown()** functions that set up the object and respond to messages, but these aren't functions that you'd call from application code; they're called for you.

The only real function (other than the constructor) that the BMenuBar class adds to those it inherits is **SetBorder()**, which determines how the list of items is bordered.

Therefore, for most BMenuBar operations—adding submenus, finding items, temporarily disabling the menu bar, and so on—you must call inherited functions and treat the object like the BMenu that it is.

See also: the BMenu class

Constructor and Destructor

BMenuBar()

BMenuBar(BRect *frame*, const char **name*,
ulong *resizingMode* = **FOLLOW_LEFT_TOP_RIGHT**,
menu_layout *layout* = **ITEMS_IN_ROW**,
bool *resizeToFit* = **FALSE)**

Initializes the BMenuBar by assigning it a *frame* rectangle, a *name*, and a *resizingMode*, just like other BViews. These values are passed up the inheritance hierarchy to the BView constructor. The "real" menu bar in a window should have a frame rectangle just high enough to accommodate a single row of items and a border. Given the default font currently used for menu items, the *frame* height should be about 14.0 coordinate units.

The *layout* of the menu determines how items are arranged. By default, they're arranged in a row as befits a true menu bar. If an instance of this class isn't being used to implement an actual menu bar, items can be laid out in a column (**ITEMS_IN_COLUMN**) or in a matrix (**ITEMS_IN_MATRIX**).

If the *resizeToFit* flag is **TRUE**, the frame rectangle of the BMenuBar will be resized to exactly fit the items that are added to the object. This usually is not what's desired. For a true menu bar, the frame rectangle should stretch all the way across the window, from the left side to the right, no matter how many items it contains. The default resizing mode of **FOLLOW_LEFT_TOP_RIGHT** permits the menu bar to adjust itself to changes in the window's width, while keeping it glued to the top of the window.

Change the *resizingMode*, the *layout*, and the *resizeToFit* flag for BMenuBars that are used for a purpose other than to implement a true menu bar.

See also: the BMenu constructor

~BMenuBar()

virtual **~BMenuBar(**void**)**

Frees all the items and submenus in the entire menu hierarchy, and all memory allocated by the BMenuBar.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Finishes the initialization of the BMenuBar by setting up its graphics environment, and by making the BWindow to which it has become attached the target receiver for all items in the menu hierarchy, except for those items for which a target has already been set.

This function also makes the BMenuBar the "main menu bar," the BMenuBar object whose menu hierarchy the user can navigate from the keyboard. If a window contains more than one BMenuBar in its view hierarchy, the last one that's added to the window gets to keep this designation. However, the "main" menu bar should always be the real menu bar at the top of the window. It can be explicitly set with BWindow's **SetMainMenuBar()** function.

See also: SetMainMenuBar() in the BWindow class

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the menu—whether as a true menu bar, as some other kind of menu list, or as a single item that controls a pop-up menu. This function is called as the result of update messages; you don't need to call it yourself.

See also: Draw() in the BView class

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Initiates mouse tracking and keyboard navigation of the menu hierarchy. This function is called when the BMenuBar is notified of a mouse-down event.

See also: MouseDown() in the BView class

SetBorder()

void **SetBorder(**ulong *border***)**

Determines how the menu list is bordered. The *border* argument can be:

BORDER_FRAME	The border is drawn around the entire frame rectangle.
BORDER_CONTENTS	The border is drawn around just the list of items.
BORDER_EACH_ITEM	A border is drawn around each item.

The default is **BORDER_FRAME**.

BMenuItem

Derived from:	public BObject
Declared in:	<interface/MenuItem.h>

Overview

A BMenuItem is an object that contains and displays one item within a menu. By default, Menu items are displayed simply as textual labels, like “Options...” or “Save As”. Derived classes can be defined to draw something other than a label—or something in addition to the label.

Kinds of Items

Some menu items play a role in helping users navigate the menu hierarchy. They give the user access to submenus. A submenu remains hidden until the user operates the item that controls it.

Other items accomplish specific actions. When the user invokes the item, a message is posted to a target BReceiver, usually the window where the menu at the root of the hierarchy (a BMenuBar object) is displayed. The action that the item initiates, or the state that it sets, depends entirely on the message and the receiver’s response to it.

The target receiver and the message can be customized for every item. Each BMenuItem retains a model for the BMessage it posts and can have a target that’s different from other items in the same menu.

Items can also have a visual presence, but do nothing. Instances of the BSeparatorItem class, which is derived from BMenuItem, serve only to visually separate groups of items in the menu.

Shortcuts and Triggers

Any menu item (except for those that control submenus) can be associated with a keyboard shortcut, a character that the user can type in combination with the Command key (and possibly other modifiers) to invoke the item. The shortcut character is displayed in the menu item to the right of the label. All shortcuts for menu items require the user to hold down the Command key.

A shortcut works even when the item it invokes isn't visible on-screen. It, therefore, has to be unique within the window (within the entire menu hierarchy).

Every menu item is also associated with a *trigger*, a character that the user can type (without the Command key) to invoke the item. The trigger works only while the menu is both open on-screen and can be operated using the keyboard. It therefore must be unique only within a particular branch of the menu hierarchy (within the menu).

The trigger is one of the characters that's displayed within the item—either the keyboard shortcut or a character in the label. When it's possible for the trigger to invoke the item, the character is drawn in a distinctive color. Like shortcuts, triggers are case-insensitive.

For an item to have a keyboard shortcut, the application must explicitly assign one when constructing the object. However, by default, the Interface Kit chooses and assigns triggers for all items. The default choice can be altered by the **SetTrigger()** function.

Marked Items

An item can also be marked (with a check mark drawn to the left of the label) in order to indicate that the state it sets is currently in effect. Items are marked by the **SetMarked()** function. A menu can be set up so that items are automatically marked when they're selected and exactly one item is marked at all times. (See **SetRadioMode()** in the BMenu class.)

Disabled Items

Items can also be enabled or disabled (by the **SetEnabled()** function). A disabled item is drawn in muted tones to indicate that it doesn't work. It can't be selected or invoked. If the item controls a specific action, it won't post the message that initiates the action. If it controls a submenu, it will still bring the submenu to the screen, but all the items in submenu will be disabled. If an item in the submenu brings its own submenu to the screen, items in that submenu will also be disabled. Disabling the superitem for a submenu in effect disables a whole branch of the menu hierarchy.

See also: the BMenu class, the BSeparatorItem class

Hook Functions

All BMenuItem hook functions are protected. They should be implemented only if you design a special type of menu item that displays something other than a textual label.

Draw()	Draws the entire item; can be reimplemented to draw the item in a different way.
DrawContents()	Draws the item label; can be reimplemented to draw something other than a label.
GetContentSize()	Provides the width and height of the item's content area, which is based on the length of the label and the current font; can be reimplemented to provide the size required to draw something other than a label.
Highlight()	Highlights the item when it's selected; can be reimplemented to do highlighting in some way other than the default.

Constructor and Destructor

BMenuItem()

BMenuItem(const char **label*, BMessage **message*,
char *shortcut* = **NULL**, ulong *modifiers* = **NULL)**
BMenuItem(BMenu **submenu***)**

Initializes the BMenu to display *label* (which can be **NULL** if the item belongs to a derived class that's designed to display something other than text) and assigns it a model *message*.

Whenever the user invokes the item, the model message is copied and the copy is posted to the target receiver. Three pieces of information are added to the copy before it's posted:

Data name	Type code	Description
"when"	**LONG_TYPE**	The time the item was invoked, as measured in milliseconds since the machine was last booted.
"source"	**OBJECT_TYPE**	A pointer to the BMenuItem object.
"index"	**LONG_TYPE**	The index of the item, its ordinal position in the menu. Indices begin at 0.

These names should not be used for any data that you place in the *message*.

By default, the target of the message is the window associated with the item's menu hierarchy—the window where the BMenuBar at the root of the hierarchy is located. Another target can be designated by calling the **SetTarget()** function.

The constructor can also optionally set a keyboard shortcut for the item. The character that's passed as the *shortcut* parameter will be displayed to the right of the item's label. It's the accepted practice to display uppercase shortcut characters only, even though the actual character the user types may not be uppercase.

The *modifiers* mask, not the *shortcut* character, determines which modifier keys the user must hold down for the shortcut to work—including whether the Shift key must be down. The mask can be formed by combining any of the modifiers constants, especially these:

SHIFT_KEY
CONTROL_KEY
OPTION_KEY
COMMAND_KEY

However, **COMMAND_KEY** is required for all keyboard shortcuts; it doesn't have to be explicitly included in the mask. For example, setting the *shortcut* to 'U' with no *modifiers* would mean that the letter 'U' would be displayed alongside the item label and Command-*u* would invoke the item. The same *shortcut* with a **SHIFT_KEY** *modifiers* mask would mean that the uppercase character (Command-Shift-*U*) would invoke the item.

If the BMenuItem is constructed to control a *submenu*, it doesn't post messages—its role is to bring up the submenu—and it can't take a shortcut. The item's initial label will be taken from the name of the submenu. It can be changed after construction by calling **SetLabel()**.

See also: SetTarget(), SetMessage(), SetLabel()

~BMenuItem()

virtual **~BMenuItem(**void**)**

Frees the item's label and its model BMessage object. If the item controls a submenu, that menu and all its items are also freed. Deleting a BMenuItem destroys the entire menu hierarchy under that item.

Member Functions

Command() see **SetMessage()**

ContentLocation()

protected:

BPoint **ContentLocation(**void**)** const

Returns the left top corner of the content area of the item, in the coordinate system of the BMenu to which it belongs. The content area of an item is the area where it displays its label (or whatever graphic substitutes for the label). It doesn't include the part of the item where a check mark or a keyboard shortcut could be displayed, nor the border and background around the content area.

You would need to call this function only if you're implementing a **DrawContent()** function to draw the contents of the menu item (likely something other than a label). The content rectangle can be calculated from the point returned by this function and the size specified by **GetContentSize()**.

If the item isn't part of a menu, the return value is indeterminate.

See also: GetContentSize(), **DrawContent()**

Draw(), DrawContent()

protected:

virtual void **Draw(**void**)**

virtual void **DrawContent(**void**)**

These functions draw the menu item and highlight it if it's currently selected. They're called by the **Draw()** function of the BMenu where the item is located whenever the menu is required to display itself; they don't need to be called from within application code.

However, they can both be overridden by derived classes that display something other than a textual label. The **Draw()** function is called first. It draws the background for the entire item, then calls **DrawContent()** to draw the label within the item's content area. After **DrawContent()** returns, it draws the check mark (if the item is currently marked) and the keyboard shortcut (if any). It finishes by calling **Highlight()** if the item is currently selected.

Both functions draw by calling functions of the BMenu in which the item is located. For example:

```
void MyItem::DrawContent()
{
    . . .
    Menu()->DrawBitmap(image);
    . . .
}
```

A derived class can override either **Draw()**, if it needs to draw the entire item, or **DrawContent()**, if it needs to draw only within the content area. A **Draw()** function can find the frame rectangle it should draw within by calling the BMenuItem's **Frame()** function; a **DrawContent()** function can calculate the content area from the point returned by **ContentLocation()** and the dimensions provided by **GetContentSize()**.

When **DrawContent()** is called, the pen is positioned to draw the item's label and the front color is appropriately set. The front color may be a shade of gray, if the item is disabled, or black if it's enabled. If some other distinction is used to distinguish disabled from enabled items, **DrawContent()** should check the item's current state by calling **IsEnabled()**.

Note: If a derived class implements its own **DrawContent()** function, but still want to draw a textual string, it should do so by assigning the string as BMenuItem's label and calling the inherited version of **DrawContent()**, not by calling **DrawString()**. This preserves the BMenuItem's ability to display a trigger character in the string.

See also: Highlight(), **Frame()**, **ContentLocation()**, **GetContentSize()**

Frame()

BRect **Frame(**void**)** const

Returns the rectangle that frames the entire menu item, in the coordinate system of the BMenu to which the item belongs. If the item hasn't been added to a menu, the return value is indeterminate.

See also: AddItem() in the BMenu class

GetContentSize()

protected:
virtual void **GetContentSize(**float *width, float *height**)**

Writes the size of the item's content area into the variables referred to by *width* and *height*. The content area of an item is the area where its label (or whatever substitutes for the label) is drawn.

A BMenu calls **GetContentSize()** for each of its items as it arranges them in a column or a row; the function is not called for items in a matrix. The information it provides helps determine where each item is located and the overall size of the menu.

GetContentSize() must report a size that's large enough to display the content of the item (and separate one item from another). By default, it reports an area just large enough to display the item's label. This area is calculated from the label and the BMenu's current font.

If you design a class derived from BMenuItem and implement your own **Draw()** or **DrawContent()** function, you should also implement a **GetContentSize()** function to report how much room will be needed to draw the item's contents.

See also: DrawContent(), **ContentLocation()**

Highlight()

protected:

virtual void **Highlight(**bool *flag***)**

Highlights the menu item when *flag* is **TRUE**, and removes the highlighting when *flag* is **FALSE**. Highlighting simply inverts all the colors in the item's frame rectangle (except for the check mark).

This function is called by the **Draw()** function whenever the item is selected and needs to be drawn in its highlighted state. There's no reason to call it yourself, unless you define your own version of **Draw()**. However, it can be reimplemented in a derived class, if items belonging to that class need to be highlighted in some way other than simple inversion.

See also: Draw()

IsEnabled() see SetEnabled()

isMarked() see SetMarked()

IsSelected()

protected:

bool **IsSelected(**void**)** const

Returns **TRUE** if the menu item is currently selected, and **FALSE** if not. Selected items are highlighted.

Label() see SetLabel()

Menu()

BMenu ***Menu(**void**)** const

Returns the menu where the item is located, or **NULL** if the item hasn't yet been added to a menu.

See also: AddItem() in the BMenu class

Message() see SetMessage()

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *flag***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenuItem if *flag* is **TRUE**, disables it if *flag* is **FALSE**, and updates the item if it's visible on-screen. If the item controls a submenu, this function calls the submenu's **SetEnabled()** virtual function, passing it the same *flag*. This ensures that the submenu is enabled or disabled as well.

IsEnabled() returns **TRUE** if the BMenuItem is enabled, its menu is enabled, and all menus above it in the hierarchy are enabled. It returns **FALSE** if the item is disabled or any objects above it in the menu hierarchy are disabled.

Items and menus are enabled by default.

When using these functions, keep in mind that:

- Disabling a BMenuItem that controls a submenu serves to disable the entire menu hierarchy under the item.
- Passing an argument of **TRUE** to **SetEnabled()** is not sufficient to enable the item if it's located in a disabled branch of the menu hierarchy. It can only undo a previous **SetEnabled()** call (with an argument of **FALSE**) on the same item.

See also: SetEnabled() in the BMenu class

SetLabel(), Label()

virtual void **SetLabel(**const char **string***)**

const char ***Label(**void**)** const

SetLabel() frees the item's current label and copies *string* to replace it. If the menu is visible on-screen, it will be redisplayed with the item's new label. If necessary, the menu will become wider (or narrower) so that it fits the new label.

The Interface Kit calls this virtual function to:

- Set the initial label of an item that controls a submenu to the name of the submenu, and
- Subsequently set the item's label to match the marked item in the submenu, if the submenu was set up to have this feature.

Label() returns a pointer to the current label.

See also: **SetLabelFromMarked()** in the BMenu class, the BMenuItem constructor

SetMarked(), IsMarked()

virtual void **SetMarked(**bool *flag***)**

bool **IsMarked(**void**)** const

SetMarked() adds a check mark to the left of the item label if *flag* is **TRUE**, or removes an existing mark if *flag* is **FALSE**. If the menu is visible on-screen. it's redisplayed with or without the mark.

IsMarked() returns whether the item is currently marked.

See also: **SetLabelFromMarked()** and **FindMarked()** in the BMenu class

SetMessage(), Message(), Command()

virtual void **SetMessage(**BMessage **message***)**

BMessage ***Message(**void**)** const

ulong **Command(**void**)** const

SetMessage() makes *message* the model BMessage for the menu item, deleting any previous message assigned to the item. The model message is first set by the BMenuItem constructor; **SetMessage()** allows you to change the message in midstream. You might need to change it, for example, when the item's label changes.

When a menu item is invoked, its model message is copied, relevant information is added to the copy, and the copy is posted to the target BReceiver. (The information that gets added to the copy is described under the BMenuItem constructor.)

Message() returns a pointer to the BMenuItem's model message and **Command()** returns its **what** data member. If the BMenuItem doesn't post a message—if, for example, it controls a submenu or is a separator item—both functions return **NULL**.

The BMessage that **Message()** returns belongs to the BMenuItem. You can modify it by adding and removing data, but you shouldn't delete it or do anything that will cause it to be deleted. In particular, you shouldn't post or send the message anywhere, since that would transfer ownership to a message loop and subject the message to automatic deletion.

It's possible to set and return a model BMessage for a separator item or an item that controls a submenu. However, the message will never be used.

See also: the BMenuItem constructor, **SetTarget()**

SetTarget(), Target()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper* = **NULL)**

BReceiver ***Target(**BLooper ***looper* = **NULL)** const

These functions set and return the object that's targeted to receive messages posted by the BMenuItem.

SetTarget() sets the *target* BReceiver, but is successful only if it can also discern a BLooper object where the BMenuItem can post messages to that target. The BMenuItem calls the BLooper's **PostMessage()** function and names the *target* as the object that should receive the message:

```
looper->PostMessage(theMessage, target);
```

If the *target* receiver passed to **SetTarget()** is itself a BLooper object (such as a BWindow) or if it's associated with a BLooper object (as BViews are associated with BWindows), the *looper* argument can be **NULL**. **SetTarget()** can discover the BLooper from the *target* (by calling the *target*'s **Looper()** function).

However, if the *target* can't supply a BLooper object, a specific *looper* must be named as an argument. If a *looper* isn't named and the *target* can't supply one, the function fails and returns **BAD_VALUE** to indicate that the *target* alone is insufficient.

Moreover, it also fails if a specific *looper* is named but the *target* is associated with some other BLooper object. **MISMATCHED_VALUES** is returned to indicate that there's a conflict between the two arguments.

It's also possible to name a specific *looper*, but a **NULL** *target*. In this case, messages will be targeted to the *looper*'s preferred receiver (the object returned by its **PreferredReceiver()** function). For a BWindow, the preferred receiver is the current focus view. Therefore, by passing a **NULL** *target* and a BWindow *looper* to **SetTarget()**,

```
myItem->SetTarget(NULL, myItem->Window());
```

the BMenuItem can be targeted to whatever BView happens to be in focus at the time it's invoked. This is useful for items like "Cut" and "Copy" that act on the current selection. (Note, however, that if the **PreferredReceiver()** is **NULL**—if there's no current focus view—the BWindow itself will be the target.)

At least one of the two arguments must point to a real object. If both *target* and *looper* are **NULL**, **SetTarget()** fails and returns **BAD_VALUE**. When successful, it returns **NO_ERROR**.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where the item will post messages. By default, both roles are filled by the BWindow at the root of the menu hierarchy (the BWindow where the menu bar is located). These defaults are established when the BMenuItem becomes part of a menu hierarchy that's rooted in a window, but only if another *target* (or *looper*) hasn't already been set. If a target hasn't been set and the BMenuItem isn't part of a rooted menu hierarchy, **Target()** returns **NULL**.

See also: **Looper()** in the BReceiver and BView classes, **PreferredReceiver()** in the BLooper and BWindow classes

SetTrigger(), Trigger()

virtual void **SetTrigger(**char *trigger***)**

char **Trigger(**void**)** const

SetTrigger() sets the *trigger* character that the user can type to invoke the item while the item's menu is open on-screen. If a *trigger* is not set, the Interface Kit will select one for the item, so it's not necessary to call **SetTrigger()**.

The character passed to this function has to match a character displayed in the item—either the keyboard shortcut or a character in the label. The case of the character doesn't matter; lowercase arguments will match uppercase characters in the item and uppercase arguments will match lowercase characters. When the item can be invoked by its trigger, the trigger character is drawn in an eye-catching color.

If more than one character in the item matches the character passed, **SetTrigger()** tries first to mark the keyboard shortcut. Failing that, it tries to mark an uppercase letter at the beginning of a word. Failing that, it marks the first instance of the character in the label.

If the *trigger* doesn't match any characters in the item, the item won't have a trigger, not even one selected by the system.

Trigger() returns the character set by **SetTrigger()**, or **NULL** if **SetTrigger()** didn't succeed or if **SetTrigger()** was never called and the trigger is selected automatically.

See also: **SetTriggersEnabled()** in the BMenu class

Shortcut()

char **Shortcut(**ulong **modifiers* = **NULL)** const

Returns the character that's used as the keyboard shortcut for invoking the item, and writes a mask of all the modifier keys the shortcut requires to the variable referred to by *modifiers*. Since the Command key is required to operate the keyboard shortcut for any menu item, **COMMAND_KEY** will always be part of the *modifiers* mask. The mask can also be tested against the **CONTROL_KEY**, **OPTION_KEY**, and **SHIFT_KEY** constants.

The shortcut is set by the BMenuItem constructor.

See also: the BMenuItem constructor

Submenu()

BMenu ***Submenu(**void**)** const

Returns the BMenu object that the item controls, or **NULL** if the item doesn't control a submenu.

See also: the BMenuItem constructor, the BMenu class

Target() see SetTarget()

Trigger() see SetTrigger()

BPoint

Derived from:	*none*
Declared in:	<interface/Point.h>

Overview

BPoint objects represent points on a two-dimensional coordinate grid. Each object holds an *x* coordinate value and a *y* coordinate value declared as public data members. These values locate a specific point, (*x*, *y*), relative to a given coordinate system.

Because the BPoint class defines a basic data type for graphic operations, its data members are publicly accessible and it declares no virtual functions. It's a simple class that doesn't inherit from BObject or any other class and doesn't retain class information that it can reveal at run time. In the Interface Kit, BPoint objects are typically passed and returned by value, not through pointers.

For an introduction to coordinate geometry on the Be machine, see "The Coordinate Space" on page 14.

Data Members

float **x**	The coordinate value measured horizontally along the *x*-axis.
float **y**	The coordinate value measured vertically along the *y*-axis.

Constructor

BPoint()

inline **BPoint(**float *x*, float *y***)**
inline **BPoint(**const BPoint& *point***)**
inline **BPoint(**void**)**

Initializes a new BPoint object to (*x*, *y*), or to the same values as *point*. For example:

```
BPoint somePoint(155.7, 336.0);
BPoint anotherPoint(somePoint);
```

Here, both *somePoint* and *anotherPoint* are initialized to (155.7, 336.0).

If no coordinate values are assigned to the BPoint when it's declared,

```
BPoint emptyPoint;
```

its initial values are indeterminate.

BPoint objects can also be initialized or modified using the **Set()** function,

```
emptyPoint.Set(155.7, 336.0);
anotherPoint.Set(221.5, 67.8);
```

or the assignment operator:

```
somePoint = anotherPoint;
```

See also: **Set()**, the assignment operator

Member Functions

ConstrainTo()

void **ConstrainTo(**BRect *rect***)**

Constrains the point so that it lies inside the *rect* rectangle. If the point is already contained in the rectangle, it remains unchanged. However, if it falls outside the rectangle, it's moved to the nearest edge. For example, this code

```
BPoint point(54.9, 76.3);
BRect rect(10.0, 20.0, 40.0, 80.0);
point.Constrain(rect);
```

modifies the point to (40.0, 76.3).

See also: **Contains()** in the BRect class

PrintToStream()

void **PrintToStream(**void**)** const

Prints the contents of the BPoint object to the standard output stream (**stdout**) in the form:

```
"BPoint(x, y)"
```

where *x* and *y* stand for the current values of the BPoint's data members.

Set()

inline void **Set(**float *x*, float *y***)**

Assigns the coordinate values *x* and *y* to the BPoint object. For example, this code

```
BPoint point;
point.Set(27.0, 53.4);
```

is equivalent to:

```
BPoint point;
point.x = 27.0;
point.y = 53.4;
```

See also: the BPoint constructor

Operators

= (assignment)

inline BPoint& **operator =(**const BPoint&**)**

Assigns the *x* and *y* values of one BPoint object to another BPoint:

```
BPoint a(21.5, 17.0);
BPoint b = a;
```

Point *b*, like point *a*, is set to (21.5, 17.0).

== (equality)

bool **operator ==(**const BPoint&**)** const

Compares the data members of two BPoint objects and returns **TRUE** if each one exactly matches its counterpart in the other object, and **FALSE** if not. In the following example, the equality operator would return **FALSE**:

```
BPoint a(21.5, 17.0);
BPoint b(17.5, 21.0);
if ( a == b )
    . . .
```

!= (inequality)

bool **operator !=(**const BPoint&**)** const

Compares two BPoint objects and returns **TRUE** unless their data members match exactly (the two points are the same), in which case it returns **FALSE**. This operator is the inverse of the **==** (equality) operator.

+ (addition)

BPoint **operator +(**const BPoint&**)** const

Combines two BPoint objects by adding the *x* coordinate of the second to the *x* coordinate of the first and the *y* coordinate of the second to the *y* coordinate of the first, and returns a BPoint object that holds the result. For example:

```
BPoint a(77.0, 11.0);
BPoint b(55.0, 33.0);
BPoint c = a + b;
```

Point *c* is initialized to (132.0, 44.0).

+= (addition and assignment)

BPoint& **operator +=(**const BPoint&**)**

Modifies a BPoint object by adding another point to it. As in the case of the **+** (addition) operator, the members of the second point are added to their counterparts in the first point:

```
BPoint a(77.0, 11.0);
BPoint b(55.0, 33.0);
a += b;
```

Point *a* is modified to (132.0, 44.0).

– (subtraction)

BPoint **operator –(**const BPoint&**)** const

Subtracts one BPoint object from another by subtracting the *x* coordinate of the second from the *x* coordinate of the first and the *y* coordinate of the second from the *y* coordinate of the first, and returns a BPoint object that holds the result. For example:

```
BPoint a(99.0, 66.0);
BPoint b(44.0, 88.0);
BPoint c = a - b;
```

Point *c* is initialized to (55.0, –22.0).

–= (subtraction and assignment)

BPoint& **operator –=(**const BPoint&**)**

Modifies a BPoint object by subtracting another point from it. As in the case of the **–** (subtraction) operator, the members of the second point are subtracted from their counterparts in the first point. For example:

```
BPoint a(99.0, 66.0);
BPoint b(44.0, 88.0);
a -= b;
```

Point *a* is modified to (55.0, –22.0).

BPolygon

Derived from:	public BObject
Declared in:	<interface/Polygon.h>

Overview

A BPolygon object represents a *polygon*—a closed, many-sided figure that describes an area within a two-dimensional coordinate system. It differs from a BRect object in that it can have any number of sides and the sides don't have to be aligned with the coordinate axes.

A BPolygon is defined as a series of connected points. Each point is a potential vertex in the polygon. An outline of the polygon could be constructed by tracing a straight line from the first point to the second, from the second point to the third, and so on through the whole series, then by connecting the first and last points if they're not identical.

The BView functions that draw a polygon—**StrokePolygon()** and **FillPolygon()**—take BPolygon objects as arguments.

Constructor and Destructor

BPolygon()

BPolygon(BPoint **pointList*, long *numPoints***)**
BPolygon(const BPolygon **polygon***)**
BPolygon(void**)**

Initializes the BPolygon by copying *numPoints* from *pointList*, or by copying the list of points from another *polygon*. If one polygon is constructed from another, the original and the copy won't share any data; independent memory is allocated for the copy to hold a duplicate list of points.

If a BPolygon is constructed without a point list, points must be set with the **AddPoints()** function.

See also: AddPoints()

~BPolygon()

virtual **~BPolygon(**void**)**

Frees all the memory allocated to hold the list of points.

Member Functions

AddPoints()

void **AddPoints(**const BPoint **pointList*, long *numPoints***)**

Appends *numPoints* from *pointList* to the list of points that already define the polygon.

See also: the BPolygon constructor

CountPoints()

inline long **CountPoints(**void**)** const

Returns the number of points that define the polygon.

Frame()

inline BRect **Frame(**void**)** const

Returns the polygon's frame rectangle—the smallest rectangle that encloses the entire polygon.

MapTo()

void **MapTo(**BRect *source*, BRect *destination***)**

Modifies the polygon so that it fits the *destination* rectangle exactly as it originally fit the *source* rectangle. Each vertex of the polygon is modified so that it has the same proportional position relative to the sides of the destination rectangle as it originally had to the sides of the source rectangle.

The polygon doesn't have to be contained in either rectangle. However, to modify a polygon so that it's exactly inscribed in the destination rectangle, you should pass its frame rectangle as the source:

```
BRect frame = myPolygon->Frame();
myPolygon->MapTo(frame, anotherRect);
```

PrintToStream()

void **PrintToStream(**void**)** const

Prints the BPolygon’s point list to the standard output stream (**stdout**). The BPoint version of this function is called to report each point as a string in the form

```
"BPoint(x, y)"
```

where *x* and *y* stand for the coordinate values of the point in question.

See also: PrintToStream() in the BPoint class

Operators

= (assignment)

BPolygon& **operator =(**const BPolygon&**)**

Copies the point list of one BPolygon object and assigns it to another BPolygon. After the assignment, the two objects describe the same polygon, but are independent of each other. Destroying one of the objects won’t affect the other.

BPopUpMenu

Derived from:	public BMenu
Declared in:	<interface/PopUpMenu.h>

Overview

A BPopUpMenu is a specialized menu that's typically used in isolation, rather than as part of an extensive menu hierarchy. By default, it operates in radio mode—the last item selected by the user, and only that item, is marked in the menu.

A menu of this kind can be used to choose one from among a limited set of mutually exclusive states—to pick a paper size or paragraph style, for example, or to select a category of information. It should not be used to group different kinds of choices (as other menus may), nor should it include items that initiate actions rather than set states, except in certain well-defined cases.

A pop-up menu can be used in any of three ways:

- It can be controlled by a BMenuBar object, often one that contains just a single item. The BMenuBar, in effect, functions as a button that pops up a list. The label of the marked item in the list can be displayed as the label of the controlling item in the BMenuBar. In this way, the BMenuBar is able to show the current state of the hidden menu. When this is the case, the menu pops up so its marked item is directly over the controlling item.

- A BPopUpMenu can also be controlled by a view other than a BMenuBar. It might be associated with a particular image the view displays, for example, and appear over the image when the user moves the cursor there and presses the mouse button. Or it might be associated with the view as a whole and come up under the cursor wherever the cursor happens to be. When the view is notified of a mouse-down event, it calls BPopUpMenu's **Go()** function to show the menu on-screen.

- Finally, the application's master menu must be a BPopUpMenu object. This menu should be set up to behave like an ordinary menu, even though it's not included in an ordinary menu hierarchy. (The master menu is the one that holds items with application-wide significance, like "About . . ." and "Quit". It's accessible when the application is the active application by pressing on the application icon in the left top corner of the screen. See **SetAppMenu()** in the BApplication class.)

Other than **Go()** (and the constructor), this class implements no functions that you'd ever need to call from application code. In all other respects, a BPopUpMenu can be treated like any other BMenu.

Constructor and Destructor

BPopUpMenu()

BPopUpMenu(const char **name*, bool *radioMode* = **TRUE**,
bool *labelFromMarked* = **TRUE**,
menu_layout *layout* = **ITEMS_IN_COLUMN)**

Initializes the BPopUpMenu object. If the object is added to a BMenuBar, its *name* also becomes the initial label of its controlling item (just as for other BMenus).

If the *labelFromMarked* flag is **TRUE** (as it is by default), the label of the controlling item will change to reflect the label of the item that the user last selected. In addition, the menu will operate in radio mode (regardless of the value passed as the *radioMode* flag). When the menu pops up, it will position itself so that the marked item appears directly over the controlling item in the BMenuBar.

If *labelFromMarked* is **FALSE**, the menu pops up < so that its first item is over the controlling item >.

If the *radioMode* flag is **TRUE** (as it is by default), the last item selected by the user will always be marked. In this mode, one and only one item within the menu can be marked at a time. If *radioMode* is **FALSE**, items aren't automatically marked or unmarked.

However, the *radioMode* flag has no effect unless the *labelFromMarked* flag is **FALSE**. As long as *labelFromMarked* is **TRUE**, radio mode will also be **TRUE**.

The BPopUpMenu that's used as the application's master menu should have both *labelFromMarked* and *radioMode* set to **FALSE**.

The *layout* of the items in a BPopUpMenu can be either **ITEMS_IN_ROW** or the default **ITEMS_IN_COLUMN**. It should never be **ITEMS_IN_MATRIX**. The menu is resized so that it exactly fits the items that are added to it.

The new BPopUpMenu is empty; you add items to it by calling BMenu's **AddItem()** function.

See also: SetRadioMode() and **SetLabelFromMarked()** in the BMenu class

~BPopUpMenu()

virtual **~BPopUpMenu(**void**)**

Does nothing. The BMenu destructor is sufficient to clean up after a BPopUpMenu.

Member Functions

Go()

BMenuItem ***Go(**BPoint *screenPoint*, bool *deliversMessage* = **FALSE)**

Places the pop-up menu on-screen and keeps it there as long as the user holds a mouse button down. The menu appears on-screen so that its left top corner is located at *screenPoint* in the screen coordinate system. When the user releases the mouse button, the menu is hidden again and **Go()** returns. If the user invoked an item in the menu, it returns a pointer to the item. If no item was invoked, it returns **NULL**.

Go() is typically called from within the **MouseDown()** function of a BView. For example:

```
void MyView::MouseDown(BPoint point)
{
    BMenuItem *selected;
    BMessage *copy;
    . . .
    ConvertToScreen(&point);
    selected = myPopUp->Go(point);
    . . .
    if ( selected ) {
        BLooper *looper;
        BReceiver *target = selected->Target(&looper);
        if ( target == NULL )
            target = looper->PreferredReceiver();
        copy = new BMessage(selected->Message());
        looper->PostMessage(copy, target);
    }
    . . .
}
```

Go() operates in two modes:

- If the *deliversMessage* flag is **TRUE**, the BPopUpMenu works just like a menu that’s controlled by a BMenuBar. When the user invokes an item in the menu, the item posts a message to its target receiver.

- If the *deliversMessage* flag is **FALSE**, a message is not posted. Invoking an item doesn’t automatically accomplish anything. It’s up to the application to look at the returned BMenuItem and decide what to do. It can mimic the behavior of other menus and post the message—as shown in the example above—or it can take some other course of action.

In the example, a copy of the BMessage returned by the item's **Message()** function was posted, not the returned message itself. Posting the returned message would turn it over to a message loop, which would eventually delete it. It would then be unavailable the next time the item was invoked.

See also: **SetMessage()** in the BMenuItem class

ScreenLocation()

protected:

virtual BPoint **ScreenLocation(**void**)**

Determines where the pop-up menu should appear on-screen (when it's being run automatically, not by **Go()**). As explained in the description of the class constructor, this largely depends on whether the label of the superitem changes to reflect the item that's currently marked in the menu. The point returned is stated in the screen coordinate system.

This function is called only for BPopUpMenus that have been added to a menu hierarchy (a BMenuBar). You should not call it to determine the point to pass to **Go()**. However, you can override it to change where a customized pop-up menu defined in a derived class appears on-screen when it's controlled by a BMenuBar.

See also: **SetLabelFromMarked()** and **ScreenLocation()** in the BMenu class, the BPopUpMenu constructor

BRadioButton

Derived from:	public BControl
Declared in:	<interface/RadioButton.h>

Overview

A BRadioButton object draws a labeled, two-state button that's displayed in a group along with other similar buttons. The button itself is a round icon that has a filled center when the BRadioButton is turned on, and is empty when it's off. The label appears next to the icon.

Only one radio button in the group can be on at a time. When the user clicks a button to turn it on, the button that's currently on is turned off. The user can turn a button off only by turning another one on; one button in the group must be on at all times. The button that's on has a value of 1; the others have a value of 0.

The BRadioButton class handles the interaction between radio buttons in the following way: A direct user action can only turn on a radio button, not turn it off. However, when the user turns a button on, the BRadioButton object turns off all sibling BRadioButtons—all BRadioButtons that have the same parent as the one that was turned on.

This means that a parent view should have no more than one group of radio buttons among its children. Each set of radio buttons should be assigned a separate parent—perhaps an empty BView that simply contains the radio buttons and does no drawing of its own.

Constructor

BRadioButton()

BRadioButton(BRect *frame*, const char **name*, const char **label*,
BMessage **message*,
ulong *resizingMode* = **FOLLOW_LEFT_TOP,**
ulong *flags* = **WILL_DRAW)**

Initializes the BRadioButton by passing all arguments to the BControl constructor without change. BControl initializes the radio button's *label* and assigns it a model *message* that identifies the action that should be taken when the radio button is turned on. When the user turns the button on, the BRadioButton posts a copy of the *message* to the target receiver.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed without change from BControl to the BView constructor.

The frame rectangle of a BRadioButton must be at least 12 units high (a difference of 11 between the bottom and the top) to accommodate the icon and the label in the default font. Anything over a height of 12 is superfluous; the BRadioButton draws at the bottom of the rectangle beginning at the left side. It ignores any extra space at the top or on the right. (However, the user can click anywhere within *frame* to turn on the radio button).

See also: the BControl and BView constructors

Member Functions

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the radio button—the circular icon—and its label. The center of the icon is filled when the BRadioButton's value is 1; it's left empty when the value is 0.

See also: **Draw()** in the BView class

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event in the radio button by tracking the cursor while the user holds the mouse button down. If the cursor is pointing to the radio button when the user releases the mouse button, this function turns the button on (and consequently turns all sibling BRadioButtons off), calls the BRadioButton's **Draw()** function, and posts a message that will be delivered to the target BReceiver. Unlike a BCheckBox, a BRadioButton posts the message—it's "invoked"—only when it's turned on, not when it's turned off.

To set the value of each radio button in the group, this function calls **SetValue()** (a hook function defined in the BControl class).

See also: **Invoke()** and **SetTarget()** in the BControl class

SetValue()

virtual void **SetValue(**long *value***)**

Augments the BControl version of **SetValue()** to turn all sibling BRadioButtons off (set their values to 0) when this BRadioButton is turned on (when the *value* passed is anything but 0).

See also: **SetValue()** in the BControl class

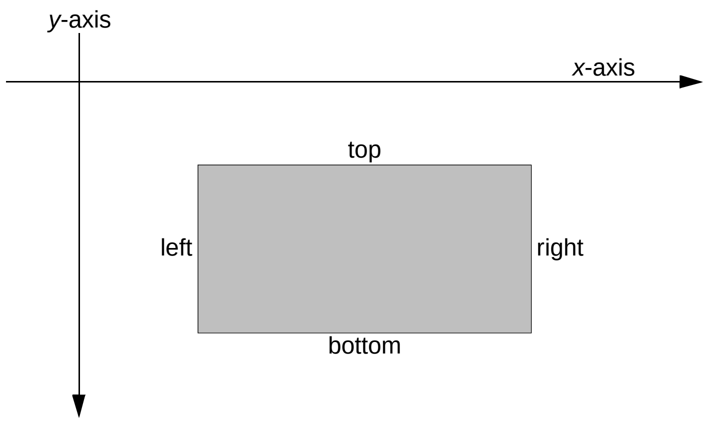
BRect

Derived from: *none*

Declared in: <interface/Rect.h>

Overview

A BRect object represents a *rectangle*, one with sides that parallel the *x* and *y* coordinate axes. The rectangle is defined by its left, top, right, and bottom coordinates, as illustrated below:



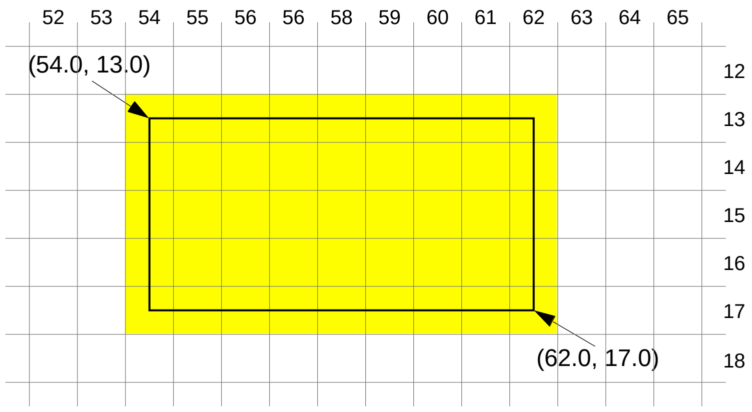
In a valid rectangle, the top *y* coordinate value is never greater than the bottom *y* coordinate, and the left *x* coordinate value is never greater than the right.

A BRect is the simplest, most basic way of specifying an area in a two-dimensional coordinate system. Windows, scroll bars, buttons, text fields, and the screen itself are all specified as rectangles. For more details on the definition of a rectangle, see "Coordinate Geometry" on page 16 in the chapter introduction.

When used to define the frame of a window or a view, or the bounds of a bitmap, the sides of the rectangle must line up on screen pixels. For this reason, the rectangle can't have any fractional coordinates. Coordinate units have a one-to-one correspondence with screen pixels.

Integral coordinates fall at the center of screen pixels, so frame rectangles cover a larger area than their coordinate values would indicate. Just as the number of elements in an array is one greater than the largest index, a frame rectangle covers one more column of pixels than its width and one more row than its height.

The figure below illustrates why this is the case. It shows a rectangle with a right side 8.0 units from its left (62.0–54.0) and a bottom 4.0 units below its top (17.0–13.0). Because the pixels that lie on all four sides of the rectangle are considered to be inside it, there's an extra pixel in each direction. When the rectangle is filled on-screen, it covers a 9-pixel-by-5-pixel area.



Because the BRect structure is a basic data type for graphic operations, it's constructed more simply than most other Interface Kit classes: All its data members are publicly accessible, it doesn't have virtual functions, it doesn't inherit from BObject or any other class, and it doesn't retain class information that it can reveal at run time. Within the Interface Kit, BRect objects are passed and returned by value.

Data Members

float **left**	The coordinate value of the rectangle's leftmost side (the smallest *x* coordinate in a valid rectangle).
float **top**	The coordinate value of the rectangle's top (the smallest *y* coordinate in a valid rectangle).
float **right**	The coordinate value of the rectangle's rightmost side (the largest *x* coordinate in a valid rectangle).
float **bottom**	The coordinate value of the rectangle's bottom (the largest *y* coordinate in a valid rectangle).

Constructor

BRect()

inline **BRect(**float *left*, float *top*, float *right*, float *bottom***)**
inline **BRect(**BPoint *leftTop*, BPoint *rightBottom***)**
inline **BRect(**const BRect& *rect***)**
inline **BRect(**void**)**

Initializes a BRect with its four coordinate values—*left*, *top*, *right*, and *bottom*. The four values can be directly stated,

```
BRect rect(11.0, 24.7, 301.5, 99.0);
```

or they can be taken from two points designating the rectangle's left top and right bottom corners,

```
BPoint leftTop(11.0, 24.7);
BPoint rightBottom(301.5, 99.0);
BRect rect(leftTop, rightBottom);
```

or they can be copied from another rectangle:

```
BRect anotherRect(11.0, 24.7, 301.5, 99.0);
BRect rect(anotherRect);
```

A rectangle that's not assigned any initial values,

```
BRect rect;
```

is constructed to be invalid (its top and left are greater than its right and bottom), until a specific assignment is made, typically with the **Set()** function:

```
rect.Set(77.0, 2.25, 510.8, 393.0);
```

See also: Set()

Member Functions

Contains()

bool **Contains(**BPoint *point***)** const
bool **Contains(**BRect *rect***)** const

Returns **TRUE** if *point*—or *rect*—lies inside the area the BRect defines, and **FALSE** if not. A rectangle contains a point even if the point coincides with one of the rectangle's corners or lies on one of its edges.

One rectangle contains another if their union is the same as the first rectangle and their intersection is the same as the second—that is, if the second rectangle lies entirely within the first. A rectangle is considered to be inside another rectangle even if they have one or more sides in common. Two identical rectangles contain each other.

See also: **Intersects()**, the **&** (intersection) and **|** (union) operators, **ConstrainTo()** in the BPoint class

Height() see **Width()**

InsetBy()

void **InsetBy(**float *horizontal*, float *vertical***)**
void **InsetBy(**BPoint *point***)**

Modifies the BRect by insetting its left and right sides by *horizontal* units and its top and bottom sides by *vertical* units. (If a *point* is passed, its *x* coordinate value substitutes for *horizontal* and its *y* coordinate value substitutes for *vertical*.)

For example, this code

```
BRect rect(10.0, 40.0, 100.0, 140.0);
rect.InsetBy(20.0, 30.0);
```

produces a rectangle identical to one that could be constructed as follows:

```
BRect rect(30.0, 70.0, 80.0, 110.0);
```

If *horizontal* or *vertical* is negative, the rectangle becomes larger in that dimension, rather than smaller.

See also: **OffsetBy()**

Intersects()

bool **Intersects(**BRect *rect***)** const

Returns **TRUE** if the BRect has any area—even a corner or part of a side—in common with *rect*, and **FALSE** if it doesn't.

See also: the **&** (intersection) operator

IsValid()

inline bool **IsValid(**void**)** const

Returns **TRUE** if the BRect's right side is greater than or equal to its left and its bottom is greater than or equal to its top, and **FALSE** otherwise. An invalid rectangle doesn't designate any area, not even a line or a point.

LeftBottom() see SetLeftBottom()

LeftTop() see SetLeftTop()

OffsetBy(), OffsetTo()

void **OffsetBy(**float *horizontal*, float *vertical***)**
void **OffsetBy(**BPoint *point***)**

void **OffsetTo(**BPoint *point***)**
void **OffsetTo(**float *x*, float *y***)**

These functions reposition the rectangle in its coordinate system, without altering its size or shape.

OffsetBy() adds *horizontal* to the left and right coordinate values of the rectangle and *vertical* to its top and bottom coordinates. (If a *point* is passed, *point.x* substitutes for *horizontal* and *point.y* for *vertical*.)

OffsetTo() moves the rectangle so that its left top corner is at *point*—or at (*x*, *y*). The coordinate values of all its sides are adjusted accordingly.

See also: InsetBy()

PrintToStream()

void **PrintToStream(**void**)** const

Prints the contents of the BRect object to the standard output stream (**stdout**) in the form:

```
"BRect(left, top, right, bottom)"
```

where *left*, *top*, *right*, and *bottom* stand for the current values of the BRects's data members.

RightBottom() see SetRightBottom()

RightTop() see **SetRightTop()**

Set()

inline void **Set(**float *left*, float *top*, float *right*, float *bottom***)**

Assigns the values *left*, *top*, *right*, and *bottom* to the BRect's corresponding data members. The following code

```
BRect rect;
rect.Set(0.0, 25.0, 50.0, 75.0);
```

is equivalent to:

```
BRect rect;
rect.left = 0.0;
rect.top = 25.0;
rect.right = 50.0;
rect.bottom = 75.0;
```

See also: the BRect constructor

SetLeftBottom(), LeftBottom()

void **SetLeftBottom(**const BPoint *point***)**

inline BPoint **LeftBottom(**void**)** const

These functions set and return the left bottom corner of the rectangle. **SetLeftBottom()** alters the BRect so that its left bottom corner is at *point*, and **LeftBottom()** returns its current left and bottom coordinates as a BPoint object.

See also: **SetLeftTop()**, **SetRightBottom()**, **SetRightTop()**

SetLeftTop(), LeftTop()

void **SetLeftTop(**const BPoint *point***)**

inline BPoint **LeftTop(**void**)** const

These functions set and return the left top corner of the rectangle. **SetLeftTop()** alters the BRect so that its left top corner is at *point*, and **LeftTop()** returns its current left and top coordinates as a BPoint object.

See also: **SetLeftBottom()**, **SetRightTop()**, **SetRightBottom()**

SetRightBottom(), RightBottom()

void **SetRightBottom(**const BPoint *point***)**

inline BPoint **RightBottom(**void**)** const

These functions set and return the right bottom corner of the rectangle. **SetRightBottom()** alters the BRect so that its right bottom corner is at *point*, and **RightBottom()** returns its current right and bottom coordinates as a BPoint object.

See also: **SetRightTop()**, **SetLeftBottom()**, **SetLeftTop()**

SetRightTop(), RightTop()

void **SetRightTop(**const BPoint *point***)**

inline BPoint **RightTop(**void**)** const

These functions set and return the right top corner of the rectangle. **SetRightTop()** alters the BRect so that its right top corner is at *point*, and **RightTop()** returns its current right and top coordinates as a BPoint object.

See also: **SetRightBottom()**, **SetLeftTop()**, **SetLeftBottom()**

Width(), Height()

inline float **Width(**void**)** const

inline float **Height(**void**)** const

These functions return the width of the rectangle (the difference between its bottom and top coordinates) and its height (the difference between its right and left sides). If either value is negative, the rectangle is invalid.

The width and height of a rectangle are not accurate guides to the number of pixels it covers on screen. As illustrated in the "Overview" to this class, a rectangle without fractional coordinates covers an area that's one pixel broader than its coordinate width and one pixel taller than its coordinate height.

Operators

= (assignment)

inline BRect& **operator =(**const BRect&**)**

Assigns the data members of one BRect object to another BRect:

```
BRect a(27.2, 36.8, 230.0, 359.1);
BRect b = a;
```

Rectangle *b* is identical to rectangle *a*.

== (equality)

bool **operator ==(**BRect**)** const

Compares the data members of two BRect objects and returns **TRUE** if each one exactly matches its counterpart in the other object, and **FALSE** if any of the members don't match. In the following example, the equality operator would return **FALSE**, since the two objects have different right boundaries:

```
BRect a(11.5, 22.5, 66.5, 88.5);
BRect b(11.5, 22.5, 46.5, 88.5);
if ( a == b )
    . . .
```

!= (inequality)

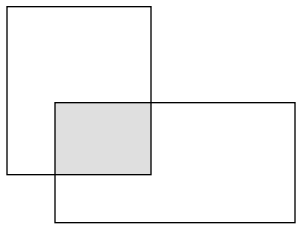
char **operator !=(**BRect**)** const

Compares two BRect objects and returns **TRUE** unless their data members match exactly (the two rectangles are identical), in which case it returns **FALSE**. This operator is the inverse of the **==** (equality) operator.

& (intersection)

BRect **operator &(**BRect**)** const

Returns the intersection of two rectangles—a rectangle enclosing the area they have in common. The shaded area below shows where the two outlined rectangles intersect.



The intersection is computed by taking the greatest left and top coordinate values of the two rectangles, and the smallest right and bottom values. In the following example,

```
BRect a(10.0, 40.0, 80.0, 100.0);
BRect b(35.0, 15.0, 95.0, 65.0);
BRect c = a & b;
```

rectangle *c* will be identical to one constructed as follows:

```
BRect c(35.0, 40.0, 80.0, 65.0);
```

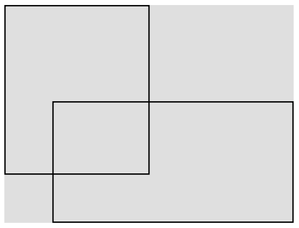
If the two rectangles don't actually intersect, the result will be invalid. You can test for this by calling the **Intersects()** function on the original rectangles, or by calling **IsValid()** on the result.

See also: **Intersects()**, **IsValid()**, the **|** (union) operator

| (union)

BRect **operator |(**BRect**)** const

Returns the union of two rectangles—the smallest rectangle that encloses them both. The shaded area below illustrates the union of the two outlined rectangles. Note that it includes areas not in either of them.



The union is computed by selecting the smallest left and top coordinate values from the two rectangles, and the greatest right and bottom coordinate values. In the following example,

```
BRect a(10.0, 40.0, 80.0, 100.0);
BRect b(35.0, 15.0, 95.0, 65.0);
BRect c = a | b;
```

rectangle *c* will be identical to one constructed as follows:

```
BRect c(10.0, 15.0, 95.0, 100.0);
```

Note that two rectangles will have a valid union even if they don't intersect.

See also: the **&** (intersection) operator

BRegion

Derived from:	public BObject
Declared in:	<interface/Region.h>

Overview

A BRegion object describes an arbitrary area within a two-dimensional coordinate system. The area can have irregular boundaries, contain holes, or be discontinuous. It’s convenient to think of a region as a set of locations or points, rather than as a closed shape like a rectangle or a polygon.

The points that a region includes can be described by a set of rectangles. Any point that lies within at least one of the rectangles belongs to the region. You can define a region incrementally by passing rectangles to functions like **Set()**, **Include()**, and **Exclude()**.

BView's **GetClippingRegion()** function modifies a BRegion object so that it represents the current clipping region of the view. A BView can pass **GetClippingRegion()** a pointer to an empty BRegion,

```
BRegion temp;
GetClippingRegion(&temp);
```

then call BRegion’s **Intersects()** and **Contains()** functions to test whether the potential drawing it might do falls within the region:

```
if ( temp.Intersects(someRect) )
    . . .
```

Constructor and Destructor

BRegion()

BRegion(const BRegion& *region***)**
BRegion(void**)**

Initializes the BRegion object to have the same area as another *region*—or, if no other region is specified, to an empty region.

The original BRegion object and the newly constructed one each have their own copies of the data describing the region. Altering or freeing one of the objects will not affect the other.

BRegion objects can be allocated on the stack and assigned to other objects:

```
BRegion regionOne(anotherRegion);
BRegion regionTwo = regionOne;
```

However, due to their size, it's more efficient to pass them by pointer rather than by value.

~BRegion

virtual **~BRegion(**void**)**

Frees any memory that was allocated to hold data describing the region.

Member Functions

Contains()

bool **Contains(**BPoint *point***)** const

Returns **TRUE** if *point* lies within the region, and **FALSE** if not.

Exclude()

void **Exclude(**BRect *rect***)**
void **Exclude(**const BRegion **region***)**

Modifies the region so that it excludes all points contained within *rect* or *region* that it might have included before.

See also: Include(), **IntersectWith()**

Frame()

BRect **Frame(**void**)** const

Returns the frame rectangle of the BRegion—the smallest rectangle that encloses all the points within the region.

If the region is empty, the rectangle returned won't be valid.

See also: IsValid() in the BRect class

Include()

> void **Include(**BRect *rect***)**
> void **Include(**const BRegion **region***)**

Modifies the region so that it includes all points contained within the *rect* or *region* passed as an argument.

See also: Exclude()

IntersectWith()

> void **IntersectWith(**const BRegion **region***)**

Modifies the region so that it includes only those points that it has in common with another *region*.

See also: Include()

Intersects()

> bool **Intersects(**BRect *rect***)** const

Returns **TRUE** if the BRegion has any area in common with *rect*, and **FALSE** if not.

MakeEmpty()

> void **MakeEmpty(**void**)**

Empties the BRegion of all its points. It will no longer designate any area and its frame rectangle won't be valid.

See also: the BRegion constructor

OffsetBy()

> void **OffsetBy(**long *horizontal*, long *vertical***)**

Offsets all points contained within the region by adding *horizontal* to each *x* coordinate value and *vertical* to each *y* coordinate value.

PrintToStream()

void **PrintToStream(**void**)** const

Prints the contents of the BRegion to the standard output stream (**stdout**) as an array of strings. Each string describes a rectangle in the form:

```
"BRect(left, top, right, bottom)"
```

where *left*, *top*, *right*, and *bottom* are the coordinate values that define the rectangle.

The first string in the array describes the BRegion's frame rectangle. Each subsequent string describes one portion of the area included in the BRegion.

See also: PrintToStream() in the BRect class, **Frame()**

Set()

void **Set(**BRect *rect***)**

Modifies the BRegion so that it describes an area identical to *rect*. A subsequent call to **Frame()** should return the same rectangle (unless some other change was made to the region in the interim).

See also: Include(), **Exclude()**

Operators

= (assignment)

BRegion& **operator =(**const BRegion&**)**

Assigns the region described by one BRegion object to another BRegion:

```
BRegion region = anotherRegion;
```

After the assignment, the two regions will be identical, but independent, copies of one another. Each object allocates its own memory to store the description of the region.

BScrollBar

Derived from:	public BView
Declared in:	<interface/ScrollBar.h>

Overview

A BScrollBar object displays a scroll bar that users can operate to scroll the contents of another view, a *target view.* Scroll bars usually come in pairs, one horizontal and one vertical, and are often grouped as siblings of the target view under a common parent. That way, when the parent is resized, the target and scroll bars can be automatically resized to match. (A companion class, BScrollView, defines just such a container view; a BScrollView object sets up the scroll bars for a target view and makes itself the parent of the target and the scroll bars.)

The Update Mechanism

BScrollBars are different from other views in one important respect: All their drawing and event handling is carried out within the Application Server, not in the application. A BScrollBar object doesn't receive **Draw()** or **MouseDown()** notifications; the Server intercepts updates and events that would otherwise be reported to the BScrollBar and handles them itself. As the user moves the knob on a scroll bar or presses a scroll button, the Application Server continuously refreshes the scroll bar's image on-screen and informs the application with a steady stream of messages reporting value-changed events.

The window dispatches these event messages by calling the BScrollBar's **ValueChanged()** function. Each function call notifies the BScrollBar of a change in its value and, consequently, of a need to scroll the target view.

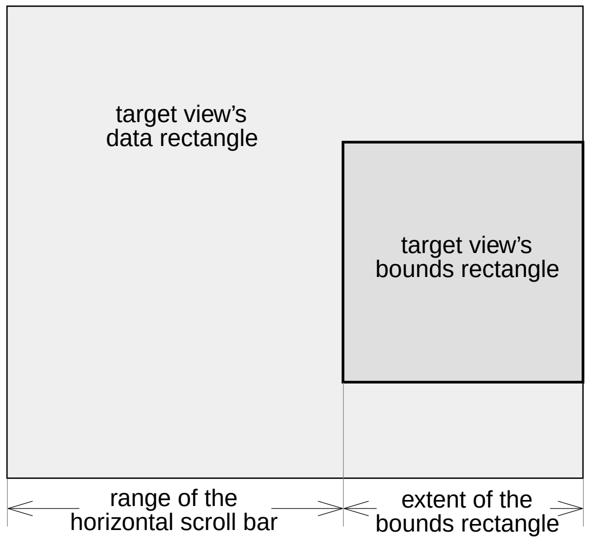
Confining the update mechanism for scroll bars to the Application Server limits the volume of communication between the application and Server and enhances the efficiency of scrolling. The application's messages to the Server can concentrate on updating the target view as its contents are being scrolled, rather than on updating the scroll bars themselves.

Value and Range

A scroll bar's value determines what the target view displays. The default assumption is that the left coordinate value of the target view's bounds rectangle should match the value of the horizontal scroll bar, and the top of the target view's bounds rectangle should match the value of the vertical scroll bar. When a BScrollBar is notified of a change of value (through its **ValueChanged()** function), it scrolls the target view to put the new value at the left or top of the bounds rectangle.

The value reported in a **ValueChanged()** notification depends on where the user moves the scroll bar's knob and on the range of values the scroll bar represents. The range is first set in the BScrollBar constructor and can be modified by the **SetRange()** function.

The range must be large enough to bring all the coordinate values where the target view can draw into its bounds rectangle. If everything the target view can draw is conceived as being enclosed in a "data rectangle," the range of a horizontal scroll bar must extend from a minimum that makes the left side of the target's bounds rectangle coincide with the left side of its data rectangle, to a maximum that puts the right side of the bounds rectangle at the right side of the data rectangle. This is illustrated in part below:


target view's data rectangle
target view's bounds rectangle
range of the horizontal scroll bar
extent of the bounds rectangle

As this illustration helps demonstrate, the maximum value of a horizontal scroll bar can be no less than the right coordinate value of the data rectangle minus the width of the bounds rectangle. Similarly, for a vertical scroll bar, the maximum value can be no less than the bottom coordinate of the data rectangle minus the height of the bounds rectangle. The range of a scroll bar subtracts the dimensions of the target's bounds rectangle from its data rectangle. (The minimum values of horizontal and vertical scroll bars can be no greater than the left and top sides of the data rectangle.)

What the target view can draw may change from time to time as the user adds or deletes data. As this happens, the range of the scroll bar should be updated with the **SetRange()** function. The range may also need to be recalculated when the target view is resized.

Hook Functions

ValueChanged()	Scrolls the target view when the BScrollBar is informed that its value has changed; can be implemented to alter the default interpretation of the scroll bar's value.

Constructor and Destructor

BScrollBar()

BScrollBar(BRect *frame*, const char **name*, BView **target*,
long *min*, long *max*, orientation *posture***)**

Initializes the BScrollBar and connects it to the *target* view that it will scroll. It will be a horizontal scroll bar if *posture* is **HORIZONTAL** and a vertical scroll bar if *posture* is **VERTICAL**.

The range of values that the scroll bar can represent at the outset is set by *min* and *max*. These values should be calculated from the boundaries of a rectangle that encloses the entire contents of the target view—everything that it can draw. If *min* and *max* are both 0, the scroll bar is disabled and the knob is not drawn.

The object's initial value is 0 < even if that falls outside the range set for the scroll bar >.

The other arguments, *frame* and *name*, are the same as for other BViews:

- The *frame* rectangle locates the scroll bar within its parent view. A horizontal scroll bar should be exactly 12.0 units high, and a vertical scroll bar should be exactly 12.0 pixels wide. < These values may change as the user interface changes. >
- The BScrollBar's *name* identifies it and permits it to be located by the **FindView()** function. It can be **NULL**.

Unlike other BViews, the BScrollBar constructor doesn't set an automatic resizing mode. By default, scroll bars have the resizing behavior that befits their posture—horizontal scroll bars resize themselves horizontally (as if they had a resizing mode of **FOLLOW_LEFT_RIGHT_BOTTOM**) and vertical scroll bars resize themselves vertically (as if their resizing mode was **FOLLOW_TOP_RIGHT_BOTTOM**).

~BScrollBar()

virtual **~BScrollBar(**void**)**

Disconnects the scroll bar from its target.

Member Functions

GetRange() see **SetRange()**

GetSteps() see **SetSteps()**

Orientation()

inline orientation **Orientation(**void**)** const

Returns **HORIZONTAL** if the object represents a horizontal scroll bar and **VERTICAL** if it represents a vertical scroll bar.

See also: the BScrollBar constructor

SetRange(), GetRange()

void **SetRange(**long *min*, long *max***)**

void **GetRange(**long **min*, long **max***)** const

These functions modify and return the range of the scroll bar. **SetRange()** sets the minimum and maximum values of the scroll bar to *min* and *max*. **GetRange()** places the current minimum and maximum in the variables that *min* and *max* refer to.

If the scroll bar's current value falls outside the new range, it will be reset to the closest value—either *min* or *max*—within range. **ValueChanged()** is called to inform the BScrollBar of the change whether or not it's attached to a window.

If the BScrollBar is attached to a window, any change in its range will be immediately reflected on-screen. The knob will move to the appropriate position to reflect the current value.

Setting both the minimum and maximum to 0 disables the scroll bar. It will be drawn without a knob.

See also: the BScrollBar constructor

SetSteps(), GetSteps()

void **SetSteps(**long *smallStep*, long *bigStep***)**

void **GetSteps(**long **smallStep*, long **bigStep***)** const

SetSteps() sets how much a single user action should change the value of the scroll bar—and therefore how far the target view should scroll. **GetSteps()** provides the current settings.

When the user presses one of the scroll buttons at either end of the scroll bar, its value changes by a *smallStep*. When the user clicks in the bar itself (other than on the knob), it changes by a *bigStep*. For an application that displays text, the small step of a vertical scroll bar should be large enough to bring another line of text into view.

The default small step is 1, which should be too small for most purposes; the default large step is 10, which is also probably too small.

< Currently, a BScrollBar's steps can be successfully set only after it's attached to a window. >

See also: ValueChanged()

SetTarget(), Target()

void **SetTarget(**BView **view***)**
void **SetTarget(**char **name***)**

inline BView ***Target(**void**)** const

These functions set and return the target of the BScrollBar (the view that the scroll bar scrolls). **SetTarget()** sets the target to *view*, or to the BView identified by *name*. **Target()** returns the current target view. The target can also be set when the BScrollBar is constructed.

SetTarget() can be called either before or after the BScrollBar is attached to a window. If the target is set by *name*, the named view must eventually be found within the same window as the scroll bar. Typically, the target and its scroll bars are children of a container view that serves to bind them together as a unit.

See also: the BScrollBar constructor, **ValueChanged()**

SetValue(), Value()

void **SetValue(**long *value***)**

long **Value(**void**)** const

These functions modify and return the value of the scroll bar. The value is usually set as the result of user actions; **SetValue()** provides a way to do it programmatically. **Value()** returns the current value, whether set by **SetValue()** or by the user.

SetValue() assigns a new *value* to the scroll bar and calls the **ValueChanged()** hook function, whether or not the new value is really a change from the old. If the *value* passed lies outside the range of the scroll bar, the BScrollBar is reset to the closest value within range—that is, to either the minimum or the maximum value previously specified.

If the scroll bar is attached to a window, changing its value updates its on-screen display. The call to **ValueChanged()** enables the object to scroll the target view so that it too is updated to conform to the new value.

The initial value of a scroll bar is 0.

See also: ValueChanged(), **SetRange()**

Target() see **SetTarget()**

Value() see **SetValue()**

ValueChanged()

virtual void **ValueChanged(**long *newValue***)**

Responds to a notification that the value of the scroll bar has changed to *newValue*. For a horizontal scroll bar, this function interprets *newValue* as the coordinate value that should be at the left side of the target view's bounds rectangle. For a vertical scroll bar, it interprets *newValue* as the coordinate value that should be at the top of the rectangle. It calls **ScrollTo()** to scroll the target view's contents accordingly.

ValueChanged() does nothing if a target BView hasn't been set—or if the target has been set by name, but the name doesn't correspond to an actual BView within the scroll bar's window.

Derived classes can override this function to interpret *newValue* differently, or to do something in addition to scrolling the target view.

ValueChanged() is called as the result both of value-changed event messages received from the Application Server and of **SetValue()** and **SetRange()** function calls within the application.

See also: SetTarget()

BScrollView

Derived from: public BView

Declared in: <interface/ScrollView.h>

Overview

A BScrollView object is a container for another view, a *target view*, typically a view that can be scrolled. The BScrollView creates and positions the scroll bars the target view needs and makes itself the parent of the scroll bars and the target view. It's a convenient way to set up scroll bars for another view.

If requested, the BScrollView draws a one-pixel wide black border around its children. Otherwise, it does no drawing and simply contains the family of views it set up.

The **ScrollBar()** function provides access to the scroll bars the BScrollView creates, so you can set their ranges and values as needed.

Constructor and Destructor

BScrollView()

BScrollView(const char **name*, BView **target*,
ulong *resizingMode* = **FOLLOW_LEFT_TOP**, ulong *flags* = 0,
bool *horizontal* = **FALSE**, bool *vertical* = **FALSE**,
bool *bordered* = **TRUE)**

Initializes the BScrollView. It will have a frame rectangle large enough to contain the *target* view and any scroll bars that are requested. If *horizontal* is **TRUE**, there will be a horizontal scroll bar. If *vertical* is **TRUE**, there will be a vertical scroll bar. Scroll bars are not provided unless you ask for them.

If *bordered* is **TRUE**, as it is by default, the frame rectangle will also be large enough to draw a narrow black border around the target view and scroll bars. A BScrollView can be used without scroll bars to simply contain and border the target view.

The BScrollView adapts its frame rectangle from the frame rectangle of the target view. It positions itself so that its left and top sides are exactly where the left and top sides of the target view originally were. It then adds the target view as its child along with any requested scroll bars. In the process, it modifies the target view's frame rectangle (but not its bounds rectangle) so that it will fit within its new parent.

If the resize mode of the target view is **FOLLOW_ALL_SIDES**, it and the scroll bars will be automatically resized to fill the container view whenever the container view is resized.

The scroll bars created by the BScrollView have an initial range extending from a minimum of 0 to a maximum of 1000. You'll generally need to ask for the scroll bars (using the **ScrollBar()** function) and set their ranges to more appropriate values.

The *name*, *resizeMode*, and *flags* arguments are identical to those declared in the BView class and are passed unchanged to the BView constructor.

See also: the BView constructor

~BScrollView()

virtual **~BScrollView(**void**)**

Does nothing.

Member Functions

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws a one-pixel wide black border around the target view and scroll views, provided the *bordered* flag wasn't set to **FALSE** in the BScrollView constructor.

See also: the BScrollView constructor, **Draw()** in the BView class

ScrollBar()

BScrollBar ***ScrollBar(**orientation *posture***)** const

Returns the horizontal scroll bar if *posture* is **HORIZONTAL** and the vertical scroll bar if *posture* is **VERTICAL**. If the BScrollView doesn't contain a scroll bar with the requested orientation, this function returns **NULL**.

See also: the BScrollBar class

BSeparatorItem

Derived from:	public BMenuItem
Declared in:	<interface/MenuItem.h>

Overview

A BSeparatorItem is a menu item that serves only to separate the items that precede it in the menu list from the items that follow it. It's drawn as a horizontal line across the menu from the left border to the right. Although it has an indexed position in the menu list just like other items, it doesn't have a label, can't be selected, posts no messages, and is permanently disabled.

Since the separator is drawn horizontally, it's assumed that items in the menu are arranged in a column, as they are by default. It's inappropriate to use a separator in a menu bar or another menu where the items are arranged in a row.

A separator can be added to a BMenu by constructing an object of this class and calling BMenu's **AddItem()** function. As a shorthand, you can simply call BMenu's **AddSeparatorItem()** function, which constructs the object for you and adds it to the list.

A BSeparatorItem that's returned to you (by BMenu's **ItemAt()** function, for example) will always respond **NULL** to **Message()**, **Command()**, and **Submenu()** queries and **FALSE** to **IsEnabled()**.

See also: AddSeparatorItem() in the BMenu class

Constructor and Destructor

BSeparatorItem()

BSeparatorItem(void**)**

Initializes the BSeparatorItem and disables it.

~BSeparatorItem()

virtual **~BSeparatorItem(**void**)**

Does nothing.

Member Functions

Draw()

protected:
  virtual void **Draw(**void**)**

Draws the item as a horizontal line across the width of the menu.

GetContentSize()

protected:
  virtual void **GetContentSize(**float **width*, float **height***)**

Provides a minimal size for the item so that it won't constrain the size of the menu.

SetEnabled()

  virtual void **SetEnabled(**bool *flag***)**

Does nothing. A BSeparatorItem is disabled when it's constructed and must stay that way.

BStringView

Derived from:	public BView
Declared in:	<interface/StringView.h>

Overview

A BStringView object draws a static character string. The user can't select the string or edit it; a BStringView doesn't respond to events. An instance of this class can be used to draw a label or other text that simply delivers a message of some kind to the user. Use a BTextView object for selectable and editable text.

You can also draw strings by calling BView's **DrawString()** function. However, assigning a string to a BStringView object locates it in the view hierarchy. The string will be updated automatically, just like other views. And, by setting the resizing mode of the object, you can make sure that it will be positioned properly when the window or the view it's in (the parent of the BStringView) is resized.

Constructor and Destructor

BStringView()

BStringView(BRect *frame*, const char **name*, const char **text*,
    ulong *resizingMode* = **FOLLOW_LEFT_TOP**,
    ulong *flags* = **WILL_DRAW)**

Initializes the BStringView by assigning it a *text* string. The *frame* rectangle needs to be large enough to display the entire string in the current font. The string is drawn at the bottom of the frame rectangle and, by default, is aligned to the left side. A different horizontal alignment can be set by calling **SetAlignment()**.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class. They're passed unchanged to the BView constructor.

~BStringView()

virtual **~BStringView(**void**)**

Frees the text string.

Member Functions

Alignment() see SetAlignment()

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Sets the default font for drawing the label to the 9-point “geneva” bitmap font. This function is called by the Interface Kit; you shouldn’t call it yourself. However, you can reimplement it to set the front color or a different font for drawing the string—or simply to take notice when the BStringView becomes part of a window’s view hierarchy.

See also: AttachedToWindow() in the BView class

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the string along the bottom of the BStringView’s frame rectangle in the current front color.

See also: Draw() in the BView class

SetAlignment(), Alignment()

void **SetAlignment(**alignment *flag***)**

inline alignment **Alignment(**void**)** const

These functions align the string within the BStringView’s frame rectangle and return the current alignment. The alignment *flag* can be:

ALIGN_LEFT	The string is aligned at the left side of the frame rectangle.
ALIGN_RIGHT	The string is aligned at the right side of the frame rectangle.
ALIGN_CENTER	The string is aligned so that the center of the string falls midway between the left and right sides of the frame rectangle.

The default is **ALIGN_LEFT**.

SetText(), Text()

> void **SetText(**const char **string***)**
>
> inline const char ***Text(**void**)** const

These functions set and return the text string that the BStringView draws. **SetText()** frees the previous string and copies *string* to replace it. **Text()** returns the null-terminated string.

BTextView

Derived from:	public BView
Declared in:	<interface/TextView.h>

Overview

The BTextView class defines a view that displays text on-screen and supports a standard user interface for entering, selecting, and editing text from the keyboard and mouse. It also supports the principal editing commands—"Cut," "Copy," "Paste," "Delete," and "Select All."

BTextView objects are suitable for displaying small amounts of text in the user interface and for creating textual data in ASCII format. Full-scale text editors and word processors will need to define their own objects to handle richer data formats.

A BTextView displays all its text in a single font, the font that it inherits as a BView graphics parameter. Multiple fonts are not supported. Paragraph properties—such as alignment, tab widths, and interline spacing—are similarly uniform for all text displayed within the view.

Resizing

A BTextView can be made to resize itself to exactly fit the text that the user enters. This is sometimes appropriate for small one-line text fields. See the **MakeResizable()** function.

Shortcuts and Menu Items

When a BTextView is the focus view for its window, it responds to these standard keyboard shortcuts for cutting, copying, and pasting text:

- Command-*x* to cut text and copy it to the clipboard,
- Command-*c* to copy text without cutting it, and
- Command-*v* to paste text taken from the clipboard.

These shortcuts work even in the absence of "Cut," "Copy," and "Paste" menu items; they're implemented by the BWindow for any view that might be the focus view. All the focus view has to do is cooperate, as a BTextView does, by handling the messages the shortcuts generate.

The only trick is to set up menu items that are compatible with the shortcuts. Follow these guidelines if you put a menu with editing commands in a window that has a BTextView:

- Create "Cut", "Copy", and "Paste" menu items and assign them the Command-*x*, Command-*c*, and Command-*v* shortcuts.
- Assign the items model **CUT**, **COPY** and **PASTE** messages. These messages don't need to contain any information (other than a **what** data member initialized to the proper constant).
- Target the messages to the BWindow's focus view (or directly to the BTextView). No changes to the BTextView are necessary. When it gets these messages, the BTextView calls its **Cut()**, **Copy()**, and **Paste()** functions.

You can also set up menu items that trigger calls to other BTextView editing and layout functions. Simply create menu items like "Select All" or "Double Space" that are targeted to the focus view of the window where the BTextView is located, or to the BTextView itself. The model messages assigned to these items can be structured with whatever command constants and data entries you wish; the BTextView class imposes no constraints.

Then, in a class derived from BTextView, implement a **MessageReceived()** function that responds to messages posted from the menu items by calling BTextView functions like **SelectAll()** and **SetSpacing()**. For example:

```
void myText::MessageReceived(BMessage *message)
{
    switch ( message->what ) {
    case SELECT_ALL:
        SelectAll();
        break;
    case SINGLE_SPACE:
        SetSpacing(1);
        break;
    case DOUBLE_SPACE:
        SetSpacing(2);
        break;
    . . .
    default:
        BTextView::MessageReceived(message);
        break;
    }
}
```

The **MessageReceived()** function you implement should be sure to call BTextView's version of the function, which already handles **CUT**, **COPY**, and **PASTE** messages.

Hook Functions

AcceptsChar()	Can be implemented to preview the characters the user types and either accept or reject them before they're added to the display.
BreaksAtChar()	Breaks word selection on spaces, tabs, and other invisible characters, permitting all adjacent visible characters to be selected when the user double-clicks a word. This function can be augmented to break word selection on other characters in addition to the invisible ones.

Constructor and Destructor

BTextView()

BTextView(BRect *frame*, const char **name*, BRect *textRect*,
ulong *resizingMode*, ulong *flags***)**

Initializes the BTextView to the *frame* rectangle, stated in its eventual parent's coordinate system, assigns it an identifying *name*, sets its resizing behavior to *resizingMode* and its drawing behavior with *flags*. These four arguments—*frame*, *name*, *resizingMode*, and *flags*—are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The text rectangle, *textRect*, is stated in the BTextView's coordinate system. It determines where text in placed within the view's bounds rectangle:

- The first line of text is placed at the top of the text rectangle. As additional lines of text are entered into the view, the text grows downward and may actually extend beyond the bottom of the rectangle.
- The left and right sides of the text rectangle determine where lines of text are placed within the view. Lines can be aligned to either side of the rectangle, or they can be centered between the two sides. See the **SetAlignment()** function.
- When lines wrap on word boundaries, the width of the text rectangle determines the maximum length of a line; each line of text can be as long as the rectangle is wide. When word wrapping isn't turned on, lines can extend beyond the boundaries of the text rectangle. See the **SetWordWrap()** function.

The bottom of the text rectangle is ignored; it doesn't limit the amount of text the view can contain. The text can be limited by the number of characters, but not by the number of lines.

The constructor establishes the following default properties for a new BTextView:

- The text is left-aligned and single-spaced.
- The tab width is 44.0 coordinate units.
- Automatic indenting and word wrapping are turned off.
- The text is selectable and editable.
- All characters the user may type are acceptable.

A BTextView isn't fully initialized until it's assigned to a window and it receives an **AttachedToWindow()** notification.

See also: **AttachedToWindow()**, the BView constructor

~BTextView()

virtual **~BTextView(**void**)**

Frees the memory the BTextView allocated to hold the text and to store information about it.

Member Functions

AcceptsChar()

virtual bool **AcceptsChar(**ulong *aChar***)** const

Implemented by derived classes to return **TRUE** if *aChar* designates a character that the BTextView can add to its text, and **FALSE** if not. By returning **FALSE**, this function prevents the character from being displayed or retained by the object.

AcceptsChar() is called for every character the user types (including those, like **BACKSPACE** and **RIGHT_ARROW**, that are used for editing the text). The default version of this function always returns **TRUE**, but it can be overridden in a derived class to restrict the text the user can enter. For example, a BTextView might reject uppercase letters, or permit only numbers, or allow only those characters that are valid in a pathname.

Sometimes, a character will be meaningful and trigger a response of some kind, even though it can't be displayed. For example, a **TAB** (0x09) might be rejected as a character to display, and instead shift the selection to another text field. Similarly, a BTextView that has room to display only a single line of text might return **FALSE** for the newline character (**ENTER**, 0x0a), yet take the occasion to simulate a click on a button.

When rejecting a character outright (not using it to take some other action), an application has an obligation to explain to the user why the character is unacceptable, perhaps by displaying an alert panel or dialog box.

As an alternative to implementing an **AcceptsChar()** function, you can simply inform the BTextView at the outset that certain characters should not be allowed. Call **DisallowChar()** when setting up the BTextView to tell it that certain characters won't be acceptable.

See also: KeyDown(), **DisallowChar()**

Alignment() see SetAlignment()

AllowChar() see DisallowChar()

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Completes the initialization of the BTextView object after it becomes attached to a window. This function sets up the object so that it can correctly format text and display it. It allocates memory for the text and makes sure that all properties that were previously set—for example, word wrapping, tab width, and alignment—are correctly reflected in the display on-screen. In addition, it calls **SetFontName()** and **SetFontSize()** to set the font to the 9-point "geneva" bitmap font (no rotation, 90° shear).

This function is called for you when the BTextView becomes part a window's view hierarchy; you shouldn't call it yourself, though you can override it to set a different default font and do other graphics initialization. For more information on when it's called, see the BView class.

An **AttachedToWindow()** function that's implemented by a derived class should begin by incorporating the BTextView version:

```
void MyText::AttachedToWindow()
{
    BTextView::AttachedToWindow()
    . . .
}
```

If it doesn't, the BTextView won't be able to properly display the text.

See also: AttachedToWindow() in the BView class, **SetFontName()**

BreaksAtChar()

virtual bool **BreaksAtChar(**ulong *aChar***)** const

Implemented by derived classes to return **TRUE** if the *aChar* character can break word selection, and **FALSE** if it cannot. The BTextView class calls this function when the user selects a word by double-clicking it. A return of **TRUE** means that the character breaks the selection—it cannot be selected as part of the word. A return of **FALSE** means that the character will be included in the selected word.

By default, **BreaksAtChar()** returns **TRUE** if the character is a **SPACE** (0x20), a **TAB** (0x09), a newline (**ENTER**, 0x0a), or some other character with an ASCII value less than that of a space, and **FALSE** otherwise.

It can be reimplemented to add hyphens to the list of characters that break word selection, as follows:

```
bool MyTextView::BreaksAtChar(ulong someChar)
{
    if ( someChar == '-' )
        return TRUE;
    return BTextView::BreaksAtChar(someChar);
}
```

See also: Text()

Copy()

virtual void **Copy(**BClipboard **clipboard***)**

Copies the current selection to the clipboard. The *clipboard* argument is identical to the global **be_clipboard** object.

See also: Paste(), **Cut()**

CountLines() see GoToLine()

CurrentLine() see GoToLine()

Cut()

virtual void **Cut(**BClipboard **clipboard***)**

Copies the current selection to the clipboard, deletes it from the BTextView's text, and removes it from the display. The *clipboard* argument is identical to the global **be_clipboard** object.

See also: Paste(), **Copy()**

Delete()

void **Delete(**void**)**

Deletes the current selection from the BTextView's text and removes it from the display, without copying it to the clipboard.

See also: Cut()

DisallowChar(), AllowChar()

void **DisallowChar(**ulong *aChar***)**

void **AllowChar(**ulong *aChar***)**

These functions inform the BTextView whether the user should be allowed to enter *aChar* into the text. By default, all characters are allowed. Call **DisallowChar()** for each character you want to prevent the BTextView from accepting, preferably when first setting up the object.

AllowChar() reverses the effect of **DisallowChar()**.

Alternatively, and for more control over the context in which characters are accepted or rejected, you can implement an **AcceptsChar()** function for the BTextView. **AcceptsChar()** is called for each key-down event that's reported to the object.

See also: AcceptsChar()

DoesAutoindent() see SetAutoindent()

DoesWordWrap() see SetWordWrap()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the text on-screen. The Interface Kit calls this function for you whenever the text display needs to be updated—for example, whenever the user edits the text, enters new characters, or scrolls the contents of the BTextView.

See also: Draw() in the BView class

GetSelection()

void **GetSelection(**long **start*, long **finish***)**

Provides the current selection by writing the offset before the first selected character into the variable referred to by *start* and the offset after the last selected character into the variable referred to by *finish*. If no characters are selected, both offsets will record the position of the current insertion point.

The offsets designate positions between characters. The position at the beginning of the text is offset 0, the position between the first and second characters is offset 1, and so on. If the 175th through the 202nd characters were selected, the *start* offset would be 174 and the *finish* offset would be 202.

If the text isn't selectable, both offsets will be 0.

See also: Select()

GoToLine(), CountLines(), CurrentLine()

void **GoToLine(**long *index***)**

long **CurrentLine(**void**)**

inline long **CountLines(**void**)**

GoToLine() moves the insertion point to the beginning of the line at *index*. The first line has an index of 0, the second line an index of 1, and so on. If the *index* is out-of-range, the insertion point is moved to the beginning of the line with the nearest in-range index—that is, to either the first or the last line.

CurrentLine() returns the index of the line where the first character of the selection—or the character following the insertion point—is currently located.

CountLines() returns how many lines of text the BTextView currently contains.

Like other functions that change the selection, **GoToLine()** doesn't automatically scroll the display to make the new selection visible. Call **ScrollToSelection()** to be sure that the user can see the start of the selection.

See also: ScrollToSelection()

Highlight()

void **Highlight(**long *start*, long *finish***)**

Highlights the characters from *start* through *finish*, where *start* and *finish* are the same sort of offsets into the text array as are passed to **Select()**.

Highlight() is the function that the BTextView calls to highlight the current selection. You don't need to call it yourself for this purpose. It's in the public API just in case you may need to highlight a range of text in some other circumstance.

See also: Select()

IndexAtPoint()

long **IndexAtPoint(**BPoint *point***)** const
long **IndexAtPoint(**float *x*, float *y***)** const

Returns the index of the character displayed closest to *point*—or (*x, y*)—in the BTextView's coordinate system. The first character in the text array is at index 0.

If the point falls after the last line of text, the return value is the index of the last character in the last line. If the point falls before the first line of text, or if the BTextView doesn't contain any text, the return value is 0.

Insert()

void **Insert(**const char **text*, long *length***)**
void **Insert(**const char **text***)**

Inserts *length* characters of *text*—or if a *length* isn't specified, all the characters of the *text* string up to the null character that terminates it—at the beginning of the current selection. The current selection is not deleted and the insertion is not selected.

See also: SetText()

IsEditable() see MakeEditable()

IsSelectable() see MakeSelectable()

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Enters text at the current selection in response to the user's typing. This function is called from the window's message loop for every report of a key-down event—once for every character the user types. However, it does nothing unless the BTextView is the focus view and the text it contains is editable.

If *aChar* is one of the arrow keys (**UP_ARROW**, **LEFT_ARROW**, **DOWN_ARROW**, or **RIGHT_ARROW**), **KeyDown()** moves the insertion point in the appropriate direction. If *aChar* is the **BACKSPACE** character, it deletes the current selection (or one character at the current insertion point). Otherwise, it checks whether the character was registered as unacceptable (by **DisallowChar()**) and it calls the **AcceptsChar()** hook function to give the application a chance to reject the character or handle it in some other way. If the character isn't disallowed and **AcceptsChar()** returns **TRUE**, it's entered into the text and displayed.

See also: KeyDown() in the BView class, **AcceptsChar()**, **DisallowChar()**

LineWidth()

float **LineWidth(**long *index* = 0**)** const

Returns the width of the line at *index*—or, if no index is given, the width of the first line. The value returned is the sum of the widths (in coordinate units) of all the characters in the line, from the first through the last, including tabs and spaces.

Line indices begin at 0.

If the *index* passed is out-of-range, it's reinterpreted to be the nearest in-range index—that is, as the index to the first or the last line.

MakeEditable(), IsEditable()

void **MakeEditable(**bool *flag* = **TRUE)**

bool **IsEditable(**void**)** const

The first of these functions sets whether the user can edit the text displayed by the BTextView; the second returns whether or not the text is currently editable. Text is editable by default.

To edit text, the user must be able to select it. Therefore, when **MakeEditable()** is called with an argument of **TRUE** (or with no argument), it makes the text both editable and selectable. Similarly, when **IsEditable()** returns **TRUE**, the text is selectable as well as editable; **IsSelectable()** will also return **TRUE**.

A value of **FALSE** means that the text can't be edited, but implies nothing about whether or not it can be selected.

See also: MakeSelectable()

MakeFocus()

virtual void **MakeFocus(**bool *flag* = **TRUE)**

Overrides the BView version of **MakeFocus()** to highlight the current selection when the BTextView becomes the focus view (when *flag* is **TRUE**) and to unhighlight it when the BTextView no longer is the focus view (when *flag* is **FALSE**). However, the current selection is highlighted only if the BTextView's window is the current active window.

This function is called for you whenever the user's actions make the BTextView become the focus view, or force it to give up that status.

See also: MakeFocus() in the BView class, **MouseDown()**

MakeResizable()

void **MakeResizable(**BView **containerView***)**

Makes the BTextView's frame rectangle and text rectangle automatically grow and shrink to exactly enclose all the characters entered by the user. The *containerView* is a view that should be resized with the BTextView; typically it's a view that draws a border around the text (like a BScrollView object) and is the parent of the BTextView. This function won't work without a container view.

MakeResizable() is an alternative to the automatic resizing behavior provided in the BView class. It triggers resizing on the user's entry of text, not on a change in the parent view's size. The two schemes are incompatible; the BTextView and the container view should not automatically resize themselves when their parents are resized.

< This function currently requires the text to be either left aligned or center aligned; it doesn't work for text that's right aligned. >

See also: SetAlignment()

MakeSelectable(), IsSelectable()

void **MakeSelectable(**bool *flag* = **TRUE)**

bool **IsSelectable(**void**)** const

The first of these functions sets whether it's possible for the user to select text displayed by the BTextView; the second returns whether or not the text is currently selectable. Text is selectable by default.

When text is selectable but not editable, the user can select one or more characters to copy to the clipboard, but can't position the insertion point (an empty selection), enter characters from the keyboard, or paste new text into the view.

Since the user must be able to select text to edit it, calling **MakeSelectable()** with an argument of **FALSE** causes the text to become uneditable as well as unselectable. Similarly, if **IsSelectable()** returns **FALSE**, the user can neither select nor edit the text; **IsEditable()** will also return **FALSE**.

A value of **TRUE** means that the text is selectable, but says nothing about whether or not it's also editable.

See also: MakeEditable()

MessageDropped()

virtual bool **MessageDropped(**BMessage **message*, BPoint *point***)**

Takes textual data from the dropped *message* and pastes it into the text. The text replaces the current selection, or is placed at the site of the current insertion point.

This function first looks in the BMessage for an entry named “text” registered as **ASCII_TYPE**. Failing that, it looks for a single character named “char” registered as **LONG_TYPE**. If successful in finding either entry, it adds the data to the text, updates the display on-screen, and returns **TRUE**. If unsuccessful, it returns **FALSE**.

See also: AcceptsChar()

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Overrides the BReceiver function to handle **CUT**, **COPY**, and **PASTE** messages, by calling the **Cut()**, **Copy()**, and **Paste()** virtual functions.

For the BTextView to get these messages, “Cut”, “Copy”, and “Paste” menu items should be:

- Assigned model messages with **CUT**, **COPY**, and **PASTE** as their **what** data members, and
- Targeted to the BTextView, or to the current focus view in the window that displays the BTextView.

The BTextView, through this function, takes care of the rest.

To inherit this functionality, **MessageReceived()** functions implemented by derived classes should be sure to call the BTextView version.

See also: SetMessage() and **SetTarget()** in the BMenuItem class

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Selects text and positions the insertion point in response to the user’s mouse actions. If the BTextView isn’t already the focus view for its window, this function calls **MakeFocus()** to make it the focus view.

MouseDown() is called for each mouse-down event that occurs inside the BTextView’s frame rectangle.

See also: MouseDown() and **MakeFocus()** in the BView class

Paste()

virtual void **Paste(**BClipboard **clipboard***)**

Takes textual data from the clipboard and pastes it into the text. The new text replaces the current selection, or is placed at the site of the current insertion point.

The *clipboard* argument is identical to the global **be_clipboard** object.

See also: Cut(), **Copy()**

Pulse()

virtual void **Pulse(**void**)**

Turns the caret marking the current insertion point on and off when the BTextView is the focus view in the active window. **Pulse()** is called by the system at regular intervals.

This function is first declared in the BView class.

See also: Pulse() in the BView class

ScrollToSelection()

void **ScrollToSelection(**void**)**

Scrolls the text so that the beginning of the current selection is within the visible region of the view, provided that the BTextView is equipped with a scroll bar that permits scrolling in the required direction (horizontal or vertical).

See also: ScrollBy() in the BView class

Select()

void **Select(**long *start*, long *finish***)**

Selects the characters from *start* up to *finish*, where *start* and *finish* are offsets into the BTextView's text. The offsets designate positions between characters. For example,

```
Select(0, 2);
```

selects the first two characters of text,

```
Select(17, 18);
```

selects the eighteenth character, and

```
Select(0, TextLength());
```

selects the entire text just as the **SelectAll()** function does. If *start* and *finish* are the same, the selection will be empty (an insertion point).

Normally, the selection is changed by the user. This function provides a way to change it programmatically.

If the BTextView is the current focus view in the active window, **Select()** highlights the new selection (or displays a blinking caret at the insertion point). However, it doesn't automatically scroll the contents of the BTextView to make the new selection visible. Call **ScrollToSelection()** to be sure that the user can see the start of the selection.

See also: Text(), **GetSelection()**, **ScrollToSelection()**, **GoToLine()**, **MouseDown()**

SelectAll()

void **SelectAll(**void**)**

Selects the entire text of the BTextView, and highlights it if the BTextView is the current focus view in the active window.

See also: Select()

SetAlignment(), Alignment()

void **SetAlignment(**alignment *where***)**

alignment **Alignment(**void**)** const

These functions set the way text is aligned within the text rectangle and return the current alignment. Three settings are possible:

ALIGN_LEFT	Each line is aligned at the left boundary of the text rectangle.
ALIGN_RIGHT	Each line is aligned at the right boundary of the text rectangle.
ALIGN_CENTER	Each line is centered between the left and right boundaries of the text rectangle.

The default is **ALIGN_LEFT**.

SetAutoindent(), DoesAutoindent()

void **SetAutoindent(**bool *flag***)**

bool **DoesAutoindent(**void**)** const

These functions set and return whether a new line of text is automatically indented the same as the preceding line. When set to **TRUE** and the user types Return at the end of a line that begins with tabs or spaces, the new line will automatically indent past those tabs and spaces to the position of the first visible character.

The default value is **FALSE**.

SetFontName(), SetFontSize(), SetFontRotation(), SetFontShear(), SetFontSymbolSet()

virtual void **SetFontName(**const char **name***)**

virtual void **SetFontSize(**float *points***)**

virtual void **SetFontRotation(**float *degrees***)**

virtual void **SetFontShear(**float *angle***)**

virtual void **SetFontSymbolSet(**const char **name***)**

These functions override their BView counterparts to redisplay the text in the new font, and to prevent the text displayed by a BTextView object from being rotated.

Font rotation is disabled; the BTextView version of **SetFontRotation()** does nothing. The other four functions invoke their BView counterparts to change the font, then make sure the entire text is rewrapped and redisplayed in the new font.

SetFontName() and **SetFontSize()** are called by **AttachedToWindow()** to set the BTextView's default font to 9-point "geneva".

See also: **SetFontName()** in the BView class

SetMaxChars()

void **SetMaxChars(**long *max***)**

Sets the maximum number of characters that the BTextView can accept. The default is the maximum number of characters that can be designated by a **long** integer, a number sufficiently large to accommodate all uses of a BTextView. Use this function only if you need to restrict the number of characters that the user can enter in a text field.

SetSpacing(), Spacing()

void **SetSpacing(**long *spacing***)**

long **Spacing(**void**)** const

These functions set and return the spacing between lines of text. A value of 1 indicates single spacing, 2 double spacing, 3 triple spacing, and so on.

Single spacing is the default.

SetTabWidth()

void **SetTabWidth(**float *width***)**

Sets the distance between tab stops to *width* coordinate units. All tabs have a uniform width.

The default tab width is 44.0.

SetText()

void **SetText(**const char **text*, long *length***)**
void **SetText(**const char **text***)**

Removes any text currently in the BTextView and copies *length* characters of *text* to replace it—or all the characters in the *text* string, up to the null character, if a *length* isn't specified. If *text* is **NULL** or *length* is 0, this function empties the BTextView. Otherwise, it copies the required number of *text* characters passed to it.

This function is typically used to set the text initially displayed in the view. If the BTextView is attached to a window, it's updated to show its new contents.

See also: **Text()**, **TextLength()**

SetTextRect(), TextRect()

void **SetTextRect(**BRect *rect***)**

inline BRect **TextRect(**void**)**

SetTextRect() makes *rect* the BTextView's text rectangle—the rectangle that locates where text is placed within the view. This replaces the text rectangle originally set in the BTextView constructor. The layout of the text is recalculated to fit the new rectangle, and the text is redisplayed.

TextRect() returns the current text rectangle.

See also: the BTextView constructor

SetWordWrap(), DoesWordWrap()

void **SetWordWrap(**bool *flag***)**

bool **DoesWordWrap(**void**)** const

These functions set and return whether the BTextView wraps lines on word boundaries, dropping entire words that don't fit at the end of a line to the next line. Words break on tabs, spaces, and other invisible characters; all adjacent visible characters wrap together.

By default, word wrapping is turned off (**DoesWordWrap()** returns **FALSE**). Lines break only on a newline character (where the user types return).

See also: SetTextRect()

Spacing() see **SetSpacing()**

Text()

const char ***Text(**void**)** const

Returns a pointer to the text contained in the BTextView, or **NULL** if it's empty. The returned pointer can be used to read the text, but not to alter it (use **SetText()**, **Insert()**, **Delete()**, and other BTextView functions to do that). The pointer may no longer be valid after the user or the program next changes the text.

This function returns the text as a null-terminated string. However, once the text is changed, the string will no longer be null-terminated.

See also: TextLength()

TextLength()

long **TextLength(**void**)** const

Returns the number of characters the BTextView currently contains—the number of characters that **Text()** returns (not counting the null terminator).

See also: Text(), **SetMaxChars()**

TextRect() see **SetTextRect()**

WindowActivated()

virtual void **WindowActivated(**bool *flag***)**

Highlights the current selection when the BTextView's window becomes the active window (when *flag* is **TRUE**)—provided that the BTextView is the current focus view—and removes the highlighting when the window ceases to be the active window (when *flag* is **FALSE**).

If the current selection is empty (if it's an insertion point), it's highlighted by turning the caret on and off (blinking it).

The Interface Kit calls this function for you whenever the BTextView's window becomes the active window or it loses that status.

See also: **WindowActivated()** in the BView class, **MakeFocus()**

BView

Derived from:	public BObject
Declared in:	<interface/View.h>

Overview

BView objects are the agents of drawing and event handling within windows. Each object sets up and takes responsibility for a particular *view*, a rectangular area that's associated with at most one window at a time. The object draws within the view rectangle and responds to reports of events elicited by the images drawn.

Classes derived from BView implement the actual functions that draw and handle events; BView merely provides the framework. For example, a BTextView object draws and edits text in response to the user's activity on the keyboard and mouse. A BButton draws the image of a button on-screen and responds when the button is clicked. BTextView and BButton inherit from the BView class—as do most classes in the Interface Kit.

The following Kit classes derive, directly or indirectly, from BView:

BMenu	BControl	BScrollBar
BMenuBar	BButton	BScrollView
BPopUpMenu	BCheckBox	BStringView
BListView	BRadioButton	BTextView
	BBox	

Serious applications will need to define their own classes derived from BView.

Views and Windows

For a BView to do its work, you must attach it to a window. The views in a window are arranged in a hierarchy—there can be views within views—with those that are most directly responsible for drawing and event handling located at the terminal branches of the hierarchy and those that contain and organize other views situated closer to its trunk and root. A BView begins life unattached. You can add it to a hierarchy by calling the **AddChild()** function of the BWindow, or of another BView.

Within the hierarchy, a BView object plays two roles:

- It's a BReceiver for messages delivered to the window thread. BViews implement the functions that respond to the most common system messages—including those that report keyboard and mouse events. They can also be targeted to receive application-defined messages that affect what they view displays.

- It's an agent for drawing. Adding a BView to a window gives it an independent graphics environment. A BView draws on the initiative of the BWindow and the Application Server, whenever they determine that the appearance of any part of the view rectangle needs to be "updated." It also draws on its own initiative in response to events.

The relationship of BViews to BWindows and the framework for drawing and handling events were discussed in the introduction to this chapter. The concepts and terminology presented there are assumed in this class description. See especially "BView Objects" on page 11, "The View Hierarchy" on page 13, "Drawing" beginning on page 18, and "Handling Events" beginning on page 40.

BViews can also be called upon to create bitmap images. See the BBitmap class for details.

Drag and Drop

The BView class supports a drag-and-drop user interface. The user can transfer a parcel of information from one place to another by dragging an image from a source view and dropping it on a destination view—perhaps a view in a different window or even a different application.

A source BView initiates dragging by calling **DragMessage()** from within its **MouseDown()** function. The BView bundles all information relevant to the dragging session into a BMessage object and passes it to **DragMessage()**. It also passes an image to represent the data package on-screen.

The Application Server then takes charge of the BMessage object and animates the image as the user drags it on-screen. As the image moves across the screen, the views it passes over are informed with **MouseMoved()** function calls. These notifications give views a chance to show the user whether or not they're willing to accept the message being dragged. When the user releases the mouse button, dropping the dragged message, the destination BView's **MessageDropped()** virtual function is called. The dragged BMessage is passed to the BView as a **MessageDropped()** argument.

Aside from creating a BMessage object and passing it to **DragMessage()**, or implementing **MouseMoved()** and **MessageDropped()** functions to handle any messages that come its way, there's nothing an application needs to do to support a drag-and-drop user interface. The bulk of the work is done by the Application Server and Interface Kit.

Locking the Window

If a BView is attached to a window, any operation that affects the view might also affect the window and the BView's shadow counterpart in the Application Server. For this reason, any code that calls a BView function should first lock the window—so that one thread can't modify essential data structures while another thread is using them. A window can be locked by only one thread at a time.

By default, before they do anything else, almost all BView functions check to be sure the caller has the window locked. If the window isn't properly locked, they print warning messages and fail.

This check should help you develop an application that correctly regulates access to windows and views. However, it adds a certain amount of time to each function call. Once your application has been debugged and is ready to ship, you can turn the check off by calling BWindow's **SetDiscipline()** function and passing it an argument of **FALSE**. The discipline flag is separately set for each window.

BView functions can require the window to be locked only if the view has a window to lock; the requirement can't be enforced if the BView isn't attached to a window. However, as discussed under "Views and the Server" on page 30 of the introduction to this chapter, many BView functions, including all those that depend on graphics parameters, don't work at all unless the view is attached—in which case the window must be locked.

Whenever the system calls a BView function to notify it of something—whenever it calls **WindowActivated()**, **Draw()**, **MessageReceived()** or another hook function—it first makes sure that the window is locked. The application doesn't have to explicitly lock the window when responding to an update, an event message, or some other notification. The window is already locked.

Derived Classes

When it comes time for a BView to draw, its **Draw()** virtual function is called automatically. When it needs to respond to an event, a virtual function named after the kind of event is called—**MouseDown()**, **KeyDown()**, **MessageDropped()**, and so on. Classes derived from BView implement these hook functions to do the particular kind of drawing and event handling characteristic of the derived class.

- Some classes derived from BView implement control devices—buttons, dials, selection lists, check boxes, and so on—that translate user actions on the keyboard and mouse into more explicit instructions for the application. In the Interface Kit, BMenu, BListView, BButton, BCheckBox, and BRadioButton are examples of control devices.
- Other BViews visually organize the display—for example, a view that draws a border around and arranges other views, or one that splits a window into two or

more resizable panels. The BBox, BScrollBar, and BScrollView classes fall into this category.

- Some BViews implement highly organized displays the user can manipulate, such as a game board or a scientific simulation.
- Perhaps the most important BViews are those that permit the user to create, organize, and edit data. These views display the current selection and are the focus of most user actions. They carry out the main work of an application. BTextView is the only Interface Kit example of such a view.

Almost all the BView classes defined in the Interface Kit fall into the first two of these groups. Control devices and organizational views can serve a variety of different kinds of applications, and therefore can be implemented in a kit that's common to all applications

However, the BViews that will be central to most applications fall into the last two groups. Of particular importance are the BViews that manage editable data. Unfortunately, these are not views that can be easily implemented in a common kit. Just as most applications devise their own data formats, most applications will need to define their own data-handling views.

Nevertheless, the BView class structures and simplifies the task of developing application-specific objects that draw in windows and interact with the user. It takes care of the lower-level details and manages the view's relationship to the window and other views in the hierarchy. You should make yourself familiar with this class before implementing you own application-specific BViews.

Hook Functions

AttachedToWindow()	Can be implemented to finish initializing the BView once it becomes part of a window's view hierarchy.
Draw()	Can be implemented to draw the view.
FrameMoved()	Can be implemented to respond to a message notifying the BView that it has moved in its parent's coordinate system.
FrameResized()	Can be implemented to respond to a message informing the BView that its frame rectangle has been resized.
KeyDown()	Can be implemented to respond to a message reporting a key-down event.

MakeFocus()	Makes the BView the focus view, or causes it to give up being the focus view; can be augmented to take any action the change in status may require.
MessageDropped()	Can be implemented to accept or reject a BMessage dropped on the view.
MouseDown()	Can be implemented to respond to a message reporting a mouse-down event.
MouseMoved()	Can be implemented to respond to a notification that the cursor has entered the view's visible region, moved within the visible region, or exited from the view.
Pulse()	Can be implemented to do something at regular intervals. This function is called repeatedly when no other messages are pending.
WindowActivated()	Can be implemented to respond to a notification that the BView's window has become the active window, or has lost that status.

Constructor and Destructor

BView()

BView(BRect *frame*, const char **name*, ulong *resizingMode*, ulong *flags***)**

Sets up a view with the *frame* rectangle, which is specified in the coordinate system of its eventual parent, and assigns the BView an identifying *name*, which can be **NULL**.

When it's created, a BView doesn't belong to a window and has no parent. It's assigned a parent by having another BView adopt it with the **AddChild()** function. If the other view is in a window, the BView becomes part of that window's view hierarchy. A BView can be made a child of the window's top view by calling BWindow's version of the **AddChild()** function.

When the BView gains a parent, the values in *frame* are interpreted in the parent's coordinate system. The sides of the view must be aligned on screen pixels. Therefore, the *frame* rectangle should not contain coordinates with fractional values. Fractional coordinates will be rounded to the nearest whole number.

The *resizingMode* flag determines the behavior of the view when its parent is resized. It can be any one of the following constants:

FOLLOW_LEFT_TOP
FOLLOW_TOP_RIGHT
FOLLOW_RIGHT_BOTTOM
FOLLOW_LEFT_BOTTOM
FOLLOW_LEFT_TOP_RIGHT
FOLLOW_LEFT_TOP_BOTTOM
FOLLOW_TOP_RIGHT_BOTTOM
FOLLOW_LEFT_RIGHT_BOTTOM
FOLLOW_ALL
FOLLOW_NONE

Before resizing, each side of a view's frame rectangle lies a certain distance from the corresponding side of its parent. The constants above state which of those distances should be maintained after the parent is resized—which of its parent's sides the child view should follow.

If a constant names opposite sides of the rectangle—left and right, or top and bottom—the view will necessarily be resized in that dimension when its parent is. **FOLLOW_ALL** means that the view will be resized in tandem with its parent, both horizontally and vertically.

If a side is not mentioned, the distance between that side of the view and the corresponding side of the parent is free to fluctuate. If the left side of a view doesn't follow the left side of its parent, or the top of the view doesn't follow its parent's top, the view will be able to move within its parent's coordinate system when the parent is resized. **FOLLOW_RIGHT_BOTTOM**, for example, keeps a view from being resized, but the view will move to follow the right bottom corner of its parent whenever the parent is resized. **FOLLOW_LEFT_TOP** prevents a view from being resized *and* from being moved.

FOLLOW_NONE keeps the view at its absolute position on-screen; the parent view is resized around it. (Nevertheless, because the parent is resized, the view may wind up being moved in its parent's coordinate system.)

Typically, a parent view is resized because the user resizes the window it's in. When the window is resized, the top view is too. Depending on how the *resizingMode* flag is set for the top view's children and for the descendants of its children, automatic resizing can cascade down the view hierarchy. A view can also be resized programmatically by the **ResizeTo()** and **ResizeBy()** functions.

The resizing mode can be changed after construction with the **SetResizingMode()** function.

The *flags* mask determines what kinds of notifications the BView will receive. It can be any combination of these four constants:

WILL_DRAW	Indicates that the BView has a **Draw()** function that needs to be called on updates—the view isn't simply a container for other views; it does some drawing on its own. If this flag isn't set, the BView won't receive update notifications.
PULSE_NEEDED	Indicates that the BView should receive **Pulse()** notifications.
FRAME_EVENTS	Indicates that the BView should receive **FrameResized()** and **FrameMoved()** notifications when its frame rectangle changes—typically as a result of the automatic resizing behavior described above. **FrameResized()** is called when the dimensions of the view change; **FrameMoved()** is called when the position of its left top corner in its parent's coordinate system changes.
FULL_UPDATE_ON_RESIZE	Indicates that the entire view should be updated when it's resized. If this flag isn't set, only the portions that resizing adds to the view will be included in the clipping region.

If none of these constants apply, *flags* can be **NULL**. The flags can be reset after construction with the **SetFlags()** function.

See also: SetResizingMode(), **SetFlags()**

~BView()

virtual **~BView(**void**)**

Removes the BView from the view hierarchy and ensures that each of its descendants is also removed and destroyed.

Member Functions

AddChild()

virtual void **AddChild(**BView **aView***)**

Makes *aView* a child of the BView. If *aView* already has a parent, it's removed from that view and added to this one. A view can have only one parent.

If the BView is attached to a window, *aView* and all of its descendants become attached to the same window. Each of them is notified of this change through an **AttachedToWindow()** function call.

See also: **AddChild()** in the BWindow class, **AttachedToWindow()**, **RemoveChild()**

AddLine() see **BeginLineArray()**

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Implemented by derived classes to complete the initialization of the BView when it's assigned to a window. A BView is assigned to a window when it, or one of its ancestors in the view hierarchy, becomes a child of a view already attached to a window.

AttachedToWindow() is called immediately after the BView is formally made a part of the window's view hierarchy and after it has become known to the Application Server. The **Window()** function can identify which BWindow the BView belongs to.

All of the BView's children, if it has any, also become attached to the window and receive their own **AttachedToWindow()** notifications. However, the BView receives the notification before any of its children do and before they are recognized as part of the window's view hierarchy. This function should therefore do nothing that depends on descendent views being attached to the window. However, it can depend on ancestor views being attached.

AttachedToWindow() is often implemented to set up a view's graphics environment, something that can't be done before the view belongs to a window. For example:

```
void MyView::AttachedToWindow()
{
    BRect bounds = Bounds();

    MyBaseClass::AttachedToWindow();

    ScrollTo(0, dataRect.bottom - bounds.Height());
    SetFontName("chicago");
    SetFontSize(14);
    SetBackColor(192, 192, 192);
}
```

The default (BView) version of **AttachedToWindow()** is empty.

See also: AddChild(), **Window()**

BackColor() see **SetFrontColor()**

BeginLineArray(), AddLine(), EndLineArray()

void **BeginLineArray(**long *count***)**

void **AddLine(**BPoint *start*, BPoint *end*, rgb_color *color***)**

void **EndLineArray(**void**)**

These functions provide a more efficient way of drawing a large number of lines than repeated calls to **StrokeLine()**. **BeginLineArray()** signals the beginning of a series of up to *count* **AddLine()** calls; **EndLineArray()** signals the end of the series. Each **AddLine()** call defines a line from the *start* point to the *end* point, associates it with a particular *color*, and adds it to the array. The lines can each be a different color; they don't have to be contiguous. When **EndLineArray()** is called, all the lines are drawn—using the then current pen size—in the order that they were added to the array.

These functions don't change any graphics parameters. For example, they don't move the pen or change the current front and background colors. Parameter values that are in effect when **EndLineArray()** is called are the ones used to draw the lines. The front and background colors are ignored in favor of the *color* specified for each line.

The *count* passed to **BeginLineArray()** is an upper limit on the number of lines that can be drawn. Keeping the count close to accurate and within reasonable bounds helps the efficiency of the line-array mechanism. It's a good idea to keep it less than 256; above that number, memory requirements begin to impinge on performance.

See also: StrokeLine()

BeginRectTracking(), EndRectTracking()

void **BeginRectTracking(**BRect *rect*, track_style *style* = **TRACK_WHOLE_RECT)**

void **EndRectTracking(**void**)**

These functions instruct the Application Server to display a rectangular outline that will track the movement of the cursor. **BeginRectTracking()** puts the rectangle on-screen and initiates tracking; **EndRectTracking()** terminates tracking and removes the rectangle. The initial rectangle, *rect*, is specified in the BView's coordinate system.

This function supports two kinds of tracking, depending on the constant passed as the *style* argument:

TRACK_WHOLE_RECT	The whole rectangle moves with the cursor. Its position changes, but its size remains fixed.
TRACK_RECT_CORNER	The left top corner of the rectangle remains fixed within the view while its right and bottom edges move with the cursor.

Tracking is typically initiated from within a BView's **MouseDown()** function and is allowed to continue as long as a mouse button is held down. For example:

```
void MyView::MouseDown(BPoint point)
{
    ulong buttons;

    BRect rect(point, point);
    BeginRectTracking(rect, TRACK_RECT_CORNER);
    do {
        snooze(40);
        GetMouse(&point, &buttons);
    } while ( buttons );
    EndRectTracking();

    rect.SetRightBottom(point);
    . . .
}
```

This example uses **BeginRectTracking()** to drag out a rectangle from the point reported in the mouse-down event. It sets up a modal loop to periodically check on the state of the mouse buttons. Tracking ends when the user releases all buttons. The right and bottom sides of the rectangle are then updated from the cursor location last reported by the **GetMouse()** function.

See also: ConvertToScreen(), GetMouse()

Bounds()

BRect **Bounds(**void**)** const

Returns the BView's bounds rectangle. If the BView is attached to a window, this function gets the current bounds rectangle from the Application Server. If not, it assigns the BView a default coordinate system and returns a bounds rectangle that's the same size and shape as the frame rectangle, but with the left and top sides at 0.

See also: Frame()

ChildAt(), CountChildren()

BView ***ChildAt(**long *index***)** const

long **CountChildren(**void**)** const

< The first of these functions returns the child BView at *index*, or **NULL** if the BView has no such child. The second returns the number of children the BView has. Indices begin at 0 and the children are not arranged in any particular order. Don't rely on these functions as they may not remain in the API in this form >

ConvertToParent(), ConvertFromParent()

void **ConvertToParent(**BPoint **localPoint***)** const
void **ConvertToParent(**BRect **localRect***)** const

void **ConvertFromParent(**BPoint **parentPoint***)** const
void **ConvertFromParent(**BRect **parentRect***)** const

These functions convert points and rectangles to and from the coordinate system of the BView's parent. **ConvertToParent()** converts the point referred to by *localPoint*, or the rectangle referred to by *localRect*, from the BView's coordinate system to the coordinate system of its parent. **ConvertFromParent()** does the opposite; it converts the point referred to by *parentPoint*, or the rectangle referred to by *parentRect*, from the coordinate system of the BView's parent to the BView's own coordinate system.

Both functions fail if the BView isn't attached to a window.

See also: ConvertToScreen()

ConvertToScreen(), ConvertFromScreen()

void **ConvertToScreen(**BPoint **localPoint***)** const
void **ConvertToScreen(**BRect **localRect***)** const

void **ConvertFromScreen(**BPoint **screenPoint***)** const
void **ConvertFromScreen(**BRect **screenRect***)** const

These functions convert points and rectangles to and from the global screen coordinate system. **ConvertToScreen()** converts the point referred to by *localPoint*, or the rectangle referred to by *localRect*, from the BView's coordinate system to the screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts the point referred to by *screenPoint*, or the rectangle referred to by *screenRect*, from the screen coordinate system to the BView's local coordinate system.

Neither function will work if the BView isn't attached to a window.

See also: **ConvertToScreen()** in the BWindow class, **ConvertToParent()**

CopyBits()

void **CopyBits(**BRect *source*, BRect *destination***)**

Copies the image displayed in the *source* rectangle to the *destination* rectangle, where both rectangles lie within the view and are stated in the BView's coordinate system.

If the two rectangles aren't the same size, the source image is scaled to fit. If not all of the destination rectangle lies within the BView's visible region, the source image is clipped rather than scaled.

If not all of the source rectangle lies within the BView's visible region, only the visible portion is copied. It's mapped to the corresponding portion of the destination rectangle. The BView is then invalidated so its **Draw()** function will be called to update the part of the destination rectangle that can't be filled with the source image.

CountChildren() see ChildAt()

DragMessage()

void **DragMessage(**BMessage **message*, BBitmap **image*, BPoint *point***)**
void **DragMessage(**BMessage **message*, BRect *rect***)**

Initiates a drag-and-drop session. The first argument, *message*, is a BMessage object that bundles the information that will be dragged and dropped on the destination view. Once passed to **DragMessage()**, this object becomes the responsibility of—and will eventually be freed by—the system. You shouldn't free it yourself or try to access it later. (Since data is copied when it's added to a BMessage, only the copies are automatically freed, not the originals).

The second argument, *image*, represents the message on-screen; it's the visible image that the user drags. Like the BMessage, this BBitmap object becomes the responsibility of the system; it will be freed when the message is dropped. If you want to keep the image yourself, make a copy to pass to **DragMessage()**. The image isn't dropped on the destination BView; if you want the destination to have the image, you must add it to the *message* as well as pass it as the *image* argument.

The final argument, *point*, locates the point within the image that's aligned with the hot spot of the cursor—that is, the point that's aligned with the location passed to **MouseDown()** or returned by **GetMouse()**. It's stated within the coordinate system of the source image and should lie somewhere within its bounds rectangle. The bounds rectangle and coordinate system of a BBitmap are set when the object is constructed.

Alternatively, you can specify that an outline of a rectangle, *rect*, should be dragged instead of an image. The rectangle is stated in the BView's coordinate system. (Therefore, a *point* argument isn't needed to align it with the cursor.)

This function works only for BViews that are attached to a window.

See also: the BMessage class in the Application Kit, **MessageDropped()**, the BBitmap class

Draw()

virtual void **Draw(**BRect *updateRect***)**

Implemented by derived classes to draw the *updateRect* portion of the view. The update rectangle is stated in the BView's coordinate system. It's the smallest rectangle that encloses the current clipping region for the view.

Since the Application Server won't render anything a BView draws outside its clipping region, applications will be more efficient if they avoid sending drawing instructions to the Server for images that don't intersect with *updateRect*. For more efficiency and precision, you can ask for the clipping region itself (by calling **GetClippingRegion()**) and confine drawing to images that intersect with it.

A BView's **Draw()** function is called (as the result of an update message) whenever the view needs to present itself on-screen. This may happen when:

- The window the view is in is first shown on-screen, or shown after being hidden (see the BWindow version of the **Hide()** function).
- The view is made visible after being hidden (see BView's **Hide()** function).
- Obscured parts of the view are revealed, as when a window is moved from in front of the view or an image is dragged across the view.
- The view is resized.
- The contents of the view are scrolled (see **ScrollBy()**).

- A child view is added, removed, or resized.
- A rectangle has been invalidated that includes at least some of the view (see **Invalidate()**).
- **CopyBits()** can't completely fill a destination rectangle within the view.

See also: **UpdateIfNeeded()** in the BWindow class, **Invalidate()**, **GetClippingRegion()**

DrawBitmap()

void **DrawBitmap(**const BBitmap *_image_, BPoint _point_**)**
void **DrawBitmap(**const BBitmap *_image_, BRect _destination_**)**
void **DrawBitmap(**const BBitmap *_image_, BRect _source_, BRect _destination_**)**

Places a bitmap *image* in the view, either at *point* or within the *destination* rectangle. These locations are specified in the BView's coordinate system.

If a *source* rectangle is given, only that part of the bitmap image is drawn. Otherwise, the entire bitmap is placed at the location specified. The source rectangle is stated in the internal coordinates of the BBitmap object.

If the source image is bigger than the destination rectangle, it's scaled to fit.

See also: "Drawing Modes" on page 27 in the chapter introduction, the BBitmap class

DrawChar()

void **DrawChar(**char *c***)**

Draws the character *c* at the current pen position and moves the pen to a position immediately to the right of the character. This function is equivalent to passing a string of one character to **DrawString()**.

See also: **DrawString()**

DrawingMode() see SetDrawingMode()

DrawString()

void **DrawString(**const char *_string_**)**
void **DrawString(**const char *_string_, long _length_**)**

Draws *length* characters of *string*—or, if the number of characters isn't specified, all the characters in the string, up to the null terminator ('\0').

This function places the first character on a baseline that begins at the current pen position and moves the pen to the baseline after the last character drawn. A series of **DrawString()** calls will produce a continuous string. For example, these two lines of code,

```
DrawString("tog");
DrawString("ether");
```

will produce the same result as this one:

```
DrawString("together");
```

See also: MovePenBy(), **SetFontName()**

EndLineArray() see BeginLineArray()

EndRectTracking() see BeginRectTracking()

FillArc() see StrokeArc()

FillEllipse() see StrokeEllipse()

FillPolygon() see StrokePolygon()

FillRect() see StrokeRect()

FillRoundRect() see StrokeRoundRect()

FillTriangle() see StrokeTriangle()

FindView()

BView ***FindView(**const char **name***)** const

Returns the BView identified by *name*, or **NULL** if the view can't be found. Names are assigned by the BView constructor and can be modified by the **SetName()** function.

FindView() begins the search by checking whether the BView's name matches *name*. If not, it continues to search down the view hierarchy, among the BView's children and more distant descendants. To search the entire view hierarchy, use the BWindow version of this function.

See also: FindView() in the BWindow class, **SetName()**

Flags() see SetFlags()

Flush(), Sync()

void **Flush(**void**)** const

void **Sync(**void**)** const

These functions flush the window's connection to the Application Server. If the BView isn't attached to a window, neither function has an effect.

For reasons of efficiency, the window's connection to the Application Server is buffered. Drawing instructions destined for the Server are placed in the buffer and dispatched as a group when the buffer becomes full. Flushing empties the buffer, sending whatever instructions happen to be in it to the Server, even if it's not yet full.

The buffer is automatically flushed on every update. However, if you do any drawing outside the update mechanism—in response to event messages, for example—you need to explicitly flush the connection so that drawing instructions won't languish in the buffer while waiting for it to fill up or for the next update.

Flush() simply flushes the buffer and returns. It does the same work as BWindow's function of the same name.

Sync() flushes the connection, then waits until the Server has executed the last instruction that was in the buffer before returning. This alternative to **Flush()** prevents the application from getting ahead of the Server (ahead of what the user sees on-screen) and keeps both processes synchronized.

(Note that all BViews attached to a window share the same connection to the Application Server. Calling **Flush()** or **Sync()** for any one of them flushes the buffer for all of them.)

See also: **Flush()** in the BWindow class

Frame()

BRect **Frame(**void**)** const

Returns the BView's frame rectangle. The frame rectangle is first set by the BView constructor and is altered only when the view is moved or resized. It's stated in the coordinate system of the BView's parent.

See also: **MoveBy()**, **ResizeBy()**, the BView constructor

FrameMoved()

virtual void **FrameMoved(**BPoint *parentPoint***)**

Implemented by derived classes to respond to a notification that the view has moved within its parent's coordinate system. *parentPoint* gives the new location of the left top corner of the BView's frame rectangle.

FrameMoved() is called only if the **FRAME_EVENTS** flag is set and the BView is attached to a window.

If the view is both moved and resized, **FrameMoved()** is called before **FrameResized()**. This might happen, for example, if the BView's automatic resizing mode is **FOLLOW_TOP_RIGHT_BOTTOM** and its parent is resized both horizontally and vertically.

The default (BView) version of this function is empty.

< Currently, **FrameMoved()** is also called when a hidden window is shown on-screen. >

See also: **MoveBy()**, **FrameMoved()** in the BWindow class, **SetFlags()**

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Implemented by derived classes to respond to a notification that the view has been resized. The arguments state the new *width* and *height* of the view. The resizing could have been the result of a user action (resizing the window) or of a programmatic one (calling **ResizeTo()** or **ResizeBy()**).

FrameResized() is called only if the **FRAME_EVENTS** flag is set and the BView is attached to a window.

BView's version of this function is empty.

See also: **ResizeBy()**, **FrameResized()** in the BWindow class, **SetFlags()**

FrontColor() see SetFrontColor()

GetCharEscapements(), GetCharEdges()

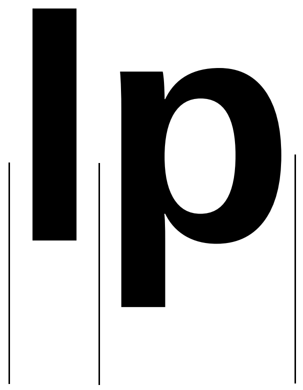
void **GetCharEscapements(**char *charArray*[], long *numChars*,
short *escapementArray*[], float **factor***)** const

void **GetCharEdges(**char *charArray*[], long *numChars*,
edge_info *edgeArray*[]**)** const

These two functions are designed for programmers who want to precisely position characters on the screen or printed page. For each character passed in the *charArray*, they write information about the horizontal dimension of the character into the *escapementArray* or the *edgeArray*. Both functions assume the BView's current font. (Therefore, neither has any effect unless the BView is attached to a window.)

Escapement. An "escapement" is simply the width of a character recorded in very small units. The units are sufficiently tiny to permit detailed information to be kept in integer form for every character in the font. Because the units are small, escapement values are quite large. (The term “escapement” has its historical roots in the fact that the carriage of a typewriter had to move or “escape” a certain distance after each character was typed to make room for the next character.)

The escapement of a character measures the amount of horizontal room it requires when positioned between other characters in a line of text. It includes a measurement of the space required to display the character itself, plus some extra room on the left and right edges to separate the character from its neighbors. In a proportionally spaced font, each character has a distinctive escapement. The illustration below shows the approximate escapements for the letters ‘l’ and ‘p’ as they might appear together in a word like “help” or “ballpark.” The escapement for each character is the distance between the vertical lines:



GetCharEscapements() measures the same space that functions such as BView’s **StringWidth()** and BTextView’s **LineWidth()** do, though it measures each character individually and records the result in arbitrary (rather than coordinate) units.

The escapement of a character in a particular font is a constant no matter what the font size. To convert an escapement value to coordinate units, you must multiply it by three values:

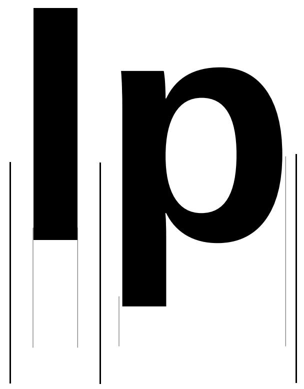
- A floating-point conversion factor,
- The font size (in points), and
- The resolution of the output device.

GetCharEscapements() writes the conversion factor into the variable referred to by *factor*. **GetFontInfo()** can provide the current font size. When the output device is a printer, the resolution should be the actual resolution (the dpi or “dots per inch”) at which it prints. When the output device is the screen, the resolution should be 72. (This reflects the fact that screen pixels are taken to equal coordinate units—and one coordinate unit is 1/72 of an inch, or roughly equivalent to one typographical point.)

Edges. Edge values measure how far a character outline is inset from its left and right escapement boundaries. **GetCharEdges()** provides edge values in standard coordinate units, not escapement units, < although those units are currently declared **short** rather than **float** >. It takes into account the size of the current font. It places the edge values into an array of **edge_info** structures. Each structure has a **left** and a **right** data member, as follows:

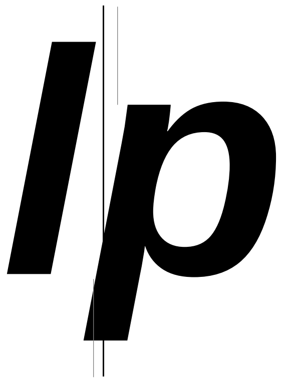
```
typedef struct {
    short left;
    short right;
} edge_info;
```

The illustration below shows typical character edges. As in the illustration above, the solid vertical lines mark escapement boundaries. The dotted lines mark off the part of each escapement that's an edge, the distance between the character outline and the escapement boundary:



This is the normal case. The left edge is a positive value measured rightward from the left escapement boundary. The right edge is a negative value measured leftward from the right escapement boundary.

However, if the characters of a font overlap, the left edge can be a negative value and the right edge can be positive. This is illustrated below:



Note that the italic '*l*' extends beyond its escapement to the right, and that the '*p*' begins before its escapement to the left. In this case, instead of separating the adjacent characters, the edges determine how much they overlap.

Edge values are specific to each character and depend on nothing but the character (and the font). They don't take into account any contextual information; for example, the right edge for italic '*l*' would be the same no matter what letter followed. Edge values therefore aren't sufficient to decide how character pairs can be kerned. Kerning is contextually dependent on the combination of two particular characters.

See also: GetFontInfo()

GetClippingRegion()

void **GetClippingRegion(**BRegion **region***)** const

Modifies the BRegion object passed as an argument so that it describes the current clipping region of the BView, the region where the BView is allowed to draw. It's most efficient to allocate temporary BRegions on the stack:

```
BRegion clipper;
GetClippingRegion(&clipper);
. . .
```

Ordinarily, the clipping region is the same as the visible region of the view, the part of the view currently visible on-screen. The visible region is equal to the view's bounds rectangle minus:

- The frame rectangles of its children,
- Any areas that are clipped because the view doesn't lie wholly within the frame rectangles of all its ancestors in the view hierarchy, and
- Any areas that are obscured by other windows or that lie in a part of the window that's off-screen.

The clipping region can be smaller than the visible region if the program restricted it by calling **SetClippingRegion()**. It will exclude any area that doesn't intersect with the region passed to **SetClippingRegion()**.

While the BView is being updated, the clipping region contains just those parts of the view that need to be redrawn. This may be smaller than the visible region, or the region restricted by **SetClippingRegion()**, if:

- The update occurs during scrolling. The clipping region will exclude any of the view's visible contents that the Application Server is able to shift to their new location and redraw automatically.
- The view rectangle has grown (because, for example, the user resized the window larger) and the update is needed only to draw the new parts of the view.

- The update was caused by **Invalidate()** and the rectangle passed to **Invalidate()** didn't cover all of the visible region.
- The update was necessary because **CopyBits()** couldn't fill all of a destination rectangle.

This function works only if the BView is attached to a window. Unattached BViews can't draw and therefore have no clipping region.

See also: SetClippingRegion(), **Draw()**, **Invalidate()**

GetFontInfo()

void **GetFontInfo(**font_info **fontInfo***)** const

Writes information about the BView's current font into the structure referred to by *fontInfo*. The **font_info** structure contains the following fields:

font_name **name**	The name of the font, which can be as long as 32 characters, plus a null terminator. The name can be set by BView's **SetFontName()** function.
short **size**	The size of the font in points. It can be set by **SetFontSize()**.
short **shear**	The shear angle, which is 90.0° by default and can vary between 45.0° and 135.0°. It can be set by **SetFontShear()**.
short **rotation**	The angle of rotation, which is 0.0° by default. It's set by **SetFontRotation()**.
short **ascent**	How far characters ascend above the baseline.
short **descent**	How far characters descend below the baseline.
short **leading**	The amount of space separating lines (between the descent of the line above and the ascent of the line below).

The ascent, descent, and leading are measured in coordinate units. < The **font_info** structure will be converted to floating-point values in a future release. >

See also: SetFontName()

GetKeys()

void **GetKeys(**key_info **keyInfo*, bool *checkQueue***)**

Writes information about the state of the keyboard into the **key_info** structure referred to by *keyInfo*. This structure contains the following fields:

ulong **char_code**	An ASCII character value, such as 'a' or **BACKSPACE**.
ulong **key_code**	A code identifying the key that produced the character.
ulong **modifiers**	A mask indicating which modifier keys are down and which keyboard locks are on.
uchar **key_states**[16]	A bit field that records the state of all the keys on the keyboard, and all keyboard locks.

These fields match the BMessage entries that record information about a key-down event.

If the *checkQueue* flag is **FALSE**, **GetKeys()** provides information about the current state of the keyboard.

However, if the *checkQueue* flag is **TRUE**, **GetKeys()** first checks the message queue to see whether it contains any messages reporting keyboard (key-down or key-up) events. If there are keyboard messages waiting in the queue, it takes the information from the oldest event, places it in the *keyInfo* structure, and removes the message from the queue. Each time **GetKeys()** is called, it gets another keyboard message from the queue. If the queue doesn't contain any keyboard messages, it reports the current state of the keyboard, just as if *checkQueue* were **FALSE**.

When called repeatedly in a loop, **GetKeys()** will empty the queue of keyboard messages and then reflect the current state of the keyboard. In this way, you can be sure that your application has not jumped ahead of the user and overlooked any reports of the user's keyboard actions.

This function never looks at the current message, even if it happens to report a keyboard event and *checkQueue* is **TRUE**. The current message isn't in the queue. To get information about the current message, you must call BLooper's **CurrentMessage()** function:

```
BMessage *current == myView->Window()->CurrentMessge();
```

If **GetKeys()** takes a keyboard message from the queue, all the **key_info** fields are filled in from the event message. However, if it captures the current state of the keyboard, the **char_code** and **key_code** fields are set to 0; these fields are appropriate only for reporting particular events.

When the **modifiers** field reflects the current keyboard state, it contains the same information that the **Modifiers()** function returns.

The **key_states** array works identically to the "states" array passed in a key-down message. See "Key States" on page 61 for information on how to read the array.

See also: **Modifiers()**, **KeyDown()**, "Keyboard Information" on page 53 of the chapter introduction

GetMouse()

void **GetMouse(**BPoint **cursor*, ulong **buttons*, bool *checkQueue* = **TRUE)** const

Provides the location of the cursor and the state of the mouse buttons. The position of the cursor is recorded in the variable referred to by *cursor*; it's provided in the BView's own coordinates. A bit is set in the variable referred to by *buttons* for each mouse button that's down. < There currently is no API for distinguishing between the buttons. >

The cursor doesn't have to be located within the view for this function to work; it can be anywhere on-screen. However, the BView must be attached to a window.

If the *checkQueue* flag is set to **FALSE**, **GetMouse()** provides information about the current state of the mouse buttons and the current location of the cursor.

If *checkQueue* is **TRUE**, as it is by default, this function first looks in the message queue for any pending reports of mouse-moved or mouse-up events. If it finds any, it takes the one that has been in the queue the longest (the oldest event), removes it from the queue, and reports the *cursor* location and *button* states that were recorded in the message. Each **GetMouse()** call removes another message from the queue. If the queue doesn't hold any **MOUSE_MOVED** or **MOUSE_UP** messages, **GetMouse()** reports the current state of the mouse and cursor, just as if *checkQueue* were **FALSE**.

This function is typically called from within a **MouseDown()** function to track the location of the cursor and wait for the mouse button to go up. By having it check the message queue, you can be sure that you haven't overlooked any of the cursor's movement or missed a mouse-up event (quickly followed by another mouse-down) that might have occurred before the first **GetMouse()** call.

See also: **Modifiers()**

Hide(), Show()

virtual void **Hide(**void**)**

virtual void **Show(**void**)**

These functions hide a view and show it again.

Hide() makes the view invisible without removing it from the view hierarchy. The visible region of the view will be empty and the BView won't receive update messages. If the BView has children, they also are hidden.

Show() unhides a view that had been hidden. This function doesn't guarantee that the view will be visible to the user; it merely undoes the effects of **Hide()**. If the view didn't have any visible area before being hidden, it won't have any after being shown again (given the same conditions).

Calls to **Hide()** and **Show()** can be nested. For a hidden view to become visible again, the number of **Hide()** calls must be matched by an equal number of **Show()** calls.

However, **Show()** can only undo a previous **Hide()** call on the same view. If the view became hidden when **Hide()** was called to hide the window it's in or to hide one of its ancestors in the view hierarchy, calling **Show()** on the view will have no effect. For a view to come out of hiding, its window and all its ancestor views must be unhidden.

Hide() and **Show()** can affect a view before it's attached to a window. The view will reflect its proper state (hidden or not) when it becomes attached. Views are created in an unhidden state.

See also: **Hide()** in the BWindow class, **IsHidden()**

Invalidate()

void **Invalidate(**BRect *rect***)**
void **Invalidate(**void**)**

Invalidates the *rect* portion of the view, causing update messages—and consequently **Draw()** notifications—to be generated for the BView and all descendants that lie wholly or partially within the rectangle. The rectangle is stated in the BView's coordinate system.

If no rectangle is specified, the BView's entire bounds rectangle is invalidated.

Since only BViews that are attached to a window can draw, only attached BViews can be invalidated.

See also: **Draw()**, **GetClippingRegion()**, **UpdateIfNeeded()** in the BWindow class

InvertRect()

void **InvertRect(**BRect *rect***)**

Inverts all the colors displayed within the *rect* rectangle. A subsequent **InvertRect()** call on the same rectangle restores the original colors.

The rectangle is stated in the BView's coordinate system.

See also: **system_colors()** global function

IsFocus()

bool **IsFocus(**void**)** const

Returns **TRUE** if the BView is the current focus view for its window, and **FALSE** if it's not. The focus view changes as the user chooses one view to work in and then another—for example, as the user moves from one text field to another when filling out an on-screen form. The change is made programmatically through the **MakeFocus()** function.

See also: CurrentFocus() in the BWindow class, **MakeFocus()**

IsHidden()

bool **IsHidden(**void**)** const

Returns **TRUE** if the view has been hidden by the **Hide()** function, and **FALSE** otherwise.

This function returns **TRUE** whether **Hide()** was called to hide the BView itself, to hide an ancestor view, or to hide the BView's window. When a window is hidden, all its views are hidden with it. When a BView is hidden, all its descendants are hidden with it.

If the view has no visible region—perhaps because it lies outside its parent's frame rectangle or is obscured by a window in front—this function may nevertheless return **FALSE**. It reports only whether the **Hide()** function has been called to hide the view, hide one of the view's ancestors in the view hierarchy, or hide the window where the view is located.

If the BView isn't attached to a window, **IsHidden()** returns the state that it will assume when it becomes attached. By default, views are not hidden.

See also: Hide()

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Implemented by derived classes to respond to a message reporting a key-down event. Whenever a BView is the focus view of the active window, it receives a **KeyDown()** notification for each character the user types, except for those that:

- Are produced while a Command key is held down. Command key events are interpreted as keyboard shortcuts.

- Can operate the default button in a window. The BButton object's **KeyDown()** function is called, rather than the focus view's.

The argument, *aChar,* names the character reported in the event. It's an ASCII value that takes into account the affect of any modifier keys that were held down or keyboard locks that were in effect at the time. For example, Shift-*i* is reported as uppercase 'I' (0x49) and Control-*i* is reported as a **TAB** (0x09).

The character can be tested against ASCII codes and these constants:

BACKSPACE	**LEFT_ARROW**	**INSERT**
ENTER	**RIGHT_ARROW**	**DELETE**
SPACE	**UP_ARROW**	**HOME**
TAB	**DOWN_ARROW**	**END**
ESCAPE		**PAGE_UP**
	FUNCTION_KEY	**PAGE_DOWN**

Only keys that generate characters produce key-down events; the modifier keys on their own do not.

You can determine which modifier keys were being held down at the time of the event by calling BLooper's **CurrentMessage()** function and looking up the "modifiers" entry in the BMessage it returns. If *aChar* is **FUNCTION_KEY** and you want to know which key produced the character, you can look up the "key" entry in the BMessage and test it against these constants:

F1_KEY	**F6_KEY**	**F11_KEY**
F2_KEY	**F7_KEY**	**F12_KEY**
F3_KEY	**F8_KEY**	**PRINT_KEY** (Print Screen)
F4_KEY	**F9_KEY**	**SCROLL_KEY** (Scroll Lock)
F5_KEY	**F10_KEY**	**PAUSE_KEY**

For example:

```
if ( aChar == FUNCTION_KEY ) {
    BMessage *msg = Window()->CurrentMessage();
    long key = msg->FindLong("key");
    if ( msg->Error == NO_ERROR ) {
        switch ( key ) {
        case F1_KEY:
            . . .
            break;
        case F2_KEY:
            . . .
            break;
         . . .
        }
    }
}
```

The BView version of **KeyDown()** is empty.

See also: "Key-Down Events" on page 46 and "Keyboard Information" on page 53 of the chapter introduction, **FilterKeyDown()** and **SetDefaultButton()** in the BWindow class, **Modifiers()**

LeftTop()

BPoint **LeftTop(**void**)** const

Returns the coordinates of the left top corner of the view—the smallest *x* and *y* coordinate values within the bounds rectangle.

See also: **LeftTop()** in the BRect class, **Bounds()**

Looper() see Window()

MakeFocus()

virtual void **MakeFocus(**bool *flag* = **TRUE)**

Makes the BView the current focus view for its window (if *flag* is **TRUE**), or causes it to give up that status (if *flag* is **FALSE**). The focus view is the view that displays the current selection and is expected to handle reports of key-down events when the window is the active window. There can be no more than one focus view per window at a time.

When called to make a BView the focus view, this function invokes **MakeFocus()** for the previous focus view, passing it an argument of **FALSE**. It's thus called twice—once for the new and once for the old focus view.

Calling **MakeFocus()** is the only way to make a view the focus view; the focus doesn't automatically change on mouse-down events. BViews that can display the current selection (including an insertion point) or that can accept pasted data should call **MakeFocus()** in their **MouseDown()** functions.

A derived class can override **MakeFocus()** to add code that takes note of the change in status. For example, a BView that displays selectable data may want to highlight the current selection when it becomes the focus view, and remove the highlighting when it's no longer the focus view.

If the BView isn't attached to a window, this function has no effect.

See also: **CurrentFocus()** in the BWindow class, **IsFocus()**

MessageDropped()

virtual bool **MessageDropped(**BMessage **message*, BPoint *point***)**

Implemented by derived classes to read data from a *message* that the user dragged and dropped on the view and to initiate whatever course of action this new information entails. The BMessage object is freed after **MessageDropped()** returns, so you must copy any of its data you want to keep.

When the message was dropped, the cursor was located at *point* within the BView's coordinate system.

If the BView accepts the message, it should return **TRUE**. A return of **FALSE** rejects the message and causes **MessageDropped()** to be called for the BView's parent. The notification works its way up the view hierarchy until it finds a BView that will return **TRUE**, or it reaches the top view.

The BView version of this function always returns **FALSE**; by default, views don't accept dropped messages.

Often the messages that can be successfully dropped on a view hold data that could also be pasted from the clipboard. To handle this data in common code, **MessageDropped()** and the **Paste()** function you define for the view can pass the data to a third function implemented for this purpose. **MessageDropped()** would extract the data from the *message* and **Paste()** would get it from the clipboard.

If a BView displays any of the data it takes from the message, it should generally make itself the focus view:

```
bool MyView::MessageDropped(BMessage *message, BPoint point)
{
    MakeFocus(TRUE);
    . . .
    return TRUE;
}
```

The messages that a user drags and drops on a view might have their source in any application. The Browser will probably be a common source, since it permits users to drag representations of database records. The message in which the Browser packages the dragged information is identical to one that reports a refs-received event. It has a single entry named "refs" containing one or more **record_ref** (**REF_TYPE**) items and **REFS_RECEIVED** as the command constant.

You can choose whether your version of **MessageDropped()** should handle these messages or not. If it does, it might simply pass them to the **RefsReceived()** function you implemented in a class derived from BApplication.

See also: "Message-Dropped Events" on page 49, **FilterMessageDropped()** in the BWindow class, **RefsReceived()** in the BApplication class of the Application Kit, **MouseMoved()**, the BMessage class

Modifiers()

ulong **Modifiers(**void**)** const

Returns a mask that has a bit set for each keyboard lock that's on and for each modifier state that's set because the user is holding down a modifier key. The mask can be tested against these constants:

SHIFT_KEY	**COMMAND_KEY**	**CAPS_LOCK**
CONTROL_KEY	**MENU_KEY**	**SCROLL_LOCK**
OPTION_KEY		**NUM_LOCK**

No bits are set (the mask is 0) if no locks are on and none of the modifiers keys are down.

If it's important to know which physical key the user is holding down, the one on the right or the one on the left, the mask can be further tested against these constants:

LEFT_SHIFT_KEY	**RIGHT_SHIFT_KEY**
LEFT_CONTROL_KEY	**RIGHT_CONTROL_KEY**
LEFT_OPTION_KEY	**RIGHT_OPTION_KEY**
LEFT_COMMAND_KEY	**RIGHT_COMMAND_KEY**

By default, on a 101-key keyboard, the keys labeled "Alt(ernate)" function as the Command modifiers, the key on the right labeled "Control" functions as the right Option key, and only the left "Control" key is available to function as a Control modifier. However, users can change this configuration with the Keyboard utility.

See also: "Modifier Keys" on page 57 of the introduction to the chapter, **GetKeys()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Implemented by derived classes to respond to a mouse-down event within the view. The location of the cursor at the time of the event is given by *point* in the BView's coordinates.

MouseDown() functions are often implemented to track the cursor while the user holds the mouse button down and then respond when the button goes up. You can call the

GetMouse() function to learn the current location of the cursor and the state of the mouse buttons. For example:

```
void MyView::MouseDown(BPoint point)
{
    ulong buttons;
    . . .
    do {
        snooze(30);
        GetMouse(&point, &buttons, TRUE);
        . . .
    } while ( buttons );
    . . .
}
```

To get complete information about the mouse-down event, look inside the BMessage object returned by BLooper's **CurrentMessage()** function.

The BView version of **MouseDown()** is empty.

See also: "Mouse-Down Events" on page 47, **FilterMouseDown()** in the BWindow class, **GetMouse()**

MouseMoved()

virtual void **MouseMoved(**BPoint *point*, ulong *transit*, BMessage **message***)**

Implemented by derived classes to respond to mouse-moved events associated with the view. As the user moves the cursor over a window, the Application Server generates a continuous stream of messages reporting where the cursor is located.

The first argument, *point*, gives the cursor's new location in the BView's coordinate system. The second argument, *transit*, is one of three constants,

ENTERED_VIEW,
INSIDE_VIEW, or
EXITED_VIEW

which explains whether the cursor has just entered the visible region of the view, is now inside the visible region having previously entered, or has just exited from the view. When the cursor crosses a boundary separating the visible regions of two views (perhaps moving from a parent to a child view, or from a child to a parent), **MouseMoved()** is called for each of the BViews, once with a *transit* code of **EXITED_VIEW** and once with a code of **ENTERED_VIEW**.

If the user is dragging a bundle of information from one location to another, the final argument, *message*, is a pointer to the BMessage object that holds the information. If a message isn't being dragged, *message* is **NULL**.

A **MouseMoved()** function might be implemented to ignore the **INSIDE_VIEW** case and respond only when the cursor enters or exits the view. For example, a BView might alter its display to indicate whether or not it can accept a message that has been dragged to it. Or it might be implemented to change the cursor image when it's over the view.

MouseMoved() notifications should not be used to track the cursor inside a view. Use the **GetMouse()** function instead. **GetMouse()** provides the current cursor location plus information on whether any of the mouse buttons are being held down.

The default version of **MouseMoved()** is empty.

See also: "Mouse-Moved Events" on page 48, **FilterMouseMoved()** in the BView class, **DragMessage()**

MoveBy(), MoveTo()

> void **MoveBy(**float *horizontal*, float *vertical***)**
>
> void **MoveTo(**BPoint *point***)**
> void **MoveTo(**float *x*, float *y***)**

These functions move the view in its parent's coordinate system without altering its size.

MoveBy() adds *horizontal* coordinate units to the left and right components of the frame rectangle and *vertical* units to the top and bottom components. If *horizontal* and *vertical* are positive, the view moves downward and to the right. If they're negative, it moves upward and to the left.

MoveTo() moves the upper left corner of the view to *point*—or to (*x*, *y*)—in the parent view's coordinate system and adjusts all coordinates in the frame rectangle accordingly.

Neither function alters the BView's bounds rectangle or coordinate system.

None of the values passed to these functions should specify fractional coordinates; the sides of a view must line up on screen pixels. Fractional values will be rounded to the closest whole number.

If the BView is attached to a window, these functions cause its parent view to be updated, so the BView is immediately displayed in its new location. If it doesn't have a parent or isn't attached to a window, these functions merely alter its frame rectangle.

See also: **FrameMoved()**, **ResizeBy()**

MovePenBy(), MovePenTo(), PenLocation()

void **MovePenBy(**float *horizontal*, float *vertical***)**

void **MovePenTo(**BPoint *point***)**
void **MovePenTo(**float *x*, float *y***)**

BPoint **PenLocation(**void**)**

These functions move the pen (without drawing a line) and report the current pen location.

MovePenBy() moves the pen *horizontal* coordinate units to the right and *vertical* units downward. If *horizontal* or *vertical* are negative, the pen moves in the opposite direction.

MovePenTo() moves the pen to *point*—or to (*x*, *y*)—in the BView's coordinate system. If the pen is the size of just one pixel on the display device, the point positions that pixel. If the pen is the size of a square with more than one pixel on a side, it positions the pixel at the left top corner of the square. The pen extends to the right and below the point specified.

PenLocation() returns the point where the pen is currently positioned in the BView's coordinate system. The default pen position is at (0.0, 0.0).

Some drawing functions also move the pen—to the end of whatever they draw. In particular, this is true of **StrokeLine()**, **DrawString()**, and **DrawChar()**. Functions that stroke a closed shape (such as **StrokeEllipse()**) don't move the pen.

Like other functions that set graphics parameters, **MovePenBy()**, **MovePenTo()**, and **PenLocation()** work only for BViews that are attached to a window.

See also: SetPenSize()

MoveTo() see MoveBy()

Name() see SetName()

Parent()

BView ***Parent(**void**)** const

Returns the BView's parent, or **NULL** if the BView doesn't have one.

See also: AddChild()

PenLocation() see MovePenBy()

PenSize() see **SetPenSize()**

Pulse()

virtual void **Pulse(**void**)**

Implemented by derived classes to do something at regular intervals. Pulses are regularly timed events, like the tick of a clock or the beat of a steady pulse. A BView receives **Pulse()** notifications when no other messages are pending, but only if it asks for them with the **PULSE_NEEDED** flag.

The interval between **Pulse()** calls can be set with BWindow's **SetPulseRate()** function. The default interval is around 500 milliseconds. The pulse rate is the same for all views within a window, but can vary between windows.

Derived classes can implement a **Pulse()** function to do something that must be repeated continuously. However, for time-critical actions, you should implement your own timing mechanism.

The BView version of this function is empty.

See also: **SetFlags()**, the BView constructor, **SetPulseRate()** in the BWindow class

RemoveChild()

virtual bool **RemoveChild(**BView **childView***)**

Severs the link between the BView and *childView*, so that *childView* is no longer a child of the BView. The *childView* retains all its own children and descendants, but they become an isolated fragment of a view hierarchy, unattached to a window.

If it succeeds in removing *childView*, this function returns **TRUE**. If it fails, it returns **FALSE**. It will fail if *childView* is not, in fact, a child of the BView.

See also: **AddChild()**, **RemoveSelf()**

RemoveSelf()

bool **RemoveSelf(**void**)**

Removes the BView from its parent and returns **TRUE**, or returns **FALSE** if the BView doesn't have a parent or for some reason can't be removed from the view hierarchy.

This function acts just like **RemoveChild()**, except that it removes the BView itself rather than one of its children.

See also: **AddChild()**, **RemoveChild()**

ResizeBy(), ResizeTo()

void **ResizeBy(**float *horizontal*, float *vertical***)**

void **ResizeTo(**float *width*, float *height***)**

These functions resize the view, without moving its left and top sides. **ResizeBy()** adds *horizontal* coordinate units to the width of the view and *vertical* units to the height. **ResizeTo()** makes the view *width* units wide and *height* units high. Both functions adjust the right and bottom components of the frame rectangle accordingly.

Since a BView's frame rectangle must be aligned on screen pixels, only integral values should be passed to these functions. Values with fractional components will be rounded to the nearest whole integer.

If the BView is attached to a window, these functions cause its parent view to be updated, so the BView is immediately displayed in its new size. If it doesn't have a parent or isn't attached to a window, these functions merely alter its frame and bounds rectangles.

See also: **FrameResized()**, **MoveBy()**, **Width()** and **Height()** in the BRect class

ResizingMode() see SetResizingMode()

ScrollBy(), ScrollTo()

void **ScrollBy(**float *horizontal*, float *vertical***)**

void **ScrollTo(**BPoint *point***)**
void **ScrollTo(**float *x*, float *y***)**

These functions scroll the contents of the view.

ScrollBy() adds *horizontal* to the left and right components of the BView's bounds rectangle, and *vertical* to the top and bottom components. This serves to shift the display *horizontal* coordinate units to the left and *vertical* units upward. If *horizontal* and *vertical* are negative, the display shifts in the opposite direction.

ScrollTo() shifts the contents of the view as much as necessary to put *point*—or (*x*, *y*)—at the upper left corner of its bounds rectangle. The point is specified in the BView's coordinate system.

Anything in the view that was visible before scrolling and also visible afterwards is automatically redisplayed at its new location. The remainder of the view is invalidated, so the BView's **Draw()** function will be called to fill in those parts of the display that were previously invisible. The update rectangle passed to **Draw()** will be the smallest rectangle that encloses just these new areas. If the view is scrolled in only one direction, the update rectangle will be exactly the area that needs to be drawn.

These function don't work on BViews that aren't attached to a window.

See also: GetClippingRegion()

SetBackColor() see SetFrontColor()

SetClippingRegion()

virtual void **SetClippingRegion(**BRegion **region***)**

Restricts the drawing that the BView can do to *region*.

The Application Server keeps track of a clipping region for each BView that's attached to a window. It clips all drawing the BView does to that region; the BView can't draw outside of it.

By default, the clipping region contains only the visible area of the view and, during an update, only the area that actually needs to be drawn. By passing a *region* to this function, an application can further restrict the clipping region. When calculating the clipping region, the Server intersects it with the *region* provided. The BView can draw only in areas common to the *region* passed and the clipping region as the Server would otherwise calculate it. The region passed can't expand the clipping region beyond what it otherwise would be.

If called during an update, **SetClippingRegion()** restricts the clipping region only for the duration of the update.

Calls to **SetClippingRegion()** are not additive; each *region* that's passed replaces the *region* that was passed in the previous call.

See also: GetClippingRegion(), Draw()

SetDrawingMode(), DrawingMode()

virtual void **SetDrawingMode(**drawing_mode *mode***)**

drawing_mode **DrawingMode(**void**)** const

These functions set and return the BView's drawing mode. They work only for BViews that are attached to a window.

The mode can be set to any of the following nine constants:

OP_COPY	**OP_MIN**	**OP_ADD**
OP_OVER	**OP_MAX**	**OP_SUBTRACT**
OP_ERASE	**OP_INVERT**	**OP_BLEND**

The default drawing mode is **OP_COPY**. It and the other modes are explained under "Drawing Modes" on page 27 of the introduction to this chapter.

See also: "Drawing Modes" in the chapter introduction

SetFlags(), Flags()

virtual void **SetFlags(**ulong *mask***)**

inline ulong **Flags(**void**)** const

These functions set and return the flags that inform the Application Server about the kinds of notifications the BView should receive. The *mask* set by **SetFlags()** and the return value of **Flags()** is formed from combinations of the following constants:

WILL_DRAW,
FULL_UPDATE_ON_RESIZE,
FRAME_EVENTS, and
PULSE_NEEDED

The flags are first set when the BView is constructed; they're explained in the description of the BView constructor.

To set just one of the flags, combine it with the current setting:

```
myView->SetFlags(Flags() | FRAME_EVENTS);
```

The *mask* passed to **SetFlags()** and the value returned by **Flags()** can be 0.

See also: the BView constructor, **SetResizingMode()**

SetFontName(), SetFontSize(), SetFontRotation(), SetFontShear(), SetFontSymbolSet()

virtual void **SetFontName(**const char **name***)**

virtual void **SetFontSize(**float *points***)**

virtual void **SetFontRotation(**float *degrees***)**

virtual void **SetFontShear(**float *angle***)**

virtual void **SetFontSymbolSet(**const char **name***)**

These functions set characteristics of the font in which the BView draws text. The font is part of the BView's graphics state. It's used by **DrawString()** and **DrawChar()** and assumed by **StringWidth()**, **GetFontInfo()**, and **GetCharEdges()**.

SetFontName() sets the precise name of the font, including the designation of whether it's bold, italic, oblique, black, narrow, or some other style. The name passed to this function must be the same as the name assigned to the font by the vendor. For example, this code

```
    SetFontName("Futura II Italic ATT");
```

sets the BView's font to the TrueType™ italic Futura II font.

For **SetFontName()** to be successful, the name it's passed must select a font that's installed on the user's machine. The global **get_font_name()** function can provide the names of all fonts that are currently installed. (Users can see the names listed in the Keyboard application's "Font" menu.)

A handful of fonts are provided with the release, including Times CG ATT, Futura II ATT, Baskerville MT, and their stylistic variations. < Additional fonts can be installed by placing them in the proper subdirectory of **/system/fonts** and rebooting the machine. >

The names of the bitmap fonts that come with the system are:

chicago
geneva
monaco

They're available only in one size—9.0 points. The default font is "monaco". If you ask for a font that isn't available, you'll get monaco instead.

< Currently, you must specifically ask for a bitmap font. In the future, bitmap equivalents to the outline fonts will be automatically provided for on-screen display. >

SetFontSize() sets the size of the font. Valid sizes range from 4 points through 999 points. < Currently, fractional font sizes are not supported. >

SetFontRotation() sets the rotation of the baseline. The baseline rotates counterclockwise from an axis on the left side of the character. The default (horizontal) baseline is at 0°. For example, this code

```
    SetFontRotation(45.0);
    DrawString("to the northeast");
```

would draw a string that extended upwards and to the right. < Currently, fractional angles of rotation are not supported. >

SetFontShear() sets the angle at which characters are drawn relative to the baseline. The default (perpendicular) shear for all font styles, including oblique and italic ones, is 90.0°. The shear is measured counterclockwise and can be adjusted within the range 45.0° (slanted to the right) through 135.0° (slanted to the left). < Currently, fractional shear angles are not supported. >

SetFontSymbolSet() determines the set of characters that can be displayed. A character set maps graphic symbols (glyphs) to character values (ASCII codes). Sets differ mainly in which symbols they associate with character values beyond the traditional ASCII range (above 0x7f).

The default symbol set is “Macintosh”; there are many other possibilities, including: “ISO 8859/9 Latin 5”, “Legal”, “PC-850 Multilingual”, and “Windows 3.1 Latin 2”. The **get_symbol_set_name()** global function can provide a list of all currently available symbol sets.

Except for the bitmap fonts, every font implements every symbol set. However, some fonts may not provide all the characters in every set.

These five font functions work only for BViews that are attached to a window. < The **SetFontSize()**, **SetFontRotation()**, and **SetFontShear()** functions don’t work for bitmap fonts. >

Derived classes can override these functions to take any collateral measures required by the font change. For example, BTextView and BListView override them to redisplay the text in the new font.

See also: GetFontInfo(), AttachedToWindow(), get_font_name(), get_symbol_set_name()

SetFrontColor(), FrontColor(), SetBackColor(), BackColor()

virtual void **SetFrontColor(**rgb_color *aColor***)**
void **SetFrontColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**

rgb_color **FrontColor(**void**)**

virtual void **SetBackColor(**rgb_color *aColor***)**
void **SetBackColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**

rgb_color **BackColor(**void**)**

These functions set and return the current front and background colors of the BView. They work only for BViews that are attached to a window.

The front and background colors combine to form a pattern that’s passed as an argument to most **Stroke...()** and **Fill...()** drawing functions. The **solid_front** pattern is the front color alone, and **solid_back** is the background color alone.

The default front color is black—*red*, *green*, and *blue* values all equal to 0. The default background color is white—*red*, *green*, and *blue* values all equal to 255. < The *alpha* component of the color is currently ignored. >

The versions of **SetFrontColor()** and **SetBackColor()** that take separate arguments for the *red*, *blue*, and *green* color components work by creating an **rgb_color** data structure and passing it to the corresponding function that’s declared **virtual**. Therefore, if you want to augment either function in some way, you need only override the **rgb_color** version.

See also: “Patterns” on page 25 of the chapter introduction, **SetViewColor()**

SetName(), Name()

void **SetName(**const char **string***)**

const char ***Name(**void**)** const

These functions set and return the name that identifies the BView. The name is originally set by the BView constructor. **SetName()** assigns the BView a new name, and **Name()** returns the current name. The string returned by **Name()** belongs to the BView object; it shouldn't be altered or freed.

See also: the BView constructor, **FindView()**

SetPenSize(), PenSize()

virtual void **SetPenSize(**float *size***)**

float **PenSize(**void**)**

SetPenSize() sets the size of the BView's pen—the graphics parameter that determines the thickness of stroked lines—and **PenSize()** returns the current pen size.

The pen size is translated from coordinate units to a device-specific number of pixels. The pen is a square with that number of pixels on a side. When it strokes a line, the left top pixel in the square follows the path of the line from pixel to pixel. Therefore, if the pen square has more than one pixel on a side, it extends to the right and hangs below the path being stroked. As it moves along the path, the pen paints the pixels that it touches.

The default pen size is 1.0 coordinate unit. It can be set to any non-negative value, including 0.0. If set to 0.0, the size is translated to 1 pixel for all output devices. This guarantees that it will always draw the thinnest possible line no matter what the device.

Thus, lines drawn with pen sizes of 1.0 and 0.0 will look alike on the screen (one pixel thick), but the line drawn with a pen size of 1.0 will be 1/72 of an inch thick when printed, however many printer pixels that takes, while the line drawn with a 0.0 pen size will be just one pixel thick.

These functions can set and return the pen size only if the BView is attached to a window.

See also: "The Pen" on page 24 and "Picking Pixels to Stroke and Fill" on page 34 of the chapter introduction, **StrokeArc()** and the other **Stroke...()** functions, **MovePenBy()**

SetResizingMode(), ResizingMode()

virtual void **SetResizingMode(**ulong *mode***)**

inline ulong **ResizingMode(**void**)** const

These functions set and return the BView's automatic resizing mode. The resizing mode is first set when the BView is constructed. The various possible modes are explained where the constructor is described.

See also: the BView constructor, **SetFlags()**

SetViewColor()

void **SetViewColor(**rgb_color *color***)**

Sets the color that's shown in all areas of the view rectangle that the BView doesn't cover with its own drawing. When the clipping region is erased prior to an update, it's erased to the view color. When a view is resized to expose new areas that it doesn't draw in, the new areas are displayed in the view color.

The view color can be set only after the view is attached to a window. It's best to set it before the window is shown on-screen. The default color is white.

See also: "The View Color" on page 22 of the introduction to the chapter, **SetFrontColor()**

Show() see Hide()

StringWidth()

float **StringWidth(**const char **string***)** const
float **StringWidth(**const char **string*, long *length***)** const

Returns how much room is required to draw *length* characters of *string* in the BView's current font. If no length is specified, the entire string is measured, up to the null character, '\0', which terminates it. The return value totals the width of all the characters. It measures, in coordinate units, the length of the baseline required to draw the string.

This function works only for BViews that are attached to a window (since only attached views have a current font).

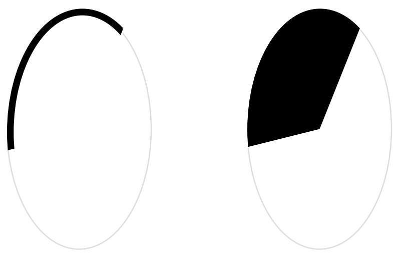
See also: **GetFontInfo()**, **GetCharEscapements()**

StrokeArc(), FillArc()

void **StrokeArc(**BRect *rect*, float *angle*, float *span*,
const pattern **aPattern* = &**solid_front)**
void **StrokeArc(**BPoint *center*, float *xRadius*, float *yRadius*,
float *angle*, float *span*,
const pattern **aPattern* = &**solid_front)**

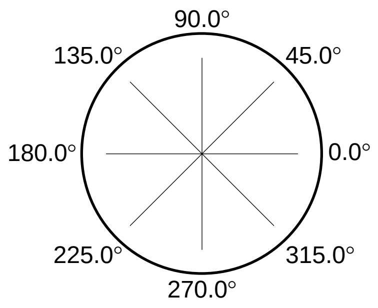
void **FillArc(**BRect *rect*, float *angle*, float *span*,
const pattern **aPattern* = &**solid_front)**
void **FillArc(**BPoint *center*, float *xRadius*, float *yRadius*,
float *angle*, float *span*,
const pattern **aPattern* = &**solid_front)**

These functions draw an arc, a portion of an ellipse. **StrokeArc()** strokes a line along the path of the arc. **FillArc()** fills the wedge defined by straight lines stretching from the center of the ellipse of which the arc is a part to the end points of the arc itself. For example:



The arc is a section of the ellipse inscribed in *rect*—or the ellipse located at *center*, where the horizontal distance from the center to the edge of the ellipse is measured by *xRadius* and the vertical distance from the center to the edge is measured by *yRadius*.

The arc starts at *angle* and stretches along the ellipse for *span* degrees, where angular coordinates are measured counterclockwise with 0° on the right, as shown below:



For example, if *angle* is 180.0° and *span* is 90.0°, the arc would be the lower left quarter of the ellipse. The same arc would be drawn if *angle* were 270.0° and *span* were –90.0°. < Currently, *angle* and *span* measurements in fractions of a degree are not supported. >

The width of the line drawn by **StrokeArc()** is determined by the current pen size. Both functions draw using *aPattern*—or, if no pattern is specified, using the current front color. Neither function alters the current pen position.

See also: StrokeEllipse()

StrokeEllipse(), FillEllipse()

void **StrokeEllipse(**BRect *rect*, const pattern **aPattern* = &**solid_front)**
void **StrokeEllipse(**BPoint *center*, float *xRadius*, float *yRadius*,
                    const pattern **aPattern* = &**solid_front)**

void **FillEllipse(**BRect *rect*, const pattern **aPattern* = &**solid_front)**
void **FillEllipse(**BPoint *center*, float *xRadius*, float *yRadius*,
                    const pattern **aPattern* = &**solid_front)**

These functions draw an ellipse. **StrokeEllipse()** strokes a line around the perimeter of the ellipse and **FillEllipse()** fills the area the ellipse encloses.

The ellipse has its center at *center.* The horizontal distance from the center to the edge of the ellipse is measured by *xRadius* and the vertical distance from the center to the edge is measured by *yRadius*. If *xRadius* and *yRadius* are the same, the ellipse will be a circle.

Alternatively, the ellipse can be described as one that's inscribed in *rect.* If the rectangle is a square, the ellipse will be a circle.

The width of the line drawn by **StrokeEllipse()** is determined by the current pen size. Both functions draw using *aPattern*—or, if no pattern is specified, using the current front color. Neither function alters the current pen position.

See also: SetPenSize()

StrokeLine()

void **StrokeLine(**BPoint *start*, BPoint *end*, const pattern **aPattern* = &**solid_front)**
void **StrokeLine(**BPoint *end*, const pattern **aPattern* = &**solid_front)**

Draws a straight line between the *start* and *end* points—or, if no starting point is given, between the current pen position and *end* point—and leaves the pen at the end point.

This function draws the line using the current pen size and the specified pattern. If no pattern is specified, the line is drawn in the current front color.

See also: SetPenSize(), BeginLineArray()

StrokePolygon(), FillPolygon()

void **StrokePolygon(**BPolygon **polygon*,
const pattern **aPattern* = &**solid_front)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*,
const pattern **aPattern* = &**solid_front)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*, BRect *rect*,
const pattern **aPattern* = &**solid_front)**

void **FillPolygon(**BPolygon **aPolygon*,
const pattern **aPattern* = &**solid_front)**
void **FillPolygon(**BPoint **pointList*, long *numPoints*,
const pattern **aPattern* = &**solid_front)**
void **FillPolygon(**BPoint **pointList*, long *numPoints*, BRect *rect*,
const pattern **aPattern* = &**solid_front)**

These functions draw a polygon with an arbitrary number of sides. **StrokePolygon()** strokes a line around the edge of the polygon using the current pen size. If a *pointList* is specified rather than a BPolygon object, this function strokes a line from point to point, connecting the first and last points if they aren't identical. **FillPolygon()** fills in the entire area enclosed by the polygon.

Both functions must calculate the frame rectangle of a polygon constructed from a point list—that is, the smallest rectangle that contains all the points in the polygon. If you know what this rectangle is, you can make the function somewhat more efficient by passing it as the *rect* parameter.

Both functions draw using the specified pattern—or, if no pattern is specified, in the current front color. Neither function alters the current pen position.

< Currently, **StrokePolygon()** doesn't accept pen sizes other than 1 or patterns other than the default. >

See also: SetPenSize(), the BPolygon class

StrokeRect(), FillRect()

void **StrokeRect(**BRect *rect*, const pattern **aPattern* = &**solid_front)**

void **FillRect(**BRect *rect*, const pattern **aPattern* = &**solid_front)**

These functions draw a rectangle. **StrokeRect()** strokes a line around the edge of the rectangle; the width of the line is determined by the current pen size. **FillRect()** fills in the entire rectangle.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current front color. Neither function alters the current pen position.

See also: SetPenSize(), **StrokeRoundRect()**

StrokeRoundRect(), FillRoundRect()

void **StrokeRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*,
const pattern **aPattern* = &**solid_front)**

void **FillRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*,
const pattern **aPattern* = &**solid_front)**

These functions draw a rectangle with rounded corners. The corner arc is one-quarter of an ellipse, where the ellipse would have a horizontal radius equal to *xRadius* and a vertical radius equal to *yRadius*.

Except for the rounded corners of the rectangle, these functions work exactly like **StrokeRect()** and **FillRect()**.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current front color. Neither function alters the current pen position.

See also: StrokeRect(), **StrokeEllipse()**

StrokeTriangle(), FillTriangle()

void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
const pattern **aPattern* = &**solid_front)**
void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
BRect *rect*,
const pattern **aPattern* = &**solid_front)**

void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
const pattern **aPattern* = &**solid_front)**
void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
BRect *rect*,
const pattern **aPattern* = &**solid_front)**

These functions draw a triangle, a three-sided polygon. **StrokeTriangle()** strokes a line the width of the current pen size from the first point to the second, from the second point to the third, then back to the first point. **FillTriangle()** fills in the area that the three points enclose.

Each function must calculate the smallest rectangle that contains the triangle. If you know what this rectangle is, you can make the function marginally more efficient by passing it as the *rect* parameter.

Both functions do their drawing using the pattern specified by *aPattern*—or, if no pattern is specified, in the current front color. Neither function alters the current pen position.

< Currently, **StrokeTriangle()** doesn't accept pen sizes other than 1 or patterns other than the default. >

See also: SetPenSize()

Sync() see **Flush()**

Window(), Looper()

BWindow ***Window(**void**)** const

virtual BLooper ***Looper(**void**)** const

Both these functions return the BWindow to which the BView belongs, or **NULL** if the BView hasn't yet been attached to a window. **Looper()** overrides the virtual function first declared in the BReceiver class to return the BWindow as a pointer to a BLooper object. **Window()** returns it more directly as a pointer to a BWindow.

See also: **Looper()** in the BReceiver class of the Application Kit, **AddChild()** in both this and the BWindow class, **AttachedToWindow()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Implemented by derived classes to take whatever steps are necessary when the BView's window becomes the active window, or when the window gives up that status. If *active* is **TRUE**, the window has become active. If *active* is **FALSE**, it no longer is the active window.

All objects in the view hierarchy receive **WindowActivated()** notifications when the status of the window changes.

BView's version of this function is empty.

See also: **WindowActivated()** in the BWindow class

BWindow

Derived from:	public BLooper
Declared in:	<interface/Window.h>

Overview

The BWindow class defines an application interface to windows. Each BWindow object corresponds to one window in the user interface.

At the most basic level, it's the Application Server's responsibility to provide an application with the windows it needs. The Server allocates the memory each window requires, renders images in the window on instructions from the application, and manages the user interface. It equips windows with all the accouterments that let users activate, move, resize, reorder, hide, and close them. These user actions are not mediated by the application; they're handled within the Application Server alone. However, the Server sends the application messages notifying it of user actions that affect the window. A class derived from BWindow can implement virtual functions such as **FrameResized()**, **QuitRequested()**, and **WindowActivated()** to respond to these messages.

BWindow objects are the application's interface to the Server's windows:

- Creating a BWindow object instructs the Application Server to produce a window that can be displayed to the user. The BWindow constructor determines what kind of window it will be and how it will behave. The window is initially hidden; the **Show()** function makes it visible on-screen.

- BWindow functions give the application the ability to manipulate the window programmatically—to activate, move, resize, reorder, hide, and close it just as a user might.

- Classes derived from BWindow can implement functions that respond to events affecting the window.

BWindow objects communicate directly with the Server. However, before this communication can take place, the constructor for the BApplication object must establish an initial connection to the Server. You must construct the BApplication object before the first BWindow.

View Hierarchy

A window can display images, but it can't produce them. To draw within a window, an application needs a collection of various BView objects. For example, a window might have several check boxes or radio buttons, a list of names, some scroll bars, and a scrollable display of pictures or text—all provided by objects that inherit from the BView class.

These BViews are created by the application and are associated with the BWindow by arranging them in a hierarchy under a *top view*, a view that fills the entire content area of the window. Views are added to the hierarchy by making them children of views already in the hierarchy, which at the outset means children of the top view.

A BWindow doesn't reveal the identity of its top view, but it does have functions that act on the top view's behalf. For example, BWindow's **AddChild()** function adds a view to the hierarchy as a child of the top view. Its **FindView()** function searches the view hierarchy beginning with the top view.

Window Threads

Each window runs in its own thread—both in the Application Server and in the application. When it's constructed, a BWindow object spawns a *window thread* for the application and begins running a message loop where it receives reports of user actions associated with the window. You don't have to call **Run()** to get the message loop going, as you do for other BLoopers; **Run()** is called for you at construction time.

Actions initiated from a BWindow's message loop are executed in the window's thread. This, of course, includes all actions that are spun off from the original event notification. For example, if the user clicks a button in a window and this initiates a series of calculations involving a variety of objects, those calculations will be executed in the thread of the window where the button is located (unless the calculation explicitly spawns other threads or posts messages to other BLoopers).

Quitting

To "close" a window is to remove the window from the screen, quit the message loop, kill the window thread, and delete the BWindow object. As is the case for other BLoopers, this process is initiated by a request to quit—a **QUIT_REQUESTED** message.

For a BWindow, a request to quit is an event that might be reported from the Application Server (as when the user clicks a window's close box) or from within the application (as when the user clicks a "Close" menu item).

To respond to quit-requested events, classes derived from BWindow implement **QuitRequested()** functions. **QuitRequested()** can prevent the window from closing, or take whatever action is appropriate before the window is destroyed. It typically interacts with the user, asking, for example, whether recent changes to a document should be saved.

QuitRequested() is a hook function declared in the BLooper class; it's not documented here. See the BLooper class in the Application Kit for information on the function and on how classes derived from BWindow might implement it.

Hook Functions

FilterKeyDown()	Can be implemented to filter reports of key-down events before they're dispatched by calling the focus view's **KeyDown()** function.
FilterMessageDropped()	Can be implemented to filter reports of message-dropped events before they're dispatched by calling a BView's **MessageDropped()** function.
FilterMouseDown()	Can be implemented to filter reports of mouse-down events before they're dispatched by calling a BView's **MouseDown()** function.
FilterMouseMoved()	Can be implemented to filter reports of mouse-moved events before they're dispatched by calling a BView's **MouseMoved()** function.
FixMenus()	Can be implemented to make sure menu data structures are up to date before the menu is displayed to the user.
FrameMoved()	Can be implemented to take note of the fact that the window has moved.
FrameResized()	Can be implemented to take note of the fact that the window has been resized.
SavePanelClosed()	Can be implemented to take note when the window's save panel closes.
SaveRequested()	Can be implemented to save the document displayed in the window when the user requests it in the save panel.
WindowActivated()	Can be implemented to take whatever action is necessary when the window becomes the active window, or when it loses that status.

Constructor and Destructor

BWindow()

BWindow(BRect *frame*, const char **title*, window_type *type*, ulong *flags***)**

Produces a new window with the *frame* content area, spawns a new thread of execution for the window, and begins running a message loop in that thread.

The first argument, *frame*, measures only the content area of the window; it excludes the border and the title tab at the top. The window's top view will be exactly the same size and shape as its frame rectangle—though the top view is located in the window's coordinate system and the window's frame rectangle is specified in the screen coordinate system.

For the window to become visible on-screen, the frame rectangle you assign it must lie within the frame rectangle of the screen. You can find the current dimensions of the screen by calling **get_screen_info()**. In addition, both the width and height of *frame* must be greater than 0.

Since a window is always aligned on screen pixels, the sides of its frame rectangle must have integral coordinate values. Any fractional coordinates that are passed in *frame* will be rounded to the nearest whole number.

The second argument, *title*, sets the title the window will display if it has a tab and also determines the name of the window thread. The thread name is a string that prefixes "w>" to the title in the following format:

```
"w>title"
```

If the *title* is long, only as many characters will be used as will fit within the limited length of a thread name. (Only the thread name is limited, not the window title.) The title (and thread name) can be changed with the **SetTitle()** function.

The *title* must be set, even if the window doesn't have a tab to display it; it can't be **NULL**, but it can be an empty string.

The *type* of window is set by one of the following constants:

MODAL_WINDOW	A modal window, one that disables other activity in the application until the user dismisses it. It has a border but no tab to display a title.
BORDERED_WINDOW	An ordinary (nonmodal) window with a border but no tab.
TITLED_WINDOW	A window with a border and a tab. Most windows are of this type. The title is displayed in the tab.
SHADOWED_WINDOW	A window with a border and tab, and a drop shadow on its right and bottom sides.

The tab, border, and drop shadow are drawn around the window's frame rectangle.

The final argument, *flags*, is a mask that determines the behavior of the window. It's formed by combining constants from the following set:

NOT_MOVABLE	Prevents the user from being able to move the window. By default, a window with a tab at the top is movable.
NOT_H_RESIZABLE	Prevents the user from resizing the window horizontally. A window is horizontally resizable by default.
NOT_V_RESIZABLE	Prevents the user from resizing the window vertically. A window is vertically resizable by default.
NOT_RESIZABLE	Prevents the user from resizing the window in any direction. This constant is a shorthand that you can substitute for the combination of **NOT_H_RESIZABLE** and **NOT_V_RESIZABLE**. A window is resizable by default.
NOT_CLOSABLE	Prevents the user from closing the window (eliminates the close box from its tab). Windows with title tabs have a close box by default.
NOT_ZOOMABLE	Prevents the user from expanding the window to the full size of the screen.
ACCEPTS_FIRST_CLICK	Enables the BWindow to receive mouse-down and mouse-up messages even when it isn't the active window. By default, a click in a window that isn't the active window brings the window to the front and makes it active, but doesn't get reported to the application. If a BWindow accepts the first click, the event gets reported to the application, but it doesn't make the window active. The BView that responds to the mouse-down message must take responsibility for activating the window.
FLOATS	Causes the window to float in front of other windows.

If *flags* is 0, the window will be one the user can move, resize, close, and zoom. It won't float or accept the first click.

The window's message loop reads messages delivered to the window and dispatches them by calling a virtual function of the responsible object. The responsible object is usually one of the BViews in the window's view hierarchy. Views are notified of event messages through **MouseDown()**, **KeyDown()**, **MessageDropped()**, **MouseMoved()** and other virtual function calls. However, sometimes the responsible object is the BWindow itself. It handles **FrameMoved()**, **QuitRequested()**, **WindowActivated()** and other notifications.

The message loop begins to run when the BWindow is constructed and continues until the window is told to quit and the BWindow object is deleted. Everything the window thread does is initiated by a message of some kind.

See also: SetFlags(), **SetTitle()**

~BWindow()

virtual **~BWindow(**void**)**

Frees all memory that the BWindow allocated for itself.

Call the **Quit()** function to destroy the BWindow object; don't use the **delete** operator. **Quit()** does everything that's necessary to shut down the window—such as remove its connection to the Application Server and get rid of its views—and invokes **delete** at the appropriate time.

See also: Quit()

Member Functions

Activate()

void **Activate(**bool *flag* = **TRUE)**

Makes the BWindow the active window (if *flag* is **TRUE**), or causes it to relinquish that status (if *flag* is **FALSE**). When this function activates a window, it reorders the window to the front <of its tier>, highlights its tab, and makes it the window responsible for handling subsequent keyboard events. When it deactivates a window, it undoes all these things. It reorders the window to the back <of its tier> and removes the highlighting from its tab. Another window (the new active window) becomes the target for keyboard events.

When a BWindow is activated or deactivated (whether programmatically through this function or by the user), it and all the BViews in its view hierarchy receive **WindowActivated()** notifications.

This function will not activate a window that's hidden.

See also: WindowActivated() in this and the BView class

AddChild()

virtual void **AddChild(**BView **aView***)**

Adds *aView* to the hierarchy of views associated with the window, making it a child of the window's top view. If *aView* already has a parent, it's forcibly removed from that family and adopted into this one. A view can live with but one parent at a time.

This function calls *aView*'s **AttachedToWindow()** function to inform it that it now belongs to the BWindow. Every view that descends from *aView* also becomes attached to the window and receives its own **AttachedToWindow()** notification.

See also: **AddChild()** and **AttachedToWindow()** in the BView class, **RemoveChild()**

AddShortcut(), RemoveShortcut()

void **AddShortcut(**ulong *aChar*, ulong *modifiers*, BMessage **message***)**
void **AddShortcut(**ulong *aChar*, ulong *modifiers*, BMessage **message*,
BView **target***)**

void **RemoveShortcut(**ulong *aChar*, ulong *modifiers***)**

These functions set up, and tear down, keyboard shortcuts for the window. A shortcut is a character (*aChar*) that the user can type, in combination with the Command key and possibly one or more other *modifiers* to issue an instruction to the application. For example, Command-*r* might rotate what's displayed within a particular view. The instruction is issued by posting a BMessage to the window thread.

Keyboard shortcuts are commonly associated with menu items. However, *do not* use these functions to set up shortcuts for menus; use the BMenuItem constructor instead. These BWindow functions are for shortcuts that aren't associated with a menu. (The version of **AddShortcut()** that takes a BMenuItem argument, declared in the header file but not documented here, is for the internal use of the Interface Kit only.)

AddShortcut() registers a new window-specific keyboard shortcut. The first two arguments, *aChar* and *modifiers*, specify the character and the modifier states that together will issue the instruction. *modifiers* is a mask that combines any of the usual modifier constants (see the **Modifiers()** function for the full list). Typically, it's one or more of these four (or it's 0):

SHIFT_KEY
CONTROL_KEY
OPTION_KEY
COMMAND_KEY

COMMAND_KEY is assumed; it doesn't have to be specified. The character value that's passed as an argument should reflect the modifier keys that are required. For example, if the shortcut is Command-Shift-*C*, *aChar* should be 'C', not 'c'.

The instruction that the shortcut issues is embodied in a model *message* that the BWindow will copy and post whenever it's notified of a key-down event matching the *aChar* and *modifiers* combination (including **COMMAND_KEY**).

Before posting the message, it adds one data entry to the copy:

Data name	Type code	Description
"when"	**LONG_TYPE**	When the key-down event occurred, as measured in milliseconds from the time the machine was last booted.

The model *message* shouldn't contain an entry of the same name.

The message is posted to the BWindow. If a *target* BView is specified, it will be named as the message receiver. If a *target* isn't specified, the current focus view will be named as the receiver. If there is no focus view, the BWindow will act as the receiver.

The message is dispatched by calling the receiver's **MessageReceived()** function. If you add a keyboard shortcut to a window, you must implement a **MessageReceived()** function that can respond to the message the shortcut generates.

(Note, however, that if the *message* has **QUIT_REQUESTED** or the constant for another interface event as its **what** data member, it will be dispatched by calling a specific function, like **QuitRequested()**, not **MessageReceived()**.)

RemoveShortcut() unregisters a keyboard shortcut that was previously added.

See also: MessageReceived(), **FilterKeyDown()**, the BMenuItem constructor

Bounds()

BRect **Bounds(**void**)** const

Returns the current bounds rectangle of the window. The bounds rectangle encloses the content area of the window and is stated in the window's coordinate system. It's exactly the same size as the frame rectangle, but its left and top sides are always 0.0.

See also: Frame()

ChildAt(), CountChildren()

BView ***ChildAt(**long *index***)** const

long **CountChildren(**void**)** const

< These first of these functions returns the child BView at *index*, or **NULL** if there's no such child of the BWindow's top view. Indices begin at 0 and there are no gaps in the list. The second function returns the number of children the top view has. Do not rely on these functions as they may not remain in the API. >

Close() see **Quit()**

CloseSavePanel() see **RunSavePanel()**

ConvertToScreen(), ConvertFromScreen()

> void **ConvertToScreen(**BPoint **windowPoint***)** const
> void **ConvertToScreen(**BRect **windowRect***)** const
>
> void **ConvertFromScreen(**BPoint **screenPoint***)** const
> void **ConvertFromScreen(**BRect **screenRect***)** const

These functions convert points and rectangles to and from the global screen coordinate system. **ConvertToScreen()** converts the point referred to by *windowPoint*, or the rectangle referred to by *windowRect*, from the window coordinate system to the screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts the point referred to by *screenPoint*, or the rectangle referred to by *screenRect*, from the screen coordinate system to the window coordinate system.

The window coordinate system has its origin, (0.0, 0.0), at the left top corner of the window's content area.

See also: **ConvertToScreen()** in the BView class

CurrentFocus(), PreferredReceiver()

> BView ***CurrentFocus(**void**)** const
>
> virtual BReceiver ***PreferredReceiver(**void**)** const

Both these functions return the current focus view for the BWindow, or **NULL** if no view is currently in focus. **CurrentFocus()** returns the object as a BView, and **PreferredReceiver()** overrides the BLooper function to return it as a BReceiver.

The focus view is the BView that's responsible for showing the current selection and handling keyboard events when the window is the active window.

Various other objects in the Interface Kit, such as BButtons and BMenuItems, call **PreferredReceiver()** to discover where they should post messages when they are targeted to post them to the BWindow, but don't have a specific receiver. This mechanism permits these objects to be targeted to the current focus view. Thus, a menu item or a control device can be set up to always act on whatever BView happens to be displaying the current selection.

See also: **MakeFocus()** and **IsFocus()** in the BView class, **SetTarget()** in the BControl, BListView, and BMenuItem classes, **PreferredReceiver()** in the BLooper class

DefaultButton() see **SetDefaultButton()**

DisableUpdates(), EnableUpdates()

void **DisableUpdates(**void**)**

void **EnableUpdates(**void**)**

These function disable automatic updating within the window, and re-enable it again. Updating is enabled by default, so every user action that changes a view and every program action that invalidates a view's contents causes the view to be automatically redrawn.

This may be inefficient when there are a number of changes to a view, or to a group of views within a window. In this case, you can temporarily disable the updating mechanism by calling **DisableUpdates()**, make the changes, then call **EnableUpdates()** to re-enable updating and have all the changes displayed at once.

See also: **Invalidate()** in the BView class, **UpdateIfNeeded()**

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message*, BReceiver **receiver***)**

Overrides the BLooper function to dispatch messages as they're received by the window thread. This function is called for you each time the BWindow takes a message from its queue. It dispatches the message by calling the virtual function that's designated to begin the application's response.

- It dispatches messages that report system-defined events by calling an event-specific virtual function implemented for the BWindow or the responsible BView. See "Hook Functions for Interface Events" on page 42 of the introduction to this chapter for a list of these functions.

- It dispatches other messages by calling the targeted *receiver*'s **MessageReceived()** function.

Derived classes can override **DispatchMessage()** to make it dispatch specific kinds of messages in other ways. For example:

```
void MyWindow::DispatchMessage(BMessage *message)
{
    if ( message->what == MAKE_PREDICTIONS )
        predictor->GuessAbout(message);
    else
        BWindow::DispatchMessage(message);
}
```

The message loop deletes every message it receives when the function that **DispatchMessage()** calls, and **DispatchMessage()** itself, return. The message should not be deleted in application code (unless **DetachCurrentMessage()** is first called to detach it from the message loop).

See also: the BMessage class, **DispatchMessage()** and **CurrentMessage()** in the BLooper class

EnableUpdates() see **DisableUpdates()**

FilterKeyDown()

virtual bool **FilterKeyDown(**ulong **aChar*, BView ***target***)**

Implemented by derived classes to interpret a key-down event before the window's focus view is notified with a **KeyDown()** function call. The first argument, *aChar*, points to the character reported in the event. The second argument, *target*, points to the focus BView that's slated to receive the **KeyDown()** notification.

FilterKeyDown() is called for every key-down event that's reported to the window, except for those that might correspond to keyboard shortcuts. If it returns **TRUE**, the **KeyDown()** virtual function implemented for the target view will be called. If it returns **FALSE**, **KeyDown()** isn't called and the key-down event isn't handled (except to the extent that **FilterKeyDown()** itself might be implemented to handle it).

Before returning **TRUE**, this function can change the *aChar* value that will be passed to **KeyDown()**. (This, however, won't change the "char" entry of the BMessage object that reported the event and that **CurrentMessage()** returns). It can also change the *target* BView to another view located within the same window. For example:

```
bool MyView::FilerKeyDown(ulong *aChar, BView **target)
{
    . . .
    if ( *target->IsVeryMuchDisabled() )
        *target = *target->Parent();
    . . .
    if ( *aChar == ENTER )
        *aChar = TAB;
    . . .
}
```

Neither **FilterKeyDown()** nor **KeyDown()** is called for key-down events that are potential keyboard shortcuts—that is, for any key-down event that's produced while holding down a Command key.

The BWindow version of **FilterKeyDown()** makes no changes to either the character or the target BView and simply returns **TRUE**.

See also: **KeyDown()** in the BView class, **AddShortcut()**, **Modifiers()**, "Key-Down Events" on page 46 of the introduction

FilterMessageDropped()

virtual bool **FilterMessageDropped(**BMessage **message*, BPoint *point*,
BView ***target***)**

Implemented by derived classes to preview a message-dropped event before **MessageDropped()** is called for any of the window's BViews. The first argument, *message*, is the dropped message (not the message that reports the message-dropped event, but the message that the user dragged and dropped). The second argument, *point*, is the location of the cursor when the message was dropped; it's stated in the window's coordinate system.

The third argument, *target*, points to the BView that's scheduled to receive the **MessageDropped()** notification. It's the view located at *point*. However, **FilterMessageDropped()** can be implemented to replace the *target* BView with another view located within the same window. The replacement BView will then be notified instead.

FilterMessageDropped() is called whenever the user drops a dragged message within the window. By returning **TRUE**, it permits the target's **MessageDropped()** function to be called. By returning **FALSE**, it prevents any BView from notified of the message-dropped event.

The default version of **FilterMessageDropped()** simply returns **TRUE**.

See also: MessageDropped() in the BView class, **CurrentMessage()** in the BLooper class, "Message-Dropped Events" on page 49 of the introduction

FilterMouseDown()

virtual bool **FilterMouseDown(**BPoint *point*, BView ***target***)**

Implemented by derived classes to return **TRUE** if the mouse-down event located at *point* should be handled by a subsequent call to the *target* view's **MouseDown()** function, and **FALSE** if **MouseDown()** should not be called. The point is stated in the target view's coordinate system.

Before returning **TRUE**, this function can alter the BView that will receive the **MouseDown()** notification—simply by changing the object that *target* points to. The replacement target must be located in view hierarchy of the same window.

FilterMouseDown() is called for every mouse-down event within the window. BWindow's default version of the function never alters *point* and always returns **TRUE**.

See also: MouseDown() in the BView class, **CurrentMessage()** in the BLooper class, "Mouse-Down Events" on page 47 in the chapter introduction

FilterMouseMoved()

virtual bool **FilterMouseMoved(**BPoint *point*, ulong *area*, BMessage **message*, BView ***target***)**

Implemented by derived classes to preview a mouse-moved event before **MouseMoved()** is called for any of the window's BViews.

FilterMouseMoved() is called once for every mouse-moved event associated with the window. The event reports that the user has moved the cursor to a new *point* in the window's coordinate system. Normally, the BView that the cursor is over is notified by calling its **MouseMoved()** virtual function. If the cursor has moved out of one view and into another, both BViews are notified. However, by returning **FALSE**, **FilterMouseMoved()** prevents any BViews from being notified of the event. A return of **TRUE** permits **MouseMoved()** to be called.

The first argument, *point*, is the current location of the cursor, stated in the window's coordinate system. The second argument, *area*, conveys which part of the window the cursor is over. It will be one of the following constants:

CONTENT_AREA	The cursor is over the content area of the window.
CLOSE_BOX	The cursor is over the close box in the title tab.
TITLE_BAR	The cursor is inside the tab, but not over the close box.
RESIZE_AREA	The cursor is over the area in the right bottom corner where the window can be resized.
UNKNOWN_AREA	It's unknown where the cursor is, probably because it just left the window.

If the cursor is over a BView in the window's content area, a pointer to the view is passed as the final argument, *target*. If the cursor isn't over a BView, *target* points to a **NULL** value.

In the normal course of events, the target view will receive a **MouseMoved()** notification, provided **FilterMouseMoved()** returns **TRUE**. However, before returning **TRUE**, **FilterMouseMoved()** can alter the target view. Depending on which BView is chosen, the replacement BView will receive a **MouseMoved()** notification informing it either that the cursor has just entered it—even though the cursor is really inside another view—or that the cursor has moved somewhere else inside it, having previously entered—even though the cursor is actually no longer inside the view. If the cursor had previously entered the *target* view passed to this function, that view will be notified that the cursor has left it, even though it really hasn't.

If the user is moving the cursor to drag a BMessage object, the third argument, *message*, points to the dragged BMessage. If nothing is being dragged, *message* is **NULL**.

The BWindow version of this function simply returns **TRUE**.

See also: **MouseMoved()** and **DragMessage()** in the BView class, "Mouse-Moved Events" on page 48 of the chapter introduction

FindView()

> BView ***FindView(**BPoint *point***)** const
> BView ***FindView(**const char **name***)** const

Returns the view located at *point* within the window, or the view tagged with *name*. The point is specified in the window's coordinate system (the coordinate system of its top view), which has the origin at the upper left corner of the window's content area.

If no view is located at the point given, or no view within the window has the name given, this function returns **NULL**.

See also: **FindView()** in the BView class

FixMenus()

> virtual void **FixMenus(**void**)**

Implemented by derived classes to make sure menus are up-to-date before they're placed on-screen. This function is called just before a menu belonging to the window is about to be shown to the user. It gives the BWindow a chance to make any required alterations—for example, disabling or enabling particular items—so that the menus are in sync with the current state of the window.

See also: the BMenu and BMenuItem classes

Flush()

> void **Flush(**void**)** const

Flushes the window's connection to the Application Server, sending whatever happens to be in the out-going buffer to the Server. The buffer is automatically flushed on every update and after each message.

This function has the same effect as the **Flush()** function defined for the BView class.

See also: **Flush** in the BView class

Frame()

BRect **Frame(**void**)** const

Asks the Application Server for the current frame rectangle for the window and returns it. The frame rectangle encloses the content area of the window and is stated in the screen coordinate system. It's first set by the BWindow constructor, and is modified as the window is resized and moved.

See also: MoveBy(), **ResizeBy()**, the BWindow constructor

FrameMoved()

virtual void **FrameMoved(**BPoint *screenPoint***)**

Implemented by derived classes to respond to a notification that the window has moved. The move—which placed the left top corner of the window's content area at *screenPoint* in the screen coordinate system—could be the result of the user dragging the window or of the program calling **MoveBy()** or **MoveTo()**. If the user drags the window, **FrameMoved()** is called repeatedly as the window moves. If the program moves the window, it's called just once to report the new location.

The default version of this function does nothing.

See also: MoveBy(), "Window-Moved Events" on page 51 of the chapter introduction

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Implemented by derived classes to respond to a notification that the window's content area has been resized to a new *width* and *height*. The resizing could be the result of the program calling **ResizeTo()** or **ResizeBy()**—in which case **FrameResized()** is called just once to report the window's new size—or of a user action—in which case it's called repeatedly as the user drags a corner of the window to resize it.

The default version of this function does nothing.

See also: ResizeBy(), "Window-Resized Events" on page 51 of the chapter introduction

GetTitle() see SetTitle()

Hide(), Show()

virtual void **Hide(**void**)**

virtual void **Show(**void**)**

These functions hide the window so it won't be visible on-screen, and show it again.

Hide() removes the window from the screen. If it happens to be the active window, **Hide()** also deactivates it. Hiding a window hides all the views attached to the window. While the window is hidden, its BViews respond **TRUE** to **IsHidden()** queries.

Show() puts the window back on-screen. It places the window in front of other windows and makes it the active window.

Calls to **Hide()** and **Show()** can be nested; if **Hide()** is called more than once, you'll need to call **Show()** an equal number of times for the window to become visible again.

A window begins life hidden (as if **Hide()** had been called once); it takes an initial call to **Show()** to display it on-screen.

See also: IsHidden()

IsActive()

bool **IsActive(**void**)** const

Returns **TRUE** if the window is currently the active window, and **FALSE** if it's not.

See also: Activate()

IsFront()

bool **IsFront(**void**)** const

Returns **TRUE** if the window is currently the frontmost window on-screen, and **FALSE** if it's not.

IsHidden()

bool **IsHidden(**void**)** const

Returns **TRUE** if the window is currently hidden, and **FALSE** if it isn't.

Windows are hidden at the outset. The **Show()** function puts them on-screen, and **Hide()** can be called to hide them again.

If **Show()** has been called to unhide the window, but the window is totally obscured by other windows or occupies coordinates that don't intersect with the physical screen, **IsHidden()** will nevertheless return **FALSE**, even though the window isn't visible.

See also: **Hide()**

IsSavePanelRunning() see RunSavePanel()

Lock(), Unlock()

bool **Lock(**void**)**

void **Unlock(**void**)**

These functions lock and unlock the BWindow, so that another thread can't alter crucial data while the current thread is in the middle of doing something. Only one thread can have the window locked at any given time. **Lock()** waits until it can lock the BWindow, then returns **TRUE**. It returns **FALSE** only if the window can't be locked at all—for example if the BWindow was destroyed.

Calls to **Lock()** and **Unlock()** can be nested. If **Lock()** has been called more than once, it will take an equal number of **Unlock()** calls to unlock the window.

It's not necessary to lock a window when calling functions defined in the BWindow class. BWindow functions are implemented to call **Lock()** and **Unlock()** when necessary.

< Currently, **Lock()** will not allow a thread to lock a BWindow if the same thread has the BApplication object locked. You must unlock the BApplication object before locking a window. >

See also: **Lock()** in the BApplication class

Modifiers()

ulong **Modifiers(**void**)** const

Returns a mask that has a bit set for each modifier key the user is holding down and for each keyboard lock that's set. The mask can be tested against these constants:

SHIFT_KEY	**COMMAND_KEY**	**CAPS_LOCK**
CONTROL_KEY	**MENU_KEY**	**SCROLL_LOCK**
OPTION_KEY	**NUM_LOCK**	

If a Shift, Command, Control, or Option key is down, the mask can be further tested against the following constants to reveal which key it is, the one on the left or the one on the right:

LEFT_SHIFT_KEY	**RIGHT_SHIFT_KEY**
LEFT_CONTROL_KEY	**RIGHT_CONTROL_KEY**
LEFT_OPTION_KEY	**RIGHT_OPTION_KEY**
LEFT_COMMAND_KEY	**RIGHT_COMMAND_KEY**

No bits are set (the mask is 0) if none of the modifiers keys are down and no locks are on.

See also: Modifiers() and **GetKeys()** in the BView class, **CurrentMessage()** in the BLooper class

MoveBy(), MoveTo()

void **MoveBy(**float *horizontal*, float *vertical***)**

void **MoveTo(**BPoint *point***)**
void **MoveTo(**float *x*, float *y***)**

These functions move the window without resizing it. **MoveBy()** adds *horizontal* coordinate units to the left and right components of the window's frame rectangle and *vertical* units to the frame's top and bottom. If *horizontal* and *vertical* are negative, the window moves upward and to the left. If they're positive, it moves downward and to the right. **MoveTo()** moves the left top corner of the window's content area to *point*—or (*x*, *y*)—in the screen coordinate system; it adjusts all coordinates in the frame rectangle accordingly.

None of the values passed to these functions should specify fractional coordinates; a window must be aligned on screen pixels. Fractional values will be rounded to the closest whole number.

Neither function alters the BWindow's coordinate system or bounds rectangle.

When a window is moved by one of these functions, a window-moved event is reported to the window. This results in the BWindow's **FrameMoved()** function being called.

See also: FrameMoved()

NeedsUpdate()

bool **NeedsUpdate(**void**)** const

Returns **TRUE** if any of the views within the window need to be updated, and **FALSE** if they're all up-to-date.

See also: UpdateIfNeeded()

PreferredReceiver() see **CurrentFocus()**

Quit(), Close()

virtual void **Quit(**void**)**

inline void **Close(**void**)**

Quit() gets rid of the window and all its views. This function removes the window from the screen, deletes all the BViews in its view hierarchy, destroys the window thread, removes the window's connection to the Application Server, and, finally, deletes the BWindow object.

Use this function, rather than the **delete** operator, to destroy a window. **Quit()** applies the operator after it empties the BWindow of views and severs its connection to the application and Server. It's dangerous to apply **delete** while these connections remain intact.

BWindow's **Quit()** works much like the BLooper function it overrides. When called from the BWindow's thread, it doesn't return. When called from another thread, it returns after all previously posted messages have been responded to and the BWindow and its thread have been destroyed.

Close() is a synonym of **Quit()**. It simply calls **Quit()** so if you override **Quit()**, you'll affect how both functions work.

See also: QuitRequested() and **Quit()** in the BLooper class, **QuitAllWindows()** and **QuitRequested()** in the BApplication class

RemoveChild()

virtual bool **RemoveChild(**BView **aView***)**

Removes *aView* from the BWindow's view hierarchy, but only if *aView* was added to the hierarchy as a child of the window's top view (by calling BWindow's version of the **AddChild()** function).

If *aView* is successfully removed, **RemoveChild()** returns **TRUE**. If not, it returns **FALSE**.

See also: AddChild()

RemoveMouseEvents()

void **RemoveMouseEvents(**void**)**

< Removes messages reporting mouse-down and mouse-up events from the window's message queue. Don't rely on this function; it's likely to be removed from the API. Instead, get the BMessageQueue and call its **RemoveMessage()** function, as follows:

```
     myWindow->MessageQueue()->RemoveMessage(MOUSE_DOWN);
     myWindow->MessageQueue()->RemoveMessage(MOUSE_UP)
```
>

See also: MessageQueue() in the BLooper class of the Application Kit

RemoveShortcut() see AddShortcut()

ResizeBy(), ResizeTo()

void **ResizeBy(**float *horizontal*, float *vertical***)**

void **ResizeTo(**float *width*, float *height***)**

These functions resize the window, without moving its left and top sides. **ResizeBy()** adds *horizontal* coordinate units to the width of the window and *vertical* units to its height. **ResizeTo()** makes the content area of the window *width* units wide and *height* units high. Both functions adjust the right and bottom components of the frame rectangle accordingly.

Since a BWindow's frame rectangle must line up with screen pixels, only integral values should be passed to these functions. Values with fractional components will be rounded to the nearest whole number.

When a window is resized, either programmatically by these functions or by the user, the BWindow's **FrameResized()** virtual function is called to notify it of the change.

See also: FrameResized()

RunSavePanel(), CloseSavePanel(), IsSavePanelRunning()

long **RunSavePanel(**const char **tentativeName* = **NULL**,
const char **windowTitle* = **NULL**,
const char **buttonLabel* = **NULL**,
BMessage **message* = **NULL)**

void **CloseSavePanel(**void**)**

bool **IsSavePanelRunning(**void**)**

RunSavePanel() requests the Browser to display a panel where the user can chose how to save the document displayed in the window. The panel permits the user to navigate the file system and type in file and directory names.

The arguments to this function are all optional. They're used to configure the panel:

- If passed a *tentativeName* for the document displayed in the window, the save panel will place it in a text field where the user can type a name for the file. The name might designate an existing file, or it might simply be a placeholder name like "UNNAMED" or "UNTITLED–3". If a *tentativeName* isn't passed, the text field will be empty.

- If another *windowTitle* is not specified, the title of the window will include the tentative filename. It will be "Save *tentativeName* As..." preceded by the name of the application. The name is enclosed in quotes. For example:

  ```
  WishMaker : Save "UNTITLED-3" As...
  ```

  If a *tentativeName* isn't passed, the quotes will be empty.

- If a *buttonLabel* label isn't provided, the principal button in the panel (the default button) will be labeled "Save". (The panel also has a "Cancel" button.)

- If a *message* is passed, it can contain entries that further configure the panel. It also serves as a model for the message that reports the directory and filename the user selected. If a *message* isn't provided, this information will be reported in a standard **SAVE_REQUESTED** message.

If the *message* has one or both of the following entries, they will be used to help configure the panel:

Data name	Type code	Description
"directory"	**REF_TYPE**	The **record_ref** for the directory that the panel should display when it first comes on-screen. If this entry is absent, the panel will initially display the current directory of the current volume.
"frame"	**RECT_TYPE**	A BRect that sets the size and position of the panel in screen coordinates. If this

entry is absent, the Browser will choose an appropriate frame rectangle for the panel.

When the user finishes choosing where to save the file and operates the “Save” (or *buttonLabel*) button, the file panel sends a message to the BApplication object. If a customized *message* is provided, it’s used as the model for the message that’s sent. If a *message* isn’t provided, a standard **SAVE_REQUESTED** message is sent instead. In either case, it has two data entries:

Data name	Type code	Description
“name”	**STRING_TYPE**	The name of the file in which the document should be saved.
“directory”	**REF_TYPE**	A **record_ref** reference to the directory where the file should reside.

A **SAVE_REQUESTED** message is dispatched by calling the **SaveRequested()** hook function; the “name” and “directory” are passed as arguments to **SaveRequested()**. This function should be implemented to create the file, if necessary, and save the document. **RunSavePanel()** doesn’t do this work; it simply delivers a BMessage object with the information you need to do the job.

A customized *message* works much like the model messages assigned to BControl objects and BMenuItems. The save panel makes a copy of the model, adds the “name” and “directory” entries (as described above) to the copy, and sends the copy to the application, which delivers it to the BWindow. Other entries in the message remain unchanged.

The *message* can have any command constant you choose. If it’s **SAVE_REQUESTED**, the “name” and “directory” will be extracted from the message and passed to **SaveRequested()**. Otherwise, nothing is extracted and the message is dispatched by calling **MessageReceived()**.

The save panel doesn’t automatically disappear when the user operates the “Save” (or *buttonLabel*) button; it remains on-screen until **CloseSavePanel()** is called (or until the application quits). You can choose to leave the panel on-screen if the user hasn’t chosen a valid filename. **IsSavePanelRunning()** will report whether the save panel is currently displayed on-screen. A BWindow can run only one save panel at a time.

The save panel is automatically closed when user operates the “Cancel” button. Whenever it’s closed, by the user or the application, a **PANEL_CLOSED** message is sent to the application and the **SavePanelClosed()** hook function is called.

RunSavePanel() returns **NO_ERROR** if it succeeds in getting the Browser to put the panel on-screen. If the Browser isn’t running or the save panel already is, it returns **SYS_ERROR**. If the Browser is running but the application can’t communicate with it, it returns an error code that indicates what went wrong; these codes are the same as those documented for the BMessenger class in the Application Kit.

See also: SaveRequested(), **SavePanelClosed()**

SavePanelClosed()

virtual void **SavePanelClosed(**BMessage **message***)**

Implemented by derived classes to take note when the save panel is closed. The *message* argument contains information about how the panel was closed and its state at the time it was closed. It has entries under the names "frame" (the panel's frame rectangle), "directory" (the directory the panel displayed), and "canceled" (whether the user closed the panel). Some of this information can be retained to configure the panel the next time it runs.

See also: "Panel-Closed Events" on page 52 of the chapter introduction, **RunSavePanel()**

SaveRequested()

virtual void **SaveRequested(**record_ref *directory*, const char **filename***)**

Implemented by derived classes to save the document displayed in the window. This function is called when the BWindow receives a **SAVE_REQUESTED** message from the save panel. It reports that the user has asked for the file to be saved in the *directory* indicated and assigned the specified *filename*. The file may already exist, or the application may need to create it to carry out the request.

There's no guarantee that the *directory* and *filename* are valid.

If the file can be saved as requested, you may want this function to call **CloseSavePanel()** to remove the panel from the screen. If the file can't be saved, **SaveRequested()** should notify the user. In some cases, you may want to leave the panel on-screen so the user can try again with a different directory or filename.

See also: **RunSavePanel()**

ScreenChanged()

virtual void **ScreenChanged(**BRect *frame*, color_space *mode***)**

Implemented by derived classes to respond to a notification that the screen configuration has changed. This function is called for all affected windows when:

- The number of pixels the screen displays (the size of the pixel grid) is altered,
- < The screen changes its location in the screen coordinate system, or
- The color mode of the screen changes. >

frame is the new frame rectangle of the screen, and *mode* is its new color space.

< Currently, there can be only one monitor per machine, so the screen can't change where its located in the screen coordinate system. Moreover, there is no way to change the screen color space. Only the pixel grid can change. >

See also: **set_screen_size()**, "Screen-Changed Events" on page 51 of the chapter introduction

SetDefaultButton(), DefaultButton()

void **SetDefaultButton(**BButton **button***)**

BButton ***DefaultButton(**void**)** const

SetDefaultButton() makes *button* the default button for the window—the button that the user can operate by pressing the Enter key. **DefaultButton()** returns the button that currently has that status, or **NULL** if there is no default button.

At any given time, only one button in the window can be the default. **SetDefaultButton()** may, therefore, affect two buttons: the one that's forced to give up its status as the default button, and the one that acquires that status. Both buttons are redisplayed, so that the user can see which one is currently the default, and both are notified of their change in status through **MakeDefault()** virtual function calls.

If the argument passed to **SetDefaultButton()** is **NULL**, there will be no default button for the window. The current default button loses its status and is appropriately notified with a **MakeDefault()** function call.

The Enter key can operate the default button only while the window is the active window. However, the BButton doesn't have to be the focus view. If another view is the focus view, it won't be notified of key-down events where the character reported is **ENTER**.

See also: **MakeDefault()** in the BButton class

SetDiscipline()

void **SetDiscipline(**bool *flag***)**

Sets a *flag* that determines how much programming discipline the system will enforce. When *flag* is **TRUE**, as it is by default, Kit functions will check to be sure various rules are adhered to. For example, most BView functions will require the caller to first lock the window. < Currently, this is the only rule that comes under the discipline flag. > When *flag* is **FALSE**, these rules are not enforced.

The discipline *flag* should be set to **TRUE** while an application is being developed. However, once it has matured, and it's clear that none of the rules are being disobeyed, the *flag* can be set to **FALSE**. This will eliminate various checking operations and improve performance.

See also: "Locking the Window" in the BView class overview

SetLimits()

void **SetLimits(**float *minWidth*, float *maxWidth*, float *minHeight*, float *maxHeight***)**

Sets limits on the size of the window. The user won't be able to resize the window to have a width less than *minWidth* or greater than *maxWidth*, nor to have a height less than *minHeight* or greater than *maxHeight*. By default, the minimums are sufficiently small and the maximums sufficiently large to accommodate any window within reason.

This function constrains the user, not the programmer. It's legal for an application to set a window size that falls outside the permitted range. The limits are imposed only when the user attempts to resize the window; at that time, the window will jump to a size that's within range.

Since the sides of a window must line up on screen pixels, the minimums and maximums should be whole numbers.

See also: the BWindow constructor

SetMainMenuBar()

void **SetMainMenuBar(**BMenuBar **menuBar***)**

Makes the specified BMenuBar object the "main" menu bar for the window—the object that's at the root of the menu hierarchy that users can navigate using the keyboard.

If a window contains only one BMenuBar view, it's automatically designated the main menu bar. If there's more than one BMenuBar in the window, the last one added to the window's view hierarchy is considered to be the main one.

If there's a "true" menu bar displayed along the top of the window, its menu hierarchy is the one that users should be able to navigate using the keyboard. This function can be called to make sure that the BMenuBar object at the root of that hierarchy is the "main" menu bar.

See also: the BMenuBar class

SetPulseRate()

void **SetPulseRate(**long *milliseconds***)**

Sets how often **Pulse()** is called for the BWindow's views.

By turning on the **PULSE_NEEDED** flag, a BView can request periodic **Pulse()** notifications. By default, pulse events are posted every 500 milliseconds, as long as no other events are pending. Each event causes **Pulse()** to be called for every BView that requested the notification.

SetPulseRate() permits you to set a different interval. The interval set should not be less than 100 milliseconds; differences less than 50 milliseconds may not be noticeable. A finer granularity can't be guaranteed.

All BViews attached to the same window share the same pulse rate.

See also: **Pulse()** in the BView class

SetTitle(), GetTitle()

void **SetTitle(**const char **newTitle***)**

void **GetTitle(**char **theTitle***)** const

These functions set and return the window's title. **SetTitle()** replaces the current title with *newTitle*. It also renames the window thread in the following format:

```
"w>newTitle"
```

where as many characters of the *newTitle* are included in the thread name as will fit.

GetTitle() copies the current title into the memory referred to by *theTitle.* Make sure that enough memory is allocated to hold the title no matter how long it might be.

A window's title and thread name are originally set by an argument passed to the BWindow constructor.

See also: the BWindow constructor

Show() see Hide()

UpdateIfNeeded()

void **UpdateIfNeeded(**void**)**

Causes the **Draw()** virtual function to be called immediately for each BView object that needs updating. If no views in the window's hierarchy need to be updated, this function does nothing.

BView's **Invalidate()** function generates an update message that the BWindow receives just as it receives other messages. Although update messages take precedence over other kinds of messages the BWindow receives, the window thread can respond to only one message at a time. It will update the invalidated view as soon as possible, but it must finish responding to the current message before it can get the update message.

This may not be soon enough for a BView that's engaged in a time-consuming response to the current message. **UpdateIfNeeded()** forces an immediate update, without waiting to return the BWindow's message loop. However, if works only if called from within the BWindow's thread.

(Because the message loop expedites the handling of update messages, they're never considered the current message and are never returned by BLooper's **CurrentMessage()** function.)

See also: **Draw()** in the BView class, **Invalidate()** in the BView class, **NeedsUpdate()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Implemented by derived classes to make any changes necessary when the window becomes the active window, or when it ceases being the active window. If *active* is **TRUE**, the window has just become the new active window, and if *active* is **FALSE**, it's about to give up that status to another window.

The BWindow receives a **WindowActivated()** notification whenever its status as the active window changes. Each of its BViews is also notified.

See also: **WindowActivated()** in the BView class

Global Functions

This section describes the global (nonmember) functions defined in the Interface Kit. All these functions deal with aspects of the system-wide environment for the user interface—the keyboard, the screen, installed fonts and symbol sets, and the list of possible colors.

The Application Server maintains this environment. Therefore, for any of these functions to work, your application needs a connection to the Server. The connection they all depend on is the one established when the BApplication object is constructed. Consequently, none of them should be called before a BApplication object is present in your application.

count_fonts() see **get_font_name()**

count_screens() see **get_screen_info()**

count_symbol_sets() see **get_symbol_set_name()**

desktop_color() see **set_desktop_color()**

get_font_name(), count_fonts()

<interface/InterfaceDefs.h>

void **get_font_name(**long *index*, font_name **name***)**

long **count_fonts(**void**)**

These two functions are used in combination to get the names of all installed fonts. For example:

```
long numFonts = count_fonts();
font_name buf;

for ( long i = 0; i < numFonts; i++ ) {
    get_font_name(i, &buf);
    . . .
}
```

The names of all installed fonts are kept in an alphabetically ordered list. **get_font_name()** reads one of the names from the list, the name at *index*, and copies it into the *name* buffer. Font names can be up to 32 characters long, plus a null terminator. Indices begin at 0.

count_fonts() returns the number of fonts currently installed, the number of names in the list.

See also: GetFontInfo() and **SetFontName()** in the BView class

get_keyboard_id()

<interface/InterfaceDefs.h>

long **get_keyboard_id(**ushort **theId***)**

Obtains the keyboard identifier from the Application Server and writes it into the variable referred to by *theId*. This number reveals what kind of keyboard is currently attached to the computer.

The identifier for the standard 101-key keyboard—and for keyboards with a similar set of keys—is 0x83ab. < Currently, this is the only value this function can provide. > See “Key Codes” on page 53 for illustrations showing the keys found on a standard keyboard.

If unsuccessful for any reason, **get_keyboard_id()** returns **SYS_ERROR**. If successful, it returns **NO_ERROR**.

get_screen_info(), count_screens()

<interface/InterfaceDefs.h>

void **get_screen_info(**screen_info **theInfo***)**
void **get_screen_info(**long *index*, screen_info **theInfo***)**

long **count_screens(**void**)**

These functions provide information about the monitors (screens) that are currently hooked up to the Be computer.

Each screen that’s attached to the Be machine is identified by an index into a system-wide screen list. The screen at index 0 is the one that has the origin of the screen coordinate system at its left top corner. Other screens in the list are unordered; they’re located elsewhere in the screen coordinate system that the first screen defines.
< Currently, multiple screens are not supported, so the screen at index 0 is the only one in the list. >

get_screen_info() writes information about the screen at *index* into the **screen_info** structure referred to by *theInfo*. If no index is mentioned, this function assumes the screen at index 0. The **screen_info** structure contains the following fields:

color_space **mode**	The depth and color interpretation of pixel data in the screen's frame buffer. (See the BBitmap class description for an explanation of the various **color_space** modes.)
BRect **frame**	The frame rectangle of the screen—the rectangle that defines the size and location of the screen in the screen coordinate system.
void ***bits**	A pointer to the frame buffer.
long **bytes_per_row**	The number of bytes used to specify one row of pixel data in the frame buffer.

count_screens() returns the number of screens (monitors) that are attached to the computer. < Currently, no more than one screen can be attached, so this function always returns 1. >

See also: the BBitmap class

get_symbol_set_name(), count_symbol_sets()

<interface/InterfaceDefs.h>

void **get_symbol_set_name(**long *index*, symbol_set_name **name***)**

long **count_symbol_sets(**void**)**

These functions are used to get the names of all available symbol sets. They work much like the parallel font functions **get_font_name()** and **count_fonts()**.

A symbol set associates character symbols (glyphs) with character codes (ASCII values). They differ mainly in how they extend the standard ASCII set—how they assign characters to codes above 0x7f.

get_symbol_set_name() gets one name from the list of symbol sets, the name at *index*, and copies it into the *name* buffer. **count_symbol_sets()** returns the total number of symbol sets (the number of names in the list).

Unlike font names, the names of symbol sets are not arranged alphabetically.

Every font implements every symbol set. However, some fonts implement particular sets more fully than others—that is, some characters in a symbol set may not be available in some fonts. But the position of each character in the set (its character code) remains the same across all fonts.

See also: **SetFont()** in the BView class, **get_font_name()**

index_for_color()

<interface/InterfaceDefs.h>

uchar **index_for_color(**rgb_color *aColor***)**
uchar **index_for_color(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**

Returns an index into the list of 256 colors that comprise the **COLOR_8_BIT** color space. The value returned picks out the listed color that most closely matches a full **RGB_24_BIT** color—specified either as an **rgb_color** value, *aColor*, or by its *red*, *green*, and *blue* components. < (The *alpha* component is currently ignored.) >

The returned index identifies a color in the **COLOR_8_BIT** color space. It can, for example, be passed to BBitmap's **SetBits()** function.

To find the fully specified color that an index picks out, you have to get the color list from the system color map. For example, if you first obtain the index for the "best fit" color that most closely matches an arbitrary color,

```
uchar index = index_for_color(134, 210, 6);
```

you can then use the index to locate that color in the color list:

```
rgb_color bestFit = system_colors()->color_list[index];
```

See also: **system_colors()**, the BBitmap class

restore_key_map() see **system_key_map()**

set_desktop_color(), desktop_color()

<interface/InterfaceDefs.h>

void **set_desktop_color(**rgb_color *color***)**

rgb_color **desktop_color(**void**)**

These functions set and return the color that will be displayed on the so-called "desktop"—the bare backdrop against which windows are displayed. The color is the same for all screens attached to the same machine.

set_keyboard_locks()

<interface/InterfaceDefs.h>

void **set_keyboard_locks(**ulong *modifiers***)**

Turns the keyboard locks—Caps Lock, Num Lock, and Scroll Lock—on and off. The keyboard locks that are listed in the *modifiers* mask passed as an argument are turned on; those not listed are turned off. The mask can be 0 (to turn off all locks) or it can contain any combination of the following constants:

CAPS_LOCK
NUM_LOCK
SCROLL_LOCK

See also: **system_key_map()**, **Modifiers()** in the BView class

set_modifier_key()

<interface/InterfaceDefs.h>

void **set_modifier_key(**ulong *modifier*, ulong *key***)**

Maps a *modifier* role to a particular *key* on the keyboard, where *key* is a key identifier and *modifier* is one of the these constants:

CAPS_LOCK	**LEFT_SHIFT_KEY**	**RIGHT_SHIFT_KEY**
NUM_LOCK	**LEFT_CONTROL_KEY**	**RIGHT_CONTROL_KEY**
SCROLL_LOCK	**LEFT_OPTION_KEY**	**RIGHT_OPTION_KEY**
MENU_KEY	**LEFT_COMMAND_KEY**	**RIGHT_COMMAND_KEY**

The *key* in question serves as the named modifier key, unmapping any key that previously played that role. The change remains in effect until the default key map is restored.

Modifier keys can also be mapped by calling **system_key_map()** and altering the **key_map** structure directly. This function is merely a convenient alternative for accomplishing the same thing.

See also: **system_key_map()**

set_screen_size()

<interface/InterfaceDefs.h>

void **set_screen_size(**long *index*, BRect *grid*, bool *makeDefault* = **TRUE)**

Sets the size of the pixel grid displayed on the monitor at *index* in the screen list.
< Since a machine currently can have only one monitor, *index* should always be 0. >

The grid is the size of the screen measured in pixels—the number of pixels that it displays horizontally and vertically. This function sets the grid to the dimensions recorded in the *grid* rectangle. Only two screen sizes are currently supported—640 × 480 and 800 × 600—so the rectangle passed should be one of the following:

```
BRect(0.0, 0.0, 639.0, 479.0)
BRect(0.0, 0.0, 799.0, 599.0)
```

Another rectangle with the same dimensions as one of these can be substituted. The rectangle doesn't have to match the frame rectangle of the screen; only its dimensions matter.

(The rectangles are specified in coordinate units, so their dimensions are one unit less than the dimensions of the screen in pixels. This is because a rectangle aligned on screen pixels covers one more column of pixels than its width and one more row than its height. See the BRect class for an explanation.)

If the *makeDefault* flag is **TRUE**, the new screen size becomes the default and will be used the next time the machine reboots. If the flag is **FALSE**, the change is for the current session only; the machine will reboot in the previously determined default screen size.

When the size of the screen grid changes, every affected BWindow object is notified with a **ScreenChanged()** function call. < Since there's currently only one screen, all windows are affected and all, whether on-screen or hidden, receive **ScreenChanged()** notifications. >

See also: **ScreenChanged()** in the BWindow class, **get_screen_info()**

system_colors()

<interface/InterfaceDefs.h>

color_map ***system_colors(**void**)**

Returns a pointer to the system's *color map*. The color map defines the set of 256 colors that can be displayed in the **COLOR_8_BIT** color space. A single set of colors is shared by all applications connected to the Application Server.

The **color_map** structure is defined in **interface/InterfaceDefs.h** and contains the following fields:

long **id**	An identifier that the Server uses to distinguish one color map from another.
rgb_color **color_list**[256]	A list of the 256 colors, expressed as **rgb_color** structures. Indices into the list can be used to specify colors in the **COLOR_8_BIT** color space. See the **index_for_color()** function above.
uchar **inversion_map**[256]	A mapping of each color in the **color_list** to its opposite color. Indices are mapped to indices. An example of how this map might be used is given below.
uchar **index_map**[32768]	An array that maps RGB colors—specified using five bits per component—to their nearest counterparts in the color list. An example of how to use this map is also given below.

The **inversion_map** is a list of indices into the **color_list** where each index locates the "inversion" of the original color. The inversion of the *n*'th color in **color_list** would be found as follows:

```
    uchar inversionIndex = system_colors()->inversion_map[n];
    rgb_color inversionColor =
                    system_colors()->color_list[inversionIndex];
```

Inverting an inverted index returns the original index, so this code

```
    uchar color = system_colors()->inversion_map[inversionIndex];
```

would return *n*. < Inverted colors are used, primarily, for highlighting. Given a color, its highlight complement is its inversion. >

The **index_map** maps every RGB combination that can be expressed in 15 bits (five bits per component) to a single **color_list** index that best approximates the original RGB data. The following example demonstrates how to squeeze 24-bit RGB data into a 15-bit number that can be used as an index into the **index_map**:

```
    long rgb15 = ( ((red & 0xf8) << 7) |
                   ((green & 0xf8) << 2) |
                   ((blue & 0xf8) >> 3) );
```

Most applications won't need to use the index map directly; the **index_for_color()** function performs the same conversion with less fuss (no masking and shifting required). However, applications that implement repetitive graphic operations, such as dithering, may want to access the index map themselves, and thus avoid the overhead of an additional function call.

You should never modify or free the **color_map** structure returned by this function.

See also: index_for_color()

system_key_map(), restore_key_map()

<interface/InterfaceDefs.h>

key_map ***system_key_map(**void**)**

void **restore_key_map(**void**)**

The first of these functions returns a pointer to the system's key map—the structure that describes the role of each key on the keyboard. The second function restores the default key map, in case any of its fields have been changed.

The system key map is shared by all applications. An application can alter values in the structure that **system_key_map()** returns—and thus alter the roles that the keys play—

but it should make sure that those changes are local to itself and don't affect other, unsuspecting applications. In particular, it should:

- Modify the key map only when one of its windows becomes the active window, and
- Restore the default key map when it no longer has the active window.

Through the Keyboard utility, users can configure the keyboard to their liking. The user's preferences affect all applications; they're captured in the default key map and stored in a file (**system/Key_map**).

When the machine reboots or when **restore_key_map()** is called, the key map is read from this file. If the file doesn't exist, the original map encoded in the Application Server is used.

The **key_map** structure contains a large number of fields, but it can be broken down into these five parts:

- A version number.
- A series of fields that determine which keys will function as modifier keys—such as Shift, Control, or Num Lock.
- A field that sets the initial state of the keyboard locks in the default key map.
- A series of ordered tables that assign character values to keys. Keys assigned a value other than 0 produce key-down events when pressed. This includes almost all the keys on the keyboard (all except for a handful of modifier keys).
- A series of tables that locate the dead keys for diacritical marks and determine how a combination of a dead key plus another key is mapped to a particular character.

The following sections describe each part of the **key_map** structure in turn.

Version. The first field of the key map is a version number:

ulong **version**	An internal identifier for the key map.

The version number doesn't change when the user configures the keyboard, and shouldn't be changed programmatically either. You can ignore it.

Modifiers. Modifier keys set states that affect other user actions on the keyboard and mouse. Eight modifier states are defined—Shift, Control, Option, Command, Menu, Caps Lock, Num Lock, and Scroll Lock. These states are discussed under "Modifier Keys" on page 57 of the introduction. They overlap, but don't exactly match the key caps found on a standard keyboard—which generally has a set of Alt(ernate) keys, rarely Option keys, and only sometimes Command and Menu keys. Because of these differences, the mapping of keys to modifiers is the area of the key map most open to the user's personal judgement and taste, and consequently to changes in the default configuration. Applications are urged to respect the user's preferences.

Since two keys, one on the left and one on the right, can be mapped to the Shift, Control, Option, and Command modifiers, the keyboard can have as many as twelve modifier keys. The **key_map** structure has one field for each key:

ulong **caps_key**	The key that functions as the Caps Lock key—by default, this is the key labeled "Caps Lock," key 0x3b.
ulong **scroll_key**	The key that functions as the Scroll Lock key—by default, this is the key labeled "Scroll Lock," key 0x0f.
ulong **num_key**	The key that functions as the Num Lock key—by default, this is the key labeled "Num Lock," key 0x22.
ulong **left_shift_key**	A key that functions as a Shift key—by default, this is the key on the left labeled "Shift," key 0x4b.
ulong **right_shift_key**	Another key that functions as a Shift key—by default, this is the key on the right labeled "Shift," key 0x56.
ulong **left_command_key**	A key that functions as a Command key—by default, this is the left "Alt" key, key 0x5d.
ulong **right_command_key**	Another key that functions as a Command key—by default, this is the right "Alt" key, key 0x5f.
ulong **left_control_key**	A key that functions as a Control key—by default, this is the key labeled "Control" on the left, key 0x5c.
ulong **right_control_key**	Another key that functions as a Control key—by default, this key is not mapped. (The value of the field is set to 0.)
ulong **left_option_key**	A key that functions as an Option key—by default, this is the key that's labeled "Command" (or that has a command symbol) on the right (*not* left) of some keyboards, key 0x67. This key doesn't exist on, and therefore isn't mapped for, a standard 101-key keyboard.

ulong **right_option_key**	A key that functions as an Option key—by default, this is the key labeled “Control” on the right, key 0x60.
ulong **menu_key**	A key that initiates keyboard navigation of the menu hierarchy—by default, this is the key labeled “Menu,” key 0x68. This key doesn’t exist on, and therefore isn’t mapped for, a standard 101-key keyboard.

Each field names the key that functions as that modifier. For example, when the user holds down the key whose code is set in the **right_option_key** field, the **OPTION_KEY** and **RIGHT_OPTION_KEY** bits are turned on in the modifiers mask that the various **Modifiers()** functions return. When the user then strikes a character key, the **OPTION_KEY** state influences the character that’s generated.

If a modifier field is set to a value that doesn’t correspond to an actual key on the keyboard (including 0), that field is not mapped. No key fills that particular modifier role.

Keyboard locks. One field of the key map sets initial modifier states:

ulong **lock_settings**	A mask that determines which keyboard locks are turned on when the machine reboots or when the default key map is restored.

The mask can be 0 or may contain any combination of these three constants:

CAPS_LOCK
SCROLL_LOCK
NUM_LOCK

It’s 0 by default; there are no initial locks.

Altering the **lock_settings** field has no effect unless the altered key map is made the default (by writing it to a file that replaces **system/Key_map**).

Character maps. The principal job of the key map is to assign character values to keys. This is done in a series of nine tables:

ulong **control_map**[128]	The characters that are produced when a Control key is down but both Command keys are up.
ulong **option_caps_shift_map**[128]	The characters that are produced when Caps Lock is on and both a Shift key and an Option key are down.

ulong **option_caps_map**[128]	The characters that are produced when Caps Lock is on and an Option key is down.
ulong **option_shift_map**[128]	The characters that are produced when both a Shift key and an Option key are down.
ulong **option_map**[128]	The characters that are produced when an Option key is down.
ulong **caps_shift_map**[128]	The characters that are produced when Caps Lock is on and a Shift key is down.
ulong **caps_map**[128]	The characters that are produced when Caps Lock is on.
ulong **shift_map**[128]	The characters that are produced when a Shift key is down.
ulong **normal_map**[128]	The characters that are produced when none of the other tables apply.

Each of these tables is an array of 128 characters (declared as **ulong**s). Key codes are used as indices into the arrays. The value stored at any particular index is the character associated with that key. For example, the code assigned to the *M* key is 0x52; the characters to which the *M* key is mapped are recorded at index 0x52 in the various arrays.

The tables are ordered. Character values from the first applicable array are used, even if another array might also seem to apply. For example, if Caps Lock is on and a Control key is down (and both Command keys are up), the **control_map** array is used, not **caps_map**. If a Shift key is down and Caps Lock is on, the **caps_shift_map** is used, not **shift_map** or **caps_map**.

Notice that the last eight tables (all except **control_map**) are paired, with a table that names the Shift key (...**_shift_map**) preceding an equivalent table without Shift:

- **option_caps_shift_map** is paired with **option_caps_map**,
- **option_shift_map** with **option_map**,
- **caps_shift_map** with **caps_map**, and
- **shift_map** with **normal_map**.

These pairings are important for a special rule that applies to keys on the numerical keypad when Num Lock is on:

- If the Shift key is down, the non-Shift table is used.
- However, if the Shift key is *not* down, the Shift table is used.

In other words, Num Lock inverts the Shift and non-Shift tables for keys on the numerical keypad.

Not every key needs to be mapped to a character. If the value recorded in a table is 0, the key corresponding to that index is not mapped to a character given the particular modifier states the table represents. Generally, modifier keys are not mapped to characters, but all other keys are.

Dead keys. Next are the tables that map combinations of keys to single characters. The first key in the combination is "dead"—it doesn't produce a key-down event until the user strikes another character key. When the user hits the second key, one of two things will happen: If the second key is one that can be used in combination with the dead key, a single key-down event reports the combination character. If the second key doesn't combine with the dead key, two key-down events occur, one reporting the dead-key character and one reporting the second character.

There are five dead-key tables:

ulong **acute_dead_key**[32]	The table for combining an acute accent (´) with other characters.
ulong **grave_dead_key**[32]	The table for combining a grave accent (`) with other characters.
ulong **circumflex_dead_key**[32]	The table for combining a circumflex (ˆ) with other characters.
ulong **dieresis_dead_key**[32]	The table for combining a dieresis (¨) with other characters.
ulong **tilde_dead_key**[32]	The table for combining a tilde (˜) with other characters

The tables are named after diacritical marks that can be placed on more than one character. However, the name is just a mnemonic; it means nothing. The contents of the table determine what the dead key is and how it combines with other characters. It would be possible, for example, to remap the **tilde_dead_key** table so that it had nothing to do with a tilde.

Each table consists of a series of up to 16 character pairs, where each character is declared as a **ulong**. The first character in the pair is the one that must be typed immediately after the dead key. The second character is the resulting character, the character that's produced by the combination of the dead key plus the first character in the pair. For example, if the first character is 'o', the second might be 'ô'—meaning that the combination of a dead key plus the character 'o' produces a circumflexed 'ô'.

The character pairs in the default **grave_dead_key** array look something like this:

```
' ', '´',
'A', 'À',
'E', 'È',
'I', 'Ì',
'O', 'Ò',
'U', 'Ù',
'a', 'à',
'e', 'è',
'i', 'ì',
'o', 'ò',
'u', 'ù',
. . .
```

By convention, the first pair in each array is a space followed by the dead-key character itself. This pair does double duty: It states that the dead key plus a space yields the dead-key character, and it also names the dead key. The system understands what the dead key is from the second character in the array. Any key that produces that character while an Option key is held down will be dead and will combine to produce the characters listed in the array.

The Option key is an essential ingredient; a key is dead only when an Option key is held down and only if it's mapped (in the four **option_..._map** tables) to the second character listed in one of the dead-key arrays.

See also: **GetKeys()** and **Modifiers()** in the BView class, "Keyboard Information" in the chapter introduction, **set_modifier_key()**

Constants and Defined Types

This section lists the various constants and types that the Interface Kit defines to support the work done by its principal classes. The Kit is a framework of cooperating classes; almost all of its programming interface can be found in the class descriptions—particularly the descriptions of member functions—presented in previous sections of this chapter. Most of the constants and types listed here have already been explained above in the class descriptions. Only one or two have not yet been mentioned in full detail. All of them are noted here and briefly described. If a more lengthy discussion is to be found under a class or a member function, you'll be referred to that location.

Constants are listed first, followed by defined types. Constants that are defined as part of an enumeration type are presented with the other constants, rather than with the type. They're listed in the "Constants" section under the type name.

Constants

alignment Constants

<interface/InterfaceDefs.h>

Enumerated constant

ALIGN_LEFT
ALIGN_RIGHT
ALIGN_CENTER

These constants define the **alignment** data type. They determine how lines of text are aligned by BTextView and BStringView objects.

See also: **SetAlignment()** in the BTextView class

Character Constants

<interface/InterfaceDefs.h>

Enumerated constant	Character value
BACKSPACE	0x08
ENTER	0x0a
RETURN	0x0a (same as **ENTER** or '\n')
SPACE	0x20 (same as ' ')
TAB	0x09 (same as '\t')
ESCAPE	0x1b
LEFT_ARROW	0x1c
RIGHT_ARROW	0x1d
UP_ARROW	0x1e
DOWN_ARROW	0x1f
INSERT	0x05
DELETE	0x7f
HOME	0x01
END	0x04
PAGE_UP	0x0b
PAGE_DOWN	0x0c
FUNCTION_KEY	0x10

These constants stand for the ASCII characters they name. Constants are defined only for characters that normally don't have visible symbols.

See also: "Function Key Constants" below

color_space Constants

<interface/InterfaceDefs.h>

Enumerated constant	Meaning
MONOCHROME_1_BIT	One bit per pixel, where 1 is black and 0 is white.
GRAYSCALE_8_BIT	256 gray values, where 255 is black and 0 is white.
COLOR_8_BIT	Colors specified as 8-bit indices into the color map.
RGB_24_BIT	Colors as 8-bit red, green, and blue components.

These constants define the **color_space** data type. A color space describes two properties of bitmap images:

- How many bits of information there are per pixel (the depth of the image), and
- How those bits are to be interpreted (whether as colors or on a grayscale, what the color components are, and so on).

See the “Colors” section in the chapter introduction for a fuller explanation of the four different color spaces currently defined for this type.

See also: “Colors” on page 24, the BBitmap class

Control States

<interface/Control.h>

Enumerated constant	Value
CONTROL_ON	1
CONTROL_OFF	0

These constants define the possible states of a typical control device.

See also: **SetValue()** in the BControl class

Cursor Transit Constants

<interface/View.h>

Enumerated constant	Meaning
ENTERED_VIEW	The cursor has just entered a view.
INSIDE_VIEW	The cursor has moved within the view.
EXITED_VIEW	The cursor has left the view

These constants describe the cursor’s transit through a view. Each **MouseMoved()** notification includes one of these constants as an argument, to inform the BView whether the cursor has entered the view, moved while inside the view, or exited the view.

See also: **MouseMoved()** in the BView class

drawing_mode Constants

<interface/InterfaceDefs.h>

Enumerated constant	Enumerated constant
OP_COPY	**OP_ADD**
OP_OVER	**OP_SUBTRACT**
OP_ERASE	**OP_MIN**
OP_INVERT	**OP_MAX**
OP_BLEND	

These constants define the **drawing_mode** data type. The drawing mode is a BView graphics parameter that determines how the image being drawn interacts with the image already in place in the area where it's drawn. The various modes are explained under "Drawing Modes" in the chapter introduction.

See also: "Drawing Modes" on page 27, **SetDrawingMode()** in the BView class

Font Name Length

<interface/InterfaceDefs.h>

Defined constant	Value
FONT_NAME_LENGTH	32

This constant defines the maximum length of a font name. It's used in the definition of the **font_name** type.

See also: **font_name** under "Defined Types" below

Function Key Constants

<interface/InterfaceDefs.h>

Enumerated constant	Enumerated constant
F1_KEY	**F9_KEY**
F2_KEY	**F10_KEY**
F3_KEY	**F11_KEY**
F4_KEY	**F12_KEY**
F5_KEY	**PRINT_KEY** (the "Print Screen" key)
F6_KEY	**SCROLL_KEY** (the "Scroll Lock" key)
F7_KEY	**PAUSE_KEY**
F8_KEY	

These constants stand for the various keys that are mapped to the **FUNCTION_KEY** character. When the **FUNCTION_KEY** character is reported in a key-down event, the application can determine which key produced the character by testing the key code against these constants. (Control-*p* also produces the **FUNCTION_KEY** character.)

See also: "Character Mapping" on page 59 of the introduction to this chapter

Menu Bar Borders

<interface/MenuBar.h>

Enumerated constant	Meaning
BORDER_FRAME	Put a border around the entire frame rectangle.
BORDER_CONTENTS	Put a border around the group of items only.
BORDER_EACH_ITEM	Put a border around each item.

These constants can be passed as an argument to BMenuBar's **SetBorder()** function.

See also: **SetBorder()** in the BMenuBar class

menu_layout Constants

<interface/Menu.h>

Enumerated constant	Meaning
ITEMS_IN_ROW	Menu items are arranged horizontally, in a row.
ITEMS_IN_COLUMN	Menu items are arranged vertically, in a column.
ITEMS_IN_MATRIX	Menu items are arranged in a custom fashion.

These constants define the **menu_layout** data type. They distinguish the ways that items can be arranged in a menu or menu bar—they can be laid out from end to end in a row like a typical menu bar, stacked from top to bottom in a column like a typical menu, or arranged in some custom fashion like a matrix.

See also: the BMenu and BMenuBar constructors

Message Constants for Interface Events

<app/AppDefs.h>

Enumerated constant	Enumerated constant
WINDOW_RESIZED	**KEY_DOWN**
WINDOW_MOVED	**KEY_UP**
WINDOW_ACTIVATED	**MOUSE_DOWN**
QUIT_REQUESTED	**MOUSE_UP**
SCREEN_CHANGED	**MOUSE_MOVED**
VIEW_RESIZED	**MESSAGE_DROPPED**
VIEW_MOVED	**SAVE_REQUESTED**
VALUE_CHANGED	**PANEL_CLOSED**
PULSE	

These constants identify the messages that report interface events. Each constant names a different type of event.

See also: "Interface Events" on page 40 of the introduction to this chapter

Modifier States

<interface/View.h>

Defined constant	Defined constant
SHIFT_KEY	**OPTION_KEY**
LEFT_SHIFT_KEY	**LEFT_OPTION_KEY**
RIGHT_SHIFT_KEY	**RIGHT_OPTION_KEY**
CONTROL_KEY	**COMMAND_KEY**
LEFT_CONTROL_KEY	**LEFT_COMMAND_KEY**
RIGHT_CONTROL_KEY	**RIGHT_COMMAND_KEY**
CAPS_LOCK	**MENU_KEY**
SCROLL_LOCK	
NUM_LOCK	

These constants designate the Shift, Option, Control, Command, and Menu modifier keys and the lock states set by the Caps Lock, Scroll Lock, and Num Lock keys. They're typically used to form a mask that describes the current, or required, modifier states.

For each variety of modifier key, there are constants that distinguish between the keys that appear at the left and right of the keyboard, as well as one that lumps both together. For example, if the user is holding the left Control key down, both **CONTROL_KEY** and **LEFT_CONTROL_KEY** will be set in the mask.

See also: **Modifiers()** in the BView and BWindow classes, **AddShortcut()** in the BWindow class, the BMenu constructor

orientation Constants

<interface/InterfaceDefs.h>

Enumerated constant

HORIZONTAL
VERTICAL

These constants define the **orientation** data type that distinguishes between the vertical and horizontal orientation of graphic objects. It's currently used only to differentiate scroll bars.

See also: the BScrollBar and BScrollView classes

Pattern Constants

<interface/InterfaceDefs.h>

const pattern **solid_front** = { 0xff, 0xff, 0xff, 0xff, 0xff,0xff, 0xff, 0xff }

const pattern **solid_back** = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }

const pattern **mixed_colors** = { 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55 }

These constants name the three standard patterns defined in the Interface Kit:

- **solid_front** is a pattern that consists of the front color only. It's the default pattern for all BView drawing functions that stroke lines and fill shapes.
- **solid_back** is a pattern with only the background color. It's used mainly to erase images (to replace them with the background color).
- **mixed_colors** alternates pixels between the front and background colors in a checkerboard pattern. The result is a halftone midway between the two colors. This pattern can produce fine gradations of color, especially when the front and background colors are set to two colors that are already quite similar.

See also: "Patterns" on page 25 of the chapter introduction, the **pattern** defined type below

Resizing Modes

<interface/View.h>

const long **FOLLOW_LEFT_TOP**
const long **FOLLOW_TOP_RIGHT**
const long **FOLLOW_RIGHT_BOTTOM**
const long **FOLLOW_LEFT_BOTTOM**
const long **FOLLOW_LEFT_TOP_RIGHT**
const long **FOLLOW_LEFT_TOP_BOTTOM**
const long **FOLLOW_TOP_RIGHT_BOTTOM**
const long **FOLLOW_LEFT_RIGHT_BOTTOM**
const long **FOLLOW_ALL**
const long **FOLLOW_NONE**

These constants are used to set the behavior of a view when its parent is resized. They're explained under the BView constructor.

See also: the BView constructor, **SetResizingMode()** in the BView class

track_style Constants

<interface/View.h>

Enumerated constant	Meaning
TRACK_WHOLE_RECT	Drag the whole rectangle around.
TRACK_RECT_CORNER	Drag only the left bottom corner of the rectangle.

These constants define the **track_style** data type. It determines how BView's **BeginRectTracking()** function permits the user to drag (or drag out) a rectangle.

See also: **BeginRectTracking()** in the BView class

Transparency Constants

<interface/InterfaceDefs.h>

const uchar **TRANSPARENT_8_BIT**
const rgb_color **TRANSPARENT_24_BIT**

These constants set transparent pixel values in a bitmap image. **TRANSPARENT_8_BIT** designates a transparent pixel in the **COLOR_8_BIT** color space, and **TRANSPARENT_24_BIT** designates a transparent pixel in the **RGB_24_BIT** color space.

Transparency is explained the "Drawing Modes" section of the chapter introduction. Drawing modes other than **OP_COPY** preserve the destination image where a source bitmap is transparent.

See also: "Drawing Modes" on page 27, the BBitmap class

View Flags

<interface/View.h>

Enumerated constant	Meaning
FULL_UPDATE_ON_RESIZE	Include the entire view in the clipping region.
WILL_DRAW	Allow the BView to draw.
PULSE_NEEDED	Report pulse events to the BView.
FRAME_EVENTS	Report view-resized and view-moved events.

These constants can be combined to form a mask that sets the behavior of a BView object. They're explained in more detail under the class constructor. The mask is passed to the constructor, or to the **SetFlags()** function.

See also: the BView constructor, **SetFlags()** in the BView class

Window Areas

<app/Window.h>

Enumerated constant
UNKNOWN_AREA
TITLE_BAR
CONTENT_AREA
RESIZE_AREA
CLOSE_BOX

These constants designate the various parts of a window. They're used in mouse-moved event messages to report the area where the cursor is located.

See also: **FilterMouseMoved()** in the BWindow class

Window Flags

<interface/Window.h>

const long **NOT_MOVABLE**
const long **NOT_H_RESIZABLE**
const long **NOT_V_RESIZABLE**
const long **NOT_RESIZABLE**
const long **ACCEPTS_FIRST_CLICK**
const long **NOT_CLOSABLE**
const long **NOT_ZOOMABLE**
const long **FLOATS**

These constants set the behavior of a window. They can be combined to form a mask that's passed to the BWindow constructor.

See also: the BWindow constructor

window_type Constants

<interface/Window.h>

Enumerated constant	Meaning
SHADOWED_WINDOW	The window has a title bar and a shadowed border.
TITLED_WINDOW	The window has a title bar.
BORDERED_WINDOW	The window has a border but no title bar.
MODAL_WINDOW	The window is a modal window.
BACKDROP_WINDOW	The window is the backdrop for the whole screen.
QUERY_WINDOW	The window displays the results of a query.

These constants define the **window_type** data type. They describe the various kinds of windows that can be requested from the Application Server. Two of them,

BACKDROP_WINDOW and **QUERY_WINDOW**, are used only by the Browser application. The others can be used by any application when constructing a window.

See also: the BWindow constructor

Defined Types

alignment

<interface/InterfaceDefs.h>

typedef enum {. . .} **alignment**

Alignment constants determine where lines of text are placed in a view.

See also: "**alignment** Constants" above and **SetAlignment()** in the BTextView class

color_map

<interface/InterfaceDefs.h>

```
typedef struct {
        long id;
        rgb_color color_list[256];
        uchar inversion_map[256];
        uchar index_map[32768];
} color_map
```

This structure contains information about the color context provided by the Application Server. There's one and only one color map for all applications connected to the Server. Applications can obtain a pointer to the color map by calling the global **system_colors()** function. See that function for information on the various fields.

See also: **system_colors()** global function

color_space

<interface/InterfaceDefs.h>

typedef enum {. . .} **color_space**

Color space constants determine the depth and interpretation of bitmap images. They're described under "Colors" in the introduction.

See also: "**color_space** Constants" above, "Colors" on page 24, the BBitmap class

drawing_mode

```
<interface/InterfaceDefs.h>

typedef enum {. . .} drawing_mode
```

The drawing mode determines how source and destination images interact. The various modes are explained in the chapter introduction under “Drawing Modes”.

See also: “Drawing Modes” on page 27, “**drawing_mode** Constants” above

edge_info

```
<interface/View.h>

typedef struct {
        short left;
        short right;
} edge_info
```

This structure records information about the location of a character outline within the horizontal space allotted to the character. Edges separate one character from adjacent characters on the left and right. They’re explained under the **GetCharEdges()** function in the BView class.

See also: **GetCharEscapements()** and **GetFontInfo()** in the BView class

font_info

```
<interface/View.h>

typedef struct {
        font_name name;
        short size;
        short shear;
        short rotation;
        short ascent;
        short descent;
        short leading;
} font_info
```

This structure holds information about a BView’s current font. Its fields are explained under the **GetFontInfo()** function in the BView class.

See also: **GetFontInfo()** and **SetFontName()** in the BView class

font_name

<interface/InterfaceDefs.h>

```
typedef char font_name[FONT_NAME_LENGTH + 1]
```

This type defines a string long enough to hold the name of a font—32 characters plus the null terminator.

See also: SetFontName() in the BView class, **get_font_name()** global function

key_info

<interface/View.h>

```
typedef struct {
        ulong char_code;
        ulong key_code;
        ulong modifiers;
        uchar key_states[16];
} key_info
```

This structure is used by BView's **GetKeys()** function to return all known information about what the user is currently doing on the keyboard.

See also: GetKeys() in the BView class, "Keyboard Information" on page 53 of the introduction to this chapter

key_map

```
<interface/InterfaceDefs.h>

typedef struct {
        ulong version;
        ulong caps_key;
        ulong scroll_key;
        ulong num_key;
        ulong left_shift_key;
        ulong right_shift_key;
        ulong left_command_key;
        ulong right_command_key;
        ulong left_control_key;
        ulong right_control_key;
        ulong left_option_key;
        ulong right_option_key;
        ulong menu_key;
        ulong lock_settings;
        ulong control_map[128];
        ulong option_caps_shift_map[128];
        ulong option_caps_map[128];
        ulong option_shift_map[128];
        ulong option_map[128];
        ulong caps_shift_map[128];
        ulong caps_map[128];
        ulong shift_map[128];
        ulong normal_map[128];
        ulong acute_dead_key[32];
        ulong grave_dead_key[32];
        ulong circumflex_dead_key[32];
        ulong dieresis_dead_key[32];
        ulong tilde_dead_key[32];
} key_map
```

This structure maps the physical keys on the keyboard to their functions in the user interface. It holds the tables that assign characters to key codes, set up dead keys, and determine which keys function as modifiers. There's just one key map shared by all applications running on the same machine. It's returned by the **system_key_map()** function.

See also: system_key_map() global function

menu_layout

<interface/Menu.h>

typedef enum {. . .} **menu_layout**

This type distinguishes the various ways that items can arranged in a menu or menu bar.

See also: the BMenu class, "**menu_layout** Constants" above

orientation

<interface/InterfaceDefs.h>

typedef enum {. . .} **orientation**

This type distinguishes between the **VERTICAL** and **HORIZONTAL** orientation of scroll bars.

See also: the BScrollBar and BScrollView classes

pattern

<interface/InterfaceDefs.h>

```
typedef struct {
        uchar data[8]
} pattern
```

A pattern is a arrangement of two colors—the front color and the background color—in an 8-pixel by 8-pixel square. Pixels are specified in rows, with one byte per row and one bit per pixel. Bits marked 1 designate the front color; those marked 0 designate the background color. An example and an illustration are given under "Patterns" on page 25 of the introduction to this chapter.

See also: "Pattern Constants" above, "Patterns" in the chapter introduction

rgb_color

<interface/InterfaceDefs.h>

```
typedef struct {
        uchar red;
        uchar green;
        uchar blue;
        uchar alpha;
} rgb_color
```

This type specifies a full color in the **RGB_24_BIT** color space. Each component, except **alpha**, can have a value ranging from a minimum of 0 to a maximum of 255.

< The **alpha** component, which is designed to specify the coverage of the color (how transparent or opaque it is), is currently ignored. However, an **rgb_color** can be made completely transparent by assigning it the special value, **TRANSPARENT_24_BIT**. >

See also: **SetFrontColor()** and **SetBackColor()** in the BView class

screen_info

<interface/InterfaceDefs.h>

```
typedef struct {
        color_space mode;
        BRect frame;
        void *bits;
        long bytes_per_row;
        long reserved;
} screen_info
```

This structure holds information about a screen. Its fields are explained under the **get_screen_info()** global function.

See also: **get_screen_info()** global function

symbol_set_name

<interface/InterfaceDefs.h>

typedef font_name **symbol_set_name**

This type defines a string long enough to hold the name of a symbol set—32 characters plus the null terminator. The names of symbol sets are subject to the same length constraint as the names fonts, which is why this type can be a redefinition of **font_name**.

See also: **get_symbol_set_name()** global function

track_style

<interface/View.h>

typedef enum {. . .} **track_style**

This type describes the ways that **BeginRectTracking()** in the BView class can permit the user to drag a rectangle.

See also: **BeginRectTracking()** in the BView class, "**track_style** Constants" above

window_type

<interface/Window.h>

typedef enum {. . .} **window_type**

This type describes the various kinds of windows that can be requested from the Application Server.

See also: the BWindow constructor, "**window_type** Constants" above