Blugs Reference Manual

Blugs List Management Engine 1.1

Document release 1.1
November 21, 2001

The list could surely go on, and there is nothing more wonderful than a list, instrument of wondrous hypotyposis.

— Umberto Eco, *The Name of the Rose*

About the silly name “Blugs” (pronounced /ˈbləgz/): it comes from my friend R. Lon Dobson, who created the name “BlugsVipah River” as part of the geography for a fantasy role-playing game. I found the name tremendously funny and eventually applied it to my list engine.

Incidentally, *blugs* (བླུགས།)

means “poured” in Classical Tibetan. I have yet to find an adequate metaphorical link, but it does make for an exotic logo.

The light source editor window in Figure 3 was inspired by the Persistence of Vision Raytracer (POV-Ray) by the POV Team. More information on this impressive software, including source code, can be found at www.povray.org.

Contact Information

Brian ‘Moses’ Hall

moses@blugs.com
moseshall@mac.com
http://www.blugs.com

841 E Copeland Rd
Montgomery, MI 49255
USA

Phone: 517-238-2921

Currently my answering machine message is in Sanskrit. Don’t let it throw you.

What’s with the nickname ‘Moses’ anyway?
It should be an interesting story but it is not. My friend Lon started calling me that in high school. A decade and a half later, it’s still with me.

If you want a copy of this documentation without the annoying “Hey, I changed this section” bars, let me know.

Credits

Lead developer

Brian ‘Moses’ Hall

Additional code and design

Kyle Hammond
Mal Paine
Andreas Petterson

Special thanks to

Peter O’Gorman for bailing me out in the 11th hour with a 1.0b1 ftp mirror
Sebastiano Pilla for a lot of work on Pascal compatibility
Peter Robinson for requesting an MPW version
Charlie Vass for some great inside information
Jordan Zimmerman for the Color MDEF code
and
R. Lon Dobson for the name *Blugs*
(yes, it’s *his* fault)

Contents

Figures

Preface

This chapter provides an overview of the Blugs list management engine. It identifies what Blugs can and cannot do, and introduces some key concepts developers should understand before programming with Blugs.

About Blugs

Blugs is a list management engine. It can be considered a replacement for the Mac OS List Manager and DataBrowser. Blugs is distributed as a set of static libraries that can be compiled into software that requires sophisticated list management.

Blugs Features

- The amount of data that can be stored in a Blugs list is limited only by available memory.
- Many library variants: PowerPC, Carbon, 68K A5 and A4, Pascal and debug, CodeWarrior and MPW. All libraries do extensive parameter checking; the debug libraries additionally report errors and warnings. With Blugs, error conditions are much easier to detect than with the List Manager.
- Blugs only requires 32-bit `GWorld` support (System 7 and later). It uses advanced OS features like the Appearance Manager and the Mac OS 8.5 Control Manager only when available. Blugs gives you nearly a decade of backward compatibility.
- Built-in support for vertical and horizontal title bars.
- Special list format that accommodates cell and title data. Routines to load, flatten, and unflatten to and from this format.
- Up to 65535 rows and columns.
- Inline text editing. Your application can begin an inline edit itself, or allow Blugs to detect a click in editable text, or both.
- Fast built-in sorting and searching. In many cases, sorting is automatic, requiring no intervention from your application. Blugs can draw a sort button to display and modify sort status.
- Blugs calls upon the Appearance Manager to draw needed user interface primitives (bevel buttons) and to set up an appropriate drawing state (when drawing colored or patterned items like cell borders and backgrounds). When Appearance is not available, Blugs' graphics routines mimic Apple's Platinum appearance. Additional Appearance-savvy drawing routines are provided as source code.
- Drag Manager support. Rows and columns can be rearranged by dragging. Drag-selection by marquee is handled automatically.
- No reliance on code resources (like the List Manager's `'LDEF'` resources). All cell content handling is done via application-supplied callbacks. Blugs' content handler architecture is richer and more flexible than the List Manager's `'LDEF'` interface. The Blugs SDK includes a variety of content handlers with source code.
- Support for disclosure (hierarchical) lists.

- Support for title rows that occupy the entire list width, regardless of how many columns exist. Title rows can be used to label sections of a list, and can have disclosure triangles.
- Individual rows and columns are always resizable, either by user manipulation or by the host application; your application can allow or disallow user changes.
- Live scrolling option with or without Appearance.
- Blugs always uses `GWorlds` for smooth drawing and scrolling.
- You can create an Appearance Manager user pane from a list with one function call.
- No reliance on other large static libraries (like the Standard Library). Blugs either calls upon OS services, or does the job itself. This helps reduce software bloat.
- Blugs is ToolsPlus-savvy. (Specifically, Blugs sets the high bit in the `refCon` field of any control it creates.)
- Blugs is theme-savvy. It can accommodate Mac OS 8.5 themes (*real* themes like Gizmo and Paper) that contain patterns as well as colors. No apparent problems with Kaleidoscope, either.
- Blugs is Aqua-savvy.

What's New in Blugs 1.1

Blugs 1.1 fixes several bugs in 1.0:

- In version 1.0 the internal Appearance user pane callback for keyboard events calls `BLKey` with the first two parameters reversed, leading to an incorrect key value being processed.
- In some cases after a call to `BLUpdate` or in other circumstances where Blugs updates the onscreen list, the host window's fore and back colors would not be reset to their original states.
- The `blCantEdit` cell flag affected by `BLSetCellEditable` was being ignored.
- In 1.0 the initial settings passed to `BLEnter` weren't modified correctly for Aqua, so Blugs could create 8-bit `GWorlds` under Aqua, which is a no-no.
- Several small fixes that would be even harder to explain.

Blugs 1.1 adds the following enhancements:

- `BLGetSelect` takes an extra parameter, a value of type `BLGetSelectMethod` that determines whether Blugs looks for a selection in all cells, one row, or one column. `BLGetSelect` now returns an `OSErr` instead of a `Boolean`. `noErr` indicates a selection was found. *Be careful about this!* If you used to call something like `if (BLGetSelect(...))` now you would call `if (BLGetSelect(...) == noErr)`.
- `blDumpGWorldsOnHide` option added to global settings flags.
- User background-drawing procedures now return `OSErr`. A nonzero error code tells Blugs to go ahead and draw its default background, as if the user routine were not installed. Thus you can now pick and choose the cells to which you apply a special background.
- The `BLHitTest` routine allows you to see what part of a list the cursor is over. If you need to support balloon help or contextual menus you will probably use this routine.
- Basic scroll bar widget support added. These widgets behave a lot like titles, but they are simpler because they can't be dragged and do not cause effects like sorting.
- User notification callback support, similar to DataBrowser notifications. Content handlers can begin to issue commands back out to the host app. (This is not fully implemented in 1.1; eventually row expand procs will be mutated into notifications.)
- `BLDrawPlacard` moved from the open source "Appearance extras" file into Blugs, for internal widget support.

When Not To Use Blugs

Because Blugs is a user interface tool, it is designed for "normal" conditions, when memory allocation and QuickDraw use are permitted. As a result, Blugs is not compatible with interrupt-level processing. From the outset you should assume that all Blugs routines can and will allocate, deallocate, relocate, or purge memory blocks, or call routines that may do so. Nor can Blugs be used safely in a faceless background application (FBA). An FBA, or daemon, is an application that does not have a graphical user interface, and thus is not permitted a QuickDraw environment under versions of Mac OS to date. Blugs relies on QuickDraw; this guarantees incompatibility in an FBA. Blugs should also not be used in an MP task; it uses non-MP safe memory routines and freely calls non-reentrant Toolbox functions.

System Requirements

Applications built with the 68K libraries require Color Quickdraw. This essentially means two requirements: a 68020 or higher processor and System 7. The `BLEnter` routine will return an error code if the minimum is not met.

The (classic) PowerPC libraries run on all systems available to PowerPC machines.

The Carbon libraries require CarbonLib 1.0.4 or later under Mac OS 8.1, or CarbonLib 1.0 or later under Mac OS 8.5. (This is because Blugs uses the `BlockZero` routine.)

Blugs Concepts

To use Blugs effectively, you should become familiar with a few new concepts, and understand some important technical terms that will be used throughout this document.

Content Handlers

You will find that the Blugs library is nominally similar in design to the classic List Manager. For Blugs, the **content handler** performs a role similar to that of the List Manager's `'LDEF'` (List DEFinition) code resource. Macintosh system software provides a default `'LDEF'` for drawing text; the developer has the opportunity to create `'LDEF'` resources to customize data display. For Blugs, content handlers provide the code necessary to draw and manage cell contents. Without at least one content handler, Blugs is like a database that cannot store data – it has no content-rendering capabilities at all. (This is a deliberate design choice.) Chapter 2 of this manual is devoted to the Blugs content handler architecture. You should read it if you aim to create your own handlers or modify those provided with the Blugs distribution.

Lists, Tables, and Spreadsheets

We use **list** as a generic term for a Blugs object; a list may have zero or more rows and columns, and thus zero or more cells. Blugs takes the generic list and divides it into two types: the somewhat specialized **table** and the more generic **spreadsheet**.

Each cell in a spreadsheet can have its own content handler. This means that, for example, in a single column there may be a cell that displays text, and another cell that displays a picture.

A table, on the other hand, is both more constrained and more powerful. Each column in a table has a content handler for all its cells. List views under the Mac OS Finder strikingly resemble the table type list: each column displays a particular kind of data. If you use Blugs to simulate a Finder list view, you might use an icon/editable text handler for the file name and icon; you might use a date handler for the creation and modification date columns. The word *table* should also be suggestive of database management: a column in a relational database contains a single type of data. A table can have a different handler for each column, or two or more columns can have the same handler. Because of data uniformity within a column, tables have capabilities not available to spreadsheets. For example, a table can be sorted (with help from a content handler) but a spreadsheet cannot.

Numbering

Blugs departs from the List Manager's convention for row and column numbers. The List Manager uses zero-based numbers; Blugs uses one-based numbers. Under the List Manager, the first row in a list is row number zero. Under Blugs, the first row is row number one; zero is typically used (when appropriate) to refer to titles. With respect to row and column numbering, Blugs is similar to the Metrowerks PowerPlant LTable class and its derivatives. Blugs uses a special data structure, `BLCell`, to refer to cells. Unlike the Mac OS `Cell` and `Point` data structures which use signed 16-bit fields, `BLCell` uses unsigned 16-bit values. This means that your lists (theoretically — we do not recommend this) can have over 65,000 rows and columns instead of the List Manager's 32,000.

This document occasionally refers to the “first,” “last,” or “next” cell (when discussing cell selection, for example). Like the List Manager, Blugs starts in the first row and goes across column by column. Thus a cell with row = 1 and column = 2 comes before a cell with row = 2 and column = 1.

Content Types

Blugs defines a special integer type, `BLContentType` (an unsigned 16-bit integer), to refer to cell content. A content handler is a routine; a **content type** is a number. When your application **registers** a content handler (via the `BLRegisterContentHandler` routine), it associates a content type with a content handler. This association exists on a per-application basis; a handler routine does not inherently possess a content type. You can think of a content type as shorthand for a handler routine, unique to your application. Blugs uses content types to organize its private table of handlers and their attributes. A content type is not the same as a data flavor (like `'TEXT'` and `'snd '`) because a content type's associated handler may (and generally should) be able to import and export more than one flavor of data.

Content type zero is not a valid content type — `BLRegisterContentHandler` will fail if you try to register a routine as type zero. Zero is reserved for the meaning “no handler.”

Host Application

We expect that Blugs will generally be called by an application program. However, you may find Blugs useful if you develop other varieties of code, such as plugs-ins executed by another application. This poses no problem. Be aware, however, that in the course of this documentation the term **host application** is used to refer to the code that calls Blugs. Occasionally the term **host window** is used to refer to an application-owned window in which a Blugs list is created. If you develop a plug-in that uses Blugs, you should understand that the term **host application** still refers to your code and the user interface items that it creates and owns.

Development Environment

Blugs 1.0 is distributed in the form of a vast array of static library flavors. You can use these libraries whether your development environment is CodeWarrior or MPW. The CodeWarrior libraries (which have no dot-suffix) are in Metrowerks' proprietary format and are for use with their products. The MPW libraries, all of which have an '-.o' suffix (e.g. BlugsLibPPC.o) are in XCOFF/MPW object format and can generally be used with both CodeWarrior and MPW. Given a choice, you should favor the MPW libraries, especially BlugsLibPPC.o and BlugsLibCARBON.o. The MPW MrC compiler is legendary for its optimization capabilities. Currently MPW 68K libraries cannot be used in CodeWarrior; attempts to link result in an error message to the effect that A5-relative 32-bit offsets are not supported.

(Let's also remember to yell at Apple a lot and persuade them to Carbonize MPW.)

We assume that your development environment includes a reasonably up-to-date version of Apple's Universal Headers. At time of writing that is version 3.4.1, but version 3.2 and possibly before should work. In order to use `Blugs.h` you will need `Appearance.h`, which is included in recent Universal Headers. For Carbon compatibility, the type `WindowPtr` is eschewed in favor of `WindowRef`. You will need a version of `Windows.h` or `MacWindows.h` that defines `WindowRef` in some manner, whether as a `WindowPtr` or as a Carbonated opaque type. Please keep in mind that backward compatibility with older Universal Interfaces releases is not a priority.

You will need to include AppearanceLib, ControlsLib, and MathLib – as appropriate to the target runtime – if your project does not already include them. ControlsLib gives Blugs access to the Control Manager additions in Mas OS 8.5 enabling proportional scroll indicators; it is only appropriate for PowerPC targets. MathLib supports the small amount of floating-point math used in adjusting scroll bars. You can weak-link ("import weak") the former two libraries.

C and C++ users should use the non-Pascal libraries. If you desire Pascal conventions for whatever reason, include the following line in your project's prefix file:
```
#define BL_PROC pascal
```
Then use the appropriate Pascal library in your project. Note there are no PowerPC Pascal libraries as such – the PowerPC architecture overrides language-specific parameter passing conventions. Pascal users should always use the Pascal libraries for 68K targets.

Blugs relies on a small number of resources that should be included in your project; they are in the file `Blugs.rsrc`. The file just consists of two `'CURS'` resources. If you forget or decide not to include the file, Blugs will run without complaint or mishap but will not display its custom cursors. If you want Blugs to use different cursors, replace the ones provided with your own, making sure to use the same numbers.

In order to use the included resource templates to view `'LiSt'` resources, you will need to use Resorcerer from Mathemæsthetics. Apple's ResEdit does not support the more sophisticated templates that Resorcerer does. (In fact, you should avoid even opening this `'TMPL'` with ResEdit.) For creating `'LiSt'` resources you should use the Rez compiler. The Blugs distribution includes Rez and DeRez definitions in `Blugs.r`.

Globals and A5

Blugs uses a small amount of global data, and must have access to its globals any time you call one of its routines. For PowerPC targets this is never an issue. For 68K non-application targets like plug-ins and code resources, the standard way to allow reference to global data (normally referenced off the A5 register) is to use register A4. This allows the host to use

A5 for its own globals without interference. To use this technique, include the appropriate Blugs 68K A4 library instead of the normal one. You will need to include `A4Stuff.h` and perform some setup at your code's entry point.

Note that there are currently no 68K A4 MPW library flavors. MPW has a different mechanism for setting up a fake A5 world in non-application code. Refer to Chapter 11 of *Building and Managing Programs in MPW* (second edition).

Memory Management

This section addresses some memory management topics relevant to Blug programming.

Content Handler Memory Allocation

The content handlers included in the Blugs distribution use a fairly primitive memory management scheme for storing their cell data: when more than 32 bits of storage are needed, they allocate a handle for storage. This strategy has the advantage of being familiar and simple. It has the disadvantage of possibly resulting in many, many small chunks of memory scattered around the application heap (in pre-OS X environments, anyway). If you find this distressing, you may want to consider a suballocator like `malloc`. The performance hit should be minimal, if you are using the C Standard Library anyway, or targeting OS X. I understand there are (were?) some open-source memory suballocators available on the internet, but I haven't used any of them. You may want to modify the provided content handlers, perhaps declaring global allocator/deallocator callbacks.

Memory Requirements

How much memory does Blugs consume? The table below summarizes. All sizes are logical sizes. Physical sizes may be rounded to the next natural boundary, or whatever black magic goes on in the Bowels of the Memory Manager. All elements are allocated as handles. You will find that the biggest memory hog is likely to be the list `GWorld`.

Globals	1616 bytes
List	500 bytes (+ `GWorld`)
Title Bar	42 bytes
Titles	8 bytes per title
Cells	8 bytes per cell
Rows	24 bytes per row
Columns	48 bytes per column
Widgets	12 bytes per widget
User data	8 bytes per entry

Chapter 1

Blugs API

This chapter details the Blugs API (application programming interface) which your application uses to create and manage scrollable lists. You can use the content handler modules in the Blugs distribution to draw and manage the data you install in your list cells. Later, when you are comfortable with the way Blugs does things, you can explore the content handler architecture presented in Chapter 2. Then you can write content handlers to manage exactly the type of data and display you need.

To get the most from this documentation, you should be familiar with the basic Mac OS human interface toolbox managers — mainly the Appearance Manager, the Window Manager, and the Control Manager. Some familiarity with the List Manager is assumed.

This chapter first discusses the user interface elements that make up a Blugs list. Then it discusses how you can

- initialize and deinitialize Blugs
- create and dispose of lists
- handle user interaction with lists
- set, get, and modify list data
- customize Blugs with optional callbacks

Introduction to Blugs Lists

Blugs, like the Mac OS List Manager, allows you to create rectangular scrolling series of data items. While a simple list might contain a single column and no other interface elements, it is likely that many developers will take advantage of Blugs' built-in functionality to handle full-featured lists. Figure 1 illustrates a simple list. Figure 2 illustrates a full-featured list with its various parts labeled.

Figure 1 **A simple list**

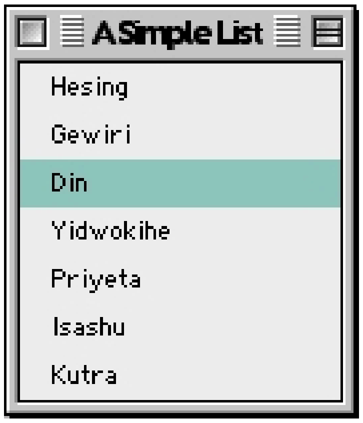
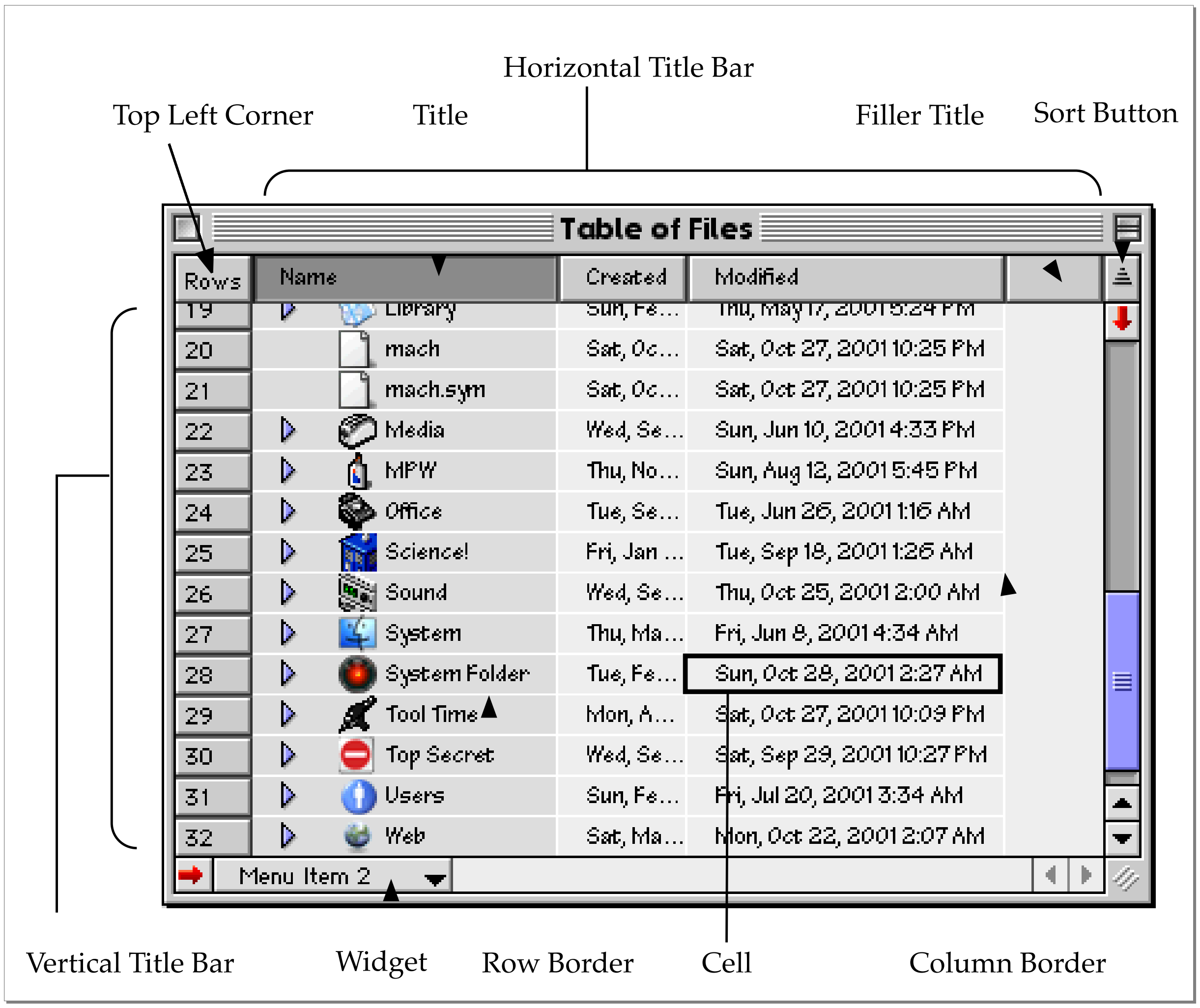


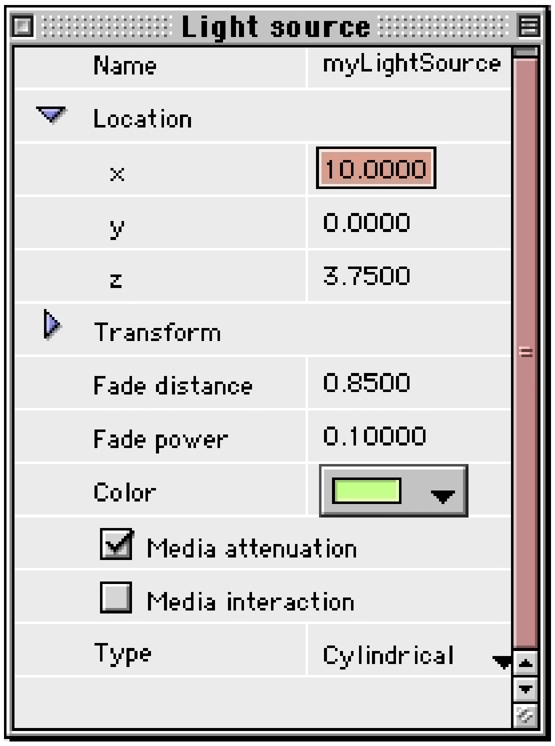
Figure 2 **A more complex list**



Rows	Name	Created	Modified
19	Library	Sun, Fe…	Thu, May 17, 2001 5:24 PM
20	mach	Sat, Oc…	Sat, Oct 27, 2001 10:25 PM
21	mach.sym	Sat, Oc…	Sat, Oct 27, 2001 10:25 PM
22	Media	Wed, Se…	Sun, Jun 10, 2001 4:33 PM
23	MPW	Thu, No…	Sun, Aug 12, 2001 5:45 PM
24	Office	Tue, Se…	Tue, Jun 26, 2001 1:16 AM
25	Science!	Fri, Jan …	Tue, Sep 18, 2001 1:26 AM
26	Sound	Wed, Se…	Thu, Oct 25, 2001 2:00 AM
27	System	Thu, Ma…	Fri, Jun 8, 2001 4:34 AM
28	System Folder	Tue, Fe…	Sun, Oct 28, 2001 2:27 AM
29	Tool Time	Mon, A…	Sat, Oct 27, 2001 10:09 PM
30	Top Secret	Wed, Se…	Sat, Sep 29, 2001 10:27 PM
31	Users	Sun, Fe…	Fri, Jul 20, 2001 3:34 AM
32	Web	Sat, Ma…	Mon, Oct 22, 2001 2:07 AM

Figure 2 also illustrates an appearance typical of a **table**. Note that the cells in any given column display the same kind of data. All cells in the column share a **content type**; for this reason they are all drawn by the same **content handler** callback function. Figure 3 illustrates a complex **spreadsheet**. In this list, content types are assigned on a cell-by-cell basis. This makes it possible for the second column in Figure 3 to contain vastly differing cell contents with differing behaviors. It would be difficult to create this kind of list as a table because it would be difficult write one content handler to draw and manage user interaction with strings, checkboxes, popup buttons, and so forth.

Figure 3 **A complex spreadsheet**



Name	myLightSource
Location	
x	10.0000
y	0.0000
z	3.7500
Transform	
Fade distance	0.8500
Fade power	0.10000
Color	🟩
☑ Media attenuation	
☐ Media interaction	
Type	Cylindrical

Note
The term *spreadsheet*, as can perhaps be appreciated from Figure 3, is not intended to convey the notion of mathematical elements and formulæ arranged in a grid. The term simply distinguishes from the fairly constrained table lists. ◆

List Parts

This section illustrates and describes the various user interface elements that comprise a Blugs list.

Cell

A list is composed of rows and columns. The intersection of a row and a column is a **cell**. The cell is the fundamental unit of data storage and display in both Blugs and the List Manager. Generally, a cell contains and displays a discrete data element of which it in some sense has ownership. Each cell has its own storage area in memory. Your application uses Blugs routines to install and retrieve cell data.

Scroll Bars

Like the List Manager, Blugs can create and manage user interaction with a horizontal scroll bar and/or a vertical scroll bar. You do not need to use the Control Manager to respond to events in these scroll bars; they are private to Blugs. You need only specify that scroll bars are to be created. You can specify optional features such as live scrolling.

Title Bars

A list can optionally contain a horizontal and/or a vertical **title bar**. A title bar is divided into **titles** along row and column boundaries. Thus, the first title in the horizontal title bar is exactly as wide as the first column of cells. A title bar always contains as many titles as there are rows or columns it labels. A list with no rows has no titles in its vertical title bar. The two most salient functional differences between cells and titles are: first, titles only move when the list is scrolled along the title bar axis (that is, the horizontal title bar only moves when the list is scrolled right or left); second, titles can produce effects (such as resizing and sorting) that cells cannot.

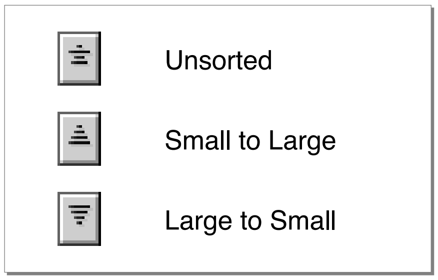
Top Left Corner

When both vertical and horizontal title bars exist, their intersection is called the **top left corner**. Rather than waste the rectangle occupied by it, Blugs lets you install and display data there like you would in a title or cell.

Sort Button

A **sort button** can be displayed above the vertical scroll bar. The sort button displays whether or not the list is sorted, and if it is, shows the direction of sorting. When a sortable list is created, the sort button is automatically drawn in an unsorted state. When the list is initially sorted (in response to the user clicking on the sort button or selecting a column title) the sort button changes to the small-to-large state. Subsequent clicks on the sort button toggle between small-to-large and large-to-small states. The small-to-large state implies smaller numbers come first, and also implies alphabetical order. Figure 4 illustrates the three possible sort button states.

Figure 4 **Sort button states**



Under Aqua, the sort button becomes obsolete. Carbon libraries detect the presence of Aqua and use Theme list header buttons instead of bevel buttons. List header buttons indicate sort direction with an arrow inside the button; you no longer need an external control to show and set sort direction. Under Aqua, Blugs draws the sort button as an empty and inert list header button.

Grow/No-grow Box

When a list can be resized by dragging, you can have Blugs draw a grow box in the lower right corner of the list. Blugs can accommodate the box and draw it if your application is running under a pre-Appearance system, or allow an Appearance-savvy window definition draw the box as part of the window frame. Blugs can also draw a grow box if the list does not occupy the entire window. Two feature flags you use when creating a list control how Blugs manages the grow box. (See the “List Flags” enumeration on page 20.) If the `blHasGrow` flag is set, Blugs makes sure scroll bars leave room for a grow box. If the `blDrawGrow` flag is set in addition, Blugs will draw a grow box in the lower-right corner of the list (either using Appearance 1.1 or its own code). If only `blDrawGrow` is set, Blugs draws a no-grow box, but only if it is needed — if both horizontal and vertical scroll bars are present — to fill in their intersection.

Scroll Bar Widgets

Placards placed in line with a scroll bar are a fairly common element in more complex user interfaces. Blugs supports these elements, called **widgets**. They can have content handlers, and respond to and report user interaction.

Row and Column Borders

Most lists will display row and column borders to indicate cell boundaries. Such a border is a one-pixel line drawn at the right or bottom of a cell. Blugs draws these borders only when the list is active. When Appearance is available they are drawn with the Theme Brush `kThemeBrushListViewSeparator`. Otherwise they are drawn in white. You can also install a callback if you want to draw them yourself.

Using Blugs

This section covers the most important tasks involved in list management using Blugs: initialization, list creation, event handling, and data manipulation. It also discusses some secondary issues like customization.

Initialization

Before you can use Blugs, it must be initialized. To initialize, call `BLEnter` and check the `OSErr` result code. If `BLEnter` return a nonzero error code, do not make any further use of Blugs.

There are two reasons initialization may fail: memory shortage and lack of 32-bit `GWorld` support. Since `BLEnter` allocates only a tiny amount of memory, such a failure would indicate your application is critically short on memory. Lack of 32-bit `GWorld` support makes Blugs inappropriate for applications targeted at, for example, Macintosh System 6 or machines with a 68000 processor (like the Mac Plus).

After initialization, prior to list creation, register the content handlers your lists will use. Call `BLRegisterContentHandler` for each handler with a unique nonzero content type you wish to use for the handler, the address of the handler function, and a variation code.

```
enum
{
	kStringContentType = 1, // Can’t use type 0.
	kPictureContentType
};

OSErr MyInitializeBlugs( void )
{
      OSErr       err;

      err = BLEnter( 0 );
      if (err) return err;
      err = BLRegisterContentHandler( kStringContentType, 0,
                                     &StringContentHandler );
      if (err) return err;
      err = BLRegisterContentHandler( kPictureContentType,
                                     0, &PictureContentHandler );
      return err;
}
```

When you are finished with Blugs, call `BLExit` to deinitialize. If your application is quitting, this is unnecessary.

Creating a List

There are two ways to create a list using Blugs. You can create an empty list based on the parameters you pass to `BLNew`, a function similar to the List Manager's `LNew`. Alternatively, you can load a list from a resource by calling `BLLoad`. The `'LiSt'` resource format allows you to embed cell and title data, so you can also populate your list depending on how much data you wish to include in the resource.

The example code shows a list being created on the fly, and another list being created from a resource.

```
#define kListResourceNumber    128
#define kListFlags             (blVerticalScroll | blLiveScroll |\
                               blAutodraw | blVisible | blActive |\
                               blDrawRowBorders | blBorderMetrics)
#define kDragFlags             0 // No Drag and Drop.

OSErr SetUpLists( void )
{
      BlugsRef    list, listFromResource;
      OSErr       err;
      Rect        listRect = {3,3,163,103};
      Point       cellSize = {20,100};

      listFromResource = BLLoad( kListResourceNumber, gWindow );
      // Most likely reason for failure is memory shortage.
      if (!listFromResource) return memFullErr;
      SetWRefCon( gWindow, (UInt32)listFromResource );
      // Now create a list by means of parameters.
      // There will not be any data in the cells.
      list = BLNew( 1, 8, &listRect, cellSize, gAnotherWindow,
                  kListFlags, kDragFlags );
      SetWRefCon( gAnotherWindow, (UInt32)list );
      return (list == nil) ? memFullErr : noErr;
}
```

Handling List Events

To allow Blugs to function as intended, your application needs to send it three types of events: mouse events, keyboard events, and idle events.

Handling Mouse Interaction

To allow Blugs to process a mouse click in a list, call `BLClick`. This is much the same as the List Manager's `LClick` routine. Both handle interaction as long as the mouse button is pressed. This section describes in detail how `BLClick` works and how you can customize it.

Clicking in Cells

Clicking in a cell generally results in cell selection or deselection. How this comes about often depends on the content handler associated with the cell. By default, a click anywhere in a cell's rectangle causes a selection effect, but the content handler may modify this behavior by defining a **cell region**. This is a region calculated by the content handler that Blugs uses for hit-testing and hiliting. If the handler does not define a cell region, Blugs uses the entire cell rectangle as the cell region. When the user clicks in the cell region, the appropriate selection effect is applied.

In addition to defining a cell region, a content handler can request that clicks be sent to it. This feature is used when the handler draws a control-like element, like a checkbox, and needs to run its own mouse-tracking loop. This means that a handler can override default behavior; you should keep this in mind if you use this kind of content handler.

Like the List Manager, Blugs takes into account the state of the modifier keys when handling cell selection. By default the shift key enables selection of cell ranges, and the command key allows discontiguous selections. When both the shift and command keys are pressed, the shift key is ignored. (Arrow-key selection treats the shift-command combination differently; see the section "Arrow-key Selection" below.)

If a click is within the rectangle normally occupied by cells, but is not in a cell region, Blugs handles marquee selection. The user can drag a dotted marquee rectangle to select cells that intersect it. Marquee selection only happens when the list's flags have the `blOnlyOne` and `blNoDisjoint` bits clear.

Customizing Cell Selection

Like the List Manager, Blugs allows you to customize cell selection behavior by setting bit flags. The List Manager defines seven features it stores in a list's `selFlags` field. Blugs only defines four features; these are set when the list is created and cannot be changed subsequently. (See the section "List Flags" on page 20.) These flags (`blOnlyOne`, `blNoExtend`, `blNoDisjoint` and `blUseSense`) are based on the similarly-named List Manager flags that are deemed most useful. The full set of seven is not implemented in Blugs. (`lNoNilHilite` should be the content handler's responsibility; `lNoRect` and `lExtendDrag` are unnecessary because Blugs implements marquee drag-selection.)

`blOnlyOne` makes it impossible for more than one cell to be selected at a time. It applies to selection by mouse and by arrow key. When a cell is selected, all other cells are deselected automatically.

`blNoExtend` essentially turns the shift key into the command key. When a cell is shift-clicked, cells that intervene between the clicked cell and another selected cell are not automatically selected. This only applies to selection by mouse, not by arrow key.

`blNoDisjoint`, the opposite of `blNoExtend`, makes it impossible to select discontiguous cells. The combination of `blNoExtend` and `blNoDisjoint` is currently undefined.

`blUseSense` only applies when the shift key is down. A click toggles selection status, where by default shift-clicking an already-selected cell would have no effect.

Below is a summary of list behavior when your application reports a mouse event in a cell.

- If the click is outside the cell region, all cells are deselected.
- If the click is inside the cell region and no modifier keys are pressed, Blugs selects the cell and deselects all others.
- If both the shift and command keys are pressed, Blugs ignores the shift key.
- If the click is inside the cell region and the command key is pressed, Blugs selects the cell.
    - If the list was created with the `blOnlyOne` flag set, all other cells are deselected.
    - Otherwise, if the list was created with the `blNoDisjoint` flag set, Blugs selects all cells between the clicked cell and any other selected cell.
- If the click is inside the cell region and the shift key is pressed, Blugs selects the cell. If there are any selected cells before the clicked cell, all intervening cells between the clicked cell and the last selected cell before it are selected. Otherwise, if any cells following the clicked cell are selected, all intervening cells between the clicked cell and the first selected cell after it are selected.
    - If the list was created with the `blOnlyOne` flag set, the shift key is ignored.

Selectability and Representatives

List cells are generally used to display and allow user interaction with data. Such interaction may involve selection, dragging, copying, and so on. Sometimes it makes sense to only display information and not allow such interaction. When the user can select a cell, the implication is that he or she can further “do something” with it. If nothing further can be done, it may be reasonable to disallow selection. This might be useful when a cell is a subordinate “view” of data contained in a different cell.

When there are multiple columns in a list, you can take selectability a step further. To redirect all selection to a single column you can set that column as the list **representative**. Now the user cannot select any cell that is not in the representative column; moreover, clicking in one selects a cell in the representative column. The Blugs representative implementation mimics the Finder’s list view. When a cell in the “Date Modified” column (for example) is clicked, the Finder selects a cell in another column: the “Name” column. (This is the apparent result – it is unclear whether these are different columns internally. Clearly the List Manager is not being used.) Blugs’ representative mechanism can be useful when multiple columns show information about a single entity.

To modify cell selectability, you can use the `BLSetCellSelectable` routine, or you can alter a column’s `blColumnCantSelect` feature flag. You can use the `BLSetRepresentativeColumn` routine to set a column as the list’s representative.

Clicking in Titles

Titles behave somewhat differently from cells with regard to mouse selection. Whereas by default multiple cells can be selected, titles always have radio-button behavior when they can be selected at all. This means that a maximum of one title in a title bar can be selected.

Titles, like cells, have content handlers, so the same comments about handlers intercepting clicks (see above) apply to titles.

When you create a title bar, you encode behavior using feature flags enumerated in the section "Title Bar Flags" on page 24. Three of these (`blTitlesSelectable`, `blTitlesReorderable` and `blTitlesResizableThickness`) affect mouse interaction. By default titles cannot be selected – they are inert labels – unless the `blTitlesSelectable` flag is set. If the `blTitlesReorderable` flag is set, a title can be dragged to a new location, moving it and an entire row or column of cells in the process. When the `blTitlesResizableThickness` flag is set, clicking and dragging the bottom edge of the horizontal title bar up or down, or dragging the right edge of the vertical title bar left or right, results in resizing the title bar's thickness. The `blTitlesPinToRight` bit affects title resizing: when set the user cannot manually resize the last column/title even if the others can be resized. Instead Blugs tries to keep the last column aligned with the right edge of the list.

Although Blugs ignores modifier keys in its internal handling of title clicks, it passes the state of the modifier key on to the title's content handler if the handler requests mouse events.

Drag and Drop

Blugs implements two strategies for dealing with the Drag Manager. The first is a specialized internal strategy for drag-rearranging titles. The second is a more general cell-dragging strategy. To use either strategy, your application must install a drag tracking handler for each window you want to contain a drag-enabled list, so you can call `BLTrackDrag`.

When Blugs detects a drag in a title whose title bar was created with the `blTitlesReorderable` flag set, it starts a drag. The user can drag the title and its associated row or column to a new location, but the drag is confined to the originating list. The row or column is actually moved in the proccess of dragging, so the user can see immediately what effect this rearranging will have. All of this behavior results from a call to `BLClick`, so there is minimal work you must do to support it: just have a drag tracking handler that can call `BLTrackDrag`.

The second drag strategy applies to cells. This is the more important and common type of dragging behavior that users are likely to expect. It also requires more work on your part because Blugs' behavior is fairly generalized. To fully enable cell dragging, your application must install callbacks in a list. There are drag-releated callbacks to:

- inspect a drag before it is tracked (optional)
- add cell data to a drag (required)
- validate the list drop location (optional)
- respond to an actual drop in a list (required)
- inspect a drag before it is disposed (optional)

The drop-oriented callbacks apply to a drag from anywhere entering a list. The others apply to a drag that begins in a list. The `BLDragDataProc` (which adds cell data to a drag) is required in order to complete creation of a drag from a list. The `BLDropProc` (which responds to a drop in a list) is required to complete a drag into a list without the "drag rejected" zoomback effect. The others are optional.

Note that Blugs does *not* automatically sort a list as a result of dragging into or within it.

Blugs draws an insertion caret similar to the one used in the CodeWarrior IDE file list, to indicate where a drop will occur, and at what disclosure level. Blugs also draws drag hiliting whenever a drag is being tracked in a list; the hiliting is drawn on the inside of the view rectangle.

Important
Blugs does not use the Drag Manger to draw its drag hilite; it does not call any of the APIs for drag hiliting. If your window has other items that can be hilited, you should detect when the drag has entered/left a Blugs list and hide/show your hiliting in response. ◆

Drag and Drop Disclosure Constraints

Blugs implements a fairly complex strategy for constraining drop location when dragging within a disclosure list (that is, when tracking in the originating list). Because rows will most likely move as a result of a drop, Blugs makes sure to draw insertion feedback only for drop targets that will not result in an impossible or confusing rearrangement.

Blugs prevents an insertion (drag to position/disclosure level) under these conditions:

- The insertion is a descendant of any of the rows being dragged. (Can't drag a row into itself.)
- The insertion would result in no net change of position and disclosure level in a single-item drag. (Can't drag an item where it already is.)
- The insertion is at a disclosure level less than that of the upper and lower row. (Can't break hierarchy.)
- The insertion disclosure level is more than one greater than that of the upper row. (Can't skip disclosure levels.)

When the cursor is at one of these illegal positions, the insertion caret disappears (although drag hiliting is still shown) and a drop results in zoomback.

Note
Because disclosure makes drag handling more complex, you can enhance performance by making sure the `blDisclosure` list flag is only set if you need it. ◆

Handling Keyboard Interaction

To allow Blugs to process keyboard input for a list, call `BLKey`. Blugs responds to user keyboard activity in a number of ways. The keyboard may be used to navigate around the list. Input may go into a cell that accepts inline editing. Another issue is keyboard focus: when a list or cell can accept key events, the element should be focused. Blugs lets you manipulate a list's focusing behavior.

Blugs tries to filter out all keys that are not relevant. Blugs does not respond, for example, to the tab key. Nor does it respond to function keys or command-key combinations as these should go to the Menu Manager.

Arrow-key Selection

Blugs automates arrow-key selection when you call `BLKey`. Below is a summary of list behavior when your application reports an arrow-key event. Behavior is determined by the arrow key and any modifier keys that are pressed.

- If no modifier keys are pressed, Blugs tries to select the next cell in the direction of the arrow key.
  - ❑ If there is no next cell, there is no effect.
  - ❑ If there is a next cell, all other cells are deselected.
- If the command key is pressed, Blugs selects the most distant cell in the direction of the arrow key. All other cells are deselected.

    - If the list was created with the `blOnlyOne` flag set, the command key is ignored and the list behaves as if the arrow key was pressed without modifier keys.
- If the shift key is pressed, Blugs extends the current selection in the direction of the arrow key.
    - If the list was created with the `blOnlyOne` flag set, the shift key is ignored and the list behaves as if the arrow key was pressed without modifier keys.
    - If the key is left or up, the first selected cell is extended.
    - If the key is right or down, the last selected cell is extended.
    - If there is no cell in the appropriate direction, there is no effect.
- If both the shift and command keys are pressed, Blugs extends the current selection as far as possible in the direction of the arrow key.
    - If the list was created with the `blOnlyOne` flag set, the shift key is ignored and the list behaves as if only the command key was pressed.
    - If the key is left or up, the first selected cell is extended and all cells between it and the top of the list are selected.
    - If the key is right or down, the last selected cell is extended and all cells between it and the bottom of the list are selected.

Keyboard Navigation

Blugs automates two types of keyboard navigation. One of these involves the secondary navigation keys: home, end, page up, and page down. Additionally, return/enter keys affect inline editing specifically. The other is key string navigation based on typed-in characters. Both behaviors occur automatically when your application calls `BLKey`.

The secondary navigation keys have standard behavior. Home and end keys scroll to the top left and bottom right, respectively. Page up and page down scroll up or down the list height minus a small number of pixels.

Key string navigation is available in sorted (table) lists. Each list maintains a string based on key events that have been directed at it within an interval that is twice the key threshold value (accessed via `LMGetKeyThresh`) with a maximum of 120 ticks. When a new character is added to this string, Blugs tries to find and select the closest matching cell in the current sort column. Blugs then deselects other cells.

Inline Editing

Inline editing has been called the “holy grail” of lists. Although writing a content handler capable of inline editing is not always easy, the payoff is great because it allows users to directly manipulate list data in a familiar way.

When there is an inline edit session in a cell, all valid keyboard activity gets sent on to the cell’s content handler. A handler can even request return and enter keys if it allows multi-line editing fields. Otherwise, return and enter keys end the session.

When a selected cell receives a return or enter key, and the cell’s content handler supports inline editing, an inline session is begun in that cell.

When an inline edit session ends, the list may be sorted automatically as a result. It happens under these conditions: the list is a table that was already sorted over the column in which the inline edit took place, and the handler supports string/data comparisons, and the handler indicates that the cell data changed as a result of the edit session.

Keyboard Focus

Inside Macintosh: More Macintosh Toolbox describes a procedure for indicating keyboard focus on a list. The old way of doing this just involved drawing a 2-pixel box around the list. The Appearance Manager introduced a consistent method for drawing keyboard focus with a hilite color. Blugs draws keyboard focus automatically, using the Appearance Manager if available, using a sort of electric blue hilite color otherwise. A focus box, like a list border, is drawn outside the list rectangle.

If a list is the only item in a window that accepts key activity, you may not need to focus it. If the list's edges extend to the window bounds, focus hiliting will not show up. For this reason, Blugs accepts keyboard events regardless of its internal focused state. This internal state is only used for determining if and when to draw the focus box.

Inline editing and focus interact, since an inline edit session receives keyboard events. If an inline edit session begins in a cell, the list becomes defocused and the cell's text box becomes focused. If the cell is big enough, Blugs may be able to draw standard focus hiliting around the text box (it's up to the content handler). Since an inline edit session can start as a result of keyboard events or after an idle event, you should call `BLGetFocusedPart` after a call to `BLKey` or `BLIdle`. If the return value is `blFocusInlineEditPart` then you should make sure any other controls in your window are defocused. Similarly, you may need to check to see if an inline edit session has ended — if a list's focus reverts to `kControlFocusNoPart` then the edit session is over and you can focus an appropriate interface item.

To determine what part of a list has focus, call `BLGetFocusedPart`. To modify focus, call `BLSetFocusedPart` and pass a focus part code. Valid codes are: `kControlFocusNoPart`, `kControlFocusNextPart`, `kControlFocusPrevPart`, `kControlListBoxPart`. The focus part codes `kControlFocusNextPart` and `kControlFocusPrevPart` are often appropriate for handling tab and shift-tab key events. Here is a summary of the effects of each part code:

- If you pass `kControlFocusNoPart`, the list becomes defocused. If there is an inline edit session, it ends.
- If you pass `kControlFocusNextPart`, the effect depends on whether there is an inline edit session in progress.
    - If there is an inline session, it ends. A new inline session begins in the next editable cell if there is one. Otherwise no part of the list is focused.
    - In searching for the next editable cell, Blugs stops at the end of the list. It does not wrap around to the beginning. Blugs only searches the current list.
    - If there is no inline session, the entire list's focus toggles between on and off.
- If you pass `kControlFocusPrevPart`, the effect depends on whether there is an inline edit session in progress.
    - If there is an inline session, it ends. A new inline session begins in the previous editable cell if there is one. Otherwise no part of the list is focused.
    - In searching for the previous editable cell, Blugs stops at the beginning of the list. It does not wrap down to the end.
    - If there is no inline session, the list's focus toggles.
- If you pass `kControlListBoxPart`, the list becomes focused. If there is an inline edit session, it ends.

Handling Idle Processing

You need to allow Blugs some idle time because there are a number of housekeeping tasks that it must do periodically. Call `BLIdle` for each list in an active window at least once each time through your main event loop.

`BLIdle` calls content handlers that want idle time and then adjusts the cursor if needed. It also starts an inline edit session if there is one pending.

Giving Time to Content Handlers

When initialized, content handlers may report that they need idle time. This idle time may be used for periodic updates or whatever. Every time you call `BLIdle`, Blugs checks the content handler for the cell under the cursor. If that handler wants idle events, Blugs calls it with `blIdleMsg`.

Idle Time and Inline Editing

Giving an active list periodic time is especially critical when inline editing is involved. TextEdit and WASTE need idle messages in order to flash the insertion caret. Whenever you call `BLIdle`, Blugs checks to see if there is an inline edit session. If there is, it calls the handler with `blIdleMsg` to allow the handler to call `TEIdle` or `WEIdle`.

When a list allows starting an inline edit session as a result of clicking on cell text, the session does not begin right away. There is a pause, then a session is begun if no other part of the list has been clicked in the meantime. That means that after a call to `BLClick`, there may be an inline edit pending, but the session cannot begin until the proper number of ticks have elapsed. This condition is checked at the end of `BLIdle`. (This Finder-like behavior gives users a chance to avoid a potentially destructive edit.) So if you don't call `BLIdle`, a potential inline edit session will never have the chance to start.

Manipulating Cell Data

This section discusses how you can install and retrieve cell and title data. It also discusses how you can install additional application-defined data to keep for reference or to identify rows and columns.

Installing Cell Data

When you store a list in a resource, you can store data for cells and titles. When you load the list, cells for which you have provided data will be set to the provided content type and data. Otherwise cells and titles begin life without content types or data.

To set a cell’s content type, call `BLSetCellContentType`. For tables, this call sets all cells in a column to a new content type. Once a cell has a valid content type, you can install data so that its content handler can draw it.

To store data in a cell, call `BLSetCellData`. You pass a buffer containing the data you wish to install, the size of the data, and its flavor. The cell’s content handler typically stores a copy of the data, or data derived from yours. (How this all works depends on the nature of the data and how the handler is written.)

Retrieving Cell Data

To find out what a cell’s content type is, call `BLGetCellContentType`. If the cell has no content handler associated with it, the result will be zero.

To find out about how many kinds of data can be retrieved from a cell, call `BLCountCellFlavors`. To find about one of these flavors, call `BLGetIndFlavorInfo`.

`BLGetIndFlavorInfo` gets the flavor type (a four-character code) and size in bytes for the indexed (1-based) flavor. You can use this returned size to make sure your buffer is large enough should you wish to get the cell data in that flavor.

To retrieve data from a cell, or just to get the data size, call `BLGetCellData`. If you pass the address of a buffer, the buffer size (maximum size the handler can copy), and the data flavor you want, the cell's content handler copies data into your buffer and indicates the number of bytes actually copied. If you pass `nil` for the buffer address, then the handler just returns the data size.

Installing and Retrieving User Data

Many Mac OS data structures, like the `ControlRecord` and `WindowRecord`, give the application some storage space. This field is usually called a `refCon`. Unfortunately, however many `refCon` fields a toolbox structure provides, it's usually one less than the number you want!

Blugs is a little more ambitious; it lets you store 32-bit data items with keys (four-character identifiers). Any number of data items can be stored as long as the keys are unique. You can use this mechanism to store any kind of data you want. Each list has its own storage area (you have to get user data from the same list you stored it in). Blugs keeps entries sorted by key, and uses a binary search algorithm for fast retrieval.

To install user data, call `BLSetUserData` and provide a key and a data item. If there is already data installed for that key, Blugs replaces it. To retrieve user data, call `BLGetUserData` and provide the key.

Row and Column Identifiers

Row and column identifiers are another method of getting application data into a list. You can use a 32-bit row or column identifier for any purpose, but it may be most useful for tracking rows and columns when they move (as a result of sorting or dragging). If your application needs to locate a specific row or column that may have been renumbered, these identifiers can provide position-independent information.

To set an identifier, call `BLSetRowIdentifier` or `BLSetColumnIdentifier`. To retrieve an identifier, call `BLGetRowIdentifier` or `BLGetColumnIdentifier`. Blugs does not enforce any uniqueness constraints on identifiers. (You can set all rows to the same identifier, for example.)

Row and Column Unique Identifiers

The problem with identifying a cell by means of a `BLCell` structure is that it is a *position* (row and column) masquerading as an *entity*. When you look at it from the data's perspective, a cell is just a location which can change if, for example, the list becomes sorted. Exclusively using `BLCell` makes for some serious reentrancy issues. Ideally, a content handler should be able to sort the list and Blugs should be able to recover the current cell even though its coordinates have changed. Unique identifiers – UIDs -- are the first step toward this enhanced functionality.

When Blugs adds a row or column to a list, it assigns a unique 64-bit integer. The first row or column added to a list is given a UID of 0x00000000 00000001 and the numbers increase from there. Blugs does not repeat UIDs; 64 bits pretty much ensures that a server running for months and years at a time, constantly adding and deleting rows, will never cycle through the complete range. I think even taking into account the most wildly optimistic variations on Moore's Law we're assured a thousand year lifespan.

A row or column UID is encoded with the `BLUID` type, an `UnsignedWide`. A `BLCellUID` is a data structure consisting of a row UID and a column UID. To get an identifier, call `BLGetRowUID`, `BLGetColumnUID`, or `BLGetCellUID`. To get the current coordinates of an item for which you have the identifier, call `BLGetRowFromUID`, `BLGetColumnFromUID`, or `BLGetCellFromUID`.

Customizing Blugs Using Callbacks

When you create a list, you have a fairly wide range of choices for behavior. When necessary, you can extend Blugs with your own callback routines to change the user interface or other features. For API-specific details, see the sections "Registering User-Defined Routines" on page 96 and "User-Defined Routines" on page 103.

Customizing Cell Background Drawing

By default, Blugs uses one of two colors (or patterns, when alternative themes are used under Mac OS 8.5 or Kaleidoscope) for drawing a cell's background: one for normal cells (under Platinum and pre-Appearance , a light shade of gray), and another (typically a darker gray) for cells in a sorted column. If you wish to change the appearance of cell backgrounds, you can supply your own background-drawing callback.

Blugs calls background-drawing routines before drawing any other part of a cell. In a very real sense, the background drawing procedure "erases" and begins the process of building a cell's appearance. It may be drawing over garbage pixels, so you should make sure your procedure overwrites the *entire* cell rectangle.

If you want to shade differently depending on the cell's position, a background procedure is the way to go. This is how you can color alternating cells differently, for example. You return an error code from your routine; anything other than `noErr` causes Blugs to apply its normal shading.

Customizing Bevel Button Drawing

Blugs uses bevel buttons for drawing title bar elements (titles, fillers, top left), as the basis for the sort button, and (when the Appearance 1.1 routine `DrawThemeGrowBox` is unavailable) the grow box. (Under Aqua, Blugs uses `kThemeListHeaderButton` in the horizontal title bar.) As a result, Blugs' UI is consistent, both internally and with Apple's Platinum appearance. If, however, you need to bring Blugs in line with specialized interface requirements, you can make radical changes just by supplying a custom bevel button callback.

Your bevel button routine is given the opportunity to pick and choose which specific list parts to draw. (If you wish, you can customize only the horizontal title bar, for example, and let Blugs apply its default drawing routines to everything else.)

Customizing Cell Borders

Cell borders are light-colored 1-pixel lines drawn at the bottom and right of each cell. You can install a callback routine to draw the borders differently. You can apply different effects to different cells.

Customizing Cell Hiliting

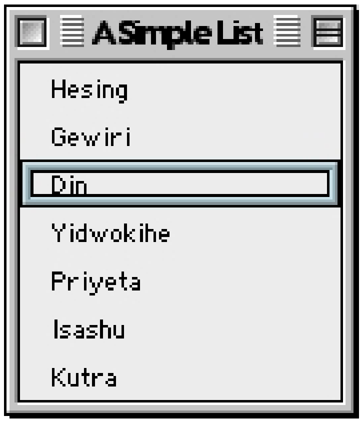
Blugs uses a slightly unusual method for hiliting cells. Traditionally, to hilite something you call `LMSetHiliteMode` and then `EraseRect` to set all pixels matching the background color to the hilite color. However, this is not a good method if the current Appearance Theme uses a background pattern. (You may have seen dire warnings about this in `Appearance.h` and related documents.) The problem is that a background pattern overrides a background color. Calling `LMSetHiliteMode` and `EraseRect` typically sets only a few randomly scattered pixels (if any) to the hilite color and leaves the rest unchanged. So Blugs applies hiliting after the cell background is drawn but before anything else is drawn by calling `LMGetHiliteRGB` and `RGBForeColor`, and then painting in this color. By means of this strange method, hiliting becomes a kind of background upon which cell content is drawn. It can overwrite any Theme background pattern.

When applying hiliting, if the cell's content handler reports a special region for hiliting and hit-testing, Blugs hilites only that region. Otherwise it hilites the whole cell rectangle. If you need to, you can bypass default behavior and install your own cell hiliting callback. It will be drawn after the background but before disclosure triangles, cell content, and cell borders.

Important
A cell hilite procedure completely bypasses the retrieval of any special region. The content handler is never called with the region request. Be aware that cell content can overlap your hiliting because content is drawn later and "on top" of your graphics. ◆

Figure 5 shows a list which has this kind of routine installed. Whether the customization is any improvement over the default is subject to debate.

Figure 5 **Effects of customized hiliting**



Customizing Row Expansion

The `RowExpandProc` callback can only be installed in a disclosure list. It enables you to add child rows on the fly when the user toggles a row's disclosure triangle down. Also, you can delete rows on the fly when a row is collapsed. This may be desirable if you need a large multilevel list and don't want to take an inordinate amount of time setting it up. You can add only the visible rows when creating the list, then display child rows on demand.

▲ **WARNING**
Adding rows on the fly in response to expansion is a tricky business, especially if the user option-clicks the disclosure triangle and thus recursively expands rows. This can tie up the machine for quite a while. If you don't know in advance how many disclosure levels deep the list may become, this might not be your best strategy. (Expanding rows outside of a tight loop would invite more reentrancy problems, however.) ▲

Enabling Secondary Sorting

This is not so much a customization as a way to get Blugs to do something it normally cannot. When Blugs sorts a list, it uses the information in a single column. If you wish Blugs to resolve sorting ambiguities based on another column, supply a `SecondarySortColumnProc` callback that calculates which column to use, based on the primary sort column.

Example

```
// My table has 3 columns; all are sortable.
// Prefer to use column 1 as the secondary column,
//    but if 1 is primary, use 2 as secondary.
pascal UInt16 MySecondarySortColumnProc( UInt16 inPrimaryColumn )
{
      UInt16      secondaryColumn;

      if (inPrimaryColumn == 1) secondaryColumn = 2;
      else secondaryColumn = 1;
      return secondaryColumn;
}
```

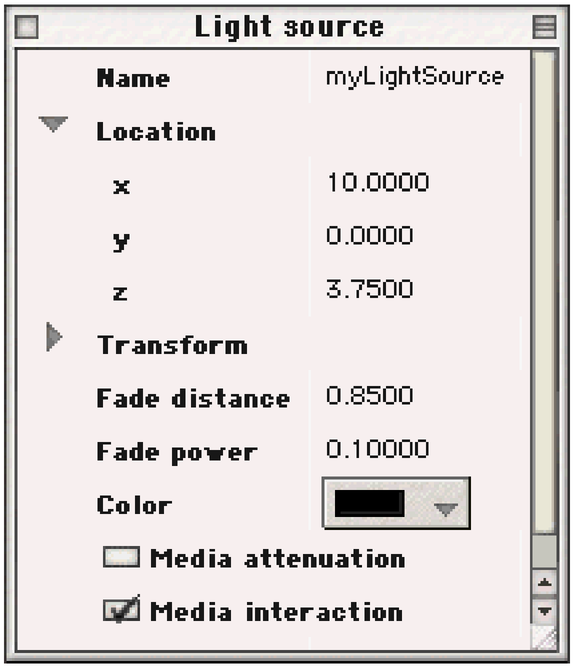
Responding to Notifications

Blugs 1.1 begins to support better reporting of user interation. Following in the footsteps of DataBrowser, you can install a notification callback to get information about user interaction as it happens. Supply a `BLNotificationProc` to a list, and it will be called when events of possible interest occur.

Blugs and Themes

We have tried to make Blugs as compatible with Appearance and Kaleidoscope Themes as possible. This entails calling Appearance Manager routines to draw UI primitives when such routines are available. Figure 6 shows a Blugs list running under a third-party Theme (Paper 2.0.1 by Eric Lob).

Figure 6 **Blugs under a custom Theme**



Name	myLightSource
Location	
x	10.0000
y	0.0000
z	3.7500
Transform	
Fade distance	0.8500
Fade power	0.10000
Color	
Media attenuation	
✓ Media interaction	

Currently, the only problem area is with `DrawThemeButton` disclosure triangles: Blugs is not set up to easily draw the portion of the cell that needs to be erased by the `ThemeEraseProc` callback when the triangle is toggled. When you have a custom background or hilite proc installed, in general you should try to leave the background color or pattern set so that when `DrawThemeButton` calls `EraseRect` (which is apparently unavoidable), your color will be applied in the rectangle surrounding the disclosure triangle.

Patterned Themes can interfere with TextEdit hiliting. Read the section "Appearance Themes and TextEdit" on page 124.

Blugs Reference

This reference section details the portions of Blugs' public interface that are used by applications to manage Blugs lists. For details on the portions of the interface that apply to content handlers, see Chapter 2.

Types and Constants

This section describes the data types and constants provided by Blugs. They are extracted from `Blugs.h`.

Global Settings Flags

When you call `BLEnter` you pass `blNoSettings` or one or more of these OR-combined flags, to control how Blugs allocates `GWorld` memory. You can change this setting with `BLSettings`.

```
enum
{
      blNoSettings                  = 0,
      blTempGWorlds                 = 0x00000001,
      blMax16BitGWorlds             = 0x00000002,
      blMax8BitGWorlds              = 0x00000004,
      blDumpGWorldsOnHide           = 0x00000008
};
```

Constant descriptions

`blNoSettings`	Use none of these flags.
`blTempGWorlds`	Blugs uses temporary memory when creating GWorlds for lists. Ignored under OS X.
`blMax16BitGWorlds`	Use 16 as the depth for calls to `NewGWorld` and `UpdateGWorld`. Overrides `blMax8BitGWorlds`.
`blMax8BitGWorlds`	Use 8 as the depth for calls to `NewGWorld` and `UpdateGWorld`. Under Aqua, if this bit is set, Blugs also sets `blMax16BitGWorlds`. This is because the Appearance Manager apparently cannot correctly execute `DrawThemeButton` in an 8-bit `GWorld`.
`blDumpGWorldsOnHide`	Blugs only allocates a `GWorld` for a list when the list is visible. When it is hidden, the `GWorld` is disposed. The world is allocated when the list is made visible.

Environment Flags

When you call `BLEnter` Blugs calls `Gestalt` a number of times to find out about the Mac OS environment. You can call `BLEnvironment` to retrieve these values.

```
enum
{
      blInitialized               = 1 << 0,
      blHasAppearance             = 1 << 1,
      blHasAqua                   = 1 << 2,
      blHasDragMgr                = 1 << 3,
      blHasTranslucentDrags       = 1 << 4,
      blHasControlMgr2            = 1 << 5
};
```

Constant descriptions

`blInitialized`	Blugs is initialized.
`blHasAppearance`	Appearance Manager is present.
`blHasAqua`	Running under Aqua on OS X.
`blHasDragMgr`	Drag Manager is present.
`blHasTranslucentDrags`	Drag Manager 7.5 is present.
`blHasControlMgr2`	Control Manager 2 is present (Mac OS 8.5 and later).

List Flags

When you create a list by means of `BLNew` you pass an `inListFlags` parameter. You derive this 32-bit number by ORing zero or more of the following constants. (You can add them, but logical OR is safer in case you repeat a constant.) You also use these features when storing a list in a `'LiSt'` resource.

```
enum
{
        blResizableHeight           = 0x00000001,
        blResizableWidth            = 0x00000002,
        blHorizontalScroll          = 0x00000004,
        blVerticalScroll            = 0x00000008,
        blLiveScroll                = 0x00000010,
        blSmallScroll               = 0x00000020,
        blAutodraw                  = 0x00000040,
        blHasGrow                   = 0x00000080,
        blDrawGrow                  = 0x00000100,
        blCanFocus                  = 0x00000200,
        blVisible                   = 0x00000400,
        blActive                    = 0x00000800,
        blInlineEditOnClick         = 0x00001000,
        blDrawColumnBorders         = 0x00002000,
        blDrawRowBorders            = 0x00004000,
        blDisclosure                = 0x00008000,
        blBorderMetrics             = 0x00010000,
        blDrawBorder                = 0x00020000,
        blSortable                  = 0x00040000,
        blDrawSortButton            = 0x00080000,
        blTable                     = 0x00100000,
        blOnlyOne                   = 0x00200000,
        blUseSense                  = 0x00400000,
        blNoExtend                  = 0x00800000,
        blNoDisjoint                = 0x01000000
};
```

Constant descriptions

`blResizableHeight`	The user can resize a row by clicking and dragging the top or bottom of a title in the vertical title bar. By default only your application can change row heights.
`blResizableWidth`	The user can resize a column by clicking and dragging the left or right edge of a title in the horizontal title bar. By default only your application can change column widths.
`blHorizontalScroll`	Blugs creates a horizontal scroll bar in the list.
`blVerticalScroll`	Blugs creates a vertical scroll bar in the list.
`blLiveScroll`	Dragging the scroll bar indicator scrolls the cells and titles. By default Blugs calls the Control Manager to draw a ghosted or outlined indicator and only scrolls when the mouse button is released.
`blSmallScroll`	The list is in a utility window. Blugs uses 11-pixel scroll bars instead of the the more common 16-pixel ones.
`blAutodraw`	Any changes made to the list cause the onscreen representation to be updated. It is useful to *temporarily* disable this feature when you have a number of changes to make and wish to update only once. You can also keep this feature turned off for a list whose host window is not visible.
`blHasGrow`	Blugs makes sure there is room for a grow box under the vertical scroll bar and/or to the right of the horizontal scroll bar.
`blDrawGrow`	Blugs draws a grow or no-grow box. You should set this flag only if the list does not extend to the window frame or if there is no Appearance Manager or Window Manager grow box. If `blHasGrow` is set, Blugs draws a grow box. If `blHasGrow` is clear, Blugs draws a no-grow box.
`blCanFocus`	The list can receive keyboard focus and thus be drawn surrounded by a focus ring.
`blVisible`	The list is drawn onscreen.
`blActive`	Blugs draws list contents in an active state. If this flag is clear, contents are drawn as inactive (dimmed).
`blInlineEditOnClick`	When the user clicks in the text region of an editable cell, Blugs starts an inline edit session after a suitable pause. If this flag is clear, only the host application can start an inline edit session, for example, by calling `BLBeginInlineEdit` (see page 67).
`blDrawColumnBorders`	Blugs draws a light-colored 1-pixel line to the far right of each column.
`blDrawRowBorders`	Blugs draws a separator line as it does for column borders, only at the bottom of each row.
`blDisclosure`	The list allows rows to contain other rows in a hierarchy. Blugs draws a disclosure triangle in any row that contains other rows and uses indentation to show which rows are contained in others.
`blBorderMetrics`	Truncates title bevel buttons by 1 pixel on the left and/or top. This way the scroll bar ends can be overlapped by a list border or window border without an ugly gutter between the border and

	the button. (Blugs draws the button as though it were 1 pixel larger so the dark border is clipped out.) Automatically set if `blDrawBorder` is set.
`blDrawBorder`	Blugs draws a 1-pixel black rectangle (pre-Appearance), or an Appearance Manager list frame, around the list. The list border lies outside the list's bounding rectangle by a maximum of 3 pixels.
`blSortable`	The arrangement of rows can be modified by sorting by cell contents. This feature is only valid if the `blTable` flag (see below) is set.
`blDrawSortButton`	Blugs draws a bevel button with an icon representing the current sort state (unsorted, sorted low-to-high, or sorted high-to-low) in the upper right corner of the list, above the vertical scroll bar. This feature is only valid if the `blSortable` flag (see above) is set and there is a vertical scroll bar. The sort button is always blank and inert under Aqua, and in a non-sortable list.
`blTable`	The list is a table, meaning that each column has a single content type for all cells in that column. Changing a cell's type changes the type of all cells in that column. By default a list is a spreadsheet, in which each cell can have its own content type.
`blOnlyOne`	A maximum of one cell at a time can be selected.
`blUseSense`	Shift-clicking a cell toggles its selection status. By default, shift-clicking a selected cell has no effect.
`blNoExtend`	Shift-clicking does not select intervening cells.
`blNoDisjoint`	Only contiguous ranges of cells can be selected. Ignored if `blOnlyOne` (see above) is set.

Drag Flags

When you call `BLNew` you pass an `inDragFlags` parameter to indicate the list's behavior with respect to the Drag Manager. You derive this number by ORing zero or more of the following constants. You also use these features when storing a list in a `'LiSt'` resource.

```
enum
{
      blCanStartDrags              = 0x0001,
      blAllowDragsOnlyToSelf       = 0x0002,
      blImmediateDrag              = 0x0004,
      blReceiveDrags               = 0x0008,
      blReceiveDragsFromSelf       = 0x0010,
      blReceiveDragsOnlyFromSelf   = 0x0020
};
```

Constant descriptions

`blCanStartDrags`	Cells can be dragged.
`blAllowDragsOnlyToSelf`	Cells can only be dragged within a list. They cannot be dragged outside.
`blImmediateDrag`	Click and drag in a single gesture.
`blReceiveDrags`	The list can receive any valid drag.
`blReceiveDragsFromSelf`	The list can receive drags that originated within it.

`blReceiveDragOnlyFromSelf`	The list cannot receive drags from any source other than itself.

Row Data Flags

These flags are used to encode a row's properties. You set a row's flags when you encode a list in a `'LiSt'` resource and create data entries for individual rows. You can use the `BLSetRowFlags` routine to alter the properties.

```
enum
{
      blRowHasChildren              = 0x0001,
      blRowIsExpanded               = 0x0002,
      blRowIsTitleRow               = 0x0004,
      blRowDrawBorder               = 0x0008,
      blRowMarkedForMovement        = 0x0010
};
```

Constant Descriptions

`blRowHasChildren`	The row has a disclosure triangle and zero or more children. Ignored if the `blDisclosure` list flag is clear.
`blRowIsExpanded`	The disclosure triangle points down. Ignored if the `blRowHasChildren` flag is clear, or if the `blDisclosure` list flag is clear.
`blRowIsTitleRow`	The row consists of a single cell that extends the full list width.
`blRowDrawBorder`	Draw a border at the bottom of this row.
`blRowMarkedForMovement`	Row will be moved in next call to `BLMoveMarkedRows`. Generally you will only use this flag in `BLSetRowFlags`; it doesn't make much sense to use it in a `'LiSt'` resource, although you can if you want.

Column Data Flags

These flags are used to encode a column's properties. You set a column's flags when you encode a list in a `'LiSt'` resource and create data entries for individual columns. You can use the `BLSetColumnFlags` routine to alter the properties.

```
enum
{
      blColumnCantSelect            = 0x0001,
      blColumnDrawBorder            = 0x0002
};
```

Constant Descriptions

`blColumnCantSelect`	The user can't select cells in this column.
`blColumnDrawBorder`	Draw a border to the right of this column.

Title Bar Flags

When you create a title bar with `BLNewTitleBar` you pass an `inFlags` parameter to describe the bar's features and capabilities. You derive this number by ORing zero or more of the following constants. These flags are also used to encode title bar data in a `'LiSt'` resource.

```
enum
{
      blTitlesSelectable              = 0x0001,
      blTitlesReorderable             = 0x0002,
      blTitlesResizableThickness      = 0x0004,
      blTitlesOneContentType          = 0x0008,
      blTitlesPinToRight              = 0x0010
};
```

Constant Descriptions

`blTitlesSelectable`	Title bevel buttons can be clicked and selected by the user. When this feature is set, the bevel buttons have radio button behavior. By default titles cannot be selected.
`blTitlesReorderable`	The titles can be drag-rearranged. When the user drags a title, the row or column is moved to a new location. If this flag is clear, titles cannot be dragged.
`blTitlesResizableThickness`	The user can drag-resize the title in its typically "shorter" dimension, changing its thickness. This means changing the height of the horizontal title bar, and the width of the vertical bar. If this flag is set, the cursor automatically changes to indicate that drag-resizing is possible when the cursor is over the title edge that can be dragged. This flag is ignored for the horizontal title bar under Aqua. The `kThemeListHeaderButton` `ThemeButtonKind` can only be drawn 20 pixels thick.
`blTitlesOneContentType`	For title bars this is the equivalent of the `blTitle` list flag (see above). When set, all titles in the title bar share the same content type. If you change a title's content type, you change the content type for all titles in that bar.
`blTitlesPinToRight`	(Horizontal title bar only) Blugs tries to keep the rightmost column aligned with the right edge of the list (while still respecting column minimum width). User cannot resize the last column.

Widget Flags

These flags define widget behavior.

```
enum
{
      blWidgetInert           = 0x0001,
      blWidgetOn              = 0x0002,
      blWidgetSticky          = 0x0004,
      blWidgetToggles           = 0x0008,
      blWidgetRadioGroup        = 0x0010
};
```

Constant Descriptions

`blWidgetInert`	Widget does not interact with user.
`blWidgetOn`	Widget placard is drawn pressed.
`blWidgetSticky`	Widget stays on when pressed.
`blWidgetToggles`	If the widget is on, pressing it turns it off. Ignored if `blWidgetSticky` is not set.
`blWidgetRadioGroup`	All widgets with this flag set are part of a radio group. Ignored if `blWidgetSticky` is not set.

Part Codes

Certain Blugs routines refer to specific portions of a list. You will probably not need to use all or even most of these constants; we list them here for completeness. (Blugs uses all of them internally.)

```
typedef ControlPartCode         BLPart;
enum
{
      blNoPart                  = 0,
      blCellPart                = 256,
      blCellRegionPart,
      blBottomRightBlankPart,
      blRightBlankPart,
      blBottomBlankPart,
      blHScrollPart,
      blVScrollPart,
      blHTitleBarTitlePart,
      blHTitleBarFillerPart,
      blVTitleBarTitlePart,
      blVTitleBarFillerPart,
      blTopLeftPart,
      blSortButtonPart,
      blGrowBoxPart,
      blInlineEditPart,
      blDisclosureTrianglePart,
      blWidgetPart
};
```

Constant Descriptions

`blNoPart`	No part of the list.
`blCellPart`	Some part of a cell.
`blCellRegionPart`	A cell's content region. This is the content handler-defined region which contains the cell's content; it is used for hiliting and hit testing. A cell may or may not not have a content region.
`blBottomRightBlankPart`	A portion of the view rectangle below and to the right of all cells.
`blRightBlankPart`	A portion of the view rectangle to the right of (but not below) all cells.
`blBottomBlankPart`	A portion of the view rectangle below (but not to the right of) all cells.

`blHScrollPart`	The horizontal scroll bar.
`blVScrollPart`	The vertical scroll bar.
`blHTitleBarTitlePart`	A title in the horizontal title bar.
`blHTitleBarFillerPart`	In the horizontal title bar, but not in any title.
`blVTitleBarTitlePart`	A title in the vertical title bar.
`blVTitleBarFillerPart`	In the vertical title bar, but not in any title.
`blTopLeftPart`	The rectangle where the vertical and horizontal title bars meet.
`blSortButtonPart`	The sort button in the list's upper right corner.
`blGrowBoxPart`	The grow (or no-grow) box drawn below the vertical scroll bar and to the right of the horizontal scroll bar.
`blInlineEditPart`	A cell's inline edit session/rectangle.
`blDisclosureTrianglePart`	A cell's disclosure triangle.
`blWidgetPart`	A placard in line with a scroll bar.

Title Zones

`BLHitTest` reports on a subpart when the hit is in a title bar. These title zones are portions of the title bar very close to the edges, where the user may drag-resize. When the cursor is inside one of these zones, `BLIdle` changes the cursor to indicate drag-resizing is possible.

```
typedef UInt8           BLTitleZone;
enum
{
      blNotInZone,
      blZoneLeft,
      blZoneRight,
      blZoneTop,
      blZoneBottom
};
```

Constant Descriptions

`blNotInZone`	The cursor is not in a zone (not near an edge).
`blZoneLeft`	The area to the left of a title or filler.
`blZoneRight`	The area to the right of a title or filler.
`blZoneTop`	The area to the top of a title or filler.
`blZoneBottom`	The area to the bottom of a title or filler.

Hit Test Record

`BLHitTest` uses this record to specify a part, its coordinates, its rectangle, and in some cases its subparts.

```
typedef struct
{
      BLPart      part;
      BLCell      cell;
      Rect        rect;
      union
      {
            ControlPartCode   controlPart;
            BLTitleZone       zone;
      } u;
};
```

Field Descriptions

`part`	The object hit.
`cell`	The object coordinates.
`rect`	The object bounds.
`u.controlPart`	For scroll bars, the control (sub)part.
`u.zone`	For title bars, the cursor-change zone.

User-Defined Routines

These are the procedure pointer types for the callbacks defined by the Blugs API. Most of them are used in the prototypes for the callback registration functions listed in the section “Registering User-Defined Routines” on page 96.

```
typedef pascal void
(*BLBackgroundProcPtr) (       Boolean inIsSortColumn,
                               BLCell inCell,
                               const Rect* inRect,
                               BlugsRef inList         );

typedef pascal Boolean
(*BLBevelButtonProcPtr)(       BLPart inPart,
                               UInt16 inTitle,
                               const Rect* inRect,
                               ThemeDrawState inState,
                               ThemeButtonValue inValue,
                               BlugsRef inList         );

typedef pascal void
(*BLBorderProcPtr)(            Boolean inDrawRowBorder,
                               Boolean inDrawColumnBorder,
                               BLCell inCell,
                               const Rect* inRect,
                               BlugsRef inList         );

typedef pascal void
(*BLFlattenProcPtr)(           BLCell inCell,
                               OSType* outFlavor,
                               UInt32* ioDataSize,
                               void* outData,
                               BlugsRef inList         );

typedef pascal void
(*BLHiliteProcPtr)(            const Rect* inRect,
                               BlugsRef inList         );

typedef pascal void
(*BLNotificationProcPtr)(      BLNotificationMessage inMessage,
                               BLNotificationCommand inCommand,
                               BLPart inPart,
                               BLCell inCell,
                               BlugsRef inList         );

typedef pascal void
(*BLRowExpandProcPtr)(         Boolean inExpanding,
                               UInt16 inRow,
                              BlugsRef inList         );

typedef pascal UInt16
(*BLSecondarySortColumnProcPtr)( UInt16 inPrimaryColumn,
                              BlugsRef inList          );

typedef pascal OSErr
(*BLPreDragProcPtr)(          DragReference inDragRef,
                              GWorldPtr inDragGWorld,
                              RgnHandle inDragRgn,
                              EventRecord* inEvent,
                              BlugsRef inList         );

typedef pascal OSErr
(*BLDragDataProcPtr)(         DragReference inDragRef,
                              DragItemRef inItem,
                              BLCell inCell,
                              BlugsRef inList         );

typedef pascal OSErr
(*BLDropValidationProcPtr)(   DragReference inDragRef,
                              UInt16 inUnderThisRow,
                              UInt16 inDisclosureLevel,
                              BlugsRef inList         );

typedef pascal void
(*BLDropProcPtr)(             DragReference inDragRef,
                              UInt16 inUnderThisRow,
                              UInt16 inDisclosureLevel,
                              BlugsRef inList         );

typedef pascal void
(*BLPostDragProcPtr)(         DragReference inDragRef,
                              BlugsRef inList         );
```

Type Descriptions

`BLBackgroundProcPtr`	A routine that draws cell backgrounds. See `MyBackgroundProc` on page 103.
`BLBevelButtonProcPtr`	A routine that draws bevel buttons. See `MyBevelButtonProc` on page 103.
`BLBorderProcPtr`	A routine that draws cell borders. See `MyBorderProc` on page 104.
`BLFlattenProcPtr`	A routine that saves cell or title data. See `MyFlattenProc` on page 105.
`BLHiliteProcPtr`	A routine that draws selection hiliting for cells. See `MyHiliteProc` on page 105.
`BLNotificationProcPtr`	A routine that responds to notifications about user activity. See `MyNotificationProc` on page 106.
`BLRowExpandProcPtr`	A routine that inserts or deletes rows on the fly in disclosure lists when a parent row is expanded or collapsed. See `MyRowExpandProc` on page 106.
`BLSecondarySortColumnProcPtr`	A routine that determines a column for secondary sorting based on the primary column. See `MySecondarySortColumnProc` on page 107.
`BLPreDragProcPtr`	A routine that inspects a drag before it begins. See `MyPreDragProc` on page 107.

`BLDragDataProcPtr`	A routine that adds data to a drag. See `MyDragDataProc` on page 108.
`BLDropValidationProcPtr`	A routine that determines whether a drop location in a list is valid. See `MyDropValidationProc` on page 108.
`BLDropProcPtr`	A routine that inserts dropped data in a list. See `MyDropProc` on page 109.
`BLPostDragProcPtr`	A routine that inspects a drag before it ends. See `MyPostDragProc` on page 109.

Callbacks Record

Use this structure to get and set list callback routines *en masse*. Pass a pointer to this record as a parameter to `BLGetCallbacks` and `BLSetCallbacks`. This is the more DataBrowser-like way of manipulating callbacks.

```
typedef struct
{
      BLBackgroundProcPtr                 backgroundProc;
      BLBevelButtonProcPtr                bevelButtonProc;
      BLBorderProcPtr                     borderProc;
      BLHiliteProcPtr                     hiliteProc;
      BLNotificationProcPtr               notificationProc;
      BLRowExpandProcPtr                  rowExpandProc;
      BLSecondarySortColumnProcPtr        secondaryProc;
      BLPreDragProcPtr                    preDragProc;
      BLDragDataProcPtr                   dragDataProc;
      BLDropValidationProcPtr             dropValidationProc;
      BLDropProcPtr                       dropProc;
      BLPostDragProcPtr                   postDragProc;
} BLCallbacksRec, *BLCallbacksPtr;
```

Disclosure Option

When you call `BLAddRows` with a list that can have disclosure triangles (that is, `blDisclosure` list flag is set when the list is created), you can partially specify the disclosure level of the added rows relative to the previous row.

```
typedef UInt8           BLDisclosureOption;
enum
{
      blDisclosureOptionRoot,
      blDisclosureOptionSame,
      blDisclosureOptionChild
};
```

Constant Descriptions

`blDisclosureOptionRoot`	Add rows at the shallowest disclosure level: level zero. The first row of a list is required to be at level zero.
`blDisclosureOptionSame`	Add rows at the same level as the previous row. If there is no previous row, the rows must be at level zero.

`blDisclosureOptionChild`	Add rows as children of the previous row. If there is no previous row, the rows are assigned to level zero.

Sort State

Use these flags when getting and setting a list's sort status.

```
typedef UInt8           BLSortState;
enum
{
      blSortStateUnsorted           = 0,
      blSortStateSorted             = 0x01,
      blSortStateLargeToSmall       = 0x02
};
```

Constant Descriptions

`blSortStateUnsorted`	The list is not (to be) sorted.
`blSortStateSorted`	The list is (to be) sorted.
`blSortStateLargeToSmall`	The list is (to be) sorted in reverse-alphabetical or large-to-small order.

Key Result

`BLKey` returns a result code of type `BLKeyResult` that gives an indication of what happened in the course of processing a keyboard event.

```
typedef UInt16          BLKeyResult;
enum
{
      blNothingHappenedKeyResult,
      blErrorKeyResult,
      blArrowKeyResult,
      blNavigationKeyResult,
      blSearchKeyResult,
      blSentToInlineEditKeyResult,
      blInlineEditBegunKeyResult,
      blInlineEditEndedKeyResult
};
```

Constant Descriptions

`blNothingHappenedKeyResult`	The key could not be processed. For example, most command-character combinations should be handled by the Menu Manager. Blugs ignores them.
`blErrorKeyResult`	An internal error occurred.
`blArrowKeyResult`	An arrow key (with or without modifiers) was processed.
`blNavigationKeyResult`	A navigation key (home, page up, etc.) was processed.
`blSearchKeyResult`	The key was added to the list's internal search string and the list was searched. Applies to sorted tables only.

`blSentToInlineEditKeyResult`	The key was sent to the inline session. (Whether the inline edit handler did anything with the key is not reported.)
`blInlineEditBegunKeyResult`	Return or enter key started an inline session.
`blInlineEditEndedKeyResult`	Return or enter key ended an inline session.

Click Result

`BLClick` returns a result code of type `BLClickResult` that gives an indication of what happened in the course of processing a mouse event. This enumeration will grow in future versions of Blugs.

```
typedef UInt16          BLClickResult;
enum
{
      blNothingHappenedClickResult,
      blCellRgnSingleClickResult,
      blCellRgnDoubleClickResult,
      blCellRgnTripleClickResult,
      blGrowBoxClickResult
};
```

Constant Descriptions

`blNothingHappenedClickResult`	Miscellaneous mouse-down processing.
`blCellRgnSingleClickResult`	A cell was single-clicked.
`blCellRgnDoubleClickResult`	A cell was double-clicked.
`blCellRgnTripleClickResult`	A cell was triple-clicked.
`blGrowBoxClickResult`	The click was in the list's grow box.

Get Select Method

Pass one of these constants to BLGetSelect to indicate where and how you want to search for a selected cell.

```
typedef UInt8           BLGetSelectMethod;
enum
{
      blCellGetSelectMethod,
      blRowGetSelectMethod,
      blColumnGetSelectMethod
};
```

Constant Descriptions

`blCellGetSelectMethod`	Search all cells starting with `ioCell`.
`blRowGetSelectMethod`	Search in `ioCell->row` starting with `ioCell->col`.
`blColumnGetSelectMethod`	Search in `ioCell->col` starting with `ioCell->row`.

Notification Messages

Blugs calls your notification callback, if you have installed one, with a message indicating what happened, and possibly a notification command issued by a content handler.

```
enum
{
      blInlineEditBeganNotificationMsg     = 'InlB',
      blInlineEditEndedNotificationMsg     = 'InlE',
      blWidgetClickedNotificationMsg       = 'WidC'
};
```

Constant Description

`blInlineEditBeganNotificationMsg`	An inline edit session began in the specified cell.
`blInlineEditEndedNotificationMsg`	An inline edit session ended in the specified cell.
`blWidgetClickedNotificationMsg`	Mouse down and up in non-inert widget, and/or widget content handler intercepted the click.

Error and Result Codes

Blugs provides its own error and result codes for those rare situations when there is no Apple-defined constant of an appropriate meaning. Numerical values are in the range Apple reserves for developers (1000-9999 inclusive). The `BLSearch` function returns an `OSErr` result code in the case of a successful or partially successful search. Blugs-defined codes indicate the degree of success.

```
enum
{
      blSearchResultExactMatch             = noErr,
      blSearchResultNextCell               = 1000,
      blSearchResultPrevCell,
};
```

Constant Descriptions

`blSearchResultExactMatch`	Search data was matched exactly.
`blSearchResultNextCell`	Inexact match: `BLSearch` returns the next cell greater than the search data.
`blSearchResultPrevCell`	Inexact match: there is no cell greater than the search data. `BLSearch` returns the previous cell less than the search data.

Blugs Cell

Blugs uses the `BLCell` data type to refer to cells, titles, and the top left corner pseudo-title. It is equivalent to the Mac OS `Point` structure, except that its fields are unsigned integers. When the `row` and `col` fields are nonzero, the structure is interpreted as a cell. If one of the fields contains zero, it is interpreted as a title. If both are zero, it is interpreted as the top left corner. In some cases it is not appropriate for a `BLCell` to refer to titles: selection routines like `BLSelectCell` cannot be used with titles because cells and titles have different selection mechanisms. In such cases input with `row` or `col` containing zero will be rejected; if an error code is returned typically it will be `inputOutOfBounds`.

In some cases `BLCell` is also used to indicate the coordinates of a scroll bar widget. In all such cases (such as in a `BLHitTestRec`) there will be a `BLPart` field or parameter clearly indicating that the coordinates refer to a widget. In the horizontal scroll bar, a widget's coordinates have a row index of zero and a one-based column index counting from the left. In the vertical scroll bar, a widget's coordinates have a column index of zero and a one-based row index counting from the top.

In the future, the preferred method for using `BLCell` will always be in conjunction with a `BLPart` code.

```
typedef struct
{
      UInt16              row;
      UInt16              col;
} BLCell;
```

Field descriptions

`row`	The cell's row number, or zero to indicate the horizontal title bar.
`col`	The cell's column number, or zero to indicate the vertical title bar.

Unique Identifiers

Blugs uses the `BLUID` data type to uniquely refer to rows and columns. The `BLCellUID` type is similar to `BLCell`, except that it uses UIDs to refer to cells. UID values start at `0x00000000 00000001`. The value `0x00000000 00000000` (equivalently, `{0,0}`) is not valid: it refers to a nonexistent row or column.

```
typedef UnsignedWide     BLUID;
typedef struct
{
      BLUID               rowID;
      BLUID               colID;
} BLCellUID;
```

Field descriptions

`rowID`	The cell's row UID.
`colID`	The cell's column UID.

Miscellaneous Types

Listed below are the remaining Blugs types. `BLContentType` is used when registering content handlers. The two opaque reference types prevent the host application from accessing Blugs' internals. `BlugsRef` is used in almost every Blugs routine. `BLTitleBarRef` is used with title bar routines.

```
typedef UInt16                                      BLContentType;
typedef struct OpaqueBLReference*                   BlugsRef;
typedef struct OpaqueBLTitleBarReference*           BLTitleBarRef;
```

Type descriptions

`BLContentType`	A number that corresponds to a content handler routine. You assign this number when you register a content handler with `BLRegisterContentHandler` (see page 96).
`BlugsRef`	An opaque reference to a Blugs list.
`BLTitleBarRef`	An opaque reference to a Blugs title bar.

Blugs Routines

This section describes all routines in the Blugs API.

Initialization

Before you use Blugs you must call `BLEnter` to let Blugs initialize itself. When you are finished with Blugs you can call `BLExit` but you usually don't have to.

BLEnter

Initializes Blugs. You must call this before using any other Blugs routine.

```
OSErr BLEnter( void )
```

Call `BLEnter` before you use any other Blugs routines. This routine makes a number of `Gestalt` checks to evaluate the runtime environment. It checks for and records the availability of the Appearance Manager, Drag Manager, Control Manager, and 32-bit `GWorld` capability. It then allocates a small hash table for storing content handler information.

If 32-bit `GWorlds` are not available, `BLEnter` returns `notInitErr`. If memory is so critically short that it cannot allocate the hash table, it returns `memFullErr`; in this case your application is in serious memory trouble. If `BLEnter` returns an error, you must make no further calls to Blugs.

RESULT CODES

`notInitErr` (-900)	Minimum system requirements not met (needs 32-bit `GWorld` capability). 68K only.
`memFullErr` (-108)	Not enough memory.
`noErr` (0)	No error.

BLExit

Deinitializes Blugs. Call this routine when finished with all Blugs lists.

```
void BLExit( void )
```

Call `BLExit` when your application is finished using Blugs. This routine releases certain blocks of memory `BLEnter` allocated for globals; consequently the amount of free space in your application heap will grow slightly after `BLExit` returns. This routine also unregisters all content handlers. Be sure to call `BLDispose` on all existing lists before calling `BLExit`.

If you need to use Blugs again after calling `BLExit`, you will need to reinitialize it by calling `BLEnter`. Then you must call `BLRegisterContentHandler` for each handler you need to use.

Your application does not need to call `BLExit` if it is in the process of shutting down. When you application terminates, the Mac OS Memory Manager automatically releases the memory occupied by Blugs' globals, because this memory is always in your application's heap. You might want to call `BLExit` if your application is finished using Blugs but will run for a while longer. If you are writing a plug-in that executes in a host application's heap, you may need to call `BLExit` when your plug-in terminates or is deinitialized. (For Photoshop plug-in developers, you might call `BLEnter` when called with `filterSelectorPrepare` and call `BLExit` when called with `filterSelectorFinish`.) Check the host application's plug-in API documentation to determine your best memory management strategy.

Creating and Disposing of Lists

Use the `BLNew` function to create a list based on function parameters. Use `BLLoad` or `BLUnflatten` to create a list based on resource data. Call `BLDispose` to dispose of a list created with those functions. Call `BLFlatten` to save a list to the resource format.

BLNew

Creates a new list in a window.

```
BlugsRef BLNew( UInt16 inColumns, UInt16 inRows,
                    const Rect* inRect, Point inCellSize,
                    WindowRef inWindow, UInt32 inListFlags,
                    UInt16 inDragFlags )
```

`inColumns`	The initial number of columns.
`inRows`	The initial number of rows.
`inRect`	The address of a rectangle, in coordinates local to `inWindow`, within which cells, title bars, and scroll bars are drawn.
`inCellSize`	A Mac OS `Point` whose horizontal component specifies the default column width for this list. The vertical component specifies the default height for rows. When rows and columns are created, they are initialized to these values.
`inWindow`	The window in which the list is to be created.
`inListFlags`	A set of bit values which specify the list's behavior. See the enumeration "List Flags" on page 20 for a detailed description of each flag bit.
`inDragFlags`	A set of bit values which specify the list's behavior as it applies to the Drag Manager. See "Drag Flags" on page 22.

`BLNew` creates a list based on its parameters. If it fails to create the list (if there is not enough free memory in the application heap, for example), it returns `nil`.

If the `inListFlags` bit `blAutodraw` is set, Blugs draws the list just before returning. For this reason, you should not set this bit if you need to modify the list (add rows and columns, for example) before it should be displayed.

If you wish to create a list and populate its cells using one function call, you can load a list from a resource. See the next function, `BLLoad`.

BLLoad

Creates a new list in a window by loading the list's data from a resource of type `'LiSt'`. This routine is especially useful when you know the initial list contents in advance.

```
BlugsRef BLLoad( SInt16 inResID, WindowRef inWindow )
```

`inResID`	The ID of the `'LiSt'` resource that contains the data for this list.
`inWindow`	The window in which the list is to be created.

`BLLoad` creates a list based on information it loads from the specified `'LiSt'` resource. It is a simple wrapper for `GetResource` followed by `BLUnflatten`. Blugs assumes that the resource is in the current resource file. If Blugs fails to create the list (if there is not enough free memory in the application heap, or if the specified resource cannot be found), it returns `nil`.

See the section "The `'LiSt'` Resource" on page 109 , or the file `Blugs.r`, for details on the resource structure.

BLFlatten

Saves a list to a handle.

```
OSErr BLFlatten( BLFlattenProcPtr inFlattenProc, Handle* outHandle,
                 BlugsRef inList )
```

`inFlattenProc`	The address of a routine to save cell and title data, or `nil`.
`outHandle`	On output, a handle to the flattened list.
`inList`	The list to be flattened.

`BLFlatten` creates a handle to a block of memory in the same format as the `'LiSt'` resource. Your `BLFlattenProcPtr` callback is the means by which Blugs extracts cell and title data. You can pass `nil` for `inFlattenProc` if you do not want to save cell or title data.

See the section "The `'LiSt'` Resource" on page 109 , or the file `Blugs.r`, for details on the resource structure. See the section "`MyFlattenProc`" on page 105 for details on the flatten procedure.

RESULT CODES

`errDataSizeMismatch` (-30591)	Flatten proc returned wrong size data (size mismatch between invocations for a given cell or title).
`memFullErr` (-108)	Not enough memory.
`noErr` (0)	No error.

BLUnflatten

Creates a list from a handle in the `'LiSt'` resource format.

```
BlugsRef BLUnflatten( Handle inHandle, WindowRef inWindow )
```

`inHandle`	The data in `'LiSt'` resource format that is to be made into a list.
`inWindow`	The window in which the list is to be created.

`BLUnflatten` creates a list based on `inHandle`, which is assumed to be in the `'LiSt'` resource format. If Blugs fails to create the list (if there is not enough memory), it returns `nil`.

See the section “The `'LiSt'` Resource” on page 109 , or the file `Blugs.r`, for details on the resource structure.

BLDispose

Deallocates all memory associated with a list.

```
void BLDispose( BlugsRef inList )
```

`inList`	The list whose memory is deallocated.

`BLDispose` releases all memory allocated for a list. In the process of doing so it calls the content handler for each cell and title and tells the handler to deinitialize the cell or title. It then calls `DisposeControl` for its scroll bars if they exist. It then releases all memory blocks that are part of the list’s internal structures.

If you have stored any memory handles or pointers in the list by means of the `BLSetUserData` routine, be sure to recover these memory references and either save them elsewhere or deallocate them. If you do not do so your application will suffer from a memory leak. Blugs only deallocates memory it has itself allocated.

BLWindow

Returns a list’s host window.

```
WindowRef BLWindow( BlugsRef inList )
```

`inList`	The list whose host is returned.

`BLWindow` returns the `WindowRef` originally passed to `BLNew`, `BLLoad`, or `BLUnflatten`. This routine may be useful if you need window information, e.g., in a Blugs notification or other type of callback where you are passed a `BlugsRef`.

Rows and Columns

You can use these routines to retrieve and alter the number of rows and columns in your list, to rearrange rows and columns, and to alter row and column properties.

BLAddRows

Adds one or more rows to a list.

```
OSErr BLAddRows( BLDisclosureOption inOption, UInt16 inCount,
                 UInt16 inRow, BlugsRef inList )
```

`inOption`	A constant specifying the disclosure level of the added rows in relation to the previous row. Ignored if the list is not a disclosure list. See the enumeration “Disclosure Option” on page 29.
`inCount`	The number of rows to be added.
`inRow`	The row number of the first added row.
`inList`	The list to which rows are to be added.

This function inserts a number of rows equal to `inCount`, starting at the row whose number is equal to the `inRow` parameter. If `inRow` is more than one greater than the number of rows in the list, `BLAddRows` also adds the intervening rows. (For example, if a list already contains two rows, and `inCount = 1` and `inRow = 4`, `BLAddRows` adds two rows.) If there is already a row at the `inRow` location, it (and any rows numbered higher than it) are shifted to higher row numbers to make room.

Non-disclosure lists ignore the `inOption` parameter. But if the list is a disclosure list, and `inOption` is `blDisclosureOptionChild`, rows are added as children to the previous row. If the previous row does not have any children on entry, Blugs marks it as a parent and subsequently draws it with a disclosure triangle. If `inOption` is `blDisclosureOptionSame`, rows are added at the same disclosure level as the previous one. If there is no previous row, the `inOption` parameter is treated as if it contained `blDisclosureOptionRoot`, and all rows are added at the root level (disclosure level zero).

If you pass zero in the `inCount` parameter, Blugs does nothing.

▲ **WARNING**
Blugs does not act like the List Manager when adding rows to the end of the list, when `inRow` is more than one greater than the last row. The List Manager always honors the `count` parameter, never adding more rows than `count`. Blugs, on the other hand, treats the intervening rows as ‘padding’ and honors `inRow`. To clarify the difference: imagine a list with one row. If you ask the List Manager to add one row at row number 32000, you end up with two rows in the list. If you ask Blugs to do the same thing, you end up with 32000 rows. Blugs cheerfully adds row number 32000 and the intervening 31998 rows. ▲

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; can’t add that many rows.
`nilHandleErr` (-109)	Bad list reference.
`memFullErr` (-108)	Not enough memory to add rows.

`noErr` (0)	No error.

BLAddColumns

Adds one or more columns to a list.

```
OSErr BLAddColumns( UInt16 inCount, UInt16 inColumn,
                            BlugsRef inList )
```

`inCount`	The number of columns to be added.
`inColumn`	The one-based index of the first added column.
`inList`	The list to which columns are added.

This function inserts a number of columns equal to `inCount`, starting at the column whose number is equal to `inColumn`. If the column specified by `inColumn` is more than one greater than the number of columns in the list, `BLAddColumns` also adds the intervening columns. (For example, if a list already contains two columns, and `inCount = 1` and `inColumn = 4`, `BLAddColumns` adds two columns: 3 and 4.)

If there is already a column at the `inColumn` location, it (and any columns numbered higher than it) are shifted to make room for the new columns. If there is a representative column, Blugs updates its number to reflect its new position.

If you pass zero in the `inCount` parameter, Blugs does nothing.

▲ **WARNING**
The comments above, on the difference between `BLAddRows` and the List Manager's `LAddRow`, apply to columns too. ▲

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column zero; can't add that many columns.
`nilHandleErr` (-109)	Bad list reference.
`memFullErr` (-108)	Not enough memory to add columns.
`noErr` (0)	No error.

BLDeleteRows

Deletes one or more rows from a list.

```
OSErr BLDeleteRows( UInt16 inCount, UInt16 inRow, BlugsRef inList )
```

`inCount`	The number of rows to be deleted.
`inRow`	The one-based number of the first deleted row.
`inList`	The list from which rows are to be deleted.

This function deletes a number of rows equal to the `inCount` parameter, starting at `inRow`. `BLDeleteRows` does not try to remove rows that do not exist. If `inCount` is zero, all rows are deleted. In a disclosure list, all descendants of a deleted row are also deleted. As a result, the number of rows actually deleted may be substantially greater than `inCount`. Blugs deletes rows and their descendants until the actual number of rows deleted is greater than or equal to `inCount`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLDeleteColumns

Deletes one or more columns from a list.

```
OSErr BLDeleteColumns( UInt16 inCount, UInt16 inColumn,
                       BlugsRef inList )
```

`inCount`	The number of columns to be deleted.
`inColumn`	The one-based number of the first deleted column.
`inList`	The list from which columns are to be deleted.

This function deletes a number of columns equal to `inCount`, starting at `inColumn`. `BLDeleteColumns` does not try to remove columns that do not exist. If `inCount` is zero, all columns are deleted.

This function may delete the representative column if there is one. If this happens Blugs does not try to introduce a new representative: there are no redirection effects until your application chooses a new representative.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column zero; nonexistent column.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLMoveRows

Moves one or more rows to a different position and/or disclosure level.

```
OSErr BLMoveRows( UInt16 inCount, UInt16 inRow,
                  UInt16 inAfterThisRow, UInt16 inDisclosureLevel,
                  BlugsRef inList )
```

`inCount`	The number of rows to be moved.
`inRow`	The one-based number of the first moved row.

`inAfterThisRow`	The one-based number of the row to be moved after.
`inDisclosureLevel`	The new disclosure level for `inRow`.
`inList`	The list in which rows are moved.

This function moves a number of rows equal to the `inCount` parameter, starting at `inRow`, so `inRow` ends up directly below `inAfterThisRow`. If you can zero for `inAfterThisRow` `inRow` moves to the top of the list. `BLMoveRows` does not try to move rows that do not exist. In a disclosure list, all descendants of a moved row are also moved. As a result, the number of rows actually moved may be substantially greater than `inCount`. Blugs moves rows and their descendants until the actual number of rows moved is greater than or equal to `inCount`. If you pass zero for `inCount`, or you are moving `inRow` directly before, after, or into itself, there is no movement, although the disclosure level may still change.

If `inList` is a disclosure list, `inDisclosureLevel` is taken into account. `inRow` is set to the new disclosure level and all of its descendants are updated. `BLMoveRows` checks `inDisclosureLevel` to make sure it does not violate the hierarchy already established between `inAfterThisRow` and any subsequent rows. For example, if `inAfterThisRow` has a child row, you cannot set `inRow` to the same disclosure level as `inAfterThisRow` because `inRow` would intervene, breaking the hierarchy. In that case the only possible target would be the same level as the child. If the disclosure level violates these hierarchy constraints, `BLMoveRows` does nothing.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row; bad disclosure level.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLMoveMarkedRows

Moves rows that have their `blRowMarkedForMovement` flag set.

```
OSErr BLMoveMarkedRows( UInt16 inAfterThisRow,
                        UInt16 inDisclosureLevel,
                        BlugsRef inList )
```

`inAfterThisRow`	The one-based number of the row to be moved after.
`inDisclosureLevel`	The new disclosure level for each marked row.
`inList`	The list in which rows are moved.

`BLMoveMarkedRows` finds each row that had its `blRowMarkedForMovement` flag bit set by a previous call to `BLSetRowFlags`, moves that row (and its descendants) to the position and disclosure level specified, and unmarks the row. In all other respects this function is the same as `BLMoveRows`. You do not need to mark a row's descendants for movement unless you want them promoted up to `inDisclosureLevel` by this routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row; bad disclosure level.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLMoveColumns

Moves one or more columns to a different position.

```
OSErr BLMoveColumns( UInt16 inCount, UInt16 inColumn,
                  UInt16 inAfterThisColumn, BlugsRef inList )
```

`inCount`	The number of columns to be moved.
`inColumn`	The one-based number of the first moved column.
`inAfterThisColumn`	The one-based number of the column to be moved after.
`inList`	The list in which columns are moved.

This function moves a number of columns equal to the `inCount` parameter, starting at `inColumn`, so `inColumn` ends up directly below `inAfterThisColumn`. If you can zero for `inAfterThisColumn inColumn` moves to the far left of the list. `BLMoveColumns` does not try to move columns that do not exist. If you pass zero for `inCount`, or you are moving `inColumn` directly before, after, or into itself, `BLMoveColumns` does nothing.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column zero; nonexistent column.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLCountRows

Returns the number of rows in a list.

```
UInt16 BLCountRows( BlugsRef inList )
```

`inList`	The list whose rows are counted.

This function returns the number of rows in `inList`. Note that this count includes all rows, even those that are not currently disclosed. This count does not include the horizontal title bar.

BLCountColumns

Returns the number of columns in a list.

```
UInt16 BLCountColumns( BlugsRef inList)
```

`inList`	The list whose columns are counted.

`BLCountColumns` returns the number of columns in `inList`. This count does not include the vertical title bar.

BLGetRowFlags

Returns a row's feature settings.

```
UInt16 BLGetRowFlags( UInt16 inRow, BlugsRef inList )
```

`inRow`	The row whose flags are retrieved.
`inList`	The list which contains the row.

This function returns the feature flags for `inRow`. If you pass a bad list reference, or if `inRow` does not exist, `BLGetRowFlags` returns zero.

BLSetRowFlags

Changes a row's feature flags.

```
OSErr BLSetRowFlags( UInt16 inRow, UInt16 inWhichFlags,
                           UInt16 inFlags, BlugsRef inList )
```

`inRow`	The row whose feature flags are to be set.
`inWhichFlags`	A mask in which flag bits to be changed are set.
`inFlags`	The new set of flags.
`inList`	The list which contains the row.

This function changes the row flags indicated by `inWhichFlags` to the settings in `inFlags`. For each bit in the `inWhichFlags` mask parameter, if the bit is set then the corresponding bit in the `inFlags` parameter is applied to the row. See the enumeration "Row Data Flags" on page 23.

If you alter the value of the `blRowIsExpanded` flag, then Blugs calls `BLExpandRow` or `BLCollapseRow`, with the `inDeepExpand`/`inDeepCollapse` parameter set to `false`. Blugs evaluates the bits from low to high (same as order of enumeration), so the `blRowHasChildren` bit is evaluated before `blRowIsExpanded`. If, by the time the `blRowIsExpanded` bit is evaluated, `blRowHasChildren` is clear, then Blugs makes sure `blRowIsExpanded` is also clear. In non-disclosure lists Blugs keeps clear any disclosure-specific flags.

You cannot clear the `blRowHasChildren` bit as long as the row actually has children (that is, if `inRow+1` has a disclosure level greater than `inRow`). Blugs will not report an error in this case, however. To remove parent status, you must first delete or promote the row's descendants.

Note
This function does not affect undocumented flags. All flag bits not documented here or in the interface files are reserved or used internally. ◆

Note
This mechanism – using mask and flags parameters – is the same as that used by, for example, the Collection Manager. It may save you application an extra call to `BLGetRowFlags`. ◆

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetColumnFlags

Returns a column's feature settings.

```
UInt16 BLGetColumnFlags( UInt16 inColumn, BlugsRef inList )
```

`inColumn`	The column whose flags are retrieved.
`inList`	The list which contains the column.

This function returns the feature flags for `inColumn`. If you pass a bad list reference, or if `inColumn` does not exist, `BLGetColumnFlags` returns zero.

BLSetColumnFlags

Changes a column's feature flags.

```
OSErr BLSetColumnFlags( UInt16 inColumn, UInt16 inWhichFlags,
                        UInt16 inFlags, BlugsRef inList )
```

`inColumn`	The column whose feature flags are to be set.
`inWhichFlags`	A mask in which flag bits to be changed are set.
`inFlags`	The new set of flags.
`inList`	The list which contains the column.

This function changes the column flags indicated by `inWhichFlags` to the settings in `inFlags`. For each bit in the `inWhichFlags` mask parameter, if the bit is set then the corresponding bit in the `inFlags` parameter is applied to the row. See the enumeration "Column Data Flags" on page 23.

If you set the `blColumnCantSelect` flag, then Blugs deselects all cells in the column, provided the flag was not already set.

Note
This function does not affect undocumented flags. All flag bits not documented here or in the interface files are reserved or used internally. ◆

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column zero; nonexistent column.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Metrics

Use these routines to get information about the sizes of list elements, and to change sizes of elements from their initial settings.

BLGetViewRect

Gets the rectangle containing a list's cells.

```
OSErr BLGetViewRect( Rect* outRect, BlugsRef inList )
```

`outRect`	On output, the list's view rectangle.
`inList`	The list whose view rectangle is retrieved.

`BLGetViewRect` returns a rectangle describing the area of the list which contains, or may contain, cells. It is the scrollable area minus title bars. Scroll bars also lie outside the view rectangle.

If you pass a bad list reference, `outRect` contains an empty rectangle on output.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetRect

Gets a list's bounding rectangle.

```
OSErr BLGetRect( Rect* outRect, BlugsRef inList )
```

`outRect`	On output, the list's bounding rectangle.

`inList`	The list whose rectangle is retrieved.

`BLGetRect` returns a rectangle describing the entire area of a list. This rectangle includes scroll bars and title bars. The only visual elements not contained within the rectangle are the focus and list border, if present. Typically neither of these elements extend more than three pixels beyond the edge of the list.

If you pass a bad list reference, `outRect` contains an empty rectangle on output.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetRect

Sets a list's bounding rectangle.

```
OSErr BLSetRect( const Rect* inRect, BlugsRef inList )
```

`inRect`	The address of a rectangle containing the new list bounds, expressed in coordinates local to the list's host window.
`inList`	The list whose bounding rectangle is to be set.

`BLSetRect` changes the rectangle occupied by the list to the one specified in `inRect`. This rectangle encloses all cells, title bars, and scroll bars. The only visual interface elements not contained within this rectangle are the list border and the keyboard focus, if they exist. Both of these elements are drawn outside the list rectangle, as is consistent with other Mac OS interface elements under the Appearance Manager.

This routine changes the rectangle occupied by cells and preserves the respective width and height of the vertical and horizontal title bars. It resizes scroll bars to conform to the new rectangle and redraws them.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLCellRect

Gets a cell or title's bounding rectangle.

```
OSErr BLCellRect( BLCell inCell, Rect* outRect, BlugsRef inList )
```

`inCell`	The cell or title whose bounding rectangle is retrieved.
`outRect`	On output, the cell or title bounds.

`inList`	The list that contains the cell or title.

`BLCellRect` retrieves the bounding rectangle of `inCell`, in local coordinates. If the `row` and `col` fields of `inCell` are nonzero, it is interpreted as a cell. If one of the fields contains zero, it is interpreted as a title. If both are zero, it is interpreted as the top left corner.

If the cell is scrolled so far out of view that it would overflow the 16-bit `Rect` fields, or if `inCell` does not exist, or you pass a bad list reference, `BLCellRect` returns an error code and sets all `outRect` fields to zero.

RESULT CODES

`invalidRect` (-2036)	Cell is scrolled too far out of view: degenerate rectangle.
`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Nonexistent cell or title.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetMinimumSize

Gets a list's smallest legal size.

```
void BLGetMinimumSize( UInt16* outMinWidth, UInt16* outMinHeight,
                       BlugsRef inList )
```

`outMinWidth`	On output, the smallest possible list width.
`outMinHeight`	On output, the smallest possible list height.
`inList`	The list whose minimum size is to be determined.

`BLGetMinimumSize` determines the absolute minimum width and height to which a list can be reduced and still be nominally functional. The starting size is 16 pixels (subject to change in future versions); Blugs then adds the appropriate title bars' thickness and enough space for the scroll bar thumb and scroll buttons to display without overlap. You should call this procedure when calculating the parameters to `GrowWindow` if the window has a list that grows and shrinks with it.

BLSetRowHeight

Sets a row's height.

```
OSErr BLSetRowHeight( UInt16 inRow, UInt16 inHeight,
                      BlugsRef inList )
```

`inRow`	The row to resize.
`inHeight`	The new height in pixels.
`inList`	The list that contains the row.

`BLSetRowHeight` changes the specified row's vertical measure to `inHeight` pixels. Note that in the current version of Blugs, you cannot set the height of a row to zero. If you try to do so, this function does nothing and returns `paramErr`.

If you pass zero in `inRow`, Blugs interprets it as the horizontal title bar. If there is no horizontal title bar, `BLSetRowHeight` returns `inputOutOfBounds`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row does not exist, no horizontal title bar.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-50)	New height was zero.
`noErr` (0)	No error.

BLSetColumnWidth

Sets a column's width.

```
OSErr BLSetColumnWidth( UInt16 inColumn, UInt16 inWidth,
                        BlugsRef inList )
```

`inColumn`	The column to resize.
`inWidth`	The new width in pixels.
`inList`	The list that contains the column.

`BLSetColumnWidth` changes the specified column's horizontal measure to `inWidth` pixels. Note that in the current version of Blugs, you cannot set the width of a column to zero. If you try to do so, this function does nothing and returns `paramErr`.

If you pass zero for `inColumn`, Blugs interprets it as the vertical title bar. If there is no vertical title bar, `BLSetColumnWidth` returns `inputOutOfBounds`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column does not exist, no vertical title bar.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-50)	New width was zero.
`noErr` (0)	No error.

BLSetDefaultCellSize

Sets the size for rows and columns subsequently added to a list.

```
OSErr BLSetDefaultCellSize( Point inSize, BlugsRef inList )
```

`inSize`	The new default size in pixels.
`inList`	The list whose default cell size is set.

`BLSetDefaultCellSize` changes the size for rows and columns subsequently added to the list. If either of the values `inSize.h` or `inSize.v` is zero, Blugs ignores that value. This routine is useful when your application may be deployed on Mac OS X, where typically larger fonts may require additional row height. You can create a list (passing one value to `BLNew`) and then call `BLSetDefaultCellSize` to set another value if necessary, before adding any rows/columns.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetIndent

Gets a list's indent.

```
UInt8 BLGetIndent( BlugsRef inList )
```

`inList`	The list whose indent is retrieved.

`BLGetIndent` returns the pixel value of the current indent setting. When a list is created via `BLNew` or `BLLoad`, indent is initialized to the default value (currently defined as 12 pixels). You can use the following routine, `BLSetIndent`, to change this setting.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetIndent

Sets a list's indent.

```
OSErr BLSetIndent( UInt8 inIndent, BlugsRef inList )
```

`inIndent`	The new indent in pixels.
`inList`	The list whose indent is set.

`BLSetIndent` changes the pixel value of the current indent setting. Although the `inIndent` value is encoded with 8 bits, Blugs enforces a maximum of 100 pixels. Note that this maximum is subject to change in future versions.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Indent greater than maximum.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Events

Call these routines to handle events reported by the Event Manager.

BLClick

Processes a `mouseDown` event in a list.

```
BLClickResult BLClick( UInt32 inWhen, Point inWhereLocal,
                  EventModifiers inModifiers, BlugsRef inList )
```

`inWhen`	The `when` field of the Event Manager event record.
`inWhereLocal`	The location, in coordinates local to the list's host window, of the `mouseDown` event. Call `GlobalToLocal` on (a copy of) the `where` field of the Event Manager event record.
`inModifiers`	The `modifiers` field of the Event Manager event record.
`inList`	The list in which the `mouseDown` event occurred.

Call `BLClick` when the Event Manager reports a `mouseDown` event in a list. `BLClick` handles all mouse interaction with list elements and returns a code indicating how many clicks it has processed in the same cell. See the section "Click Result" on page 31. If the click was not in a cell, `BLClick` returns `blNothingHappenedClickResult`. If the click was in the list's grow box, `BLClick` returns `blGrowBoxClickResult`.

Since this function does not report in great detail whether or how the click was handled, you should only call it after you have ascertained the click was not in another user interface item in your window or dialog. The best way to do this is to call `BLGetRect` on the list and then call `PtInRect` to determine whether the `mouseDown` occurred inside the list bounds.

BLKey

Handles a keyboard event in a list.

```
BLKeyResult BLKey( UInt32 inMessage, UInt32 inWhen,
                  EventModifiers inModifiers, BlugsRef inList )
```

`inMessage`	The `message` field of the Event Manager event record.
`inWhen`	The `when` field of the Event Manager event record.
`inModifiers`	The `modifiers` field of the Event Manager event record.
`inList`	The list in which a key event is to be processed.

Call `BLKey` in response to a `keyDown` or `autoKey` event in a window in which a Blugs list can receive keyboard activity (by being active and visible). `BLKey` returns a value of type

`BLKeyResult` to indicate what happened as a result of the event. See the section "Key Result" on page 30.

If the key represents a display character (alphanumeric, punctuation), Blugs responds as follows: if the list currently has an inline edit session in progress, the key event is sent to the content handler in control of the inline session. If there is no inline session the key is added to the list's internal search string and Blugs searches for the cell with the closest matching string and selects the cell if one is found.

Non-display characters are handled as follows: those that TextEdit should be able to process (such as from delete, arrow keys, page up, etc.) are sent to the inline edit session if there is one, as are return and enter if requested by the handler. Otherwise Blugs modifies cell selection as appropriate for navigation keys. Return and enter end the inline session if there is one. Delete, tab, function keys and other non-display keys are ignored.

Command-key combinations are almost universally ignored by Blugs (since they should go to the Menu Manager); however, command-arrow combinations are valid and are handled appropriately.

For further details, see the section "Handling Keyboard Interaction" on page 10.

BLIdle

Handles idle processing when there is a list in the frontmost window.

```
void BLIdle( Point inWhere, BlugsRef inList )
```

`inWhere`	The current mouse location in local coordinates.
`inList`	The list which is in the active window.

At least once during your main event loop, call `BLIdle` for each list in an active window or dialog.

`BLIdle` first calls the content handler associated with an inline edit session, if there is one. This allows the handler to call on TextEdit or WASTE to flash the insertion caret. It then calls the handler for the cell under the cursor if that handler requests idle messages. It then adjusts the cursor based on its current position relative to titles, inline edit fields, and other elements that may cause cursor changes. The last thing `BLIdle` does is to check if there is an inline edit pending; if there is, it starts the inline edit session if an appropriate delay has passed since the inline edit region was last clicked.

If the list is invisible or inactive, `BLIdle` does nothing.

Cell Selection

Use these routines to control and get information about cell selection in your lists. All the routines in this section are restricted to cell selection; to handle title selection you must use routines from the "Title Bars" section starting on page 83. All selection effects update the list unless autodraw is disabled.

BLSetCellSelectable

Makes it possible or impossible for the user and your application to select a particular cell.

```
OSErr BLSetCellSelectable( BLCell inCell, Boolean inSelectable,
                                 BlugsRef inList )
```

`inCell`	The cell whose selectablitiy is to be set.
`inSelectable`	`true` if the cell can be selected.
`inList`	The list which contains the cell.

`BLSetCellSelectable` sets the selectability of a given cell. If a cell is selectable, the user can select it by means of mouse clicks and keyboard navigation, and your application can select it with routines like `BLSelectCell`. A non-selectable cell cannot be selected by any of these methods.

If you pass `false` in the `inSelectable` parameter, `BLSetCellSelectable` makes sure the cell is deselected before returning.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetRepresentativeColumn

Redirects selection to a single column.

```
OSErr BLSetRepresentativeColumn( UInt16 inColumn, BlugsRef inList )
```

`inColumn`	The column to which selection is redirected.
`inList`	The list which contains the columns.

`BLSetRepresentativeColumn` prevents cells in any column other than `inColumn` from being selected. Instead, Blugs will subsequently select a cell in the same row, but in the column passed as `inColumn`. The resulting behavior can be similar to Finder list views, in which only cells in the filename/icon column can be selected, and clicking in another column selects cells in the former. If you pass zero in the `inColumn` parameter, it clears any previously defined representative, and all columns can be selected as normal.

Routines like `BLDeleteColumns` and `BLAddColumns` that may renumber columns, as well as user-initiated changes like drag-rearranging, update the list's representative. In other words, representative status moves with the column.

`BLSetRepresentativeColumn` does not alter existing selections.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetSelect

Selects or deselects a cell.

```
OSErr BLSetSelect( Boolean inSelect, BLCell inCell,
                   BlugsRef inList )
```

`inSelect`	`true` if the cell is to be selected, `false` if it is to be deselected.
`inCell`	The cell which is to be selected or deselected.
`inList`	The list which contains the cell.

`BLSetSelect` selects or deselects a cell if possible. Blugs redirects selection or deselection effects to a cell in the representative column if there is one.

If `inSelect` is `true`, Blugs selects the cell and scrolls to make it more fully visible if needed. If the cell has been made non-selectable by means of `BLSetCellSelectable` or `BLSetColumnSelectable`, `BLSetSelect` does nothing. If the list allows only one selected cell at a time (that is, if the `blOnlyOne` flag is set), any other selected cells are deselected.

If `inSelect` is `false`, Blugs deselects the cell.

If the cell does not exist or you pass a bad list reference, or the cell's selection status is already the same as `inSelect`, `BLSetSelect` does nothing.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSelectOneCell

Selects a cell and deselects all others.

```
void BLSelectOneCell( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell which is to be selected.
`inList`	The list which contains the cell.

`BLSelectOneCell` selects a cell if possible and deselects all other cells. Blugs redirects selection to a cell in the representative column if there is one. If appropriate, it scrolls to make the cell more fully visible. If the cell has been made non-selectable by means of `BLSetCellSelectable` or `BLSetColumnSelectable`, or does not exist, or you pass a bad list reference, `BLSelectOneCell` does nothing.

BLSelectAll

Selects a cell and deselects all others.

```
OSErr BLSelectAll( BlugsRef inList )
```

`inList`	The list whose cells are selected.

`BLSelectAll` selects all cells in the list, except those which have been made non-selectable by means of `BLSetCellSelectable` or `BLSetColumnSelectable`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLDeselectAll

Deselects all cells.

```
void BLDeselectAll( BlugsRef inList )
```

`inList`	The list whose cells are deselected.

`BLDeselectAll` removes selection from all cells in the list. If you pass a bad list reference, `BLDeselectAll` does nothing.

BLIsCellSelected

Determines if a given cell is selected.

```
Boolean BLIsCellSelected( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell whose selection status is to be tested.
`inList`	The list which contains the cell.

`BLIsCellSelected` returns `true` if the cell is selected and `false` if it is not. It does not take into account any representative column but only reports the status of the cell itself. If the cell does not exist or you pass a bad list reference, `BLIsCellSelected` returns `false`.

BLGetSelect

Searches for the next selected cell.

```
OSErr BLGetSelect( BLGetSelectMethod inMethod, BLCell* ioCell,
                   BlugsRef inList)
```

`inMethod`	A constant indicating where to look for selected cells.
`ioCell`	On input, the cell at which the search begins. On output, the first selected cell that was found.
`inList`	The list to be searched.

`BLGetSelect` searches, starting with the cell pointed to by `ioCell`, for a selected cell. See the enumeration “Get Select Method” on page 31. If `inMethod` is `blCellGetSelectMethod`, Blugs searches all cells. If `inMethod` is `blRowGetSelectMethod`, Blugs only searches `ioCell->row`. If `blColumnGetSelectMethod`, Blugs only searches `ioCell->col`. In all cases Blugs only searches cells starting with `ioCell`.

If Blugs finds a selected cell, it places it in `ioCell` and returns `noErr`. If neither the cell initially passed in nor any subsequent cells are selected, `BLGetSelect` returns `errAENoUserSelection` and the contents of `ioCell` are unchanged.

RESULT CODES

`errAENoUserSelection` (-10013)	No selection found.
`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`ioCell` does not exist.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-50)	Unknown method.
`noErr` (0)	Selection found.

BLHitTest

Identifies the object under the sursor.

```
OSErr BLGetSelect( Point inWhere, BLHitTestPtr outHitTest,
                   BlugsRef inList)
```

`inWhere`	A point in local coordinates.
`outHitTest`	The address of a `BLHitTestRec`.
`inList`	The list which is to be hit tested.

`BLHitTest` finds the Blugs part at the point `inWhere` and returns the type of part, its list coordinates, and its rectangle or (in the case of `blDisclosureTrianglePart`) the rectangle of its containing part. Additionally, if the part hit is `blHTitleBarTitlePart`, `blHTitleBarFillerPart`, `blVTitleBarTitlePart`, or `blVTitleBarFillerPart`, Blugs returns a title zone code in `outHitTest->u.titleZone`. If the part code is `blHScrollPart` or `blVScrollPart`, Blugs returns the scroll bar part (returned by `FindControl` or `FindControlUnderMouse`) in `outHitTest->u.controlPart`.

Note that when the list is in a user pane control, the scroll bars have the list set as their supervisor, so `BLHitTest` will not return `blHScrollPart` or `blVScrollPart` in that case.

See the sections “Hit Test Record” on page 26 and “Title Zones” on page 26.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	Selection found.

Drag and Drop

Use these routines to invoke Blugs’ built in drag-handling capabilities. You must invoke them from a drag tracking handler you have installed in a window or dialog.

BLTrackDrag

Handles a drag’s interaction with a list.

```
OSErr BLTrackDrag( DragTrackingMessage inMessage,
                   DragReference inDragRef, BlugsRef inList )
```

`inMessage`	The drag tracking message received from the Drag Manager.
`inDragRef`	The current drag.
`inList`	The list within which the drag is currently being tracked.

`BLTrackDrag` attempts to handle a drag as it interacts with `inList`. Call this function from a drag tracking handler you have installed in an application window or dialog.

RESULT CODES

`dragNotAcceptedErr` (-1857)	Bad list or drag reference.
Drag Manager errors (-1861 to -1850)	Drag Manager returned error.
`noErr` (0)	No error.

BLReceiveDrag

Handles a drop on a list.

```
OSErr BLReceiveDrag( DragReference inDragRef, BlugsRef inList )
```

`inDragRef`	The current drag.

`inList`	The list within which the drop occurred.

`BLReceiveDrag` attempts to handle a drop on `inList`. Call this function from a drag receiving handler you have installed in an application window or dialog.

RESULT CODES

`dragNotAcceptedErr` (-1857)	Bad list or drag reference.
Drag Manager errors (-1861 to -1850)	Drag Manager returned error.
`noErr` (0)	No error.

BLGetListFromDrag

Retrieves a reference to the list in which a drag began.

```
OSErr BLGetListFromDrag( DragReference inDragRef,
                         BlugsRef* outList )
```

`inDragRef`	The current drag.
`inList`	On output, the list from which the drag originated.

`BLGetListFromDrag` attempts to retrieve the list which was stored as a private data item in the drag. If `inDragRef` did not originate in a Blugs list owned by the current process, `outList` contains `nil` on output and `BLGetListFromDrag` returns `cantGetFlavorErr`.

RESULT CODES

`cantGetFlavorErr` (-1854)	No list stored in drag.
`noErr` (0)	No error.

BLGetCellFromDragItemRef

Retrieves a reference to the list in which a drag began.

```
OSErr BLGetCellFromDragItemRef( DragReference inDragRef,
                                DragItemRef inDragItem,
                                BLCell* outCell )
```

`inDragRef`	The current drag.
`inDragItem`	The item reference number, starting with 1.
`outCell`	On output, the cell that is being dragged.

`BLGetCellFromDragItemRef` attempts to retrieve one of the cells stored as a private data item in the drag. When Blugs starts a cell drag it adds each cell, in a private data flavor, as a `flavorSenderOnly` drag item, with the item reference starting at 1 for the first cell and going sequentially. If `inDragRef` did not originate in a Blugs list owned by the current process, `outCell` is unchanged on output and `BLGetCellFromDragItemRef` returns `cantGetFlavorErr`.

RESULT CODES

`cantGetFlavorErr` (-1854)	No cell stored in drag item.
`noErr` (0)	No error.

List Display

Use these routines to control the display of your lists.

BLSetAutodraw

Enables or disables automatic updating of a list.

```
OSErr BLSetAutodraw( Boolean inDraw, BlugsRef inList )
```

`inDraw`	`true` if Blugs is to automatically redraw the list.
`inList`	The list whose autodraw status is to be set.

When you create a list, you set autodraw either on or off (see the `blAutodraw` list flag on page 20). You enable or disable this automatic onscreen updating using `BLSetAutodraw`. If your application needs to make a number of changes such as adding a number of rows or columns, you can improve performance by temporarily disabling autodraw, then re-enabling it when all changes have been made. As a result of this strategy the screen is only updated once, when all changes have been made. Also, Blugs avoids updating its offscreen buffer while autodraw is suspended.

If the list's autodraw status is already set as indicated by `inDraw`, `BLSetAutodraw` does nothing. If `inDraw` is `true`, both the offscreen and onscreen images are automatically updated. You do not need to call `BLUpdate` when you enable autodraw.

Important
You should only disable autodraw for a short time if your list is displayed onscreen. If you leave autodraw disabled, the list will not display correctly, and will not respond properly to user interaction. ◆

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLUpdate

Draws all or part of a list onscreen.

```
void BLUpdate( RgnHandle inUpdateRgn, BlugsRef inList )
```

`inUpdateRgn`	The portion of the list that needs to be updated, or `nil` to update the entire list.
`inList`	The list that is to be updated.

Your application typically calls `BLUpdate` in response to an Event Manager `updateEvent`. `BLUpdate` redraws all portions of the list that intersects `inUpdateRgn`. If you pass `nil` for the update region, Blugs redraws the entire list and draws the list border and focus if they exist.

If the list has scroll bars, Blugs calls `Draw1Control` to update them. Be aware that if your application calls a more generic routine like `DrawControls` to update controls in a window that contains a list, and calls `BLUpdate`, the list's scroll bars will be drawn twice; users may find this distracting. You may find it desirable to update controls in your window individually.

If the list is not currently visible, or if `inUpdateRgn` does not intersect the list, `BLUpdate` does nothing.

BLIsVisible

Determines whether a list is hidden.

```
Boolean BLIsVisible( BlugsRef inList )
```

`inList`	The list to test for visibility.

When you create a list you can set or clear the `blVisible` list feature flag (see page 20). To determine the current visibility setting, call this function. `BLIsVisible` returns `true` if the list is visible.

Note that this function does not compute whether or not the list's host window is hidden, or any other external condition which might obscure the list. It only checks the list's internal state.

BLSetVisible

Hides or shows a list.

```
OSErr BLSetVisible( Boolean inMakeVisible, BlugsRef inList )
```

`inMakeVisible`	`true` if the list is to become visible.
`inList`	The list that is to be hidden or shown.

Use this routine to hide or show a list. If you pass `true` for `inMakeVisible`, Blugs shows the list; if you pass `false`, Blugs hides it. If the list contains scroll bars, `BLSetVisibility` calls `HideControl` or `ShowControl` to change scroll bar visibility. If `inMakeVisible` is `false`, Blugs calls `InvalRect` to force an update over the list rectangle (inset by -3 pixels when appropriate to cover the focus or border area).

If the list's visibility is already set to the state passed in the `inMakeVisible` parameter, `BLSetVisible` does nothing.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLIsActive

Determines whether a list is active.

```
Boolean BLIsActive( BlugsRef inList )
```

`inList`	The list to test for active status.

When you create a list you set or clear the `blActive` feature flag (see page 20). You can activate or deactivate a list using the `BLSetActive` procedure.To determine the flag's current setting, call this function. `BLIsActive` returns `true` if the list is active. If you pass a bad list reference, `BLIsActive` returns `false`.

BLSetActive

Activates or deactivates a list.

```
void BLSetActive( Boolean inMakeActive, BlugsRef inList )
```

`inMakeActive`	`true` if the list is to be activated.
`inList`	The list to activate or deactivate.

Use this routine to change a list's active state. If you pass `true` for the `inMakeActive` parameter, Blugs activates the list; if you pass `false`, Blugs deactivates it. If the list contains scroll bars, `BLSetActive` calls `HiliteControl`, `ActivateControl`, or `DeactivateControl`, as appropriate, to activate or deactivate them.

If the list's active state is already the same as the state passed in the `inMakeActive` parameter, or you pass a bad list reference, `BLSetActive` does nothing.

BLGetFocusedPart

Determines whether and what part of a list has keyboard focus.

```
ControlPartCode BLGetFocusedPart( BlugsRef inList )
```

`inList`	The list whose focus state is determined.

`BLGetFocusedPart` returns a part code indicating whether a list is currently focused, and what part of the list is focused.

RETURN VALUES

`kControlFocusNoPart`	The list is not focused; no inline edit session.
`kControlListBoxPart`	The list is focused; no inline edit session.
`blInlineEditPart`	The list has an inline edit session that is focused.

BLSetFocusedPart

Sets a list's keyboard focus.

```
ControlPartCode BLSetFocusedPart( ControlPartCode inPart,
                                   BlugsRef inList )
```

`inPart`	One of the following constants defined by the Control Manager and Blugs: `kControlFocusNoPart`, `kControlFocusNextPart`, `kControlFocusPrevPart`, `kControlListBoxPart`.
`inState`	The list whose focus state is set.

`BLSetFocusedPart` applies focusing as appropriate for the part code passed in the `inPart` parameter and returns a part code indicating the list part that has become focused. See the section "Keyboard Focus" on page 12 for more information on the part codes you can pass to this function.

If the list's focus changes, Blugs redraws the list as necessary.

Important
The constant `kBLInlineEditPart` is not appropriate as an input to this function. If you pass `kBLInlineEditPart` Blugs simply defocuses the list and returns `kControlFocusNoPart`. If you want to start an inline edit session, call `BLBeginInlineEdit`. ◆

RETURN VALUES

`kControlFocusNoPart`	The list is not focused; no inline edit session.
`kControlListBoxPart`	The list is focused; no inline edit session.

BLGetColumnFontStyle

Gets font information for a column.

```
OSErr BLGetColumnFontStyle( UInt16 inColumn,
                            ControlFontStylePtr outFontStyle,
                            BlugsRef inList )
```

`inColumn`	The column whose font information you want, or zero for the vertical title bar.
`outFontStyle`	The address of a `ControlFontStyleRec` which Blugs fills in with the column's font information.
`inState`	The list that contains the column.

Pass the address of a `ControlFontStyleRec` to `BLGetColumnFontStyle` to have Blugs fill in the record fields with font information for that column. If you pass zero for `inColumn`, Blugs gets the font information from the vertical title bar. When a column (or vertical title bar) is created, the fields of its associated `ControlFontStyleRec` are initialized as follows:

```
fontStyle.flags = kControlUseFontMask | kControlUseSizeMask;
fontStyle.font = 1;
fontStyle.size = 9;
```

Blugs sets the current port to the values in the appropriate `ControlFontStyleRec` before calling a content handler. After calling the handler, the port's state is restored.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetColumnFontStyle

Sets font information for a column.

```
OSErr BLSetColumnFontStyle( UInt16 inColumn,
                            const ControlFontStyleRec* inFontStyle,
                            BlugsRef inList )
```

`inColumn`	The column whose font information you want, or zero for the vertical title bar.
`outFontStyle`	The address of a `ControlFontStyleRec` which Blugs copies to the column's font information.
`inState`	The list that contains the column.

Pass the address of a `ControlFontStyleRec` to `BLGetColumnFontStyle` to have Blugs copy the information to the appropriate column. If you pass zero for `inColumn`, Blugs applies the font information to the vertical title bar. When a column (or vertical title bar) is created, its associated `ControlFontStyleRec` is initialized as follows:

```
fontStyle.flags = kControlUseFontMask | kControlUseSizeMask;
fontStyle.font = 1;
fontStyle.size = 9;
```

Blugs sets the current port to the values in the appropriate `ControlFontStyleRec` before calling a content handler. After calling the handler, the port's state is restored.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Blugs User Pane Controls

When the Appearance Manager is available, you can use these routines to create a user pane control that encapsulates an entire list. These routines only work under Appearance 1.0 and later.

BLMakeUserPaneControl

Sets up a list as an Appearance Manager user pane control.

```
OSErr BLMakeUserPaneControl( ControlRef* outNewCntl,
                             BlugsRef inList )
```

`outNewCntl`	On output, an Appearance user pane control which encapsulates the list.
`inList`	The list from which the user pane control is created.

`BLMakeUserPaneControl` creates a user pane control containing a list. This routine will only be successful if the Appearance Manager is installed.

RESULT CODES

Control Manager errors (-30580 to -30599)

`paramErr` (-50)	Bad list or control reference.
`unimpErr` (-4)	Appearance Manager not installed.
`noErr` (0)	No error.

BLConvertUserPaneControl

Sets up a list as an Appearance Manager user pane control, in an existing control.

```
OSErr BLConvertUserPaneControl( ControlRef ioControl,
                                BlugsRef inList )
```

`ioControl`	A user pane control that is to be converted to a Blugs list user pane control.
`inList`	The list from which the user pane control is created.

`BLConvertUserPaneControl` modifies `ioControl`, installing the callbacks and data necessary to turn it into a Blugs user pane control. This routine will only be successful if the Appearance Manager is installed.

This routine is provided mainly to support those dealing with the Dialog Manager. If you need a Blugs list in a dialog, you can create a user item in your `'DITL'` resource, then transmogrify it into a Blugs pane using this routine.

Note

For best results, you should set up the user pane with the same feature flags Blugs uses when it creates a user pane. If you create a user pane using a resource, use 58 (`0x003A`) as the inital value to encode the feature set `kControlHandlesTracking | kControlWantsActivate | kControlWantsIdle | kControlSupportsEmbedding`. If you user pane supports focus, use the value 318 (`0x013E`), which additionally encodes the features `kControlSupportsFocus | kControlGetsFocusOnClick`. ◆

RESULT CODES

Control Manager errors (-30580 to -30599)

`paramErr` (-50)	Bad list or control reference.
`unimpErr` (-4)	Appearance Manager not installed.
`noErr` (0)	No error.

BLRefFromUserPaneControl

Returns an embedded list from a user pane control created with `BLMakeUserPaneControl`.

```
BlugsRef BLRefFromUserPaneControl( ControlRef inBlugsCntl )
```

`inBlugsCntl`	The user pane control from which a list reference is to be extracted.

You can use `BLRefFromUserPaneControl` to extract a reference to a Blugs list from an Appearance Manager user pane control created with `BLMakeUserPaneControl`. If you pass a `nil` control, or if Blugs cannot retrieve a valid list reference from the control, `BLRefFromUserPaneControl` returns `nil`.

BLDisposeUserPaneControl

Disposes of an Appearance user pane control based on a Blugs list.

```
OSErr BLDisposeUserPaneControl( Boolean inDisposeOfList,
                                        ControlRef inBlugsCntl )
```

`inDisposeOfList`	`true` of this function is to call `BLDispose` on the list contained in the user pane control.
`inBlugsCntl`	The Appearance Manager user pane control to be disposed.

This routine disposes of a Blugs user pane control and optionally disposes of the embedded list as well. You must use this routine to dispose of controls created with `BLMakeUserPaneControl`; do not simply call `DisposeControl` or the memory occupied by the list will be leaked and the list's scroll bars will (in most cases) be automatically disposed without the list knowing about it.

Ordinarily, disposing of a user pane would dispose of the list's scroll bars in cases where an embedding hierarchy is established in the host window (as is typically the case, and recommended for Blugs user panes). This is because Blugs embeds list scroll bars in the user pane. Because the list may not be disposed of (and in any case would end up with stale `ControlRefs` in its data structure), `BLDisposeUserPaneControl` re-embeds the list's scroll bars in the host window's root control before calling `DisposeControl` on the user pane. If there is no embedding hierarchy (`GetRootControl` returns an error) then Blugs does not try to re-embed the scroll bars.

RESULT CODES

`paramErr` (-50)	Bad list or control reference.
`noErr` (0)	No error.

Inline Editing

You can use these routines if you need more control over inline editing than is provided by Blugs' event-handling routines. You can use these routines if you need to start or stop inline editing based on factors not available to Blugs.

Inline editing is allowed in cells but not in titles.

BLIsCellEditable

Determines if a cell's content handler supports inline editing.

```
Boolean BLIsCellEditable( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell whose editablility is checked.
`inList`	The list which contains the cell.

`BLIsCellEditable` returns `true` if `inCell` can be edited. If the cell does not support inline editing, editing has been disabled with `BLSetCellEditable`, or if Blugs is unable to retrieve the cell's content handler, it returns `false`.

BLSetCellEditable

Allows or disallows inline editing.

```
OSErr BLSetCellEditable( BLCell inCell, Boolean inEditable,
                         BlugsRef inList )
```

`inCell`	The cell whose editablility is set.
`inEditable`	`true` if the cell is to be editable, `false` if not.
`inList`	The list which contains the cell.

`BLSetCellEditable` disables or enables inline editing in a cell. It is particularly useful in a lists where a few otherwise editable cells are read-only in nature. By calling this routine you can keep the user from typing in new values for a few cells, and still have the advantage of using a table instead of a spreadsheet.

You can make a cell editable even if the current content handler does not support inline editing: `BLIsCellEditable` will still report `false` because of the content handler. If you pass `false` for `inEditable` and there is already an inline session in `inCell`, `BLSetCellEditable` does *not* end the inline session.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLIsInlineEdit

Determines if a list has an inline edit session in progress.

```
Boolean BLIsInlineEdit( BlugsRef inList )
```

`inList`	The list which is to be tested for an inline edit session.

`BLIsInlineEdit` returns `true` if `inList` has an inline edit session in progress. If there is no inline edit session, or if you pass a bad list reference, it returns `false`.

BLGetInlineEditCell

Retrieves the cell which is being edited.

```
OSErr BLGetInlineEditCell( BLCell* outCell, BlugsRef inList )
```

`outCell`	On output, the cell which is the target of the current inline edit session.
`inList`	The list whose inline edit cell is to be retrieved.

`BLGetInlineEditCell` returns the cell which is the target of the current inline edit session. If there is an inline edit session, `BLGetInlineEditCell` passes the inline edit cell back in the `outCell` parameter (it does not matter what values it contains on input). Otherwise, it returns `invalidEditState` and the contents of `outCell` are unchanged.

RESULT CODES

`invalidEditState` (-2023)	No inline edit session.
`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLBeginInlineEdit

Begins an inline edit session in a cell.

```
OSErr BLBeginInlineEdit( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell in which an inline edit session is to take place.
`inList`	The list which contains the cell.

This routine begins an inline edit session in the specified cell. It first checks the cell's content handler to make sure it supports inline editing. It then deselects all selected cells, after which it sets the list's keyboard focus to the cell's text region. It then calls the cell's content handler to begin the inline edit session; usually this means the content handler allocates and prepares a TextEdit environment or the equivalent. Finally, Blugs updates the cell.

If the cell's content handler does not support inline editing, `BLBeginInlineEdit` returns `editingNotAllowed`. Otherwise it returns `noErr`.

RESULT CODES

`editingNotAllowed` (-9995)	Content handler does not support inline editing.
`noErr` (0)	No error.

BLEndInlineEdit

Terminates a list's current inline edit session.

```
void BLEndInlineEdit( BlugsRef inList )
```

`inList`	The list in which an inline edit session is in progress.

This routine ends an inline edit session in the specified list. It calls the content handler for the cell currently being edited, informing it that the edit session is ending. If the content handler indicates that the cell's contents have changed as a result of the edit session, and if the list was previously sorted over the column which contains the inline edit cell, Blugs re-sorts the list. Blugs removes keyboard focus from the edited cell's content, and (if appropriate) focuses the entire list.

If there is no current inline edit session, `BLEndInlineEdit` does nothing.

Cell Data

These routines allow you to manipulate data contained and displayed in list cells and titles. A cell must have a content type (hence a content handler) in order to contain and display data. Use `BLGetCellContentType` and `BLSetCellContentType` to manipulate these. Use `BLGetCellData` to retrieve a copy of cell data, and `BLSetCellData` and `BLClearCell` to change the data. `BLCountCellFlavors` and `BLGetIndFlavorInfo` can be used for fine-tuning cell data acquisition.

BLGetCellContentType

Retrieves a cell's content type.

```
BLContentType BLGetCellContentType( BLCell inCell,
                                     BlugsRef inList )
```

`inCell`	The cell or title whose content type is retrieved.
`inList`	The list which contains the cell.

This routine retrieves a cell or title's content type regardless of whether the list is a table or a spreadsheet. If the `row` and `col` fields of `inCell` are nonzero, it is interpreted as a cell. If one of the fields contains zero, it is interpreted as a title. If both are zero, it is interpreted as the top left corner.

If `inList` is a bad list reference, or if `inCell` is not valid, `BLGetCellContentType` returns zero. Otherwise it returns the cell's content type (which may also be zero if it has not had a content type assigned).

Note that when a list or title bar has the table property (`blTable`/`blTitlesOneContentType`) `inCell` does not have to exist. For example, if `inList` is a table and you pass `inCell` = {100,1}, Blugs gets the content type for column 1 and ignores the row number (100). As a further example, if you pass {0,100} and there is a horizontal title bar with the `blTitlesOneContentType` property, Blugs gets its content type and ignores the column (100). When in doubt, pass 1 (not zero, since Blugs will interpret it as a title bar or the top left corner). Blugs only validates cell fields that are relevant.

BLSetCellContentType

Sets a cell's content type. In a table, this has the effect of setting the content type for the whole column.

```
OSErr BLSetCellContentType( BLCell inCell, BLContentType inContent,
                            BlugsRef inList )
```

`inCell`	The cell or title whose content type is set.
`inContent`	The new content type.
`inList`	The list which contains `inCell`.

This routine sets a cell or title's content type. If the `row` and `col` fields of `inCell` are nonzero, it is interpreted as a cell. If one of the fields contains zero, it is interpreted as a title. If both are zero, it is interpreted as the top left corner.

If `inCell` is interpreted as a cell, and the list is a spreadsheet, this routine sets a single cell's content type to `inContent`. In a table, however, the content type for the entire column that contains `inCell` becomes set to `inContent`.

If `inCell` is interpreted as a title, `BLSetCellContentType` sets the title's content type to `inContent`. If the title bar which contains the title was created with the feature flag `blTitlesOneContentType` set, all titles in the title bar are set to `inContent`. Otherwise only the one title is changed. Note that the top left corner is not considered a part of either title bar, so the `blTitlesOneContentType` flag will not cause the top left corner to change.

If `inCell` is interpreted as the top left corner, its content type is changed as if it were a single cell.

In all cases, if the cell(s) affected by this function initially have a different content type, they are deinitialized before being assigned the new type. `BLSetCellContentType` calls the old handler with the `blCellDeinitMsg` so it can dispose of any allocated memory before the cell is reinitialized. Note that when a cell is deinitialized and reinitialized, it is empty of data on output.

To have a cell associated with no content handler and holding no data, set the content type to zero.

As with `BLGetCellContentType` (see above), Blugs only validates cell fields that are relevant. For table-type lists and title bars, you can pass any number except zero for the irrelevant field.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetCellData

Retrieves a copy of a cell's data in a particular flavor.

```
OSErr BLGetCellData( BLCell inCell, OSType inDataFlavor,
                     UInt32* ioDataSize, void* outData,
                     BlugsRef inList )
```

`inCell`	The cell or title whose data is to be retrieved.
`inDataFlavor`	A four-character code identifying the flavor of data that is to be retrieved.
`ioDataSize`	On input, the maximum number of bytes that should be retrieved. On output, the actual number of bytes retrieved.
`outData`	The location in memory to which the appropriate content handler should copy data, or `nil` to just get the data size.
`inList`	The list which contains `inCell`.

`BLGetCellData` causes Blugs to call the cell or title's content handler with the message `blCellGetDataMsg`. If the handler can export data in the desired flavor, it copies a maximum of `ioDataSize` bytes to the location in memory referenced by `outData` if `outData` is non-`nil`. It then changes the contents of `ioDataSize` to the actual number of bytes copied, or the actual data size if `outData` is `nil`. It is assumed that if the handler cannot export the desired flavor or quantity of data, it will return zero in `ioDataSize`.

If the `row` and `col` fields of `inCell` are nonzero, it is interpreted as a cell. If one of the fields contains zero, it is interpreted as a title. If both are zero, it is interpreted as the top left corner.

▲ **WARNING**
Content handlers are largely on their own when exporting data; Blugs doesn't do much except validate parameters and invoke the handler. If you use someone else's content handler code, make sure you understand how it exports data. (For example, does it give you a reference to text, or does it give you the actual characters?) Beware of passing bogus information in `ioDataSize` even if you are sure the handler "couldn't *possibly* return more than four bytes." A content handler will very cheerfully clobber your application by returning thousands of bytes of data you didn't want and didn't plan on getting. ▲

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetCellData

Sets a cell contents to data in a particular flavor.

```
OSErr BLSetCellData( BLCell inCell, OSType inDataFlavor,
                        UInt32 inDataSize, void* inData,
                        BlugsRef inList )
```

`inCell`	The cell or title whose data is to be set.
`inDataFlavor`	A four-character code for the flavor of data being installed.
`inDataSize`	The number of bytes of data pointed to by the `inData` parameter.
`inData`	The address from which the appropriate content handler should copy data.
`inList`	The list which contains `inCell`.

`BLSetCellData` causes Blugs to call the cell or title's content handler with the `blCellSetDataMsg` message. If the handler can import data in the specified flavor, it copies a maximum of `inDataSize` bytes from the location in memory referenced by `inData`.

If the `row` and `col` fields of `inCell` are nonzero, it is interpreted as a cell. If one of the fields contains zero, it is interpreted as a title. If both are zero, it is interpreted as the top left corner.

If autodraw is enabled, Blugs updates the list after the cell data is set.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.

`paramErr` (-50)	`nil` data pointer.
`noErr` (0)	No error.

BLClearCell

Sets a cell's data to a content handler-defined "clear" status.

```
OSErr BLClearCell( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell to clear.
`inList`	The containing list.

`BLClearCell` causes Blugs to call the cell's content handler with the message `blCellClearDataMsg`. The actual effect this has on the cell's contents depends on the content handler — it may deallocate all cell storage or merely set (for example) a string's length byte to zero.

If autodraw is enabled, Blugs updates the list after the cell is cleared.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLCountCellFlavors

Counts the number of data flavors a cell's content handler can export.

```
UInt32 BLCountCellFlavors( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell whose exportable data flavors are counted.
`inList`	The list which contains the cell.

`BLCountCellFlavors` causes Blugs to call the cell or title's content handler to determine how many flavors of data it can export for the cell, and to return that number.

If the cell or title does not exist, or `inList` is not valid, `BLCountCellFlavors` returns zero.

BLGetIndFlavorInfo

Returns the type and data size of the indexed flavor that a cell can export.

```
OSErr BLGetIndFlavorInfo( BLCell inCell, UInt16 inIndex,
                          OSType* outDataFlavor,
                          UInt32* outSize, BlugsRef inList )
```

`inCell`	The cell whose exportable data flavor information is to be retrieved.
`inIndex`	A 1-based index to the flavor information retrieved.
`outDataFlavor`	On output, a four-character code for the data export flavor.
`outSize`	On output, the size in bytes of the data flavor the cell can export.
`inList`	The list which contains the cell.

`BLGetIndFlavorInfo` retrieves the indexed flavor type and size of data `inCell` can export. Generally, you will call `BLCountCellFlavors` to determine what range of indices you can pass to this function.

If this routine returns an error code, the contents of `outFlavorType` and `outSize` are undefined. The content handler will presumably return a zero size in `outSize` if `inIndex` is out of range or otherwise not valid, but Blugs cannot validate `inIndex` itself.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Cell or title does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Sorting and Searching

Use these routines if you need finer control over table sorting than Blugs can do automatically, or if you need to search a column for data.

BLGetSortState

Retrieves a list's primary sort column and/or its sort status.

```
OSErr BLGetSortState( BLSortState* outState, UInt16* outColumn,
                          BlugsRef inList )
```

`outState`	On output, the list's sort status. Pass `nil` if you don't care.
`outColumn`	On output, the list's primary sort column. Pass `nil` if you don't care.
`inList`	The list whose sort state is to be retrieved.

This routine gets the column over which the list is currently sorted, and the status. The `BLSortState` bits `blSortStateSorted` and `blSortStateLargeToSmall` are set or cleared to indicate how the list is currently sorted. If there is no primary sort column or sorting is not possible in the list, `BLGetSortState` returns zero in `outColumn` and `blSortStateUnsorted` (zero) in `outState`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSort

Sorts a list over a column.

```
OSErr BLSort( BLSortState inState, UInt16 inColumn,
                  BlugsRef inList )
```

`inState`	The new sort state.
`inColumn`	The column over which the list is to be sorted.
`inList`	The list which is to be sorted.

This routine sets the sort status of inList. If the `blSortStateSorted` bit is set in `inState`, Blugs marks `inColumn` as the primary sort column, sorts the list in the direction indicated by the `inState` `blSortStateLargeToSmall` bit status, and selects the appropriate title in the horizontal title bar if possible. The sort button, if present, will reflect the new state. If the list is not a table, not sortable, or if `inColumn`'s content handler does not support data comparisons, `BLSort` does not sort the list.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSearch

Finds the closest matching cell text in a table column.

```
OSErr BLSearch( UInt16 inColumn, void* inData, UInt32 inDataSize,
                  BLCell* outCell, BlugsRef inList )
```

`inColumn`	The column which is to be searched.
`inData`	The address of the text data to be searched for.
`inDataSize`	The number of bytes of text pointed to by `inData`.
`outCell`	On output, the cell whose data matched most closely.
`inList`	The list which contains the column to be searched.

`BLSearch` attempts to find the cell whose textual contents match the input as closely as possible. Blugs first sorts the list (internally) and then executes a binary search algorithm, calling upon the table's content handler to compare the input with cell data. If the return value is non-negative, `BLSearch` has been at least partially successful. See the result codes below, and the section "Error and Result Codes" on page 32.

This function will only work under a fairly constrained set of conditions. The list must be a sorted table whose primary sort column supports comparison of text with cell data.

Row hierarchy in a disclosure list is flattened in the course of constructing the internal sorted form. It is not currently possible to exclude or give priority to disclosure levels. This flattening is internal to the search mechanism; your list hierarchy remains intact.

RESULT CODES

`errMessageNotSupported` (-30580)	Handler doesn't support text search.
`kNoClientTableErr` (-9097)	List is not a table.
`handlerNotFoundErr` (-1856)	No handler for column.
`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column does not exist.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-109)	`nil` data or zero-length data.
`blSearchResultExactMatch` (0)	Search data was matched exactly: matched cell is returned.
`blSearchResultNextCell` (1000)	Search data was not matched exactly: next cell is returned.
`blSearchResultPrevCell` (1001)	Search data was not matched exactly: previous cell is returned.

Row and Column Identifiers

When you create a list whose rows and columns can be rearranged when the user drags them, you could lose track of which row or column is which. With these routines you can set and retrieve 32-bit `refCon`-like elements for each row and column. If you use `BLLoad` to create a list from a resource, you can also embed these identifiers in the resource data. You can use these identifiers in any way you choose, but if a row or column is deleted its identifier is removed. Rows and columns have their identifiers initialized to zero when they are created.

BLSetRowIdentifier

Stores a 32-bit identifier in a row.

```
OSErr BLSetRowIdentifier( UInt16 inRow, UInt32 inIdentifier,
                                    BlugsRef inList )
```

`inRow`	The row which is to hold the identifier.
`inIdentifier`	The data to be stored in the row.
`inList`	The list which contains the row.

This routine stores a 32-bit identifier in a row. You can use this number to identify a row even if it is subsequently moved. If the row already has an identifier, it is replaced by the new one. You can retrieve the identifier as long as the row exists.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row number zero or out of bounds.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetRowIdentifier

Retrieves a row's 32-bit identifier.

```
OSErr BLGetRowIdentifier( UInt16 inRow, UInt32 outIdentifier,
                              BlugsRef inList )
```

`inRow`	The row whose identifier is retrieved.
`outIdentifier`	On output, the identifier that was stored in the row.
`inList`	The list which contains the row.

This routine retrieves a 32-bit identifier from a row. You can use this number to identify a row even if it is subsequently moved. If you have not stored an identifier in the row, `BLGetRowIdentifier` will retrieve the row's initial value of zero.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row number zero or out of bounds.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetColumnIdentifier

Stores a 32-bit identifier in a column.

```
OSErr BLSetColumnIdentifier( UInt16 inColumn, UInt32 inIdentifier,
                              BlugsRef inList )
```

`inColumn`	The column which is to hold the identifier.
`inIdentifier`	The data to be stored in the column.
`inList`	The list which contains the column.

This routine stores a 32-bit identifier in a column. You can use this number to identify a column even if it is subsequently moved. If the column already has an identifier, it is replaced by the new one. You can retrieve the identifier as long as the column exists.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column number zero or out of bounds.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetColumnIdentifier

Retrieves a column's 32-bit identifier.

```
OSErr BLGetColumnIdentifier( UInt16 inColumn, UInt32 outIdentifier,
                              BlugsRef inList )
```

`inColumn`	The column whose identifier is retrieved.
`outIdentifier`	On output, the identifier that was stored in the column.
`inList`	The list which contains the column.

This routine retrieves a 32-bit identifier from a column. You can use this number to identify a column even if it is subsequently moved. If you have not stored an identifier in the column, `BLGetColumnIdentifier` will retrieve the column's initial value of zero.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Column number zero or out of bounds.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Unique Identifiers

Use these routines to get row, column, and cell unique identifiers from row and column numbers, and to get row and column numbers from unique identifiers.

BLGetCellFromUID

Gets the coordinates of a cell from its row and column UIDs.

```
OSErr BLGetCellFromUID( BLCellUID* inID, BLCell* outCell,
                        BlugsRef inList )
```

`inID`	The address of a cell UID.
`outCell`	On output, the cell coordinates.
`inList`	The list which contains the cell.

`BLGetCellFromUID` is a wrapper for calls to `BLGetRowFromUID` and `BLGetColumnFromUID`. It calls the aforementioned routines on the `rowID` and `colID` fields, repectively, of the `inID` parameter. If either the row or column identifier cannot be resolved to an existing row or column, `BLGetCellFromUID` returns `userDataItemNotFound`. This takes time linear in the number of rows and columns.

RESULT CODES

`userDataItemNotFound` (-2026)	Row or column does not exist.
`noErr` (0)	No error.

BLGetRowFromUID

Gets a row number from a row UIDs.

```
UInt16 BLGetRowFromUID( BLUID* inID, BlugsRef inList )
```

`inID`	The address of a row UID.
`inList`	The list which contains the row.

`BLGetRowFromUID` searches a list's internal row data array for a UID equal to `inID`. If found, Blugs returns the row number. Otherwise it returns zero. This takes time linear in the number of rows.

BLGetColumnFromUID

Gets a column number from a column UID.

```
UInt16 BLGetColumnFromUID( BLUID* inID, BlugsRef inList )
```

`inID`	The address of a column UID.
`inList`	The list which contains the column.

`BLGetColumnFromUID` searches a list's internal column data array for a UID equal to `inID`. If found, Blugs returns the column number. Otherwise it returns zero. This takes time linear in the number of columns.

BLGetCellUID

Retrieves a unique identifier from cell coordinates.

```
OSErr BLGetCellUID( BLCell inCell, BLCellUID* outID,
                    BlugsRef inList )
```

`inCell`	The cell whose unique identifier is retrieved.
`outID`	On output, the identifier.
`inList`	The list which contains the cell.

`BLGetCellUID` is a wrapper for calls to `BLGetRowUID` and `BLGetColumnUID`. It calls the aforementioned routines on the `row` and `col` fields, repectively, of the `inCell` parameter. If either the row or column does not exist, `BLGetCellUID` returns `inputOutOfBounds` and `outID` contains `0x00000000 000000000` in either the `rowID` or `colID` field (or both). This routine takes constant time.

RESULT CODES

`inputOutOfBounds` (-190)	Row or column does not exist.
`noErr` (0)	No error.

BLGetRowUID

Retrieves a unique identifier from a row number.

```
void BLGetRowUID( UInt16 inRow, BLUID* outID, BlugsRef inList )
```

`inRow`	The row whose unique identifier is retrieved.
`outID`	On output, the identifier.
`inList`	The list which contains the row.

`BLGetRowUID` looks up the row's unique identifier stored in the list's row data array. If the row does not exist, `outID` contains `0x00000000 000000000` on output. This routine takes constant time.

BLGetColumnUID

Retrieves a unique identifier from a column number.

```
void BLGetColumnUID( UInt16 inColumn, BLUID* outID,
                            BlugsRef inList )
```

`inColumn`	The column whose unique identifier is retrieved.
`outID`	On output, the identifier.
`inList`	The list which contains the column.

`BLGetColumnUID` looks up the column's unique identifier stored in `inList`'s column data array. If the column does not exist, `outID` contains `0x00000000 000000000` on output. This routine takes constant time.

User Data

You can use these routines to store additional data in a list. With the Mac OS List Manager and Control Manager, you can store your own data in a `refCon` field within the list or control record. Blugs allows you to store multiple user data items, each associated with a key value.

BLSetUserData

Stores data identified by a key.

```
OSErr BLSetUserData( SInt32 inKey, SInt32 inData, BlugsRef inList )
```

`inKey`	A unique identifier for the data. If data referenced by this key already exists in the current list, that data is replaced.
`inData`	The data to be stored in the list.
`inList`	The list in which the data is stored.

This routine stores the 32-bit `inData` value in the list. If the list already holds data referenced by `inKey`, Blugs replaces the previous data with the new data. To store more than 32 bits of information, you can pass a handle or pointer to a block of memory you have allocated. You must deallocate any memory whose reference you store in the list prior to disposing of the list, or you will have a memory leak. Similarly, you must deallocate the memory if you wish to store a different memory block referenced by the same `inKey` value, or you remove the data by calling `BLRemoveUserData`. Blugs only disposes of memory blocks it has itself allocated.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`memFullErr` (-108)	Could not allocate memory to store key-data pair.
`noErr` (0)	No error.

BLGetUserData

Retrieves user data previously stored in a list, referenced by a key.

```
OSErr BLGetUserData( SInt32 inKey, SInt32* outData,
                              BlugsRef inList )
```

`inKey`	A unique identifier for the data being retrieved.
`outData`	On output, the data which was previously stored in the list.
`inList`	The list in which the data is stored.

Retrieves a 32-bit value previously stored in the list, referenced by `inKey`. If the key cannot be located in the list, the routine returns `paramErr` and passes back `nil` in the `outData` parameter.

RESULT CODES

`userDataItemNotFound` (-2026)	Key could not be located.
`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.

`noErr` (0)	No error.

BLRemoveUserData

Removes user data previously stored in a list, referenced by a key.

```
OSErr BLRemoveUserData( SInt32 inKey, BlugsRef inList )
```

`inKey`	A unique identifier for the data being removed.
`inList`	The list in which the data is stored.

`BLRemoveUserData` finds and removes a 32-bit value previously stored in the list, referenced by `inKey`. Subsequent attempts to access the data previously associated with `inKey` will fail. If your data is a reference to a block of memory you have allocated, you must retrieve and dispose of it before calling `BLRemoveUserData`. Otherwise you will suffer a memory leak.

RESULT CODES

`userDataItemNotFound` (-2026)	Key could not be located.
`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Disclosure

Use these routines for additional control over disclosure (hierarchical) lists.

BLGetRowDisclosureLevel

Gets a row’s disclosure level.

```
OSErr BLGetRowDisclosureLevel( UInt16 inRow,
                               UInt16* outDisclosureLevel,
                               BlugsRef inList )
```

`inRow`	A valid row number.
`outDisclosureLevel`	On output, `inRow`’s disclosure level.
`inList`	The list that contains the row.

Call `BLGetRowDisclosureLevel` to obtain a row’s disclosure level. Disclosure levels are zero-based: level zero is the root level. If an error occurs, `BLGetRowDisclosureLevel` returns zero in `outDisclosureLevel`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`Row` zero; nonexistent row.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetRowDisclosureLevel

Sets a row's disclosure level.

```
OSErr BLSetRowDisclosureLevel( UInt16 inRow,
                               UInt16 inDisclosureLevel,
                               BlugsRef inList )
```

`inRow`	A valid row number.
`inDisclosureLevel`	`inRow`'s new disclosure level.
`inList`	The list that contains the row.

Call `BLSetRowDisclosureLevel` to indent or outdent (is that really a word??) `inRow` and its descendants to a new disclosure level. Disclosure levels are zero-based: level zero is the root level.

Blugs checks to make sure the passed-in disclosure is valid for `inRow` and all its descendants, which must move with it. The same constraints discussed in the section "Drag and Drop Disclosure Constraints" (Blugs Reference Manual page 10) apply to this routine; Blugs uses the same code for much of the logic.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetParentRow

Gets the number of a row's immediate parent.

```
UInt16 BLGetParentRow( UInt16 inRow, BlugsRef inList )
```

`inRow`	The row number of a child row.
`inList`	The list which contains the row.

This routine returns the number of the input row's immediate parent. If the list is not a disclosure list, or if `inRow` has no parent (that is, if its disclosure level is zero because it is at the top level of the hierarchy) `BLGetParentRow` returns zero.

BLRowIsDisclosed

Tests whether all of a row's ancestors are expanded.

```
Boolean BLRowIsDisclosed( UInt16 inRow, BlugsRef inList )
```

`inRow`	The row to be tested.
`inList`	The list which contains the row.

This routine determines whether all of `inRow`'s ancestors (that is, all rows which at least indirectly "contain" the row in question) are expanded. If they are all expanded (disclosure triangles point down) then `BLRowIsDisclosed` returns `true`.

Note that this routine does not compute whether or not `inRow` is actually visible onscreen. It may be scrolled out of the view rectangle.

BLExpandRow

Causes a row, and optionally its descendants, to be expanded.

```
OSErr BLExpandRow( Boolean inDeepExpand, UInt16 inRow,
                   BlugsRef inList )
```

`inDeepExpand`	`true` if any of the row's descendants which have children are to be expanded as well. `false` if only `inRow` is to be expanded.
`inRow`	The row to expand.
`inList`	The list which contains the row.

When `inDeepExpand` is `false`, this function expands only the row passed in the `inRow` parameter, as if the user clicked its disclosure triangle. When `inDeepExpand` is `true`, all descendants with children are expanded as well, as if the user option-clicked its disclosure triangle.

If you have installed a `RowExpandProc` callback, it is called for each row expanded.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-50)	Not a disclosure list.
`noErr` (0)	No error.

BLCollapseRow

Causes a row, and optionally its descendants, to be collapsed.

```
OSErr BLCollapseRow( Boolean inDeepCollapse, UInt16 inRow,
                     BlugsRef inList )
```

`inDeepCollapse`	`true` if any of the row's descendants which have children are to be collapsed as well. `false` if only `inRow` is collapsed.
`inRow`	The row to collapse.
`inList`	The list which contains the row.

When `inDeepCollapse` is `false`, this function collapses only the row passed in the `inRow` parameter, as if the user clicked the row's disclosure triangle. When `inDeepCollapse` is `true`, all descendants with children are collapsed as well, as if the user option-clicked the row's disclosure triangle.

If you have installed a `RowExpandProc` callback, it is called for each row collapsed.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Row zero; nonexistent row.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-50)	Not a disclosure list.
`noErr` (0)	No error.

BLCountDescendants

Returns the count of a row's descendants.

```
UInt16 BLCountDescendants( UInt16 inRow, BlugsRef inList )
```

`inRow`	The row whose descendants are counted.
`inList`	The list which contains the row.

`BLCountDescendants` returns a count of the range of rows which have a higher disclosure level than the one passed in the `inRow` parameter. This count represents the number of rows which are hidden if the row in question is collapsed.

If `inRow` is zero or does not exist, or you pass a `nil` or non-disclosure list, `BLCountDescendants` returns zero.

Title Bars

You can create title bars independently of lists, or load title bar information from resource using the `BLLoad` function. To create a title bar on the fly, call `BLNewTitleBar` using a preexisting list.

BLNewTitleBar

Creates a title bar associated with a list.

```
OSErr BLNewTitleBar( UInt16 inThickness, UInt16 inFlags,
                        Boolean inVertical, BlugsRef inList )
```

`inThickness`	The number of pixels high (for horizontal title bars) or wide (for vertical title bars) the title bar is to be.
`inFlags`	A set of flags that determine the title bar's characteristics. See the section "Title Bar Flags" on page 24 for details.
`inVertical`	`true` if the title bar is vertical, `false` if it is horizontal.
`inList`	The list which is to contain the title bar.

`BLNewTitleBar` creates a title bar and associates it with a list. Blugs creates the title bar within the bounds of the list's view rectangle. If you pass zero in the `inThickness` parameter, Blugs initializes the title bar's thickness to the default value of 16 pixels. (This value is subject to change in future releases. You should not depend on it.)

If the list in which `BLNewTitleBar` creates a title bar already has a title bar of the same orientation (for example, if the list already has a horizontal title bar, and the `inVertical` parameter is `false`), the old title bar is deleted and replaced with the new one. If this happens, the content handler(s) for the old title bar's titles are called to dispose of data for individual titles.

If you wish to load and populate a list and title bar(s) with a single function call, use the `BLLoad` function.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`memFullErr` (-108)	Could not allocate memory for title bar.
`noErr` (0)	No error.

BLGetHorizontalTitleBar

Returns a reference to a list's horizontal title bar.

```
BLTitleBarRef BLGetHorizontalTitleBar( BlugsRef inList )
```

`inList`	The list which contains the title bar.

`BLGetHorizontalTitleBar` returns a reference to the list's horizontal title bar, if it exists. Returns `nil` if you pass a bad list reference or if the list does not have a horizontal title bar.

BLGetVerticalTitleBar

Returns a reference to a list's vertical title bar.

```
BLTitleBarRef BLGetVerticalTitleBar( BlugsRef inList )
```

`inList`	The list which contains the title bar.

`BLGetVerticalTitleBar` returns a reference to the list's vertical title bar, if it exists. Returns `nil` if you pass a bad list reference or if the list does not have a vertical title bar.

BLSelectTitle

Selects a title in a title bar.

```
OSErr BLSelectTitle( UInt16 inTitle, BLTitleBarRef inTitleBar,
                            BlugsRef inList )
```

`inTitle`	The title to select.
`inTitleBar`	The title bar that contains the title.
`inList`	The list that contains the title.

Call `BLSelectTitle` to select `inTitle` and deselect any other selected title in `inTitleBar`. If `inTitleBar` is the horizontal title bar, the list can be sorted, and the new selection was not selected before, Blugs makes `inTitle` the new primary sort column and sorts the list. This routine behaves exactly as if the user had clicked in `inTitle`. Note that it is not currently possible to pass zero for `inTitle`; you cannot do a "deselect all titles" operation.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Title zero; nonexistent title.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetSelectedTitle

Returns the one-based index of the currently selected title in a title bar.

```
UInt16 BLGetSelectedTitle( BLTitleBarRef inBar, BlugsRef inList )
```

`inBar`	The title bar whose selection is to be retrieved.
`inList`	The list which contains the title bar.

`BLGetSelectedTitle` returns the one-based index of the currently selected title in the title bar specified by `inBar`. If the title bar does not contain a selection, `BLGetSelectedTitle` returns zero.

Scrolling and Navigation

For the most part Blugs automates scrolling and navigation based on user input. If you need programmatic control, you can use `BLPageUp` and `BLPageDown`. In rare cases you may want to customize the scroll distance using `BLSetScrollDistance`.

BLMakeVisible

Scrolls a list to make a cell more fully visible.

```
OSErr BLMakeVisible( BLCell inCell, BlugsRef inList )
```

`inCell`	The cell which is to be made more fully visible.
`inList`	The list which contains the cell.

`BLMakeVisible` tries to make the specified cell more fully visible by scrolling. The cell's top and left sides take priority. If the cell is at least partially scrolled out of view, Blugs does an animated scroll so the user can see what navigation has taken place.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Nonexistent cell.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLPageUp

Scrolls up by approximately the height of the list.

```
void BLPageUp( BlugsRef inList )
```

`inList`	The list that is to be scrolled.

`BLPageUp` scrolls the list up (toward the beginning of the list) by a number of pixels equal to the list height minus an arbitrary 12 pixels, so the user can stay oriented by having a small portion of the original view visible. Note that `BLPageUp` and `BLPageDown`, like Blugs' internal handling of the page up and page down keys, do not attempt to align the view top or bottom to a row boundary.

The value of 12 pixels is subject to change in future versions. Blugs does not attempt to align the view rectangle with cell boundaries because rows can all differ in height; trying to align would make it impossible in some situations to keep the same views across multiple page-up/page-down alternations. From the user's point of view this could be disconcerting.

BLPageDown

Scrolls down by approximately the height of the list.

```
void BLPageDown( BlugsRef inList )
```

`inList`	The list that is to be scrolled.

`BLPageDown` scrolls the list down (toward the end of the list) by a number of pixels equal to the list height minus an arbitrary 12 pixels. Note that `BLPageUp` and `BLPageDown`, like Blugs' internal handling of the page up and page down keys, do not attempt to align the view top or bottom to a row boundary.

BLSetScrollDistance

Sets the atomic distance by which Blugs scrolls a list.

```
OSErr BLSetScrollDistance( UInt8 inDistance, Boolean inVertical,
                                 BlugsRef inList )
```

`inDistance`	The pixels per iteration by which the list will be scrolled.
`inVertical`	`true` if the distance is to be applied to the vertical scroll bar, `false` if it is to be applied to the horizontal scroll bar.
`inList`	The list whose scroll distance is to be set.

`BLSetScrollDistance` allows you to customize list scrolling by setting the number of pixels at a time that are scrolled when the user holds down one of the specified scroll bar's arrow buttons. If you need unusually slow or fast scrolling you should use this function. Most users will never need it. The default scroll distance is six pixels (subject to change in future versions).

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Widgets

Use these routines to create and manage scroll bar widgets: placards in line with list scroll bars. Only lists with scroll bars can have widgets.

BLAddWidgets

Adds one or more widgets to a list.

```
OSErr BLAddWidgets( Boolean inVertical, UInt16 inCount,
                    UInt16 inWidget, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inCount`	The number of widgets to add.
`inWidget`	The 1-based index of the first added widget.
`inList`	The list to which widgets are added.

This function inserts a number of widgets equal to `inCount`, starting at the widget whose number is equal to `inWidget`. If there is already a widget at the `inWidget` location, it (and any widgets numbered higher than it) are shifted to make room for the new widgets. If you pass zero in the `inCount` parameter, Blugs does nothing.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`Widget` zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`memFullErr` (-108)	Not enough memory.
`noErr` (0)	No error.

BLDeleteWidgets

Deletes one or more widgets from a list.

```
OSErr BLDeleteWidgets( Boolean inVertical, UInt16 inCount,
                       UInt16 inWidget, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inCount`	The number of widgets to be deleted.
`inWidget`	The one-based index of the first deleted widget.
`inList`	The list from which widgets are deleted.

This function deletes a number of widgets equal to `inCount`, starting at `inWidget`. `BLDeleteWidgets` does not try to remove widgets that do not exist. If `inCount` is zero, all widgets are deleted.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Widget zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLCountWidgets

Counts the widgets in a scroll bar.

```
UInt16 BLCountWidgets( Boolean inVertical, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inList`	The list that contains the scroll bar.

`BLCountWidgets` returns the number of widgets in line with the vertical scroll bar if `inVertical` is `true`, the horizontal scroll bar if `false`. If there is no scroll bar in the specified axis or you pass a bad list parameter, `BLCountWidgets` return zero.

BLGetWidgetSize

Measures a widget.

```
SInt16 BLGetWidgetSize( Boolean inVertical, UInt16 inWidget,
                        BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget to measure.
`inList`	The list that contains the widget.

`BLGetWidgetSize` returns the pixel size of the widget. Depending on the associated scroll bar axis, one dimension of a widget is always fixed at the scroll bar size (16 pixels in document windows). This routine returns the variable size: the *other* dimension. In the vertical scroll bar, that is height. If there is no scroll bar in the specified axis or you pass a bad list parameter, `BLGetWidgetSize` return zero.

BLSetWidgetSize

Sets a widget's height or width.

```
OSErr BLAddWidgets( Boolean inVertical, UInt16 inWidget,
                    SInt16 inSize, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inSize`	The new widget size.
`inList`	The list that contains the widget.

`BLSetWidgetSize` sets the pixel size of the widget. Depending on the associated scroll bar axis, one dimension of a widget is always fixed at the scroll bar size (16 pixels in document windows). This routine adjusts the variable size: the *other* dimension. In the vertical scroll bar, that is height. Blugs redraws the widget after resizing it.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`rgnTooBigErr` (-500)	New size is too big.
`inputOutOfBounds` (-190)	`Widget` zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetWidgetContentType

Returns a widget's content type.

```
BLContentType BLGetWidgetContentType( Boolean inVertical,
                                      UInt16 inWidget,
                                      BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inList`	The list that contains the widget.

`BLGetWidgetContentType` returns the content type of the widget. If there is no scroll bar in the specified axis or you pass a bad list parameter, `BLGetWidgetContentType` return zero.

BLSetWidgetContentType

Sets a widget's content type.

```
OSErr BLSetWidgetContentType( Boolean inVertical, UInt16 inWidget,
                              BLContentType inContent,
                              BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inContent`	The new content type.
`inList`	The list that contains the widget.

This routine sets a widget's content type. If the widget has a different content type on entry, it is deinitialized before being assigned the new type. `BLSetWidgetContentType` calls the old handler with the `blCellDeinitMsg` so it can dispose of any allocated memory before the cell is reinitialized. Note that when a widget is deinitialized and reinitialized, it is empty of data on output. Blugs redraws the widget after setting the content.

To have a widget associated with no content handler and holding no data, set the content type to zero.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`Widget` zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetWidgetData

Retrieves data from a widget.

```
OSErr BLGetWidgetData( Boolean inVertical, UInt16 inWidget,
                        OSType inDataFlavor, UInt32* ioDataSize,
                        void* outData, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inDataFlavor`	A four-character code identifying the flavor of data that is to be retrieved.
`ioDataSize`	On input, the maximum number of bytes that should be retrieved. On output, the actual number of bytes retrieved.
`outData`	The location in memory to which the appropriate content handler should copy data, or `nil` to just get the data size.
`inList`	The list that contains the widget.

`BLGetWidgetData` is a lot like `BLGetCellData`. It causes Blugs to call the widget's content handler with the message `blCellGetDataMsg`. If the handler can export data in the desired flavor, it copies a maximum of `ioDataSize` bytes to the location in memory referenced by `outData` if `outData` is non-`nil`. It then changes the contents of `ioDataSize` to the actual number of bytes copied, or the actual data size if `outData` is `nil`. It is assumed that if the handler cannot export the desired flavor or quantity of data, it will return zero in `ioDataSize`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	Widget does not exist.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetWidgetData

Installs data in a widget.

```
OSErr BLSetWidgetData( Boolean inVertical, UInt16 inWidget,
                        OSType inDataFlavor, UInt32 inDataSize,
                        void* inData, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inDataFlavor`	A four-character code for the flavor of data being installed.
`inDataSize`	The number of bytes of data pointed to by the `inData` parameter.

`inData`	The address from which the appropriate content handler should copy data.
`inList`	The list that contains the widget.

`BLSetWidgetData` is similar to `BLSetCellData`. It causes Blugs to call the widget's content handler with the `blCellSetDataMsg` message. If the handler can import data in the specified flavor, it copies a maximum of `inDataSize` bytes from the location in memory referenced by `inData`. Blugs updates the widget after the data is set.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`Widget` zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`paramErr` (-50)	`nil` data pointer.
`noErr` (0)	No error.

BLClearWidget

Removes all data from a widget.

```
void BLClearWidget( Boolean inVertical, UInt16 inWidget,
                        BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inList`	The list that contains the widget.

`BLClearWidget` is similar to `BLClearCell`. It causes Blugs to call the widget's content handler with the message `blCellClearDataMsg`. The actual effect this has on the cell's contents depends on the content handler — it may deallocate storage or merely set (for example) a string's length byte to zero. Blugs updates the widget after clearing it.

BLGetWidgetRect

Returns a widget's bounds.

```
OSErr BLGetWidgetRect( Boolean inVertical, UInt16 inWidget,
                        Rect* outRect, BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inWidget`	On output, the widget bounds.
`inList`	The list that contains the widget.

`BLGetWidgetRect` calculates the rectangle occupied by a widget and returns it in `outRect`. If it returns an error code other than `noErr`, `outRect` will contain an empty rectangle (fields all zero) on output.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`Widget` zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLGetWidgetFlags

Returns a widget's feature flags.

```
UInt16 BLGetWidgetFlags( Boolean inVertical, UInt16 inWidget,
                         BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inList`	The list that contains the widget.

`BLGetWidgetFlags` returns a widget's feature flags. See the enumeration "Widget Flags" on page 24.If there is no scroll bar in the specified axis or you pass a bad list parameter, `BLGetWidgetFlags` return zero.

BLSetWidgetFlags

Sets a widget's feature flags.

```
OSErr BLSetWidgetContentType( Boolean inVertical, UInt16 inWidget,
                              UInt16 inWhichFlags, UInt16 inFlags,
                              BlugsRef inList )
```

`inVertical`	`true` if vertical scroll bar, `false` if horizontal.
`inWidget`	The 1-based index of the widget.
`inWhichFlags`	A mask in which flag bits to be changed are set.
`inFlags`	The new set of flags.
`inList`	The list that contains the widget.

This routine modifies a widget's feature flags. See the enumeration "Widget Flags" on page 24. Blugs redraws the widget after modifying the flags.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`inputOutOfBounds` (-190)	`Widget` zero; nonexistent widget.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

Utility Routines

BLSettings

Changes Blugs' global settings.

```
void BLSettings( UInt32 inSettings )
```

`inSettings`	The new global settings.

Call `BLSettings` to change Blugs' global behavior. Currently the only option is `blTempGWorlds`, but other behaviors may be added in future versions (behaviors that should be dictated by runtime conditions such as memory availability and screen resolution).

BLEnvironment

Returns flag values for common `Gestalt` checks.

```
UInt32 BLEnvironment( void )
```

When you call `BLEnter` to initialize Blugs, a number of `Gestalt` checks are made and recorded. To get these values, and to find out if Blugs is indeed initialized, call BLEnvironment. See the enumeration Environment Flags on page 19.

BLAppearanceVersion

Returns the Appearance Manager version.

```
SInt16 BLAppearanceVersion( void )
```

Use this routine to determine whether your application or content handler is running under the Appearance Manager. If some version of Appearance is installed, `BLAppearanceVersion` returns the BCD (binary coded decimal) version number, with the major revision in the high-order byte. For example, Appearance 1.0 is represented as `0x0100`. If `BLAppearanceVersion` returns `0x0000` then Appearance is not present; do not use Appearance Manager routines in that case.

BLCredits

Retrieves the Blugs copyright string.

```
void BLCredits( Str255 outCredits )
```

`outCredits`	On output, the Blugs credits string.

Use this routine to get the credits string so you can display it in an about box or splash screen. This is the only Blugs routine (other than `BLEnter`) you can safely call when Blugs is uninitialized.

BLDrawBevelButton

Draws a 3-D button with square corners.

```
void BLDrawBevelButton( Rect* inRect, ThemeDrawState inState,
                        ThemeButtonValue inValue )
```

`inRect`	The rectangle in which the button is drawn.
`inState`	The button's draw state.
`inValue`	The button's value.

`BLDrawBevelButton` draws a control-like object with square corners and 2-pixel beveled edges filling the input rectangle. This is the routine Blugs calls for drawing titles. Pass one of the Appearance Manager draw state constants `kThemeStateActive`, `kThemeStateInactive`, or `kThemeStatePressed` in the `inState` parameter. Pass either of the Appearance Manager button value constants `kThemeButtonOn` or `kThemeButtonOff` in the `inValue` parameter. If Appearance version 1.1 or later is installed, Blugs calls `DrawThemeButton` using the constant `kThemeSmallBevelButton`. Otherwise it uses its own code.

Important
If you have installed a custom bevel button drawing routine using `BLRegisterBevelButtonProc`, that routine is *not* called by `BLDrawBevelButton`. ◆

BLDrawPlacard

Draws a 3-D raised pane.

```
void BLDrawPlacard( Rect* inRect, ThemeDrawState inState )
```

`inRect`	The rectangle in which the placard is drawn.
`inState`	The placard's draw state.

`BLDrawPlacard` draws a control-like object with square corners and 1-pixel beveled edges filling the input rectangle. This is the routine Blugs calls for drawing widgets. Pass one of the Appearance Manager draw state constants `kThemeStateActive`, `kThemeStateInactive`, or `kThemeStatePressed` in the `inState` parameter. If Appearance version 1.0 or later is installed, Blugs calls `DrawThemePlacard`. Otherwise it uses its own code.

Registering User-Defined Routines

You call the `BLRegisterContentHandler` function to make a content handler callback function available to Blugs. For other callback types, you pass a pointer to a `BLCallbacksRec` with the addresses of your callback routines, to the `BLSetCallbacks`. If you want to register callbacks individually, you can call a routine whose name is the example user routine name with "`My`" changed to "`BLRegister`". Thus you register a routine named below as `MyBevelButtonProc` by means of the function called `BLRegisterBevelButtonProc`.

Note
You can unregister any callback except a content handler by passing `nil` for its routine pointer. `BLRegisterBackgroundProc( nil, list )` is perfectly valid code. ◆

▲ **WARNING**
You are strongly encouraged not to call from within a user-defined routine any Blugs routine which may change the state of the list or call your routine recursively. If you cannot avoid this, make sure your routine is reentrant. ▲

BLRegisterContentHandler

Associates a content hander procedure with a content type, and allows Blugs to call upon that handler routine.

```
OSErr BLRegisterContentHandler( BLContentType inContent,
                                UInt16 inVariationCode,
                                BLContentHandlerProcPtr inProc )
```

`inContent`	A 16-bit number identifying the handler.
`inVariationCode`	An initial setting for the handler.
`inProc`	The address of a content handler function.

Before any content handler can be used, it must be registered. Registration associates a routine pointer with a content type. Cells, titles, and other list elements with a given content type are processed by the handler routine associated with that type.

The `inVariationCode` parameter is a value that becomes associated with the handler on a global level. Handlers will typically define whether or not the variation code has any meaning, and what that meaning is. For example, a text content handler might use this value to indicate the default font setting for all its cells.

Content handlers are registered globally to your application. Thus, if your static text handler is associated with a content type of three, all elements in your application with this type are processed by that handler, no matter what list or window contains them.

For fastest access, Blugs allocates a global hash table of content types and looks up the procedure pointer and other data from that table. The table is created when you call `BLEnter` and is populated when you call `BLRegisterContentHandler`. Blugs' globals are opaque, so you cannot access or modify this hash table. You can, however, allow Blugs to make most efficient use of it. For reasons that will be obvious to readers familiar with separate chaining hashing algorithms, the current version of Blugs can access handlers the quickest if their content types are between 1 and 101, inclusive. We consider it highly unlikely that developers will ever need to register 101 handlers simultaneously, but in case you are tempted to use a number over 101 (perhaps for mnemonic reasons) you should be aware that doing so might result in a small performance penalty. Using content types greater than 101 also uses a few extra bytes of memory.

Content type zero is not a valid type. If `inContentType` is zero, or if `inContent` is already registered, or if `inProc` is `nil`, `BLRegisterContentHandler` returns `paramErr`.

Important
You cannot unregister or replace a content handler. Since Blugs does not keep a “list of lists” (like the Window Manager’s window list) there is no way for the content handler subsystem to find all existing lists in order to let the handler deinitialize itself. In simpler language, you can’t pass `nil` for `inProc`, or a content type that is already used, for `inContent`. ◆

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`memFullErr` (-108)	Could not allocate memory needed to store handler.
`paramErr` (-50)	`nil` procedure pointer, content type zero, content type already registered.
`noErr` (0)	No error.

BLGetCallbacks

Gets all callbacks registered to a list.

```
OSErr BLGetCallbacks( BLCallbacksPtr outCallbacks,
                      BlugsRef inList )
```

`outCallbacks`	The address of a callback record.
`inList`	The list whose callbacks are retrieved.

`BLGetCallbacks` fills each field of a `BLCallbacksRec` (see page 29), to which you pass a pointer, with the user-defined routines registered to `inList`. If you call this routine on a list that has just been created, you will find that all fields of the record are set to `nil`.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLSetCallbacks

Registers a set of callbacks to a list.

```
OSErr BLSetCallbacks( const BLCallbacksPtr inCallbacks,
                      BlugsRef inList )
```

`inCallbacks`	The address of a callback record, or `nil` to remove all callbacks.
`inList`	The list whose callbacks are retrieved.

`BLSetCallbacks` registers a list callback for which you provide a non-`nil` routine pointer, for each field of a `BLCallbacksRec` (see page 29) passed by reference in `inCallbacks`. For each callback record field in which you pass `nil`, the corresponding routine is unregistered. If you pass a nil `inCallbacks` pointer, all callback routines are unregistered.

The ability to pass `nil` for `inCallbacks` is new in Blugs 1.1.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterBackgroundProc

Makes a user-defined background-drawing callback available to Blugs.

```
OSErr BLRegisterBackgroundProc( BLBackgroundProcPtr inProc,
                                BlugsRef inList )
```

`inProc`	The address of a background-drawing routine.
`inList`	The list in which the routine will be executed.

`BLRegisterBackgroundProc` installs a user-defined cell background-drawing routine in the specified list. See the description of `MyBackgroundProc` on page 103 for more information on this type of routine. After installing your routine Blugs invokes it to redraw cells with the new background, provided autodraw is enabled.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterBevelButtonProc

Makes a user-defined bevel button-drawing callback available to Blugs.

```
OSErr BLRegisterBevelButtonProc( BLBevelButtonProcPtr inProc,
                                   BlugsRef inList )
```

`inProc`	The address of a bevel button-drawing routine.
`inList`	The list in which the routine will be executed.

`BLRegisterBevelButtonProc` installs a user-defined bevel button-drawing routine in the specified list. See the description of `MyBevelButtonProc` on page 103 for more information on this type of routine. After installing your routine Blugs invokes it to redraw bevel buttons, provided autodraw is enabled.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterBorderProc

Makes a user-defined cell border-drawing callback available to Blugs.

```
OSErr BLRegisterBorderProc( BLBorderProcPtr inProc,
                            BlugsRef inList )
```

`inProc`	The address of a border-drawing routine.
`inList`	The list in which the routine will be executed.

`BLRegisterBorderProc` installs a user-defined cell border-drawing routine in the specified list. See the description of `MyBorderProc` on page 104 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterHiliteProc

Makes a user-defined cell hiliting callback available to Blugs.

```
OSErr BLRegisterHiliteProc( BLHiliteProcPtr inProc,
                            BlugsRef inList )
```

`inProc`	The address of a cell hiliting routine.
`inList`	The list in which the routine will be executed.

`BLRegisterHiliteProc` installs a user-defined cell hiliting routine in the specified list. See the description of `MyHiliteProc` on page 105 for more information on this type of routine. After installing your routine Blugs invokes it to apply the new hiliting style to any selected cells, provided autodraw is enabled.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterRowExpandProc

Makes a user-defined row expansion callback available to Blugs.

```
OSErr BLRegisterRowExpandProc( BLRowExpandProcPtr inProc,
                               BlugsRef inList )
```

`inProc`	The address of a row expansion routine.
`inList`	The list in which the routine will be executed.

`BLRegisterHiliteProc` installs a user-defined row expansion routine in the specified list. See the description of `MyRowExpandProc` on page 106 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterSecondarySortColumnProc

Makes a user-defined secondary sort column callback available to Blugs.

```
OSErr BLRegisterSecondarySortColumnProc(
                               BLSecondarySortColumnProcPtr inProc,
                               BlugsRef inList )
```

`inProc`	The address of a secondary sort column routine.
`inList`	The list in which the routine will be executed.

`BLRegisterSecondarySortColumnProc` installs a user-defined secondary sort column routine in the specified list. See the description of `MySecondarySortColumnProc` on page 107 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterPreDragProc

Makes a user-defined drag-inspecting callback available to Blugs.

```
OSErr BLRegisterPreDragProc( BLPreDragProcPtr inProc,
                              BlugsRef inList )
```

`inProc`	The address of a pre-drag routine.
`inList`	The list in which the routine will be executed.

`BLRegisterPreDragProc` installs a user-defined routine to inspect a drag before it begins, in the specified list. See the description of `MyPreDragProc` on page 107 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterDragDataProc

Makes a user-defined callback that adds cell data available to Blugs.

```
OSErr BLRegisterDragDataProc( BLDragDataProcPtr inProc,
                              BlugsRef inList )
```

`inProc`	The address of a drag data routine.
`inList`	The list in which the routine will be executed.

`BLRegisterDragDataProc` installs in the specified list a user-defined routine that adds cell data to a drag that is beginning. See the description of `MyDragDataProc` on page 108 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterDropValidationProc

Makes a user-defined drop validation callback available to Blugs.

```
OSErr BLRegisterDropValidationProc(
                              BLDropValidationProcPtr inProc,
                              BlugsRef inList )
```

`inProc`	The address of a drop validation routine.

`inList`	The list in which the routine will be executed.

`BLRegisterDropValidationProc` installs a user-defined drop validation routine in the specified list. See the description of `MyDropValidationProc` on page 108 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterDropProc

Makes a user-defined drop callback available to Blugs.

```
OSErr BLRegisterDropProc( BLDropProcPtr inProc, BlugsRef inList )
```

`inProc`	The address of a drop routine.
`inList`	The list in which the routine will be executed.

`BLRegisterDropProc` installs a user-defined drop-handling routine in the specified list. See the description of `MyDropProc` on page 109 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.
`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

BLRegisterPostDragProc

Makes a user-defined post-drag callback available to Blugs.

```
OSErr BLRegisterPostDragProc( BLPostDragProcPtr inProc,
                              BlugsRef inList )
```

`inProc`	The address of a post-drag routine.
`inList`	The list in which the routine will be executed.

`BLRegisterPostDragProc` installs a user-defined post-drag routine in the specified list. See the description of `MyPostDragProc` on page 109 for more information on this type of routine.

RESULT CODES

`notInitErr` (-900)	Blugs is not initialized.

`nilHandleErr` (-109)	Bad list reference.
`noErr` (0)	No error.

User-Defined Routines

Using the appropriate registration function (see above) you make available your implementation of one of the user-defined routines described in this section.

MyBackgroundProc

Supply your implementation of this routine to draw cell backgrounds rather than using the default background.

```
pascal void MyBackgroundProc( Boolean inSortColumn, BLCell inCell,
                              const Rect* inRect, BlugsRef inList )
```

`inSortColumn`	`true` if `inCell` is in the sorted column.
`inCell`	The cell whose background is to be drawn.
`inRect`	The address of the cell's rectangle in local coordinates.
`inList`	The list that contains the cell.

This routine is called to draw a cell's background. The `inSortColumn` parameter is provided so you can draw appropriate shading for cells in the primary sort column.

When Blugs calls your routine, the drawing environment is already set to the correct port. Your routine should not change graphics ports. You do not need to include "clean-up" code in your routine (such as code that restores the size of the graphics pen or the foreground color) since Blugs restores the graphics environment as appropriate after your routine returns.

MyBevelButtonProc

Supply your implementation of this routine to draw a bevel button as you wish, rather than using the Appearance-supplied or built-in bevel button routines.

```
pascal Boolean MyBevelButtonProc( BLPart inPart, UInt16 inTitle,
                              const Rect* inRect,
                              ThemeDrawState inState,
                              ThemeButtonValue inValue,
                              BlugsRef inList )
```

`inPart`	A code identifying the list part to be drawn.
`inTitle`	The one-based index of the title to be drawn.
`inRect`	The address of the rectangle within which the button should be drawn.
`inState`	The object's current state.

`inValue`	The object's current value.
`inList`	The list in which the title is to be drawn.

Blugs calls this routine to draw titles, filler titles, the sort button, and other objects that typically appear as bevel buttons. The following part codes may be sent in the `inPart` parameter:

```
blHTitleBarTitlePart
blHTitleBarFillerPart
blVTitleBarTitlePart
blVTitleBarFillerPart
blTopLeftPart
blSortButtonPart
blGrowBoxPart
```

For titles, the `inTitle` parameter contains a nonzero title number.
For other objects, `inTitle` is zero.

Blugs will pass one of the Appearance Manager draw state constants `kThemeStateActive`, `kThemeStateInactive`, or `kThemeStatePressed` in the `inState` parameter. Blugs will pass either of the Appearance Manager button value constants `kThemeButtonOn` or `kThemeButtonOff` in the `inValue` parameter.

Your function should return `true` if it drew the object and `false` if it did not draw. In the latter case, Blugs draws the bevel button as it normally would. You need only draw the objects whose appearance you wish to customize.

MyBorderProc

Supply your implementation of this routine to draw cell borders.

```
pascal Boolean MyBorderProc( Boolean inDrawRowBorder,
                             Boolean inDrawColumnBorder,
                             BLCell inCell, const Rect* inRect,
                             BlugsRef inList )
```

`inDrawRowBorder`	`true` if your routine should draw a horizontal border.
`inDrawColumnBorder`	`true` if your routine should draw a vertical border.
`inCell`	The cell whose border you are drawing.
`inRect`	A pointer to the cell's rectangle.
`inList`	The list which contains the cell.

Blugs calls this routine to draw cell borders. Typically you draw the border as a 1-pixel thick line to the bottom or right of the cell.

MyFlattenProc

Supply your implementation of this routine to retrieve cell and title data for `BLFlatten`.

```
pascal void MyFlattenProc( BLCell inCell, OSType* outDataFlavor,
                           UInt32* ioDataSize, void* outData,
                           BlugsRef inList );
```

`inCell`	The cell or title whose data is requested.
`outDataFlavor`	On output, the preferred data flavor to save.
`ioDataSize`	On output, the number of bytes of data to save.
`outData`	`nil` on first call, a buffer into which your routine copies cell data on second call.
`inList`	The list that is being flattened.

Pass your implementation of this routine to `BLFlatten`. The first time it is called for a cell or title, `outData` is `nil`. Return the data size and flavor. If, on output, `ioDataSize` is nonzero, your routine will be called a second time with a valid `outData` buffer. Generally this function will act as a wrapper for `BLGetCellData`. It allows you to determine what data flavors to use, and you can filter out cells/titles you don't want to save.

MyHiliteProc

Supply your implementation of this routine to draw selection hiliting as you wish, rather than using the handler-supplied or built-in hilite routines.

```
pascal Boolean MyHiliteProc( Rect* inRect, BlugsRef inList )
```

`inRect`	The address of the rectangle of the cell to be hilited.
`inList`	The list that contains the cell.

This routine is called to draw cell hiliting. When installed in a list by means of the `BLRegisterHiliteProc` procedure, it overrides Blugs' built-in functionality and any special hiliting done by content handlers. Your routine is called only for cells, not for any other elements that can be selected (like titles).

Important
Use this type of procedure with care, and only with content handlers whose hiliting requirements are compatible with yours. Registering a `HiliteProc` callback is an excellent opportunity to perform user interface butchery. ◆

MyNotificationProc

Supply your implementation of this routine to respond to user actions as they are reported by Blugs.

```
pascal void MyNotificationProc( BLNotificationMessage inMessage,
                                BLNotificationCommand inCommand,
                                BLPart inPart, BLCell inCell,
                                BlugsRef inList )
```

`inMessage`	The type of event reported.
`inCommand`	A command issued from the relevant content handler.
`inPart`	The part in which the event happened.
`inCell`	The coordinates of `inPart`.
`inList`	The list in which the event happened.

This routine is called when something happens in a list and your application may want to respond to it. The `inMessage` parameter is a predefined code for the type of event. The `inCommand` parameter is data that Blugs passes from a content handler to your notification. For example, a popup menu content handler may define a command that means the user selected a new item. When the handler sends the command back to Blugs in the content handler parameter block, Blugs passes the command along to your notification callback. Based on `inMessage`, information in `inList`, and this command, you can decide whether you need to respond to the event. Some notifications are not content handler related, so `inCommand` may not be relvant in those cases.

There is no `BLRegisterNotificationProc` routine. Use `BLSetCallbacks`.

MyRowExpandProc

Supply your implementation of this routine to add and delete rows on the fly when a disclosure triangle is expanded or collapsed.

```
pascal void MyRowExpandProc( Boolean inExpanding, UInt16 inRow,
                                BlugsRef inList )
```

`inExpanding`	`true` if the row is being expanded, `false` if it is being collapsed.
`inRow`	The number of the row that is being expanded or collapsed.
`inList`	The list that contains the row.

This routine is called to add or delete rows when a parent row is in the process of being expanded or collapsed. By registering a procedure of this kind, you can cut down on the amount of time it takes to set up a large disclosure list, adding only the necessary rows at initialization and deferring the rest until they are needed.

If you use, for example, `BLAddRows` or `BLDeleteRows`, you will generally call it with a `inRow` parameter one greater than the `inRow` passed to this procedure. To safely delete the appropriate number of rows when called with `inExpanding` equal to `false`, call

`BLCountDescendants( inRow, inList )` and pass the function return value as the `inCount` parameter of `BLDeleteRows`.

Important
Using a technique like this is not a replacement for sound memory management strategies; it is meant to speed list creation. You should prepare for the worst-case memory scenario, in which all rows are fully expanded. ◆

MySecondarySortColumnProc

Supply your implementation of this routine to derive a column number which will be used for secondary sorting.

```
pascal UInt16 MySecondarySortColumnProc( UInt16 inPrimaryColumn,
                                         BlugsRef inList )
```

`inPrimaryColumn`	The column used for the shallowest level of sorting.
`inList`	The list that is being sorted.

Blugs' default behavior is to sort a list based on the column number of the currently selected title. This is the primary sort column. By using a function of this type, you can allow Blugs to use a second column to resolve the ordering of rows which have identical (for sorting purposes at least) elements in the primary column.

Your function should return the number of the column that should be used as a secondary sort column. To indicate that Blugs should not use a secondary sort column in this particular case, return zero.

MyPreDragProc

Supply your implementation of this routine to inspect a drag just as it is beginning.

```
pascal OSErr MyPreDragProc( DragReference inDragRef,
                             GWorldPtr inDragGWorld,
                             RgnHandle inDragRgn,
                             EventRecord* inEvent,
                             BlugsRef inList )
```

`inDragRef`	The drag which is just beginning.
`inDragGWorld`	The `GWorld` that holds the drag image, if translucent drags are available on the user's system. Otherwise `nil`.
`inDragRgn`	A region enclosing the dragged material.
`inEvent`	The address of an Event Manager record for the event that has started the drag.
`inList`	The list in which the drag is starting.

This routine is called when a drag is starting in a list, just before Blugs calls `TrackDrag`. You can register an `InitiatingDragProc` if you need to inspect, modify, or prevent certain kinds of drag behavior. If your routine returns any value other than `noErr`, Blugs aborts and does not call `TrackDrag`.

MyDragDataProc

Supply your implementation of this routine to add cell data to a drag.

```
pascal OSErr MyDragDataProc( DragReference inDragRef,
                             DragItemRef inItem,
                             BLCell inCell,
                             BlugsRef inList )
```

`inDragRef`	The drag which is just beginning.
`inItem`	The drag item which will hold cell data.
`inCell`	The cell whose data your callback routine should add to the drag.
`inList`	The list in which the drag is starting.

This routine is called when a drag is starting in a list, just before Blugs calls `TrackDrag`. Blugs calls this routine is called once for each selected cell; your callback is responsible for adding data to the drag. If you return a result other than `noErr`, Blugs does not drag `inCell`. If your callback rejects all cells, Blugs does not call `TrackDrag`.

MyDropValidationProc

Supply your implementation of this routine to validate a drop location in a list.

```
pascal OSErr MyDropValidationProc( DragReference inDragRef,
                                   UInt16 inUnderThisRow,
                                   UInt16 inDisclosureLevel,
                                   BlugsRef inList )
```

`inDragRef`	The drag which is being tracked in `inList`.
`inUnderThisRow`	The row under which data may be dropped.
`inDisclosureLevel`	The disclosure level at which data may be dropped.
`inList`	The list in which the drag is being tracked.

Blugs calls your `DropValidationProc` as it tracks a drag in a list. Whenever Blugs determines that the drop location and/or drop disclosure level have changed, and the new target is legal, your routine is called. If you return `noErr`, Blugs draws the appropriate insertion caret to show the drop location, and an actual drop will be able to take place there. If you return any other error code, Blugs does not draw an insertion caret at the proposed location, and a drop will not be allowed there. The drag may have come from anywhere (another list, Finder, etc.) but you are dropping into `inList`. You can respond as appropriate, but do not dispose of `inDragRef`.

MyDropProc

Supply your implementation of this routine to handle a drop in a list.

```
pascal void MyDropProc( DragReference inDragRef,
                         UInt16 inUnderThisRow,
                         UInt16 inDisclosureLevel, BlugsRef inList )
```

`inDragRef`	The drag which has been dropped in `inList`.
`inUnderThisRow`	The row under which data is being dropped.
`inDisclosureLevel`	The disclosure level at which data is being dropped.
`inList`	The list in which the drop is happening.

Blugs calls your `DropProc` when it is time to drop in a list. The drag may have come from anywhere (another list, Finder, etc.) but you are dropping into `inList`. You can respond as appropriate, but do not dispose of `inDragRef`. Typically you will either add rows to `inList`, or move them around, depending on the drag's origin.

MyPostDragProc

Supply your implementation of this routine to inspect a drag just before it is disposed.

```
pascal void MyPostDragProc( DragReference inDragRef,
                              BlugsRef inList )
```

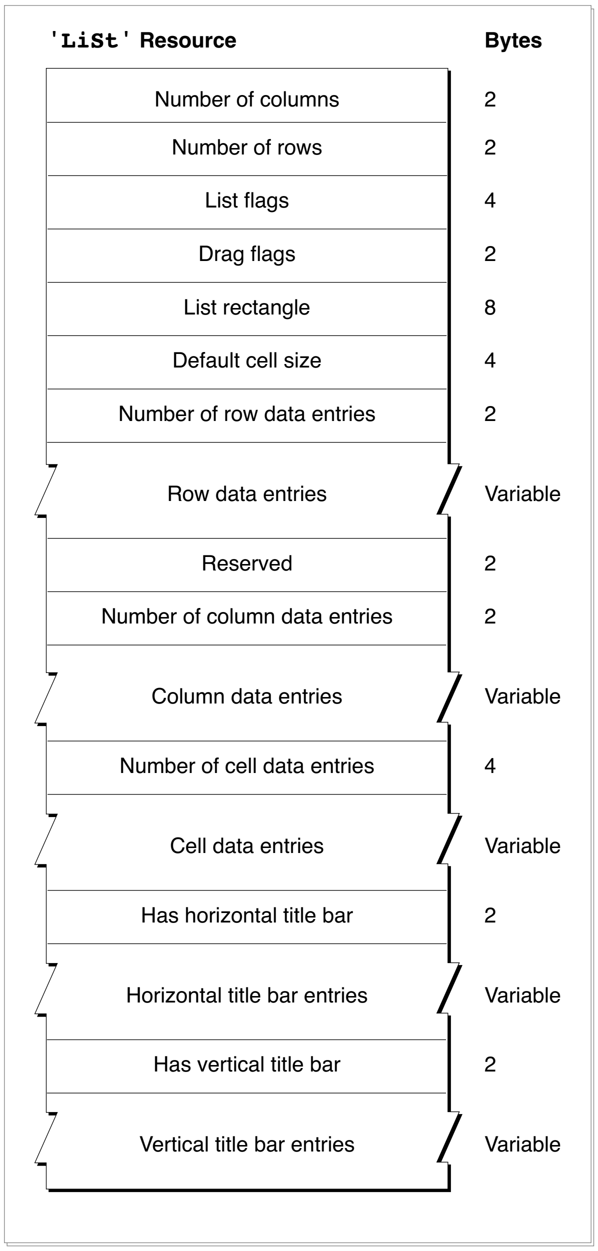
`inDragRef`	The drag which is just ending.
`inList`	The originating list.

This routine is called when a drag is ending in a list, just before Blugs calls `DisposeDrag`. You can register a `PostDragProc` if you need to inspect the drag to see if, for example, the drag ended in the Finder trash.

The `'LiSt'` Resource

You can use the `'LiSt'` resource with `BLLoad` to create and populate a list without the overhead of repeated calls to `BLAddRows`, `BLSetCell`, `BLNewTitleBar`, and so on, when you need to create a list that initially holds data. By using the `BLLoad` function you can load the data for a list and (optionally) data for individual rows, columns, cells, title bars, and titles.

Figure 7 **Structure of the `'LiSt'` resource**

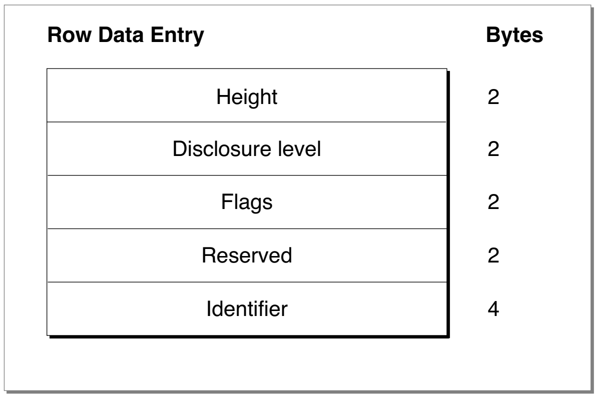


The `'LiSt'` resource consists of the following elements:

- Number of columns. The number of columns the list initially contains.
- Number of rows. The number of rows the list initially contains.
- List flags. A set of flags encoding the list's initial behavior settings. See the enumeration "List Flags" on page 20.
- Drag flags. A set of flags encoding the list's Drag and Drop behavior. See the enumeration "Drag Flags" on page 22.
- List rectangle. A Mac OS `Rect` structure with the initial list rectangle. This rectangle encloses all list elements except the list border and focus if they are present.
- Default cell size. A Mac OS `Point` structure whose horizontal component encodes the default column width, and whose vertical component encodes the list's default row height.

- ■ Number of row data entries. The number of blocks of data specific to individual rows in the list.
- ■ Row data. The number of blocks of row-specific data in the resource. Each entry contains the initial data for one row. Figure 8 shows the format of a row data entry.
- ■ Reserved. Set to zero.
- ■ Number of column data entries. The number of blocks of data specific to individual columns in the list.
- ■ Column data. Blocks of data which encode data specific to columns. Each entry contains the initial data for one column. Figure 9 shows the format of a column data entry.
- ■ Number of cell data entries. The number of blocks of data specific to individual cells in the list.
- ■ Cell data. Blocks of data which encode data specific to cells. Each entry contains the initial data for one cell. Figure 10 shows the format of a cell data entry.
- ■ Has horizontal title bar. `true` if Blugs is to load a horizontal title bar from the resource. Figure 11 shows the format of a title bar data entry.
- ■ Horizontal title bar data. This entry is skipped (is zero bytes) if the preceding field is set to `false`.
- ■ Has vertical title bar. `true` if Blugs is to load a vertical title bar from the resource. Figure 11 shows the format of a title bar data entry.
- ■ Vertical title bar data. This entry is skipped (is zero bytes) if the preceding field is set to `false`.

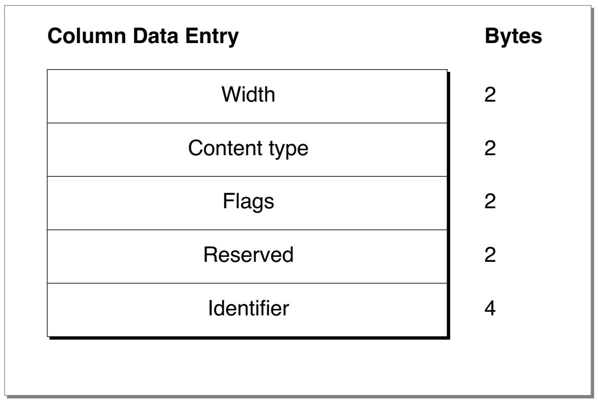
Figure 8 **Structure of a row data entry**



A row data entry encodes initial settings for one row. Each entry contains the following elements:

- Height. The row height in pixels.
- Disclosure level. The row's disclosure depth. Ignored if the list is created with the `blDisclosure` list flag cleared.
- Flags. A set of flags encoding the row's initial behavior settings. See "Row Data Flags" on page 23.
- Reserved. Set to zero.
- Identifier. A custom identifier for the row.

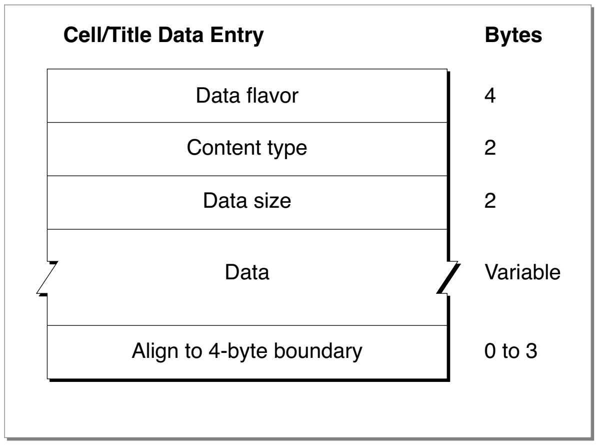
Figure 9 Structure of a column data entry



A column data entry encodes initial settings for one column. Each entry contains the following elements:

- Width. The column width in pixels.
- Content type. The content type for all cells in the column. Ignored if the list is a spreadsheet.
- Flags. A set of flags encoding the column's initial behavior settings. See "Column Data Flags" on page 23.
- Reserved. Set to zero.
- Identifier. A custom identifier for the column.

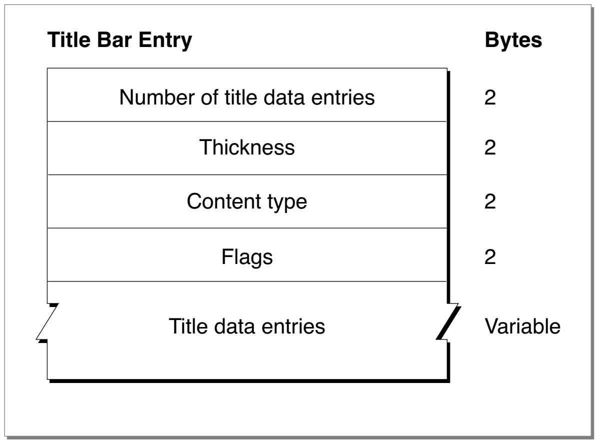
Figure 10 Structure of a cell or title data entry



A cell data entry encodes initial settings for one cell or title. Each entry contains the following elements:

- Data flavor. The flavor of the data to be installed in the cell.
- Content type. The content type for the cell or title. Ignored if the list is a table (for cells) or if the title bar has the `blTitlesOneContentType` flag bit set (for titles).
- Data size. The number of bytes of data to follow.
- Data. The data to be installed.

Figure 11 Structure of a title bar entry



- Number of title data entries. The number of title data blocks contained in the title bar data entry.
- Thickness. The number of pixels high (horizontal title bars) or wide (vertical title bars) the title is initially.
- Features. A set of flags encoding the title bar's behavior. See "Title Bar Flags" on page 24.
- Content type. The content type for all titles in the title bar. Ignored if the `blTitlesOneContentType` bit is set in the Features field is not set.
- Title data entries. Each entry contains the initial data for one title in the title bar. Figure 10 shows the format of a cell/title data entry.

Chapter 2

Content Handlers

This chapter describes the Blugs content handler architecture. You should read this chapter if you intend to modify the handlers provided with the Blugs SDK, or wish to write one from scratch.

You should be familiar with the material in Chapter 1 and thus familiar with the behavior of Blugs lists before you try to master the information to follow.

Introduction to Content Handlers

Fundamentally, content handlers provide Blugs with the code to display and allow user interaction with list data. As mentioned in the Preface, content handlers play a role similar to the List Manager's `'LDEF'` code resources. Both list definition procedures and content handlers respond to messages sent by the list engine. The most important of these messages is arguably the message which means *draw your content*. In other respects content handlers are like control definition functions. Control definitions can, for example, calculate and return a region describing the interface item; Blugs content handlers can optionally support the same kind of functionality.

The simplest content handler need only respond to four messages: get features, set, get, and draw. Typically, a handler will need to allocate storage for its contents, but this is not necessary if the handler needs to store four bytes of data or less.

What Content Handlers Can't Do

There are two important limitations on what a content handler function can do. The first limitation is on the creation of "real" Control Manager controls in cells or titles. This should be avoided. One problem with using controls is that they are by nature "owned" by the window in which they are created. As a result, the Control Manager can sidestep Blugs and draw controls at inappropriate times. An example of this is when a list that contains a control in a cell is hidden, and the application calls `DrawControls` for the list's host window. Although the list is invisible, the control is drawn. Blugs currently has no means of informing a handler that its cells are becoming invisible. Another problem scenario is when a control in a cell needs to be drawn partially obscured as a result of being scrolled partly out of view; it would be extremely difficult to manage the necessary clipping without requiring the Appearance Manager. The third problem area with controls is the issue of `GWorlds`, since Blugs draws cells into a `GWorld` and blits them to the host window. Prior to Appearance 1.0, there is no sanctioned way to draw a control in a `GWorld`. This makes it difficult for the programmer since not only is the control in the "wrong" port, it would also have to be moved and redrawn manually every time the list was scrolled. (This could require the developer to hack around Blugs and possibly call the content handler directly from the application. Needless to say, we are not in favor of such an approach.) Blugs uses "fake" controls to draw titles, grow boxes, and sort buttons. Since the Content Handler architecture allows your handler to intercept user interaction with cell contents, you can approach the power of the Control Manager with none of the aforementioned limitations.

The second major limitation on content handlers stems from the fact that the current version of Blugs is not reentrant. Under normal circumstances this is not a problem; the host application calls Blugs to possibly change the state of the list. When a callback routine like a content handler calls Blugs, however, it is possible that a change to the list's state at a deep level of the calling chain will result in invalid data at a shallower level, such as the internal Blugs routine that invoked the handler. (In a sense, Blugs would be called from two directions at once.) Also, because handlers (theoretically) have access to the entire Blugs API, a handler could call a Blugs routine that calls the same handler recursively, possibly resulting in an infinite loop. As a partial solution to the problem, Appendix A lists the Blugs routines that cannot be called from a content handler. The debug libraries enforce the restrictions by setting an internal flag when a content handler is called and asserting if anything in the restricted API is called while the flag is set. The non-debug libraries do not do this checking, so it is in your best interest to always use the debug libraries for non-distribution builds. Bear in mind that cell data is owned (at least indirectly) by the list, not the other way around; this data probably should not have the power to fundamentally and directly change the state of its container. We feel this makes a good deal of sense, but we are unable to enforce it completely.

Writing a Content Handler

The form a content handler takes resembles that of other single-entry-point multi-task routines like control definition functions. Your public routine will generally dispatch to internal routines by means of a `switch` statement on the input message.

Responding to Messages

This section discusses how your content handler should repond to each message described in the section "Content Handler Messages" on page 128.

Responding to `blHandlerInitMsg`

Blugs calls your content handler with `blHandlerInitMsg` in response to `BLRegisterContentHandler`. First, perform initialization and setup tasks, including but not limited to memory allocation and `Gestalt` checks. Second, return an integer that encodes your handler's capabilities. OR-combine zero or more of the enumerated "Content Handler Features" (page 127) and return it as your handler's function result.

Do not try to draw anything at this point. The host application is likely still in its "Display Splash Screen" stage, and there are probably no lists in existence yet anyway.

Responding to `blHandlerDeinitMsg`

Your content handler is called with `blHandlerDeinitMsg` when the host calls `BLExit`. At this point you should deallocate and deinitialize global settings. Do so under the assumption that `BLEnter` will be called again, and your handler will receive `blHandlerInitMsg` all over again. Do not assume the host application is shutting down. (Thinking of your handler in the context of a Photoshop plug-in might be helpful.)

You may not wish to deinitialize such global data as `Gestalt` results or font numbers, which are likely to be unchanged across initializations.

Responding to `blCellInitMsg`

Your handler responds to `blCellInitMsg` when its associated content type is assigned to a cell or title. This message does not correspond to any particular data assignment, but you should initialize the cell to some default empty value (such as a zero-length string). Subsequent calls with `blCellSetDataMsg` (see below) will install data. Keep in mind that in some circumstances your cell may be drawn before data is installed; your handler needs to detect the absence of data. In the example below, the handler calls `NewHandleClear` so that it can look for `nil` when called on to draw.

Blugs internally stores the `ioStorage` field of the parameter block, for subsequent use by your handler . Most handlers must allocate memory to hold cell data. Typically you will allocate a handle and pass it back to Blugs in `ioStorage`. In some circumstances you will not need to allocate: if your data is always four bytes or less you can store the value directly in the four bytes of `ioStorage`.

Example

```
void MyAllocateStorage( BLHandlerParamPtr ioParam )
{
      Handle            myStorage;

      myStorage = NewHandleClear( sizeof( MyHandlerStruct ) );
      ioParam->ioStorage = myStorage;
}
```

Responding to `blCellDeinitMsg`

Your handler responds to `blCellDeinitMsg` when a cell or title stops being associated with your content type, whether because it was deleted, or perhaps just assigned a different content type. Dispose of any memory allocated to store cell data. You do not need to set the `ioStorage` field of the parameter block to `nil`; Blugs does that automatically after calling your handler. (Yes, this means you have no choice but to dispose of allocated memory, or else leak it. When you receive `blCellDeinitMsg` you do not have the option of “hanging on to the data for a while longer.”)

If your handler supports inline editing, keep in mind that an inline session may not be terminated before your handler is called with this mesage. In other words, you may need to check your data storage and dispose of the TextEdit/MLTE/WASTE environment if there is an inline session going on when the cell is deleted.

Example

```
void MyDisposeStorage( BLHandlerParamPtr ioParam )
{
      Handle            myStorage;

      myStorage = ioParam->ioStorage;
      if (myStorage)
      {
            if ((*myStorage)->inlineTE)
                TEDispose( (*myStorage)->inlineTE );
            DisposeHandle( myStorage );
        }
    }
```

Responding to `blCellDrawMsg`

`blCellDrawMsg` is the heart and soul of your content handler. Based on the cell's data, rectangle, indent, and (possibly) selection status, your handler draws the cell in the current port. Under normal circumstances this port is a `GWorld`; however, if the cell is the target of an inline edit session the port is the host window instead. You should use `GetGWorld` and `SetGWorld` instead of `GetPort` and `SetPort`. (Mixing the two types of calls seems to cause weird results, and Inside Macontosh discourages it; Blugs always uses the `GWorld` calls.) Be sure to save and restore the current `GWorld` if you need to change ports. You need not restore the drawing state in any other way.

You do not need to erase anything before drawing. Blugs overwrites the entire cell rectangle when it draws the cell background before calling your handler.

If your handler reports that it draws its own hiliting when the cell is selected, by setting the `blWantsHilite` feature flag, then it should refer to the value contained in the `inIsSelected` field of the parameter block and draw as appropriate. Otherwise Blugs draws hiliting as part of the cell's background before your handler is called upon to draw.

Generally, your handler should draw text in the current foreground color, particularly when drawing titles. Title text should invert when the title is pressed, so Blugs sets the foreground color to an appropriate shade before calling the handler. When the list is inactive, Blugs sets the foreground to a shade appropriate for inactive text. For consistency, you should try to use these initial shades, or base your colors on the list's current state. Under normal conditions the foreground color is black. Note that a column or title bar's `ControlFontStyleRec` settings can override these colors.

Responding to `blCellSetDataMsg`

Your handler responds to `blCellSetDataMsg` by storing a copy of the passed-in data, or storing data derived from the passed-in data. You should first make sure the data is in a flavor your handler can import. Then install the data in the `ioStorage` parameter block field in an appropriate manner.

Responding to `blCellClearDataMsg`

Blugs sends `blCellClearDataMsg` when the host calls `BLClearCell`. Your handler responds by setting the cell's data to some "empty" or default value. How this is done will vary greatly with the type of data your handler stores. If text is involved, it may be set to a zero-length string, for example. You should carefully document how, and if, your handler responds to this message.

Responding to `blCellGetDataMsg`

Your handler responds to `blCellGetDataMsg` by first determining the size, in bytes, of the data in the requested flavor. Second, *provided the `ioDataBuffer` field of the parameter block is non-`nil`*, by copying data of the requested flavor into that buffer. The `ioDataSize` field indicates the maximum number of bytes that can be safely copied to this buffer. *Do* *not copy more than this amount.* If there is not enough buffer space, your handler has the option of copying nothing, or copying a truncated version of the data (this approach might be useful for strings, useless otherwise). Set the `ioDataSize` field to the actual number of bytes copied, or the total number of bytes if `ioDataBuffer` is `nil`.

Responding to `blCountImportFlavorsMsg`

Your handler responds to `blCountImportFlavorsMsg` by returning the number of data flavors that the cell in the `inCell` field can import. In many cases you can assume the number will be the same for all cells, but you are given the opportunity to determine the number on a cell-by-cell basis. Blugs sends this message when the host application requests the information.

Responding to `blCountExportFlavorsMsg`

Your handler responds to `blCountExportFlavorsMsg` by returning the number of data flavors that the cell in the `inCell` field can export. In many cases you can assume the number will be the same for all cells, but you are given the opportunity to determine the number on a cell-by-cell basis. Blugs sends this message when the host application requests the information.

Responding to `blGetIndImportFlavorInfoMsg`

Your handler responds to this message by returning the `OSType` and maximum size in bytes of the indexed (1-based) data flavor that `inCell` can import. Return the flavor type in the `ioDataFlavor` field of the parameter block, and the maximum size in the `ioDataSize` field. In many cases you can assume the size will be the same for all cells, but you are given the opportunity to determine the number on a cell-by-cell basis. Blugs sends this message when the host application requests the information.

Responding to `blGetIndExportFlavorInfoMsg`

Your handler responds to this message by returning the `OSType` and size in bytes of the indexed (1-based) data flavor that `inCell` can export. Return the flavor type in the `ioDataFlavor` field of the parameter block, and the size in the `ioDataSize` field. In many cases you can assume the size will be the same for all cells, but you are given the opportunity to determine the number on a cell-by-cell basis. Blugs sends this message when the host application requests the information.

Responding to `blInlineEditRegionMsg`

Your handler responds to `blInlineEditRegionMsg` by modifying the region passed in the `inRegion` field of the parameter block. Do not allocate a new region handle, just manipulate the one passed in. Blugs takes care of allocating and deallocating this region handle. Blugs uses the region, once you have modified it, to hit-test a cell and determine if the click should begin an inline session. If clicking anywhere in the cell will start an inline session, just call `RectRgn` on the cell's rectangle. To restrict the region to your cell's text rectangle, calculate that rectangle and call `RectRgn` on it.

Example

```
void MyGetInlineRegion( BLContentHandlerParamPtr ioParam )
{
      Rect   myTextRect;

      myTextRect = MyGetTextRect( ioParam );
      RectRgn(ioParam->inRegion, &myTextRect );
}
```

Responding to `blInlineEditBeginMsg`

Your handler responds to `blInlineEditBeginMsg` by establishing an editing environment appropriate to your content handler. Generally this will be either a TextEdit record or a WASTE reference. If you use TextEdit, you may encounter QuickDraw/TE quirks requiring you to fine-tune some metrics calculations — in the example below the TextEdit record must have its left edge shifted leftward one pixel so the text will be drawn at the same horizontal offset.

When using TextEdit, it is also recommended you set the port foreground and background colors to black and white, respectively. You should also set the background pattern to plain white (using something like `BackPat( &qd.white )`) in case the host application is running under Kaleidoscope/OS 8.5+ with a theme that uses patterns. A background pattern will prevent TextEdit from inverting colors properly when text is selected. It is recommended you do this color and pattern shuffling when receiving any message for a cell that is being edited, or is about to be. With WASTE or MLTE all this may not be necessary.

Example

```
#define kDestRectExtraRight 1000

void MyInlineEditBegin( BLHandlerParamPtr ioParam )
{
        Rect          viewRect, destRect;
        TEHandle      inlineTE;

        viewRect = MyTextRectangle( ioParam );
        // QuickDraw and TE are off by one!
        viewRect.left--;
        destRect = viewRect;
        // Make sure TE doesn't try to wrap the text to a new line.
        destRect.right += kDestRectExtraRight;
        inlineTE = TENew( &destRect, &viewRect );
        if (inlineTE)
        {
                Str31         cellString;

                // Record my editing environment in the cell's storage.
                MySetDoingInline( true, ioParam->ioStorage );
                MySetInlineTextEdit( inlineTE, ioParam->ioStorage );
                // Get cell string and initialize TE with it.
                MyGetString( ioParam->ioStorage, cellString );
                TESetText( &cellString[1], cellString[0], inlineTE );
                TEActivate( inlineTE );
                TESetSelect( 0, 32767, inlineTE );
        }
}
```

Responding to `blInlineEditKeyMsg`

Your handler responds to `blInlineEditKeyMsg` by passing a key event on to the editing environment set up for the inline edit session. Generally this will involve `TEKey` or `WEKey`. It may involve some kind of filtering to make sure the input is valid.

Example

```
Boolean MyInlineEditKey( BLHandlerParamPtr ioParam )
{
	TEHandle	inlineTE;

	inlineTE = MyGetInlineTextEdit( ioParam->ioStorage );
	if (inlineTE)
	{
		UInt8		key;

		key = ioParam->inEventMessage & charCodeMask;
		// We will not allow more than 31 characters.
		if ( MyExceedsLengthLimit( key ) ) SysBeep( 30 );
		else
		{
			Rect	oldRect, newRect;

			oldRect = MyGetInlineTextRect( ioParam );
			TEKey( key, inlineTE );
			// Resize the viewRect to hold the new character.
			newRect = MyGetInlineTextRect ( ioParam );
			// If text rectangle changed, need update.
			if ( !EqualRect( &oldRect, &newRect ) )
				return blInlineEditKeyUpdate;
		}
	}
	return blInlineEditKeyNoUpdate;
}
```

Responding to `blInlineEditClickMsg`

Your handler responds to `blInlineEditClickMsg` by passing a mouse event on to the editing environment set up for the inline edit session. Generally this will involve `TEClick` or `WEClick`.

Example

```
Boolean MyInlineEditClick( BLHandlerParamPtr ioParam )
{
	TEHandle	inlineTE;

	inlineTE = MyGetInlineTextEdit( ioParam->ioStorage );
	if (inlineTE)
	{
		Point		where;
		Rect		textRect;

		where = ioParam->inEventWhereLocal;
		textRect = MyInlineTextRect( ioParam );
		// Send to TextEdit if w/i text rectangle.
```

```
            if (PtInRect( where, &textRect ))
            {
                  TEClick( where, false, inlineTE );
                  return blInlineEditClickHandled;
            }
      }
      // Click outside text; end inline edit session.
      return blInlineEditClickNotHandled;
}
```

Responding to `blInlineEditEndMsg`

Your handler responds to `blInlineEditEndMsg` by disposing of its editing environment (via `TEDispose` or `WEDispose`) and anything else it allocated for the inline edit session. Returns one of the Boolean values `blInlineEditEndChanged` or `blInlineEditEndNotChanged` to indicate whether the contents of the cell changed as a result of the inline edit session. This return value tells Blugs whether there is a need to re-sort a sorted list after an inline session has ended.

Example

```
Boolean MyInlineEditEnd( BLHandlerParamPtr ioParam )
{
      TEHandle    inlineTE;
      Boolean     answer = blInlineEditEndNotChanged;

      inlineTE = MyGetInlineTextEdit( ioParam->ioStorage );
      MySetDoingInline( false, ioParam->ioStorage );
      if (inlineTE)
      {
            UInt8       inlineStringLength;
            Str31       string;

            // Only count the first 31 characters.
            inlineStringLength = (*inlineTE)->teLength;
            if (inlineStringLength > 31) inlineStringLength = 31;
            // Get the normal cell string.
            MyGetString( ioParam, string );
            // Check to see if the text changed.
            if (IdenticalText( *(*inlineTE)->hText, &string[1]),
                              inlineStringLength, string[0], nil ))
            {
                  // TE contents different from cell storage.
                  answer = blInlineEditEndChanged;
            }
            // Copy the inline data to normal storage.
            MySetCellFromTE( inlineTE, ioParam );
            TEDispose( inlineTE );
            MySetInlineTextEdit( nil, ioParam->ioStorage );
      }
      return answer;
}
```

Responding to `blCellRegionMsg`

Your handler responds to the optional `blCellRegionMsg` by modifying the region passed in the `inRegion` field of the parameter block. Do not allocate a new region handle, just manipulate the one passed in. Blugs takes care of allocating and deallocating this region handle. Blugs uses the region, once you have modified it, to hit-test a cell and hilite the cell's contents. If clicking anywhere in the cell will select it, and if selecting the cell hilites the entire cell, just call `RectRgn` on the cell's rectangle. (Or, better yet, do not set the `blWantsRegion` flag when reporting handler features so Blugs can do this for you.) To restrict the region to your cell's text rectangle, calculate that rectangle and call `RectRgn` on it.

Blugs always intersects this region with the cell's rectangle, so you can safely report a region outside the cell rectangle.

Example

```
void MyGetCellRegion( BLContentHandlerParamPtr ioParam )
{
      Rect  myTextRect;

      myTextRect = MyGetTextRect( param->inRect );
      RectRgn( param->inRegion, &myTextRect );
}
```

Responding to `blCellClickMsg`

Your handler responds to the optional `blCellClickMsg` by reacting to the mouse event reported in the `inEventWhereLocal` field of the parameter block. Further interaction is possible if your handler runs some kind of a `while (StillDown())` loop and tracks mouse movements. You cannot call out to Blugs to redraw your cell and blit it to screen, but Blugs sets the current graphics world to the host window before calling your handler with this message, so you can safely redraw as necessary while running this mini-event loop. When finished, you can request that Blugs call your handler with `blCellDrawMsg` immediately, so you can make sure the state of the cell is in sync with its screen representation.

If something potentially interesting to the host application happened as a result of this click, you can pass a notification command back in `outCommand`; Blugs will send this to the host's notification callback if one is installed.

Example

```
// Handle a click in our cell's "doodad" (a control-like object of
//    sublime appearance that accomplishes wondrous things when
//    clicked).
UInt32 MyCellClick( BLContentHandlerParamPtr ioParam )
{
      Rect        doodadRect;
      Boolean     wasIn = false;
      UInt32      response = 0;

      MyDoodadRect( ioParam, &doodadRect );
      while (StillDown())
      {
            Point       mouse;
```

```
                        GetMouse( &mouse );
                        // If mouse either entered or left…
                        if (PtInRect( mouse, &doodadRect) != wasIn)
                        {
                                wasIn = !wasIn;
                                MyDraw( ioParam, (wasIn) ? kThemeStatePressed :
                                                                kThemeStateActive );
                                // Click was definitely in doodad.
                                response |= blCellClickHandled;
                        }
                }
                // Mouse is released…
                if (wasIn)
                {
                        MyDoSomethingWondrous();
                        // Send a command to the app to report that something
                        //      wondrous happened.
                        ioParam->outCommand = 'Wow!';
                        // We last drew kThemeStatePressed; need update.
                        response |= blCellClickUpdate;
                }
                return response;
        }
```

Responding to `blCellIdleMsg`

Your handler responds to the optional `blCellIdleMsg` by reacting to the cursor position: this message is only sent when the cursor is within your cell's rectangle. The `inEventWhereLocal` field of the parameter block contains the mouse location in local coordinates. After calling your handler with this message Blugs adjusts the cursor unless your return value has the `blCellIdleAdjustedCursor` bit set. That is how you can prevent Blugs from automatically setting the cursor to the standard QuickDraw arrow.

When this message is sent, the port will already be set to the host window.

Responding to `blCellMinSizeMsg`

Your handler responds to the optional `blCellMinSizeMsg` by reporting minimum cell height and width. These minima apply to all cells of your handler's associated content type. You will only receive this message if the `blWantsMin` feature flag is set. You report these minima in the `outMinWidth` and `outMinHeight` fields of the parameter block. If your handler does not enforce a minimum in a dimension, set the corresponding field to zero.

Example

```
// Report to Blugs that cells have to be at least 100 pixels wide.
void MyGetMin( BLHandlerParamPtr param )
{
        param->outMinWidth = 100;
        param->outMinHeight = 0; // Who cares how high.
}
```

Responding to `blSortMsg`

Your handler responds to the optional `blSortMsg` by returning the result of comparison between the data passed in the `ioStorage` and `inStorage2` fields of the parameter block. You return one of the values `blSortFirstIsLess`, `blSortEqual`, or `blSortFirstIsGreater` to indicate how the first data (in `ioStorage`) compares with the second data item (`inStorage2`). See "Sort Replies" on page 130.The data storage fields will both be from cells of your content handler's content type.

Often you will sort text; you can use a Mac OS text utility routine to do so. You may also need to sort other types of data, such as numbers and dates. You should sort using the main data item your cell holds. (If your cell data contains a string and a text color for displaying the string, it's probably not a good idea to sort by the color value.)

Example

```
SInt32 MySort( BLHandlerParamPtr param )
{
        Str255        storageString, storage2String;
        SInt16        result;
        SInt32        longResult;

        // Extract text from my storage areas.
        MyGetStorage1String( param, storageString );
        MyGetStorage2String( param, storage2String );
        // Compare the strings.
        result = CompareString( storageString, storage2String, nil );
        // Blugs expects 32-bit result codes!
        longResult = result;
        return longResult;
}
```

TextEdit Issues

Whenever there is an inline edit session in progress with your handler involved, you should make sure to check and update the cell rectangle before drawing the text field. If a list scrolls while an inline edit is happening, the TextEdit environment must scroll along with it. This means updating the TextEdit record's `destRect` and `viewRect` before calling `TEUpdate` whenever you are called with `blDrawMsg`.

When drawing, especially with a TextEdit environment, be sure to honor the clipping region set up by Blugs. Your cell or edit field may be partially clipped if partially scrolled out of view, under a title bar or whatever. If you need to adjust clipping for your own drawing needs, make sure to do something like call `GetClip`, then `SectRgn` to intersect the old region with whatever clipping you need, then `SetClip` to restore everything the way it was. That way you are maximally conservative about where you allow drawing.

Appearance Themes and TextEdit

TextEdit is not inherently theme-savvy as it was written long before Themes were invented. The problem is one of hiliting text to show selection, this hiliting does not show up correctly on patterned themes because there is a background pattern overriding the background color. Content handlers that use TextEdit need to be built with these issues in mind since Themes will probably continue to grow in popularity.

The way to fix a content handler so it behaves properly in the world of Themes, is to do exactly what Apple recommends in their Appearance documentation and headers: save the port's background pattern and set it to normal white. When TextEdit hilites it will display correctly. When the handler exits, restore the port's background pattern. The example code shows the handler getting the desired white pattern from QuickDraw globals. (This is not Carbon-savvy code.)

Example

```
pascal UInt32 MyHandler( BLMessage inMessage,
                         BLHandlerParamPtr ioParam )
{
      UInt32              answer = 0;
      PixPatHandle        saveBkPixPat;
      CGrafPtr            port;

      if (inMessage >= blInlineEditBeginMsg &&
            inMessage <= blInlineEditEndMsg)
      {
            GetPort( (GrafPtr*)&port );
            saveBkPixPat = port->bkPixPat;
            BackPat( &qd.white );
      }
      switch (inMessage)
      { } // Do lots of stuff…

      if (inMessage >= blInlineEditBeginMsg &&
            inMessage <= blInlineEditEndMsg)
      {
            port->bkPixPat = saveBkPixPat;
      }
      return answer;
}
```

Summary of Content Handler Parameters

This section summarizes whether the pointer to the handler parameter block is valid and, if so, which of the parameter block fields are valid for each content handler message. The first two messages take `nil` instead of a parameter pointer. That's why the field cells are blacked out. For all other messages the parameter pointer is valid. Valid input fields for those messages are indicated by a bullet (•). Valid output fields (fields in which you pass data out but contain garbage on input) are indicated by a lower-case "o".

	`inCell`	`inRect`	`ioStorage`	`inStorage2`	`ioDataBuffer`	`ioDataSize`	`ioDataFlavor`	`inRegion`	`inList`
`blHandlerInitMsg`	■	■	■	■	■	■	■	■	■
`blHandlerDeinitMsg`	■	■	■	■	■	■	■	■	■
`blCellInitMsg`	•		•						•
`blCellDeinitMsg`	•		•						•
`blCellDrawMsg`	•	•	•						•
`blCellSetDataMsg`	•	•	•		•	•	•		•
`blCellClearDataMsg`	•		•						•
`blCellGetDataMsg`	•		•		•	•	•		•
`blCountImportFlavorsMsg`	•		•						•
`blCountExportFlavorsMsg`	•		•						•
`blGetIndImportFlavorInfoMsg`	•		•			o	o		•
`blGetIndExportFlavorInfoMsg`	•		•			o	o		•
`blInlineEditRegionMsg`	•	•	•					•	•
`blInlineEditKeyMsg`	•	•	•						•
`blInlineEditClickMsg`	•	•	•						•
`blInlineEditEndMsg`	•	•	•						•
`blCellRegionMsg`	•	•	•						•
`blCellClickMsg`	•	•	•						•
`blCellIdleMsg`	•	•	•						•
`blCellMinSizeMsg`	•		•						•
`blCellSortMsg`	•		•	•					•
`blCellSearchMsg`	•		•		•	•	•		•

	inEventWhereLocal	inEventMessage	inEventModifiers	outCommand	inIndex	inIndent	outMinWidth	outMinHeight	inVariationCode	inIsSelected
blHandlerInitMsg	■	■	■	■	■	■	■	■	■	■
blHandlerDeinitMsg	■	■	■	■	■	■	■	■	■	■
blCellInitMsg									●	
blCellDeinitMsg									●	
blCellDrawMsg						●			●	●
blCellSetDataMsg						●			●	
blCellClearDataMsg									●	
blCellGetDataMsg									●	
blCountImportFlavorsMsg									●	
blCountExportFlavorsMsg									●	
blGetIndImportFlavorInfoMsg					●				●	
blGetIndExportFlavorInfoMsg					●				●	
blInlineEditRegionMsg						●			●	
blInlineEditKeyMsg		●	●			●			●	
blInlineEditClickMsg	●		●			●			●	
blInlineEditEndMsg						●			●	
blCellRegionMsg						●			●	●
blCellClickMsg	●		●	o		●			●	
blCellIdleMsg	●					●			●	
blCellMinSizeMsg							o	o	●	
blCellSortMsg									●	
blCellSearchMsg									●	

Content Handler Reference

Types and Constants

This section describes the types and constants specific to the Blugs Content Handler Architecture.

Content Handler Features

Your content handler returns the result of ORing zero or more of these flags when it is first initialized with `blHandlerInitMsg`. You use these flags to report your handler's special capabilities, if any.

```
enum
{
      blWantsClick            = 0x00000001,
      blWantsIdle             = 0x00000002,
      blWantsInlineEdit       = 0x00000004,
      blWantsInlineReturn     = 0x00000008,
      blWantsInlineFocus      = 0x00000010,
      blWantsSort             = 0x00000020,
      blWantsMin              = 0x00000040,
      blWantsRegion           = 0x00000080,
      blWantsHilite           = 0x00000100
};
```

Constant Descriptions

`blWantsClick`	Set to have Blugs call the handler with `blCellClickMsg` when the user clicks in the cell's rectangle or (if the handler supports it) the cell's content region. Handler can run its own mini event loop to respond to subsequent dragging and other interaction.
`blWantsIdle`	Set if the handler wants idle events (passed to Blugs via `BLIdle`) passed on to the content handler when the cursor is over the cell.
`blWantsInlineEdit`	Set if handler can do inline text editing.
`blWantsInlineReturn`	Set if return and enter keys should be sent to the handler during inline edit sessions. By default, return and enter end the session. Set this bit to support multi-line edit fields.
`blWantsInlineFocus`	Set if Blugs should draw a focus box around inline edit text. Ignored if the handler does not support inline editing.
`blWantsSort`	Set if handler can do data comparison.
`blWantsMin`	Set if handler can return a minimum height and/or width for cells.
`blWantsRegion`	Set if the handler calculates a region for hit-testing, hiliting, and drag-selection that is not the same as the cell rectangle. Blugs sends `blCellRegionMsg` only if this bit is set.
`blWantsHilite`	Set if the handler does its own hiliting when drawing a selected cell.

Content Handler Messages

Below are all of the messages Blugs may send in calling your content handler. The message determines what action your content handler should take, and which fields of the Content Handler Parameter Block are valid.

```
typedef UInt16          BLMessage;
enum
{
      /* These are the required messages: */
      blHandlerInitMsg,
      blHandlerDeinitMsg,
      blCellInitMsg,
      blCellDeinitMsg,
      blCellDrawMsg,
      blCellSetDataMsg,
      blCellClearDataMsg,
      blCellGetDataMsg,
      blCountImportFlavorsMsg,
      blCountExportFlavorsMsg,
      blGetIndImportFlavorInfoMsg,
      blGetIndExportFlavorInfoMsg,
      /* These 5 are only sent if blWantsInlineEdit flag is set: */
      blInlineEditRegionMsg,
      blInlineEditBeginMsg,
      blInlineEditKeyMsg,
      blInlineEditClickMsg,
      blInlineEditEndMsg,
      /* End of inline edit stuff */
      /* These are optional: */
      blCellRegionMsg,
      blCellClickMsg,
      blCellIdleMsg,
      blCellMinSizeMsg,
      blCellSortMsg,
      blCellSearchMsg
};
```

Constant Descriptions

`blHandlerInitMsg`	Sent when the host application calls `BLRegisterContentHandler`. Allocate any needed memory, do setup tasks, and return handler features.
`blHandlerDeinitMsg`	Sent when the host application calls `BLExit`. Deallocate any global memory.
`blCellInitMsg`	Sent when a cell is created. Handler should allocate storage for future data if necessary and initialize as appropriate.
`blCellDeinitMsg`	Sent when a cell is removed from the list. Handler must release memory allocated for that cell.
`blCellDrawMsg`	Draw content in the current port.
`blCellSetDataMsg`	Stores data of the specified flavor in a cell.
`blCellClearDataMsg`	Reset a cell to its default or initial data content.
`blCellGetDataMsg`	Gets cell data of the specified flavor.
`blCountImportFlavorsMsg`	Return the number of data flavors the cell can import.
`blCountExportFlavorsMsg`	Return the number of data flavors the cell can export.
`blGetIndImportFlavorInfoMsg`	Get `OSType` and maximum size for the indexed (1-based) data import flavor.
`blGetIndExportFlavorInfoMsg`	Get `OSType` and maximum size for the indexed (1-based) data export flavor.
`blInlineEditRegionMsg`	Only sent if `blWantsInlineEdit` feature flag is set. Blugs asks for a region, and uses that for hit-testing to determine whether a click starts an inline edit session. It also uses the region for drawing the focus around inline text. Blugs allocates and disposes of this region handle.
`blInlineEditBeginMsg`	Only sent if `blWantsInlineEdit` feature flag is set. An inline edit session has begun. Allocate a TextEdit/WASTE/MLTE environment in the current port to handle subsequent inline edit events.
`blInlineEditKeyMsg`	Only sent if `blWantsInlineEdit` feature flag is set. Handle a key event during an inline edit.
`blInlineEditClickMsg`	Only sent if `blWantsInlineEdit` feature flag is set. Handle a click in the inline edit cell.
`blInlineEditEndMsg`	Only sent if `blWantsInlineEdit` feature flag is set. Edit session is over. Update cell contents if appropriate and

	dispose of TextEdit/WASTE/MLTE environment. See "Other Handler Replies" (below) for the `blInlineEditEndChanged` and `blInlineEditEndNotChanged` constants that your handler can return.
`blCellRegionMsg`	Only sent if `blWantsRegion` feature flag is set. Blugs asks for a region, and uses that for hit-testing, hiliting the cell contents, and drag-selecting cells. Blugs allocates and disposes of this region handle.
`blCellClickMsg`	Only sent if `blWantsClick` feature flag is set. Content handler can run a sort of event loop based on mouse movement and redraw as necessary to allow the user to manipulate the cell.
`blCellIdleMsg`	Only sent if `blWantsIdle` feature flag is set. Sent when the host application calls `BLIdle` and the cursor is over the cell or title. See "Other Handler Replies" (below) for the `blCellIdleAdjustedCursor` and `blCellIdleUpdate` flags that your handler can return.
`blCellMinSizeMsg`	Only sent if `blWantsMin` feature flag is set; minima apply to all cells of your handler's type. Return minima in `outMinWidth` and `outMinHeight` fields of the parameter block. Set to zero to indicate no minimum in that dimension.
`blSortMsg`	Only sent if `blWantsSort` feature flag is set. Handler compares data elements passed in the `ioStorage` and `inStorage2` fields of the parameter block. See "Sort Replies" (below) for the appropriate return values.
`blSearchMsg`	Only sent if `blWantsSort` feature flag is set. Handler compares its data with the data passed in the parameter block. See "Search Replies" (below) for the appropriate return values.

Sort Replies

Your content handler returns one of these constants when called with `blCellSortMsg`. These values indicate the relationship between the data referenced by the storage areas `ioStorage` and `inStorage2` passed in the parameter block.

```
enum
{
      blSortFirstIsLess         = -1L,
      blSortEqual               = 0,
      blSortFirstIsGreater      = 1
};
```

Constant Descriptions

`blSortFirstIsLess`	`ioStorage` is less than `inStorage2`.
`blSortEqual`	`ioStorage` and `inStorage2` are equal.
`blSortFirstIsGreater`	`ioStorage` is greater than `inStorage2`.

Search Replies

Your content handler returns one of these constants in response to `blCellSearchMsg`. These values indicate the relationship between the cell data and the data passed to the handler.

```
enum
{
      blSearchCellDataIsLess         = -1L,
      blSearchEqual                  = 0,
      blSearchCellDataIsGreater      = 1
};
```

Constant Descriptions

`blSearchCellDataIsLess`	Cell data is less than data passed to handler.
`blSearchEqual`	Cell data and search data are equal.
`blSearchCellDataIsGreater`	Cell data is greater than data passed to handler.

Other Handler Replies

These are miscellaneous constants returned in response to `blCellDrawMsg`, `blInlineEditKeyMsg`, `blInlineEditClickMsg`, `blInlineEditEndMsg`, `blCellClickMsg`, and `blCellIdleMsg`. These values specify additional tasks that Blugs should or should not do after calling the handler.

```
enum
{
      blInlineEditKeyUpdate          = true,
      blInlineEditKeyNoUpdate        = false,
      blInlineEditClickHandled       = true,
      blInlineEditClickNotHandled    = false,
      blInlineEditEndChanged         = true,
      blInlineEditEndNotChanged      = false,
      blCellClickUpdate              = 1 << 0,
      blCellClickHandled             = 1 << 1,
      blCellIdleAdjustedCursor       = 1 << 0,
      blCellIdleUpdate               = 1 << 1
};
```

Constant Descriptions

`blInlineEditKeyUpdate`	Blugs needs to call the handler with `blCellDrawMsg`.
`blInlineEditKeyNoUpdate`	No need for `blCellDrawMsg`.
`blInlineEditClickHandled`	Handler processed the inline edit click.
`blInlineEditClickNotHandled`	Handler did not process the click.
`blInlineEditEndChanged`	The cell's contents changed as a result of the inline edit.
`blInlineEditEndNotChanged`	The cells contents were unchanged after the inline edit ended.
`blCellClickUpdate`	Blugs needs to call the handler with `blCellDrawMsg` after handling click.
`blCellClickHandled`	Blugs should leave cell selection as it was before: the click was strictly for manipulating something in the cell, not for selecting and deselecting list items.
`blCellIdleAdjustedCursor`	Handler adjusted cursor; Blugs should not adjust the cursor.
`blCellIdleUpdate`	Blugs needs to call the handler with `blCellDrawMsg`.

Content Handler Parameter Block

When Blugs calls a content handler, it passes the address of a parameter block of this type. Generally this parameter block is used to pass data to the handler, but in many cases its fields are also used to pass data from the handler back to Blugs.

```
typedef struct
{
      BLCell                  inCell;
      Rect                    inRect;
      Handle                  ioStorage;
      Handle                  inStorage2;
      Ptr                     ioDataBuffer;
      UInt32                  ioDataSize;
      OSType                  ioDataFlavor;
      RgnHandle               inRegion;
      BlugsRef                inList;
      Point                   inEventWhereLocal;
      UInt32                  inEventMessage;
      EventModifiers          inEventModifiers;
      BLNotificationCommand   outCommand;
      UInt16                  inIndex;
      UInt16                  inIndent;
      UInt16                  outMinWidth;
      UInt16                  outMinHeight;
      UInt16                  inVariationCode;
      Boolean                 inIsSelected;
} BLHandlerParamBlock, *BLHandlerParamPtr;
```

Field descriptions

`inCell`	The cell (or title) upon which the handler is to operate.
`inRect`	The cell rectangle, in local coordinates.
`ioStorage`	A data area the handler can use to store cell data (typically via handle).
`inStorage2`	Like `ioStorage`, except it is only used when the handler is called with `blCellSortMsg`. The handler interprets the two data storage areas and returns a value indicating which should come first. Note that `inStorage2` is *only* valid for `blCellSortMsg`; its contents are undefined otherwise.
`ioDataBuffer`	A pointer to a buffer of memory used to hold data being passed to or from the content handler.
`ioDataSize`	The size of the data buffer. This represents the size of data being passed to the handler, or the maximum number of bytes that can be passed out from it. You return an actual byte count in this field when exporting data.
`ioDataFlavor`	A four-character code for the data passed in or requested.
`inRegion`	A region handle which you can adjust to reflect your cell content region. Modify the existing region; do not allocate a new region (that is, do not call `NewRgn`).
`inList`	The Blugs list whose cells your handler is operating on.
`inEventWhereLocal`	The location of a click event, in coordinates local to the host window.
`inEventMessage`	The `message` field of an `EventRecord` your content handler can respond to. (Typically a keyboard event.)
`inEventModifiers`	The `modifiers` field of an `EventRecord` your content handler can respond to. (Typically a keyboard event.)

`outCommand`	A command reported back to the host application by means of a notification.
`inIndex`	Used when Blugs requests a list of data flavors by calling your handler repeatedly. You return the (1-based) indexed flavor the cell can export.
`inIndent`	The number of pixels from the left edge of the cell's rectangle to indent cell content when drawing.
`outMinWidth`	The minimum width of your handler's cells or titles. Return zero to indicate there is no minimum in this dimension.
`outMinHeight`	The minimum height of your handler's cells or titles. Return zero to indicate there is no minimum in this dimension.
`inVariationCode`	A host application-supplied value associated with the handler on a global level. Your handler may define special meanings for this variation code, or ignore it.
`inIsSelected`	`true` if the cell or title is selected.

Content Handler Routine

The one routine in the Blugs content handler architecture is the user-defined routine `MyContentHandler`. This is the prototype for every content handler you will write.

MyContentHandler

Supply your version of this routine to handle cell and title content.

```
UInt32 MyContentHandler( BLContentHandlerMessage inMessage,
                         BLContentHandlerParamPtr ioParam )
```

`inMessage`	A content handler message.
`ioParam`	The address of a `BLContentHandlerParamBlock` structure holding the parameters appropriate to `inMessage`.

Appendix A

Restricted API

This appendix enumerates the Blugs routines that, because of reentrancy issues, are deemed too dangerous to call from within a content handler. As discussed in the section “What Content Handlers Can’t Do” on page 114, certain Blugs routines can result in stale data on the stack when called by content handlers, or could call the same handler recursively, possibly resulting in an infinite loop.

What follows are the Blugs API routines that can change the numbering of rows, columns, or cells, or invoke a content handler, or in general cause bad results when called by a content handler.

The debug libraries will alert you if you try to call one of these routines from a content handler. Furthermore, the routines will return immediately without executing, typically returning `paramErr`. The non-debug builds will execute without warning, so beware!

```
BLExit
BLNew
BLLoad
BLUnflatten
BLDispose
BLAddRows
BLAddColumns
BLDeleteRows
BLDeleteColumns
BLSetRowFlags
BLSetColumnFlags
BLMoveRows
BLMoveColumns
BLMoveMarkedRows
BLSetRect
BLSetRowHeight
BLSetColumnWidth
BLSetIndent
BLClick
BLKey
BLIdle
BLSelectCell
BLSelectOneCell
BLDeselectCell
BLDeselectAll
BLTrackDrag
BLReceiveDrag
BLSetActive
BLMakeUserPaneControl
BLDisposeUserPaneControl
BLBeginInlineEdit
BLEndInlineEdit
BLSetCellContentType
BLGetCellData
BLSetCellData
BLClearCell
BLSort
BLSearch
BLExpandRow
BLCollapseRow
BLNewTitleBar
BLSetVisible
BLPageUp
BLPageDown
```

Appendix B

Migration

This appendix attempts to help ease migration from the Mac OS List Manager to Blugs. Because Blugs is so different from the List Manager, it has been impossible to formulate a parallel API. In some ways Blugs has been designed to follow the Apple's API, for example, in parameter ordering — the list parameter is always last. Yet in most cases adopters must deal with additional functionality, additional parameters, restricted access to data, etc.

Classic API Comparison

This section compares the pre-Carbon List Manager with Blugs.

Routine	**Comments**
`LNew`	Substantially different from `BLNew`. Use caution.
`LAddColumn`, `LAddRow`	Note plural in the routine names, which is more accurate. Blugs returns an error code; LM returns the first added element. The latter information can be derived in the (I expect) rare case when it is needed. The disclosure option parameter for `BLAddRows` is only needed in some casees, so we provide the `BLAddRowsCompat` macro that resolves to `BLAddRows` with the `blDisclosureOptionSame` constant.
`LGetSelect`	Use `BLIsCellSelected` to query a single cell. Use `BLGetSelect` to query a range of cells.
`LLastClick`	Currently we have no comparable routine. Is a routine like this necessary? We plan to introduce better click event reporting, so in the future it definitely won't be necessary. For now, we have a field in the list record called `lastClickCell` that could be made available with an accessor `BLLastClick`.
`LNextCell`	Currently we have no comparable routine. Identical functionality can be made available if desired.
`LSearch`	`BLSearch` is different because it searches a single column only. To search all cells, it would be necessary to use brute-force. In the future, it's possible `BLSearch` will become more like `LSearch`. Less automation but more control for the app. Still, it will likely remain single-column-based.
`LSize`	`BLSetRect` is more flexible because it allows the list to be moved. But we could offer a `BLSize` or `BLSetSize` routine that keeps the upper left corner intact.
`LSetDrawingMode`, `LDoDraw`	Equivalent to `BLSetAutodraw`. The macros `BLSetDrawingMode` and `BLDoDraw` resolve to `BLSetAutodraw`.
`LScroll`	A problem since with variable row/column metrics Blugs uses

	pixel offsets. If we add a `BLScroll` routine, it will have to take pixels, not row and column counts.
`LAutoScroll`	This is accomplished by getting the first selected cell using `BLGetSelect`, and then calling `BLMakeVisible`.
`LActivate`	The `BLActivate` macro resolves to `BLSetActive`.
`LCellSize`	To achieve this functionality, call `BLSetRowHeight` once for each row, and `BLSetColumnWidth` for each column. (Use `BLCountRows` and `BLCountColumns` to get the data bounds.)
`LClick`	We have to take more parameters. And we will probably make the call even more elaborate in the future.
`LAddToCell`	Blugs has no comparable routine. Use `BLSetCellData`.
`LClrCell`	The `BLClrCell` macro resolves to `BLClearCell`.
`LGetCell`, `LSetCell`	The additional `inDataFlavor` param is required. We may be able to relocate the `inCell` parameter immediately before the `BlugsRef` to make it more like the LM calls.
`LRect`	`BLCellRect`
`LSetSelect`	Equivalent to `BLSetSelect`.
`LDraw`	Would this ever be needed?
`LGetCellDataLocation`	The host app is forbidden access to this information. Use `BLGetCellData`.

Carbon API Comparison

This section compares the Carbon List Manager with Blugs. The Carbon API adds accessor functions.

Routine	**Comments**
`GetListViewBounds`	Our rects are different, but we have the ability to get rect of whole list or cell area.
`GetListPort`	We may want to provide this, though it should be called `BLGetWindow`.
`GetListCellIndent`	Blugs indent is horizontal-only, we've never addressed whether we want or need vertical indentation. Probably not, given variable row heights and content handler diversity.
`GetListCellSize`	We may want to provide this, but I don't know how useful it would be. It refers to the default cell size.
`GetListVisibleCells`	Candidate for inclusion.
`GetListVerticalScrollBar`, `GetListHorizontalScrollBar`	Absolutely not!
`GetListActive`	Same as `BLIsActive`. The macro `BLGetListActive` resolves to `BLIsActive`.
`GetListClickTime`, `GetListClickLocation`	When `BLClick` has better event reporting, this kind of information will be returned. Probably not stored in the list.
`GetListMouseLocation`	I don't know what this is for!

`GetListRefCon`,
`GetListUserHandle`,
`SetListRefCon`,
`SetListUserHandle`
	Try something like this:
	
	`#define blListRefConKey        'refc'`
	`#define blListUserHandleKey    'uhnd'`
	`SInt32 BLGetListRefCon( BlugsRef inList )`
	`{`
	`      OSErr       err;`
	`      SInt32      data = nil;`
	`      err = BLGetUserData( blListRefConKey,`

	`&data, inList );`
	`return data;`
	`}`

`GetListDataBounds`	Try something like this:
	
	`void BLGetListDataBounds( Rect* outBounds,`
	`BlugsRef inList )`
	`{`
	`outBounds->top = 0;`
	`outBounds->left = 0;`
	`outBounds->bottom =`
	`(SInt16)BLCountColumns( inList );`
	`outBounds->right =`
	`(SInt16)BLCountRows( inList );`
	`}`
	
	Or this…
	
	`#define BLGetListDataBounds(rPtr,list) \`
	`SetRect( rPtr, 0, 0, \`
	`(SInt16)BLCountRows( inList ), \`
	`(SInt16)BLCountColumns( inList ) )`

`GetListDataHandle`	No equivalent.
`GetListFlags`, `GetListSelectionFlags`	Maybe; they're a single unit in Blugs, though.
`SetListViewBounds`	Maybe provide a macro to `BLSetRect`.
`SetListPort`	Interesting idea. Bad idea. (Remember those scroll bars?) The closest we could come to that is something like `BLUpdateInCurrentPort`.
`SetListCellIndent`	Maybe provide a macro to `BLSetIndent`. See notes on `GetListCellIndent`.
`SetListClickTime`, `SetListClickLoop`, `SetListLastClick`, `SetListFlags`, `SetListSelectionFlags`	Nope. Maybe later.

Compatibility Macros

These macros allow you to use some names that are a little more List Manager -like. You can find them at the end of `Blugs.h`.

Macro	Resolves to
`BLSetDrawingMode`	`BLSetAutodraw`
`BLDoDraw`	`BLSetAutodraw`
`BLAddRowsCompat(count,row,list)`	`BLAddRows(blDisclosureOptionRoot, count,row,list)`
`BLActivate`	`BLSetActive`
`BLGetListActive`	`BLIsActive`
`BLClrCell`	`BLClearCell`

Appendix C

Blugs FAQ

What follows is a more informal discussion about the Blugs project. (Note that I made up ~~some~~ most of the questions myself.)

Blugs FAQ

What is the history of Blugs? How did Blugs come about?

Blugs is an amalgam of development projects. The most primitive skeleton of Blugs was begun in late 1997 by Moses Hall to provide list management for linguistic database applications. The earliest Blugs versions included a number of rudimentary built-in content handlers, and in general bore little resemblance to Blugs as it is today. When Kyle Hammond and Moses decided join forces, Kyle's A List served as a more mature code base for extending Blugs.

So Blugs is a commercialized version of the A List?

No, although there is a certain amount of A List code in Blugs. Very little of the original Blugs code still exists in the release version, and very little of the A List code has been imported intact. Kyle wrote almost all of the title drag-reordering code. (One of the reasons it works so well!) And most of the start-drag logic is his. I went in and did a lot of the track-content-drag code later on. A lot of the basic scrolling, `GWorld`, blitting nitty-gritty code is borrowed from the A List originally. Kyle also did a lot of of the early work on Carbonization since I was clueless about Carbon. (OK, no jokes about how little times have changed.)

How does Blugs compare with StoneTable?

I (obviously) cannot offer an unbiased comparison. I have not used StoneTable, although it has gotten a favorable review in MacTech magazine. Blugs has certain capabilities StoneTable does not have (at time of writing), like the ability to sort a disclosure list and preserve hierarchy. Based only on the StoneTable demo, I feel Blugs has a dramatically better user interface. Also, Blugs does not use code resources at all. On the other hand, StoneTable is a mature product with many users. StoneTable also has a PowerPlant interface, which Blugs does not. Blugs is fairly generic, where StoneTable seems to be targeted more at true spreadsheet applications. StoneTable is patterned after the List Manager API to make the transition easier, where Blugs does not offer an easy transition but rather concentrates on flexibility. It appears that Stone Table is designed to work more like Excel, whereas Blugs mimics the Finder in many ways.

I hope that an interested third party will write a critical comparison of Blugs and StoneTable.

How does Blugs compare with DataBrowser?

Again, I can't offer an unbiased comparison. Blugs is more flexible in many respects. But DataBrowser is tightly integrated into OS X and the later CarbonLib versions and it's free. On the other hand, Blugs supports legacy machines going way back. DataBrowser has a history of being incredibly buggy, sort of like Nav Services 1.0.

Again, I hope that a third party will write a critical comparison.

How will Blugs change in the future?

We can't say for sure, but one thing we want to accomplish is greater reentrancy. Ideally, we should see Appendix A's restricted API dwindle down to nothing. The UID mechanism is the first big step in that direction.

Some sophisticated table-management products (like AreaList Pro for 4D) allow for "lockers", or rows/columns that remain visible when the main list view is scrolled. You could call this a "split pane." We will implement this in a future version, since we know (from experience with CodeWarrior) a valuable feature when we see one.

We will *definitely* migrate from the current list resource format to a more flexible Collection/Stockpile format.

The `BLCell` type is becoming seriously overloaded (with the introduction of scroll bar widgets) and in the future APIs will most likely start to access list parts by `BLPart` and `BLCell` (or perhaps a new name like `BLCoordinate`) rather than by `BLCell` alone. This would mean being able to collapse routines like `BLGetCellData` and `BLGetWidgetData` into something like `BLGetObjectData( part, coordinates, ... )`. This is what the notification callback and `BLHitTestRec` are starting to do.

I'm trying to set up a Blugs Appearance user pane, but my scroll bars don't work. What gives?

When you have a user pane, you should make sure you're calling `HandleControlClick` instead of `BLClick`. When Blugs makes a user pane, it sets the scroll bars' supervisor to the user pane. That way the Control Manager reports that the pane was clicked when you call `FindControlUnderMouse`. Blugs' user pane code does the necessary dirty work before calling `BLClick` internally. If `BLClick` is called directly on a user pane, Blugs is unable to hit-test the scroll bars. (You see how that works? In one case you want `FindControlUnderMouse` to report the user pane. Once you're inside the user pane code, you want to be able to find the scroll bars.)

Is Blugs available as a shared library? How about Mach-O?

Only as a static library for now. This puts the 68K and PowerPC versions on equal footing. In the future, we may decide to make Blugs available as a shared library, for CFM/Mach-O, but that will only be when Blugs is a mature product.

How are the Blugs binaries compiled?

Metrowerks CodeWarrior Pro 5 and MPW 3.6d7. Debug versions are compiled with inline MacsBug symbols (traceback tables), all optimizations and scheduling off. PowerPC distribution builds are targeted at generic PowerPC, highest speed optimizations, no MacsBug symbols. A tiny portion of Blugs is written in PowerPC assembly language. All 68K libraries contain 68020 instructions, but not 68040. 68K debug builds use A6 stack frames; non-debug builds do not. All 68K floating point calculations use SANE. Blugs does not use floating point enough to justify 68881 or AltiVec. (Dammit Jim, I'm a list engine, not a raytracer.)

Blugs compiles under ProjectBuilder, but not as a distinct product yet.

Is tripod.com out to get you?

The conspiracy theorist in me says yes. Twice the Blugs distribution file on my Tripod site disappeared. The first time I never got an explanation; the second time I got one to the effect that I had been erroneously targeted for abuse. What was the nature of the abuse, I do not know. The pattern that began to emerge was: whenever there was an unfavorable Napster ruling, Tripod's overly aggressive and appallingly stupid abuse-bots went on a rampage. Apparently, the more mission-critical the file, the more likely to be gobbled up.

A savvy insider told me that even the best web hosts pretty much suck. What this implies about Tripod I leave to the reader's imagination. I no longer have anything to do with Tripod and I advise readers to do the same.

Glossary

ancestor In disclosure lists, any row which "contains" the row in question. If a row's disclosure triangle points to the right, its descendants are not visible. See also **descendant**, **parent**.

autodraw A Blugs feature whereby the list's screen representation is updated whenever the list is changed. Autodraw can be disabled (either initially or on the fly) if multiple changes are to be made to the list before it is updated.

background The color and/or texture used to fill a cell's rectangle before its content is drawn.

cell The intersection of a row and a column. The fundamental unit of data display in a list.

cell region A region calculated by a content handler that Blugs uses for hit-testing and hiliting

child In disclosure lists, a row which is "directly contained" by the row in question. If this row's disclosure triangle points to the right, its children are not visible. See also **descendant**, **parent**.

collapse To make a row's disclosure triangle point to the right so that its descendants become obscured (undisclosed). See also **expand**.

content handler A user-supplied routine which provides drawing and other capabilities for the contents of cells and other interface items with which it is associated.

content type A number which Blugs' host application associates with a content handler.

descendant In disclosure lists, a row which is "contained" by the row in question. If this row's disclosure triangle points to the right, its descendants are not visible. See also **ancestor**, **child**.

disclosure list A list whose rows are able to display, and respond to user manipulation of, disclosure triangles. In a disclosure list, rows are able to "contain" other rows. We prefer the term *disclosure* to the equivalent *hierarchical*.

expand To make a row's disclosure triangle point down so that its children become visible (disclosed). See also **collapse**.

filler title An inert element that resembles a title bevel button. Blugs draws a filler title in the part of a title bar that does not contain any titles. A filler title does not respond to mouse clicks except (optionally) for drag-resizing.

host application The code which calls Blugs. Typically this is an application program, but it may be a plug-in or other type of code fragment.

host window A window owned by the host application, inside which a Blugs list is created.

inline edit session A state in which a cell receives keyboard input, and this input modifies the cell's data. Only one inline edit session is allowed in a single list. Not all lists allow inline editing, and not all content handlers support it.

list A generic type of user interface item which may have rows and columns. See also **spreadsheet** and **table**.

parent In disclosure lists, the row which "directly contains" the row in question. If a parent's disclosure triangle points to the right, its children are not visible. See also **ancestor**, **child**.

representative A column to which selection effects in all other columns are redirected.

register To make a user-defined procedure (such as a content handler) available to Blugs.

sort button A bevel button which contains an icon that indicates whether and how the list is sorted. When the sort button is clicked, an unsorted list becomes sorted and a sorted list has its sort direction toggled.

spreadsheet A type of list in which every cell can potentially have a different content type. See also **table**.

table A type of list in which all cells in a column have the same content type. See also **spreadsheet**.

title A single element of a title bar. A horizontal title bar contains as many titles as there are columns in the list; a vertical title bar has as many titles as there are rows. Titles typically can be selected by the user; they have radio-button behavior.

title bar A user interface item which may be part of a list. A title bar may be horizontal or vertical, and extends the full width or height of the list depending on its orientation. See **title**.

title row A row which extends the full width of the list. Can be used to label a section of a list.

widget A control-like placard drawn in line with a scroll bar. Widgets can have content types and data like titles or cells. They can respond to user interaction.

Index