Hyperbole Manual

Everyday Information Management

Bob Weiner
Brown University

Edition 3.07

May 21, 1993

The author's work on this project has been sponsored by Motorola Inc. under the Distinguished Student-Employee program.

Preface

This manual explains user operation and summarizes basic developer facilities of Hyperbole. It is for use with any version 3.07 or greater of Hyperbole. This major release of Hyperbole concentrates on providing convenient access to information and control over its display. The next major release will emphasize flexible views and structure manipulation within bodies of retrieved information.

Hyperbole is an open, efficient, programmable prototype information management system with a hypertextual flavor. It can be used for everyday work on any UNIX-type system and display supported by GNU Emacs. It also works with the versions of Emacs that support multiple X windows: GNU Emacs V19, Epoch (from the University of Illinois) and Lucid Emacs (from Lucid Inc.). It is easy to pick up and use since it introduces only a few new mechanisms and provides user-level facilities through a menu interface, which you control from the keyboard or the mouse. Hyperbole is wholly written in Emacs Lisp for ease of modification. It includes a mouse interface for browsing documentation in GNU Info format.

Hyperbole consists of three parts:

1. an interactive information management interface which anyone can use;
2. a set of programming library classes for system developers who want to integrate Hyperbole with another user interface or as a back-end to a distinct system; (Although Hyperbole is a prototype, it has been engineered for real usage and is well structured.)
3. a set of hyper-button types that provides core hypertext and other behaviors; users can make simple changes to button types and those familiar with Emacs Lisp can quickly prototype and deliver new types.

Hyperbole may be used simply for browsing through documents pre-configured with Hyperbole buttons, in which case, one can safely ignore most of the information in this manual. The 'DEMO' file included in the Hyperbole distribution demonstrates many of Hyperbole's standard facilities. It provides good examples of how buttons may be used and offers a much less technical introduction for Hyperbole users. So if this manual is too detailed for your taste, you can skip it entirely and just jump right into the demonstration, normally by typing {C-h h d d}, assuming Hyperbole has already been installed at your site. Otherwise, see Section 4.1 [Initializing], page 15, for Hyperbole installation and initialization information.

Many users, however, will want to do more than browse with Hyperbole, e.g. create their own buttons. The standard Hyperbole button editing user interface is GNU Emacs-based, so a basic familiarity with the Emacs editing model is useful. The material covered in the GNU Emacs tutorial, normally bound to `{C-h t}` within Emacs, is more than sufficient as background. If some GNU Emacs terms are unfamiliar to you, see section "Glossary" in *the GNU Emacs Manual*.

Acknowledgments

Peter Wegner has encouraged the growth in this work. Morris Moore and Q. Balzano have helped us pursue our own research visions and keep us striving for excellence.

Doug Engelbart has shown us the bigger picture and continues to be an inspiration. His work provides a model from which we are beginning to draw.

1 Introduction

This chapter describes what Hyperbole is, lists some of its potential applications, and then summarizes the structure of the rest of the manual.

Hyperbole Overview

A Hyperbole user works with *buttons*; he may create, modify, move or delete buttons. Each button performs a specific action, such as linking to a file or executing a shell command.

There are three categories of Hyperbole buttons:

explicit buttons
: created by Hyperbole, accessible from within a single document;

global buttons
: created by Hyperbole, accessible anywhere within a user's network of documents;

implicit buttons
: created and managed by other programs or embedded within the structure of a document, accessible from within a single document. Hyperbole recognizes implicit buttons by contextual patterns given in their type specifications (explained later).

Explicit Hyperbole buttons may be embedded within any type of text file. Implicit buttons may be recognized anywhere within a text file, depending on the implicit button types that are available. All global buttons are stored in a single location and activated by entering their names, rather than by direct selection, the means used to activate explicit and implicit buttons.

To summarize:

```
Button Category   Active Within        Activation Means      Managed By
========================================================================
Explicit          a single document    direct selection      Hyperbole
Global            any document         specifying its name   Hyperbole
Implicit          a matching context   direct selection      other tools
========================================================================
```

Hyperbole buttons may be clicked upon with a mouse to activate them or to describe their actions. Thus, a user can always check how a button will act before activating it. Buttons may also be activated from a keyboard. (In fact, virtually all Hyperbole operations, including menu usage, may be performed from any standard character terminal interface, so one need not be anchored to a workstation all day). See Section 4.2 [Smart Keys], page 17.

Hyperbole does not enforce any particular hypertext or information management model, but instead allows you to organize your information in large or small chunks as you see fit. Some of its most important features include:

- Buttons may link to information or may execute procedures, such as starting or communicating with external programs;
- One simply drags between a button source location and a link destination to create or to modify a link button. The same result can be achieved from the keyboard.
- Buttons may be embedded within electronic mail messages;
- Other hypertext and information retrieval systems may be encapsulated under a Hyperbole user interface (a number of samples are provided).

Typical Hyperbole applications include:

personal information management

Overlapping link paths provide a variety of views into an information space. A search facility locates buttons in context and permits quick selection.

documentation and code browsing

Cross-references may be embedded within documentation. One can add a point-and-click interface to existing documentation, link code with associated design documents, or jump to the definition of an identifier by selecting its name within code or documentation.

help/training systems

Tutorials containing buttons can show students how things work while explaining the concepts, e.g. an introduction to local commands. This technique can be much more effective than written documentation alone.

archive managers

Programs that manage archives from incoming information streams may be supplemented by having them add topic-based buttons that link to the archive holdings. Users can then search and create their own links to archive entries.

Manual Overview

Appendix A [Glossary], page 41, provides definitions of Hyperbole terms for quick reference, so in some cases terms are not precisely defined within the text. Be sure to reference the glossary if a term is unclear to you. Although you need not have a keen understanding of all of these terms, a quick scan of the Glossary should help throughout Hyperbole use.

Chapter 2 [Button Concepts], page 7, and Chapter 3 [Hyperbole Views], page 13, lay a conceptual foundation needed before the material in Chapter 4 [Using Hyperbole], page 15, can be utilized. See Appendix B [Smart Key Reference], page 45 for reference documentation on the innovative, context-sensitive mouse and key handling offered by Hyperbole. See Appendix D [Questions and Answers], page 57 for responses to a number of frequent questions.

Developers comfortable with Emacs Lisp will want to continue on through Chapter 5 [Developing with Hyperbole], page 33.

Appendix E [Ongoing Work], page 59 describes future directions in Hyperbole's evolution.

2 Button Concepts

This chapter explains the user-level notion of Hyperbole buttons. You should read it before reading Chapter 4 [Using Hyperbole], page 15.

Hyperbole buttons that are stored in files persist across Emacs sessions, so they provide a convenient means of linking from one information source to another.

2.1 Explicit Buttons

Hyperbole creates and manages *explicit buttons* which look like this `<(fake button)>` to a Hyperbole user. They are quickly recognizable, yet relatively non-distracting as one scans the text in which they are embedded. The text between the `<(` and `)>` delimiters is called the *button label*. Spacing between words within a button label is irrelevant to Hyperbole, so button labels may wrap across several lines without causing a problem.

Hyperbole stores the *button data* that gives an explicit button its behavior, separately from the button label, in a file named '`.hypb`' within the same directory as the file in which the button is created. Thus, all files in the same directory share a common button data file. Button data is comprised of individual *button attribute* values. A user never sees this data in its raw form but may see a formatted version by asking for help on a button.

Explicit buttons may be freely moved about within the buffer in which they are created. (No present support exists for moving buttons between buffers). A single button may also appear multiple times within the same buffer; one simply copies the button label with its delimiters to a new location in such cases.

Each explicit button is assigned an action type which determines the actions that it performs. *Link action types* connect buttons to particular types of referents. *Activation* of such buttons then displays the referents.

Hyperbole does not manage referent data; this is left to the applications that generate the data. This means that Hyperbole provides in-place linking and does not require reformatting of data to integrate it with a Hyperbole framework.

2.2 Global Buttons

Access to explicit buttons depends upon the information on your screen since they are embedded within particular buffers. Sometimes it is useful to activate buttons without regard to the information with which you are presently working. In such instances, you use *global buttons*, which are simply explicit buttons which may be activated or otherwise operated upon by entering their labels when they are prompted for, rather than selecting the buttons within a buffer.

If you want a permanent link to a file section that you can follow at any time, you can use a global button. Or what about an Emacs keyboard macro that you use frequently? Create an `exec-kbd-macro` button with an easy to type name and then you can easily activate it whenever the need arises.

2.3 Implicit Buttons and Types

Implicit buttons are those defined by the natural structure of a document. They are identified by contextual patterns which limit the locations or states in which they can appear. Their behavior is determined by one or more actions which they trigger when activated. An action is derived from either a Hyperbole action type specification, see Section 2.4 [Action Types and Actions], page 9, or an Emacs Lisp function. Implicit button types may use the same action types that explicit buttons do.

Implicit buttons never have any button data associated with them. They are recognized in context based on predicate matches defined within implicit button types. For example, Hyperbole recognizes file names enclosed in double quotes and can quickly display their associated files in response to simple mouse clicks.

See '`hibtypes.el`' for complete examples. Standard implicit button types include:

`annot-bib`
: Displays annotated bibliography entries referenced internally, delimeters = [].

`dir-summary`
: Detects filename buttons in files named "MANIFEST" or "DIR".

`doc-id`
: Displays an index entry for a site-specific document given its id.

`grep-msg`
: Jumps to line associated with grep or compilation error msgs. Messages are recognized in any buffer.

`hyp-source`
: Makes source entries in Hyperbole reports into buttons that jump to source.

`Info-node`
: Makes "(file)node" buttons display the associated Info node.

`kbd-key`
: Executes a key sequence delimited by curly braces.

`man-apropos`
: Makes man apropos entries display associated man pages when selected.

`pathname`
: Makes a delimited, valid pathname display the path entry. Also works for delimited and non-delimited ange-ftp pathnames.

`rfc`
: Retrieves and displays an Internet rfc referenced at point. Requires ange-ftp. The following formats are recognized: RFC822, rfc-822, and RFC 822.

`rfc-toc`
: Summarizes contents of an Internet rfc from anywhere within rfc buffer.

The Hyperbole Smart Keys offer extensive additional context-sensitive point-and-click type behavior beyond these standard implicit button types. See Section 4.2 [Smart Keys], page 17.

2.4 Action Types and Actions

Action types provide action procedures that specify button behavior. The arguments needed by an action type are prompted for at button creation time. When a button is activated, the stored arguments are fed to the action type's *action* body to achieve the desired result. Hyperbole handles all of this transparently.

Standard action types include:

`annot-bib`
: Follows internal ref KEY within an annotated bibliography, delimiters=[].

`eval-elisp`
: Evaluates a Lisp expression LISP-EXPR.

`exec-kbd-macro`
: Executes KBD-MACRO REPEAT-COUNT times.

`exec-shell-cmd`
: Executes a SHELL-CMD string asynchronously.

`exec-window-cmd`
: Executes an external window-based SHELL-CMD string asynchronously.

`hyp-source`
: Displays a buffer or file from a line beginning with 'hbut:source-prefix'.

`kbd-key`
: Executes the function binding for KEY-SEQUENCE, delimited by `{}`.

`link-to-buffer-tmp`
: Displays a BUFFER in another window.

`link-to-directory`
: Displays a DIRECTORY in Dired mode in another window.

`link-to-doc`
: Displays online version of a document given by DOC-ID, in other window.

`link-to-ebut`
: Performs action given by another button, specified by KEY and KEY-FILE.

`link-to-file`
: Displays a PATH in another window scrolled to optional POINT.

`link-to-file-line`
: Displays a PATH in another window scrolled to LINE-NUM.

`link-to-Info-node`
: Displays an Info NODE in another window.

`link-to-mail`
: Displays mail msg with MAIL-MSG-ID from MAIL-FILE in other window.

`link-to-regexp-match`
: Finds REGEXP's Nth occurrence in FILE and displays location at window top.

`link-to-rfc`
: Retrieves and displays an Internet rfc given by RFC-NUM. RFC-NUM may be a string or an integer. Requires ange-ftp.

`link-to-string-match`
: Finds STRING's Nth occurrence in FILE and displays location at window top.

`man-show`
: Displays man page on TOPIC, which may be of the form `<command>(<section>)`.

`rfc-toc`
: Computes and displays summary of an Internet rfc in BUF-NAME.

The use of action types provides a convenient way of specifying button behavior without the need to know how to program. Expert users who are familiar with Emacs Lisp, however, may find that they often want to tailor button actions in a variety of ways not easily captured within a type system. In such cases, *hui:ebut-prompt-for-action* should be set non-nil. This will cause Hyperbole to prompt for an action to override the button's action type at each explicit button creation. For those cases where the action type is sufficient, a nil value should be entered for the action. An action may be any Lisp form that may be evaluated.

2.5 Button and Type Precedences

Explicit buttons always take precedence over implicit buttons. Thus, if a button selection is made which falls within both an explicit and implicit button, only the explicit button will be selected. Explicit button labels are not allowed to overlap; Hyperbole's behavior in such cases is undefined.

If there is no explicit button at point during a selection request, then each implicit button type predicate is tested in turn until one returns non-nil or all are exhausted. Since two implicit button types may have overlapping domains (those contexts in which their predicates are true), only the first matching type is used. The type predicates are tested in **reverse** order of definition, i.e. most recently entered types are tested first, so that personal types defined after standard system types take precedence. It is important to keep this order in mind when defining new implicit button types. By making their match predicates as specific as possible, one can minimize any overlapping of implicit button type domains.

Once a type name is defined, its precedence relative to other types remains the same even if you redefine the body of the type, as long as you don't change its name. This allows incremental modifications to types without having to worry about shifts in type precedence. See Section 5.2 [Creating Types], page 34, for information on how to develop or modify types.

3 Hyperbole Views

This chapter explains the role of views in the display of information accessed via Hyperbole. You should read it before reading Chapter 4 [Using Hyperbole], page 15.

This release of Hyperbole does not contain **any** support for views. A general view model is in development. The descriptions here are meant to familiarize you with the notion of views to be available in future releases.

A *Hyperbole view* is a perspective on a pool of information. An outline of a document, a document extract, and a decrypted document all represent views of an original document (which may just be a file).

View specifications or view specs, for short, control the application of views to displayed information. An action type which displays a buffer of text, the *result buffer*, can apply a view spec to the buffer before display. A user can then manipulate the view by interactively altering the view spec associated with the buffer. A 'V' in an action type's interactive call string indicates that the view spec of any button which uses the action type should be applied to the result buffer of the action type.

4 Using Hyperbole

This chapter covers user interaction with Hyperbole. Often a number of overlapping interaction methods are provided to support different work styles and hardware limitations. You need learn only one with which you can become comfortable, in such instances.

Remember that the 'DEMO' file included in the Hyperbole distribution demonstrates many of Hyperbole's standard facilities, See [Top], page 1, for more details.

4.1 Initializing

Hyperbole must be installed at your site before you can use it. The 'README' file that comes with the Hyperbole distribution explains how to install it and how to join the Hyperbole interest mail lists. See the Msg/ menu item in Section 4.4 [Operating Menus], page 21, for a convenient means of joining and mailing to these lists.

If you are installing Hyperbole at your site, please read the 'README' file before continuing in this document.

If you want to customize the basic Hyperbole initialization sequence for yourself rather than the users at your site, you should make a personal copy of the 'hsite.el' file, modify it the way you want, and then load it. (If you are familiar with Emacs Lisp, see also Section 5.1 [Hook Variables], page 33.)

4.1.1 External Viewers

If you will be using Hyperbole under a window system, you may want to configure the *hpath:find-alist* variable in 'hsite.el' to support hyperlinks which open documents using non-Emacs tools, e.g. a fax reader or a bitmap viewer. Each element in this association list is a (<emacs-window-system> . <file-suffix-alist>) pair. The <emacs-window-system> part is already filled out for you, it is a string indicating a special Emacs variant or a window system. Choose the particular Emacs/window system that you want to configure. Then make the <file-suffix-alist> be an association list whose elements are (<file-suffix-regular-expression> . <viewer-program>) pairs. Any links to paths whose suffix matches a file-suffix-regular-expression will be displayed using the corresponding viewer-program. See the "next" clause within the definition of *hpath:find-alist* as an example.

4.1.2 Link Variable Substitution

Another variable to consider modifying in the '`hsite.el`' file is *hpath:variables*. This variable consists of a list of Emacs Lisp variable names, each of which may have a pathname or a list of pathnames as a value. Whenever a Hyperbole file or directory link button is created, its pathname is compared against the values in *hpath:variables*. The first match found, if any, is selected and its associated variable name is substituted into the link pathname, in place of its literal value. When the link is resolved (the button is activated), Hyperbole replaces any variables with the first matching value. This permits sharing of links over wide areas, where the variable values may differ between link creator and link activator. The entire process is wholly transparent to the user; it is explained here simply to help you in deciding whether or not to modify the value of *hpath:variables*.

4.1.3 Configuring Button Colors

When Hyperbole is run under Epoch or Lucid Emacs, it can automatically highlight any explicit buttons in a buffer and make them flash when selected. The only configuration necessary is selection of a color (or style) for button highlighting and button flashing. See the '`hui-ep*.el or hui-le*.el`' files for lists of potential colors and the code which supports this behavior. A call to `(ep:cycle-but-color)` or `(le:cycle-but-color)` within a Hyperbole initialization sequence changes the color used to highlight and flash explicit buttons.

4.1.4 Loading Hyperbole

Each time you start a new Emacs session, the Hyperbole code must be loaded for you to use it. The easiest way to achieve this is to add a line to your '`~/.emacs`' file, which Emacs reads automatically on startup. The line should be of the form: `(load "<HYP-DIR>/hyperbole")` where the local Hyperbole installation directory is substituted for `<HYP-DIR>`. This establishes a few key bindings and sets up Hyperbole to automatically load whenever you activate its menu. If you would rather have the whole Hyperbole system loaded when you start up so that you can always use the Smart Keys and other facilities, use `(load "<HYP-DIR>/hsite")` instead.

If you use mouse keys, be sure to add the above load line after any setup of mouse key bindings, to ensure that Hyperbole's mouse keys are initialized. See Section 4.2 [Smart Keys], page 17, for further details. If you use any Hyperbole mail or news support, see Section 4.6.7 [Buttons in Mail], page 27, be certain to perform all of your personal mail/news initializations before the point at which you load Hyperbole. Otherwise, the mail/news support may not be configured properly. For example, if you use the Emacs add-on Supercite package, its setup should come before Hyperbole initialization.

4.2 Smart Keys

Hyperbole provides two special keys that perform context-sensitive operation, as explained below. By default, these two keys are bound to your middle (or left on a 2-button mouse) and right mouse keys when Hyperbole is run under an external window system (mouse configuration is automatic for Lucid Emacs and Epoch under the X window system and for GNU Emacs versions 18 and 19 under X, OpenWindows, NEXTSTEP, SunView and Apollo's Display Manager, assuming your GNU Emacs program has been built with support for any of these window systems. Note that the shift-left mouse button must be used under NEXTSTEP to set point to a location, since Hyperbole takes over the left button.)

The middle key (or left on a 2-button mouse) is called the *primary Smart Key* (or just Smart Key) and the right is called the *secondary Smart Key* (or Smart Meta Key). By default (if *smart-key-init* is left with a setting of `t` in '`hsite.el`'), then `{M-RET}` may also be used as the primary Smart Key and `{C-u M-RET}` may be used as the secondary Smart Key. In many read-only modes like Dired and Rmail, `{RTN}` also functions as the Smart Key. These key bindings allow context sensitive operation from any keyboard.

If you prefer other key bindings, simply bind the commands `smart-key` and `smart-key-meta` to keyboard keys. `Smart-key-either` may be used instead if you prefer the a single key binding for both commands; a prefix argument then invokes `smart-key-meta`.

You may also bind `smart-key-mouse` and `smart-key-mouse-meta` to mouse keys, as you like.

The primary Smart Key generally selects entities, creates links and activates buttons. The secondary Smart Key generally provides help, such as reporting on a button's attributes, or serves a complementary function to whatever the primary Smart Key does within a context.

You can get a summary of what the Smart Keys do in all of their different contexts by pressing the secondary Smart Key in the right corner (within the rightmost 3 characters) of a window modeline or by using the Hyperbole Doc/SmartKy menu entry.

The following table is an example of this summary. Much of the browsing power of Hyperbole comes from use of the Smart Keys, so you should spend some time practicing how to use them. This table may appear daunting at first, but as you practice and notice that the Smart Keys do just a few context-sensitive things per editor mode, you will find it easy to just point and click and let Hyperbole do the rest.

For extensive reference documentation on the Smart Keys, see Appendix B [Smart Key Reference], page 45.

```
==============================================================================
Context                                            Smart Key
                                Primary                   Secondary
==============================================================================
Hyperbole
  On a menu item                Item is activated         Item help
  On an explicit button         Button is activated       Button help
  Reading argument
    1st press at an arg value   Value copied to minibuffer <- same
    2nd press at an arg value   Value used as argument    <- same
    In minibuffer               Minibuf arg is applied    Completion help
  On an implicit button         Button is activated       Button help
  Wrolo Match Buffer            Toggles narrow to one entry and edits entries

Mouse or Keyboard Display Control
  Line end, not end of buffer
    smart-scroll-proportional
      = nil (default)           Scrolls up a windowful    Scrolls down
      = t                       Makes curr line top line  Bottom line
  End of Any Help buffer        Screen restored to previous state

Mouse-only Control
  Modeline down & wind release  Resize window height      <- same
  Drag from shared window side  Resize window width       <- same
  Drag between windows          Create/modify a link but  Swap window buffers
  Horizontal drag within window
    Left to right               Scroll to buffer end      Split window across
    Right to left               Scroll to buffer begin    Delete window
  Vertical drag within window   Split window sideways     <- same
  Diagonal drag within window   Save ring screen-config   Restore ring config
  Click in modeline
    Left window edge            Bury buffer               Unbury bottom buf
    Right window edge           Info                      Smart Key Summ
    Otherwise                   Smart or popup menus      Popup menus

Special Modes
  C Mode                        Jumps to id def           Jumps to next def
  Any Lisp mode                 Jumps to id def           Jumps to next def
  Occur match                   Jumps to match source line <- same
  Multi-buffer occur match      Jumps to match source line <- same
  Outline Major/Minor Modes     Collapses, expands, and moves outline entries
  Man Apropos                   Displays man page entry   <- same
  Man Pages                     Follows cross refs, file refs and C code refs
  Buffer Menu                   Saves, deletes and displays buffers

Emacs Info Reader
  Menu Entry or Cross Ref       Jumps to referent         <- same
  Up, Next or Prev Header       Jumps to referent         Jumps to prior node
  File entry of Header          Jumps to top node         Jumps to (DIR) node
  End of current node           Jumps to next node        Jumps to prev node
```

```
  Anywhere else                 Scrolls up a windowful     Scrolls down a wind

Subsystems
  Calendar                      Scrolls or shows appts     Scrolls/marks dates
  Dired Mode                    Views and deletes files from directory listing
  GNUS News Reader              Toggles group subscriptions, gets new news,
                                  and browses articles
  Mail reader and Summaries     Browses, deletes and expunges messages

Any other context (defaults)    Hyperbole top menu         Smart Key summary
===============================================================================
```

A prime design criterion of Hyperbole's user interface is that one should be able to see what an operation will do before using it. The secondary Smart Key provides help for button and menu item actions. Hyperbole also shows the result of directly selecting an argument value, to provide feedback as to whether the right item has been selected.

When you use a mouse and you want to find out what either of the Smart Keys does within a context, depress the one you want to check on and hold it down, then press the other and release as you please. A help buffer will pop up explaining the actions that will be performed in that context, if any. A press of either Smart Key at the end of that help buffer will restore your display to its configuration prior to invoking help.

By default (if *smart-key-init* is left set equal to `t` in '`hsite.el`'), then `{C-h S}` will display this same context-sensitive help for the primary Smart Key while `{C-u C-h S}` will display the help for the secondary Smart Key. Note that `{C-h s}` will perform a function unrelated to Hyperbole.

When Hyperbole is installed, a key may be bound which allows you to switch between the Smart Key mouse bindings and your prior ones. `C-h w sm-mouse-toggle-bindings RTN` should show you any key which performs this command. If no key binding has been established or if you prefer one of your own, simply select a key and bind it within your '`~/.emacs`' file. For example, `(global-set-key "\C-ct" 'sm-mouse-toggle-bindings)`.

4.3 Screen Configurations

Hyperbole includes the '`wconfig.el`' package which lets you save and restore screen configurations, i.e. the layout and display parameters of the windows on an Emacs screen. This is useful to save a particular working context and then to jump back to it during an Emacs working session. It is also useful during demonstrations to pull up many informational artifacts all at once, e.g. all of the windows for a particular subsystem.

Wconfig creates a ring structure that operates just like the Emacs *kill-ring*, see section "Kill Ring" in *The GNU Emacs Manual*, but its elements are screen configurations rather than text regions. One can add an element to the ring based upon the current screen configuration. After several elements are in the ring, one can walk through all of them in sequence until the desired configuration is restored to the screen.

The prior section, see Section 4.2 [Smart Keys], page 17, mentions how screen configuration saves and restores may be performed with the Smart Keys. Alternatively, the functions may be invoked independently of the Smart Keys with the following key bindings.

`{C-x 4 w}` performs a `wconfig-ring-save` command which saves the current screen configuration to the ring. Think of it like a wipe of the screen.

`{C-x 4 y}` performs a `wconfig-yank-pop` which restores the screen configuration from the currently pointed to configuration in the ring. It does not delete this configuration from the ring but it does move the pointer to the prior ring element. Repeated calls to this command thus restore successive screen configurations until the ring pointer wraps around. Simply stop when you see the one you want.

`{C-x 4 DEL}` should be used when you want to restore a previously saved configuration and at the same time delete it from the ring. It calls the `wconfig-delete-pop` command.

These bindings are available only if these keys are not bound to any other commands before Hyperbole is invoked. Hyperbole will not overwrite any prior bindings of these keys.

The maximum number of elements the ring can hold is set by the *wconfig-ring-max* variable whose default is 10. Any saves beyond this value cause deletion of the oldest element in the ring before a new one is added.

4.4 Operating Menus

Hyperbole menus provide access to user-level commands. Only one key binding is used for the top-level menu, but menu items may be selected from either the keyboard or via mouse clicks. When used with the keyboard, they provide rapid command access similar to key sequence bindings.

The top level menu is invoked from a key given in your '`hsite.el`' file (by default, `{C-h h}`) or via a Smart Key press anywhere in the center region of a window's modeline. The menu will appear in the minibuffer and should look about like so:

```
Hypb>  Act  Butfile/  Doc/  Ebut/  Gbut/  Hist  Ibut/  Msg/  Rolo/  Types/
```

The above menu items can be summarized as follows:

Act Perform the action associated with any button at point or prompt for the name of an explicit button to activate if point is not on one.

Butfile/ Display a local or global file of buttons, providing easy access. '`HYPB`' for a local button file and '`~/.hyperb/HYPB`' for your global file. These are good places to start your button creation testing.

Ebut/ All explicit button commands.

Doc/ Hyperbole documentation quick access.

Gbut/ All global button commands. Global buttons are accessed by name rather than by direct selection.

Hist Jumps back to last position in button traversal history.

Ibut/ All implicit button commands.

Msg/ Hyperbole-specific mail and news messaging support commands. Use this to send mail to the Hyperbole discussion list or to add/modify your entry on a Hyperbole mail list.

Rolo/ Hierarchical, multi-file rolodex lookup and edit commands. See Section 4.7 [Rolodex], page 29.

Types/ Summarizes available Hyperbole types such as implicit button types and action types.

All menu items are selected via the first character of their names (letter case does not matter) or via a press of the Smart Key. "/" at the end of an item name indicates that it brings up a sub-menu. A press of the secondary Smart Key on an item displays help for the item, including the action that it performs.

While a menu is active, to re-activate the top-level Hyperbole menu, you must use `{C-t}`. This allows you to browse the submenus and then return to the top. You can quit without selecting an item by using `{q}`. `{C-g}` aborts whether you are at a menu prompt or any other Hyperbole prompt.

4.5 Entering Arguments

Many Hyperbole commands prompt you for arguments. The standard Hyperbole user interface has an extensive core of argument types that it recognizes. Whenever Hyperbole is prompting you for an argument, it knows the type that it needs and provides some error checking to help you get it right. More importantly, it allows you to press the Smart Key within an entity that you want to use as an argument and it will grab the appropriate thing and show it to you at the input prompt within the minibuffer. If you press the Smart Key again at the same point (click with a mouse) on the same thing again, it accepts the entity as the argument and moves on. Thus, a double click registers a desired argument. Double-quoted strings, pathnames, mail messages, Info nodes, dired listings, buffers, numbers, completion items and so forth are all recognized at appropriate times. All of the argument types mentioned in the documentation for the Emacs Lisp `(interactive)` function are recognized. Experiment a little and you will quickly get used to this direct selection technique.

Wherever possible, standard Emacs completion is offered, see section "Completion" in *the Gnu Emacs Manual*. Remember to use `{?}` to see what your possibilities for an argument are. Once you have a list of possible completions on screen, you can double click the Smart Key on any one to enter it as the argument.

4.6 Working with Explicit Buttons

Explicit buttons provide the building blocks for creating personal or organizational hypertext networks with Hyperbole. This section summarizes the user-level operations available for managing these buttons.

4.6.1 Creation

4.6.1.1 via Smart Mouse Key Drags

The most efficient way to create an explicit button interactively is to use the Smart Mouse Key to drag from a button source window to a window showing its link referent. More specifically, you should split your current Emacs screen into two windows: one which contains the point at which you want a button to be inserted and another which shows the point to which you want to link. Depress the Smart Mouse Key at the point at which the button should be inserted, drag to the other window and release it at the point of the link referent. The process becomes quite simple with a little practice. (See Section 4.6.1.2 [Menus], page 24, for a more detailed explanation of the explicit button creation process.)

Hyperbole uses the link referent context to determine the type of link to make. If there are a few different types of links which are applicable from the context, you will be prompted with a list of the types. Simply use the Smart Mouse Key or the first letter of the link type to select one of the type names and to finish the link creation. Hyperbole will then insert explicit button delimiters around the button label and will display a message in the minibuffer indicating both the button name and its action/link type.

You can emulate a Smart Mouse Key drag from the keyboard by hitting `{M-o}`, the `smart-key-operate` command, at the button source location, moving to the link destination, e.g. with `{C-x o}`, and then hitting `{M-o}` again. This simulates a depress and then release of the Smart Key.

4.6.1.2 via Hyperbole Menus

You can alternatively use the Hyperbole menus to create explicit buttons. First, mark a short region of text in any fashion allowed by GNU Emacs and then select the Hyperbole menu item sequence, Ebut/Create. You will be prompted for the button's label with the marked region as the default. If you accept the default and enter the rest of the information you are prompted for, the button will be created within the current buffer and Hyperbole will surround the marked region with explicit button delimiters to indicate success.

If you do not mark a region before invoking the button create command, you will be prompted for both a label and a target buffer for the button and the delimited label text will be inserted into the target buffer after a successful button creation.

After Hyperbole has the button label and its target buffer, it will prompt you for an action type for the button. Use the `{?}` completion help key to see the available types. The type selected determines any following values for which you will be prompted.

If a previous button with the same label exists in the same buffer, Hyperbole will add an *instance number* to the label when it adds the delimiters so that the name is unique. Thus, you don't have to worry about accidental button name conflicts. If you want the same button to appear in multiple places within the buffer, just enter the label again and delimit it yourself. Hyperbole will interpret all occurrences of the same delimited label within a buffer as the same button.

If you create link buttons using the Hyperbole menus, the best technique is to place on screen both the source buffer for the button and the buffer to which it will link. Mark the region of text to use for your button label, invoke the button create command from the menu, choose an action type which begins with `link-to-` and then use the direct selection techniques mentioned in Section 4.5 [Entering Arguments], page 22, to select the link referent.

4.6.2 Renaming

Once an explicit button has been created, its label text must be treated specially. Any inter-word spacing within the label may be freely changed, as may happen when a paragraph is refilled. But a special command must be invoked to rename it.

The rename command operates in two different ways. If point is within a button label when it is invoked, it will tell you to edit the button label and then invoke the rename command again. The second invocation will actually rename the button. If instead the command is originally invoked outside of any explicit button, it will prompt for the button label to replace and the label to replace it with and then will perform the rename. All occurrences of the same button in the buffer will be renamed, so you need locate only one occurrence of the button.

The rename command may be invoked from the Hyperbole menu via Ebut/Rename. A faster method is to use a key bound to the `hui:ebut-rename` command. Your site installation may include such a key. `C-h w hui:ebut-rename RTN` should show you any key it is on. If no key binding has been established or if you prefer one of your own, simply bind it within your '`~/.emacs`' file. We recommend the `{C-c C-r}` key, as in: `(global-set-key "\C-c\C-r" 'hui:ebut-rename)`.

4.6.3 Deletion

Ebut/Delete works similarly to the Rename command but deletes the selected button. The button's delimiters are removed to confirm the delete.

Presently there is no way to recover a deleted button; it must be recreated. Therefore, the *hui:ebut-delete-confirm-p* variable is true by default, causing Hyperbole to require confirmation before interactively deleting explicit buttons. Set it to nil if you prefer no confirmation.

4.6.4 Modification

Ebut/Modify prompts you with each of the elements from the button's data list and allows you to modify each in turn.

There is a quicker way to modify explicit link buttons. Simply drag with the Smart Mouse Key from within the button label to a link destination in a different window, just as you would when creating a new button with a mouse drag. Remember that drags may also be emulated from the keyboard. See Section 4.6.1 [Creation], page 23.

4.6.5 Location

The Ebut/Help menu can be used to summarize a single explicit button or all such buttons within a single buffer. The buttons summarized may then be activated directly from the summary.

Ebut/Help/BufferButs summarizes the explicit buttons in the order in which they appear in the buffer. Ebut/Help/CurrentBut summarizes only the button at point. Ebut/Help/OrderedButs summarizes the buttons in alphabetical order. All of these summary commands eliminate duplicate instances of buttons from their help displays.

Ebut/Search prompts for a search pattern and searches across all the locations in which you have previously created explicit buttons. It asks you whether to match to any part of a button label or only complete labels. It then displays a list of button matches with a single line of surrounding context from their sources. Any button in the match list may be activated as usual. A Smart Key press on the surrounding context jumps to the associated source line or a press on the filename preceding the matches jumps to the file without selecting a particular line.

There are presently no user-level facilities for globally locating buttons created by others or for searching on particular button attributes.

4.6.6 Button Files

It is often convenient to create lists of buttons that can be used as menus to provide centralized access to distributed information pools or for other purposes. These files can serve as useful roadmaps to help efficiently guide a user through both unfamiliar and highly familiar information spaces. Files that are created specifically for this purpose, we call *button files*.

The Hyperbole menu system provides quick access to two types of these button files: personal and directory-specific, through the ButFile menu. (The variable, *hbmap:filename*, contains the base name of these standard button files. Its standard value is '`HYPB`'.)

A personal button file may serve as a user's own roadmap to frequently used resources. Selection of the ButFile/PersonalFile menu item displays this file for editing. The default personal button file is stored within the directory given by the *hbmap:dir-user* variable whose standard value is '`~/.hyperb`'. The standard Hyperbole configuration also appends all global buttons to the end of this file, one per line, as they are created. So you can edit or annotate them within the file.

A directory-specific button file may exist for each file system directory. Such files are useful for explaining the contents of directories and pointing readers to particular highlights within the directories. Selection of the ButFile/DirFile menu item displays the button file for the current directory; this provides an easy means of updating this file when working on a file within the same directory. If you want to view some other directory-specific button file, simply use the normal Emacs file finding commands.

One might suggest that menu quick access be provided for group-specific and site-specific button files. Instead, link buttons to such things should be placed at the top of your personal button file. This provides a more flexible means of quick access.

4.6.7 Buttons in Mail

Hyperbole allows the embedding of buttons within electronic mail messages that are composed in Emacs with the standard `(mail)` command, normally bound to `{C-x m}` or with other Emacs-based mail composing functions. An enhanced mail reader can then be used to activate the buttons within messages just like any other buttons.

Hyperbole automatically supports the Rmail, see section "Rmail" in *the GNU Emacs Manual*, VM, see section "Introduction" in *the VM Manual*, and MH-e mail readers. Button inclusion and activation within USENET news articles is also supported in the same fashion via the GNUS news reader, see section "Introduction" in *the GNUS Manual*, if available at your site. (The '`hmail.el`' file provides a generalized interface that can be used to hook in other mail or news readers if the necessary interface functions are written.)

All explicit buttons to be mailed must be created within the outgoing message buffer. There is no present support for including text from other buffers or files which contain explicit buttons, except for the ability to yank the contents of a message being replied to, together with all of its buttons, via the `(mail-yank-original)` command bound to `{C-c C-y}`. From a user's perspective, buttons are created in precisely the same way as in any other buffer. They also appear just like any other buttons to both the message sender and the reader who uses the Hyperbole enhanced readers. Button operation may be tested any time before a message is sent. A person who does not use Hyperbole enhanced mail readers can still send messages with embedded buttons since mail composing is independent of any mail reader choice.

Hyperbole buttons embedded within received mail messages act just like any other buttons. The mail does not contain any of the action type definitions used by the buttons, so the receiver must have these or she will receive an error when she activates the buttons. Buttons which appear in message *Subject* lines are copied to summary buffers whenever such summaries are generated. Thus, they may be activated from either the message or summary buffers.

Nothing bad will happen if a mail message with explicit buttons is sent to a non-Hyperbole user. The user will simply see the text of the message followed by a series of lines of button data at its end. Hyperbole mail users never see this data in its raw form.

In order to alert readers of your mail messages that you can utilize Hyperbole mail buttons, the system automatically inserts a comment into each mail message that you compose to announce this fact. The variable, *smail:comment* controls this behavior. See its documentation for technical details. By default, it produces a message of the form:

```
Comments: Hyperbole mail buttons accepted, vX.XX.
```

where the X's indicate your Hyperbole version number. You can cut this out of particular messages before you send them. If you don't want any message at all, add the following to your '`~/.emacs`' file before the point at which you load Hyperbole.

```
(setq smail:comment nil)
```

A final mail-related facility provided by Hyperbole is the ability to save a pointer to a received mail message by creating an explicit button with a `link-to-mail` action type. When prompted for the mail message to link to, if you press the Smart Key on an Rmail message, the appropriate parameter will be copied to the argument prompt, as described in Section 4.5 [Entering Arguments], page 22.

4.6.8 Buttons in News

Explicit buttons may be embedded within outgoing USENET news articles and may be activated from news articles that are being read. This support is available for the GNUS news reader. It is enabled by default within '`hsite.el`' by autoloading the '`hgnus.el`' file.

All Hyperbole support should work just as it does when reading or sending mail. See Section 4.6.7 [Buttons in Mail], page 27. When reading news, buttons which appear in message *Subject* lines may be activated within the GNUS subject buffer as well as the article buffer. When posting news, the *post-news* buffer is used for outgoing news articles rather than the *mail* buffer.

Remember that the articles you post do not contain the action type definitions used by the buttons, so the receiver must have these or she will receive an error when he activates the buttons. You should also keep in mind that most USENET readers will not be using Hyperbole, so if they receive a news article containing explicit buttons, they will wonder what the button data at the end of the message is. You should therefore limit distribution of such messages. For example, if most people at your site read news with GNUS and use Hyperbole, it would be reasonable to embed buttons in postings to local newsgroups.

In order to alert readers of your postings that you can utilize Hyperbole mail buttons embedded within personal replies, the system automatically inserts the same comment that is included within mail messages to announce this fact. See Section 4.6.7 [Buttons in Mail], page 27, for details and an explanation of how to turn this feature off.

4.7 Rolodex Operation

Hyperbole includes a complete, advanced rolodex system, wrolo, for convenient management of hierarchical, record-oriented information, be it address records or other database items.

Hyperbole buttons may be included within rolodex records and then manually activated whenever their records are retrieved.

See the description at the top of the '`wrolo.el`' file for details on programmatic interfacing to the rolodex. The following subsections explain use and basic customization of the rolodex.

4.7.1 Rolo Concepts

The rolodex manages and searches rolodex files. A *rolodex file* consists of an optional header which starts and ends with a line of equal signs (at least three equal signs starting at the beginning of a line), followed by any non-negative number of rolodex records. You must manually add a header to any rolodex file if you want it to have one.

Here is an example of a simple rolodex file.

```
==================================================================
                        PERSONAL ROLODEX
<Last-Name>, <First>  <Co/Categ>  <Email>       W<Work#>   F<Fax#>
==================================================================
*   Smith, John       Motorola    <js@mot.com> W2001      F1892
```

We call rolodex records, *entries*. Entries begin with a delimiter, some number of '*' characters at the beginning of a line. Entries may be arranged in a hierarchy, where child entries begin with one more '*' character than do their parents. Top level entries begin with a single '*'.

Beyond this initial delimiter, entries are completely free-form text. It is best to use a "lastname, firstname" format, however, when adding contact entries into a rolodex. Then the rolodex system will automatically keep your entries alphabetized.

Any search done on the rolodex scans the full text of each entry. During a search, the rolodex file header separator lines and anything in between are appended to the buffer of matched entries before any entries are retrieved from the file. Whenever an entry is matched, it and all of its descendant entries are retrieved.

For example, a search on "Company" could retrieve the following:

```
==================================================================
                        COMPANY ROLODEX
==================================================================
*    Company
**     Manager
***      Underlings
```

Thus, searching for Company retrieves all listed employees. Searching for Manager turns up all Underlings.

4.7.2 Rolo Menu

The Rolo/ menu entry on the Hyperbole top-level menu provides the user interface to the rolodex. The rolo menu provides access to the following commands:

```
Menu Item       Function              Description
====================================================================
Add             rolo-add              Adds a rolodex entry
Display-again   rolo-display-matches  Displays last matches again
Edit            rolo-edit             Edits an existing rolodex entry
Kill            rolo-kill             Removes an entry from the rolodex
Order           rolo-sort             Sorts all levels in rolodex
Regexp-find     rolo-grep             Finds all entries containing
                                        a regular expression
String-find     rolo-fgrep            Finds all entries containing
                                        a string
Yank            rolo-yank             Inserts first matching rolodex
                                        entry at point
====================================================================
```

A prefix argument used with either of the find commands listed above limits the search to a maximum number of matches given by the argument. The search is terminated whenever that number of matches is found.

For any of the above commands that prompt for a name, you may use the form parent/child to locate a child entry below a parent entry. So for a rolodex which looked like so:

```
*    Company
**     Manager
***      Underlings
```

one could edit the Underlings entry by identifying it as Company/Manager/Underlings. Do not use this hierarchical notation in search expressions since the whole rolodex will be searched anyway. Thus, "Underlings" as a search pattern will find an entry containing "Underlings" at any level in a hierarchy, like so:

```
***      Underlings
```

4.7.3 Rolo Settings

The buffers containing the rolodex files are not killed after a search on the assumption that another search is likely to follow within this Emacs session. You may wish to change this behavior with the following setting: `(setq rolo-kill-buffers-after-use t)`.

After an entry is killed, the modified rolodex file is automatically saved. If you would rather always save files yourself, use this setting: `(setq rolo-save-buffers-after-use nil)`.

When adding an entry from within a buffer containing a mail message, the rolodex add function will extract the sender's name and e-mail address and prompt you with the name as a default. If you accept it, it will enter the name and the email address using the format given by the *rolo-email-format* variable. See its documentation if you want to change its value.

The files used in any rolodex search are given by the *rolo-file-list* variable, whose default value is `("~/.rolodex.otl")`, so that searches initially scan only your personal rolodex. Any entries added to this list should be file pathnames. If a file in the list does not exist or is not readable, it is skipped. Files are searched in the order in which they appear in the list. In general, you should leave your personal rolodex file as the first entry in the list, since this is the only file to which the rolo menu Add command adds entries.

The rolodex entry start delimiter is given by the regular expression variable, *rolo-entry-regexp*, whose default value is `"^\*+"`.

A rolodex file may begin with an optional header section which is copied to the match display buffer whenever any matches are found during a search. The start and end lines of this header are controlled by the regular expression variable, *rolo-hdr-regexp*, whose default value is `"^==="`. This allows lines of all equal signs to visually separate matching entries from multiple files retrieved from a single search.

5 Developing with Hyperbole

This chapter is only for people who are familiar with Emacs Lisp and wish to extend Hyperbole, customize Hyperbole or develop other systems using Hyperbole as a base.

5.1 Hook Variables

Hyperbole provides a number of hook variables that allow you to adjust its basic operations to meet your own needs, without requiring you to change the code for those operations.

We find it best to always set the value of hook variables either to nil or to a list of function names of no arguments, each of which will be called in sequence when the hook is triggered.

Given the name of a function, a Hyperbole hook variable triggered within that function has the same name as the function with a `-hook` appended. Hyperbole provides the following hook variables:

hyperb:init-hook
: For customization at Hyperbole initialization time. Use this to load any personal Hyperbole type definitions you might have. It is run after Hyperbole support code is loaded but before Hyperbole session initializations take place.

action:act-hook
: Run before each Hyperbole button activation. The variable *hbut:current* contains the button to be activated when this is run.

ebut:create-hook
: To add to the Hyperbole explicit button creation process.

ebut:delete-hook
: To add to the Hyperbole explicit button deletion process.

ebut:modify-hook
: Executed when an explicit button's attributes are modified.

hibtypes:begin-load-hook
: Executed prior to loading of standard Hyperbole implicit button types. Used to load site-specific low priority implicit button types since lowest priority ibtypes are loaded first.

hibtypes:end-load-hook

Executed after loading of standard Hyperbole implicit button types. Used to load site-specific high priority implicit button types since highest priority ibtypes are loaded last.

htype:create-hook

Executed when a Hyperbole type (e.g. action type or implicit button type) is added to the environment.

htype:delete-hook

Executed when a type is deleted from the environment.

Hyperbole also makes use of a number of external Emacs hook variables.

find-file-hooks

This is called whenever a file is read into a GNU Emacs buffer. Hyperbole uses it to highlight any buttons within files when run under Epoch or Lucid Emacs, X Window system-based versions of GNU Emacs.

write-file-hooks

This is called whenever a GNU Emacs buffer is written to a file. Hyperbole uses it to save any modified button data associated with the file's directory.

Hyperbole mail and news facilities also utilize a number of external hook variables. These hide button data and highlight buttons if possible. See the various support files for details.

5.2 Creating Types

To define or redefine a single Hyperbole type, you may either:

- move your Emacs point to within the type definition and use `{C-M-x} (eval-defun)` (only works in Emacs Lisp mode);
- or move your point to the end of the last line of the type definition and use `{C-x C-e} (eval-last-sexp)` (works in most modes).

The functions from the `htype` class may be applied to any Hyperbole types, if needed.

The following subsections explain the specifics of Hyperbole type definitions which are beyond standard practice for Emacs Lisp programming. See the definitions of the standard types in '`hactypes.el`' and '`hibtypes.el`' for examples.

5.2.1 Action Types

New forms of explicit buttons may be created by adding new action types to a Hyperbole environment. The file, '`hactypes.el`', provides many examples of working action types.

An action type is created, i.e. loaded into the Hyperbole environment, with the `(defact)` function (which is an alias for `(actype:create)`). The calling signature for this function is given in its documentation; it is the same as that of `(defun)` except that a documentation string is required. (An interactive calling form is also required if the action type has formal parameters and is to be used in explicit button definitions. Implicit buttons never use an action type's interactive form. It is good practice to include an interactive form since the type creator cannot know how users may choose to apply the type.)

An action type's parameters are used differently than those of a function being called. Its interactive calling form is used when an explicit button is created to prompt for type-specific button attributes. The rest of its body is used when a button with that action type is activated. Then the button attributes together with the action type body are used to form an action that is executed in response to the button activation. The action's result is returned to the action caller unless it returns nil, in which case t is returned to the caller to ensure that it registers the performance of the action.

An action type body may perform any computation using Emacs Lisp and Hyperbole functions.

The interactive calling form for an action type is of the same form as that of a regular Emacs Lisp function definition (see the documentation for the Emacs Lisp `(interactive)` form). It may additionally use Hyperbole command character extensions when the form is given as a string. The present extension characters are:

I Prompts for an existing Info node name and file.

M Prompts for a mail message date and the file name it resides in. The mail parameters prompted for by this character code are likely to change in the future.

V Prompts for a Hyperbole view specification. Not yet available for use.

Arguments are read by the functions in Hyperbole's `hargs` class, rather than the standard Lisp `read` functions, in order to allow direct selection of arguments via the Smart Keys.

If an action type create is successful, the symbol that Hyperbole uses internally to reference the type is returned. `Nil` is returned on failure so that you may test whether or not the operation succeeds.

Once you have defined an action type within your present Hyperbole environment, you can create new explicit buttons which use it. There is no explicit button type beyond its action type, so no further work is necessary.

Call `(actype:delete)` to remove an action type from a Hyperbole environment. It takes a single parameter which should be the same type symbol used in the type definition call (not the Hyperbole symbol returned by the call).

5.2.2 Implicit Button Types

An implicit button type is created or loaded via the `(defib)` function (which is an alias for `(ibtype:create)`). The calling signature for this function is given in its documentation; it is the same as that of `(defun)`, but with a number of constraints. The parameter list should always be empty since no parameters will be used. A documentation string is required. The type's body follows this.

The body of an implicit button type is a predicate which determines whether or not point is within an implicit button of the type. If not, the predicate returns `nil`. If so, it may optionally setup to flash the button and then perform one or more actions. A call of the form: `(ibut:label-set label start-pos end-pos)` is used to setup the button flashing, if desired. (This call should be made if you want buttons of the type to flash on at least one display type.) This is then typically immediately followed by an action invocation of the form: `(hact 'actype &rest actype-arguments)`. It is imperative that all actions (non-predicate code) be invoked through the `(hact)` function rather than directly or your ibtypes will not work properly. (Hyperbole first tests to see if any ibtype matches the current context before activating any type, so it ensures that `(hact)` calls are disabled during this testing.) Any action types used may be created before or after the implicit button type definition but obviously should be defined before any implicit buttons of the given type are activated; an error will result, otherwise,

If an implicit button type create is successful, the symbol that Hyperbole uses internally to reference the type is returned. `Nil` is returned on failure so that you may test whether or not the operation succeeds. Implicit button type names and action type names may be the same without any conflict. In fact, such naming is encouraged when an implicit button type is the exclusive intended user of an action type.

Call `(ibtype:delete)` to remove an implicit button type from a Hyperbole environment. It takes a single parameter which should be the same type symbol used in the type definition call (not the Hyperbole symbol returned by the call). This will not delete the action type used by the implicit button; that must be done separately.

By default, a request for help on an implicit button will display the button's attributes in the same manner as is done for explicit buttons. For some implicit button types, other forms of help will be more appropriate. If an Emacs Lisp function is defined whose name is formed from the concatenation of the type name followed by `:help`, e.g. `my-ibtype:help`, it is used to respond to requests for help on buttons of that type. Any such function should take a single argument of an implicit button construct. (This is what `(ibut:at-p)` returns when point is within an implicit button context.) The button may be queried for its attributes using functions from the `hbut` and `hattr` classes. See the '`hib-kbd.el`' file for an example of a custom help function.

5.3 Explicit Button Technicalities

5.3.1 Button Label Normalization

Hyperbole uses a normalized form of button labels called button keys (or label keys) for all internal operations. See the documentation for the function `(hbut:label-to-key)` for details of the normalization process. The normalized form permits Hyperbole to recognize buttons that are the same but whose labels appear different from one another, due to text formatting conventions. For example, all of the following would be recognized as the same button.

```
<(fake button)>     <( fake      button)>

Pam>  <(fake
Pam>    button)>

;; <(fake
;;   button)>

/* <( fake      */
/*    button )> */
```

The last three examples demonstrate how Hyperbole ignores common fill prefix patterns that happen to fall within the middle of a button label that spans multiple lines. As long as such buttons are selected with point at a location within the label's first line, the button will be recognized. The variable *hbut:fill-prefix-regexps* holds the list of fill prefixes recognized when embedded within button labels. All such prefixes are recognized (one per button label), regardless of the setting of the GNU Emacs variable, *fill-prefix*, so no user intervention is required.

5.3.2 Operational and Storage Formats

Hyperbole uses a terse format to store explicit buttons and a more meaningful one to show users and to manipulate during editing. The terse format consists solely of button attribute values whereas the edit format includes an attribute name with each attribute value. A button in edit format consists of a Lisp symbol together with its property list which holds the attribute names and values. In this way, buttons may be passed along from function to function simply by passing the symbol to which the button is attached. Most functions utilize the pre-defined *hbut:current* symbol by default to store and retrieve the last encountered button in edit format.

The `hbdata` class handles the terse, stored format. The `hbut`, `ebut`, and `ibut` classes work with the name/value format. This separation permits the wholesale replacement of the storage manager with another, with any interface changes hidden from any Hyperbole client programming.

5.3.3 Programmatic Button Creation

A common need when developing with Hyperbole is the ability to create or modify explicit buttons without user interaction. For example, an application might require the addition of an explicit summary button to a file for each new mail message a user reads that contains a set of keywords. The user could then check the summary file and jump to desired messages quickly.

The Hyperbole class `ebut` supports programmatic access to explicit buttons. See it within the '`hbut.el`' file for full details. The documentation for `(ebut:create)` explains the set of attributes settings necessary to create an explicit button. For operations over the whole set of buttons within the visible (non-narrowed) portion of a buffer, use the `(ebut:map)` function.

5.4 Encapsulating Systems

A powerful use of implicit button types is to provide a Hyperbole-based interface to external systems. The basic idea is to interpret patterns within the application's displays as implicit buttons. See the '`hsys-*`' files for examples of how to do this.

5.5 Embedding Hyperbole

[NOTE: We have never done this ourselves, though we have done similar things which leads us to infer that the task should not be difficult.]

The standard Emacs-based Hyperbole user interface has purposely been separated from the Hyperbole backend to support the development of alternative interfaces and the embedding of Hyperbole functionality within other system prototypes. The Hyperbole backend functionality that system developers can make use of is called its Application Programming Interface (API). The API may be used to make server-based calls to Hyperbole when Emacs is run as a non-interactive (batch) process, with its input/output streams attached to another process.

The public functions and variables from the following files may be considered the present Hyperbole API:

'`hact.el`', '`hargs.el`', '`hbmap.el`', '`hbut.el`', '`hhist.el`', '`hmail.el`', '`hmoccur.el`', '`hpath.el`', '`htz.el`', '`hypb.el`', '`set.el`', '`wconfig.el`', '`wrolo.el`', and '`wrolo-logic.el`'.

(Note that within a class definition, functions and variables that follow a line of dashes are private.)

Appendix A Glossary

Concepts pertinent to operational usage of Hyperbole are defined here. If some GNU Emacs terms are unfamiliar to you, see section "Glossary" in *the GNU Emacs Manual*.

`action` An executable behavior associated with a Hyperbole button. A specific class of actions which display entities are called *links*, such as a link to a file.

`action type`

A behavioral specification for use within Hyperbole buttons. Action types usually contain a set of parameters which must be given values for each button with which they are associated. An action type together with a set of values, called arguments, may be considered an *action*. *Actype* is a synonym for action type.

`activation`

Request for a Hyperbole button to perform its action. Ordinarily the user presses a key which selects and activates a button.

`ange-ftp` A non-standard GNU Emacs Lisp package which allows one to use pathnames that are accessible via the Internet File Transfer Protocol (ftp) just like other pathnames, for example when finding a file. The latest version of ange-ftp may always be obtained via anonymous ftp to: '`ftp.gnu.ai.mit.edu:ange-ftp/ange-ftp.tar.Z`'.

`argument` A button-specific value fed to a Hyperbole type specification when the button is activated.

`attributes`

Slot names associated with Hyperbole buttons. An *attribute value* is associated with each button attribute.

`button` A selectable Hyperbole construct which performs an action. A button consists of a set of attributes that includes: a textual label, a category, a type and zero or more arguments. *Explicit buttons* also have creator, create time, last modifier, and last modifier time attributes.

Buttons provide the user's gateway to information. The user sees and interacts with button labels, the rest of the button data is managed invisibly by Hyperbole and displayed only in response to user queries.

`button activation`

See *activation*.

`button attributes`

See *attributes*.

`button data`
: Lists of button attribute values explicitly saved and managed by Hyperbole. One list for each button created by Hyperbole.

`button file, local`
: A per-directory file named '`HYPB`' that may be used to store any desired buttons and may then be displayed via a menu selection whenever a user is within that directory.

`button file, personal`
: A per-user file named '`HYPB`' that may be used to store any desired buttons and may then be displayed via a menu selection.

`button key`
: A normalized form of a *button label* used internally by Hyperbole.

`button label`
: A text string that visually indicates a Hyperbole button location and provides it with a name and unique identifier. Within a buffer, buttons with the same label are considered separate views of the same button and so behave exactly alike. Since button labels are simply text strings, they may be embedded within any text to provide non-linear information or operational access points.

  A button is selected by pressing a *Smart Key* within its label. The maximum length of a button label is limited by the variable *ebut:max-len*.

`button selection`
: The act of designating a Hyperbole button upon which to operate.

`category`
: A high-level, conceptual grouping of Hyperbole buttons into classes. *Implicit* and *explicit* groupings represent categories.

`class`
: A group of functions and variables with the same prefix in their names, used to provide an interface to an internal or external Hyperbole abstraction.

`context`
: A programmatic or positional state recognized by Hyperbole. We speak of Smart Key and implicit button contexts. Both are typically defined in terms of surrounding patterns within a buffer, but may be defined by arbitrary Emacs Lisp predicates. (Context may come to have a broader meaning within future versions of Hyperbole.)

  See *Hyperbole environment*.

`explicit button`
: A button created and managed by Hyperbole. By default, explicit buttons are delimited like this `<(fake button)>`. Direct selection is used to operate upon an explicit button.

`global button`
: A form of explicit button which is typically accessed by name rather than direct selection. Global buttons are useful when one wants quick access to actions such as jumping to common file locations or for performing sequences of operations. One need not locate

them since they are always available by name, with full completion offered. All global buttons are stored in the file given by the variable *gbut:file* and may be activated as regular explicit buttons by visiting this file. By default, this is the same as the user's personal button file.

`global button file`

See *button file, personal*.

`hook variable`

A variable that permits customization of an existing function's operation without the need to edit the function's code. See also the documentation for the function `(run-hooks)`.

`Hyperbole`

A flexible, programmable information management and viewing system built on top of GNU Emacs. It utilizes a button-action model and supports hypertextual linkages. Hyperbole is all things to all people.

`Hyperbole environment`

A programmatic context within which Hyperbole operates. This includes the set of Hyperbole types defined and the set of Hyperbole code modules loaded. It does not include the set of accessible buttons. Although the entire Emacs environment is available to Hyperbole, we do not speak of this as part of the Hyperbole environment.

`hypertext`

A text or group of texts which may be explored in a non-linear fashion through associative linkages embedded throughout the text. Instead of simply referring to other pieces of work, hypertext references when followed actually take you to the works themselves.

`implicit button`

A button recognized contextually by Hyperbole. Such buttons contain no button data. See also *implicit button type*.

`implicit button type`

A specification of how to recognize and activate implicit buttons of a specific kind. Implicit button types often utilize structure internal to documents created and managed without Hyperbole assistance, for example, programming documentation. *Ibtype* is a synonym for implicit button type. See also *system encapsulation*.

`instance number`

A colon prefaced number appended to the label of a newly created button when the button's label duplicates the label of an existing button in the current buffer. This number makes the label unique and so allows any number of buttons with the same base label within a single buffer.

`link` A reference from a Hyperbole button to an entity. The referenced entity is sometimes called a *node* or *referent*. A specific class of actions which display entities are called *links*, such as a link to a file.

`local button file`
: See *button file, local*.

`mouse button`

`mouse key` See *Smart Key*.

`node` See *link*.

`predicate`
: A boolean (nil = false, non-nil = true) Lisp expression typically evaluated as part of a conditional expression.

`referent` See *link*.

`Smart Key` A context-sensitive key used within Hyperbole and beyond. Actually, there are two Smart Keys, a primary and a secondary Smart Key, sometimes referred to as the Smart Key and Smart Meta Key, respectively. The primary Smart Key, typically bound to the middle mouse key (or left mouse key on a 2-button mouse), activates Hyperbole buttons and scrolls the buffer forward a screenful when pressed at the end of a line. The secondary Smart Key, typically bound to the right mouse key, explains what a Hyperbole button does or scrolls the buffer backward a screenful when pressed at the end of a line.

To see what a Smart Key will do within a particular context, depress and hold the key at the point desired and depress the other Smart Key. A buffer containing a description of its contextual function will then be displayed. You may release the two keys in any order after you have them both depressed. A press of the secondary Smart Key in an unsupported context displays a summary of Smart Key functions in each context, as does the Doc/SmartKy menu item.

`source buffer / file`
: The buffer or file within which a Hyperbole button is embedded.

`system encapsulation`
: Use of Hyperbole to provide an improved or simply consistent user interface to another system. Typically, implicit button types are defined to recognize and activate button-type constructs managed by the other system.

`view` A perspective on some information. A view can affect the extent of the information displayed, its format, modes used to operate on it, its display location and so forth.

`view spec` A terse (and to the uninitiated, cryptic) string that specifies a particular view of a link referent or some other information.

Appendix B Smart Key Reference

This appendix supplies complete documentation on Smart Key operation. It is quite extensive and is meant for reference rather than sequential reading. See Section 4.2 [Smart Keys], page 17, for a description of the Smart Keys. That section also describes how to get context-sensitive Smart Key help, with which you can explore their operation bit by bit.

Smart Key operations are context-sensitive. Contexts are described herein as conditionals, e.g. when depressed here, if this is true, etc. Each Smart Key context is listed in the order in which it will be checked. The first matching context is always the one applied. Within each context, the actions performed by the primary and secondary Smart Keys are listed.

B.1 Smart Mouse Keys

Smart Key drags and modeline presses can only be used when running under a window system with mouse key support. So keep in mind that the operations in this section apply only if you have mouse support within Hyperbole. The Smart Key operations in, Section B.2 [Smart Keyboard Keys], page 47, apply to both mouse and keyboard Smart Key usage.

```
If dragged from a side-by-side window edge:
  PRIMARY or SECONDARY
     Resizes adjacent window sides to the point of drag release.


If dragged from inside one window to another:
  PRIMARY
     Creates a new link button at the drag start location, linked to the
     drag end location.  If drag start position is within a button,
     modifies the button to link to drag end location.
  SECONDARY
     Swaps buffers in the two windows.


If dragged horizontally within a single window while depressed
(sm-mouse-x-drag-sensitivity sets the minimal horizontal movement which
registers a drag):
  PRIMARY
     Goes to buffer end if drag was to the right, otherwise goes to beginning.
  SECONDARY
     Splits window vertically if drag was to the right, otherwise deletes
     window.
```

```
If depressed within a window mode line:
  PRIMARY
    (1) clicked on left edge of a window's modeline,
        window's buffer is buried (placed at bottom of buffer list);
    (2) clicked on right edge of a window's modeline,
        the Info buffer is displayed, or if already displayed and the
        modeline clicked belongs to a window displaying Info, the Info
        buffer is hidden;
    (3) clicked anywhere in the middle of a window's modeline,
        a Smart Menu (if available) or a popup menu (if available)
        is displayed;
    (4) dragged vertically from modeline to within a window,
        the modeline is moved to point of key release, thereby resizing
        its window and potentially its vertical neighbors.
  SECONDARY
    (1) clicked on left edge of a window's modeline,
        bottom buffer in buffer list is unburied and placed in window;
    (2) clicked on right edge of a window's modeline,
        the summary of Smart Key behavior is displayed, or if already
        displayed and the modeline clicked belongs to a window displaying
        the summary, the summary buffer is hidden;
    (3) clicked anywhere in the middle of a window's modeline,
        a popup menu (if available) is displayed;
    (4) dragged vertically from modeline to within a window,
        the modeline is moved to point of key release, thereby resizing
        its window and potentially its vertical neighbors.


If dragged vertically within a single window while depressed
(sm-mouse-y-drag-sensitivity sets the minimal vertical movement which
registers a drag):
  PRIMARY or SECONDARY
    Splits current window into two side-by-side windows.


If dragged diagonally within a single window while depressed
(sm-mouse-x-diagonal-sensitivity and sm-mouse-y-diagonal-sensitivity set the
minimal diagonal movement which registers a drag):
  PRIMARY
    Save current window configuration onto a ring of window configurations.
  SECONDARY
    Restores prior window configuration from ring.  A prefix argument N
    specifies the Nth prior configuration from the ring.
```

B.2 Smart Keyboard Keys

When prompting for a Hyperbole argument, a press in the minibuffer:
PRIMARY
Terminates this minibuffer argument.
SECONDARY
Offers completion help for current minibuffer argument.

When reading a Hyperbole menu item or a Hyperbole completion-based argument:
PRIMARY
Returns value selected at point if any, else nil. If value is the same as the contents of the minibuffer, it is used as the current minibuffer argument, otherwise, the minibuffer is erased and value is inserted there.

SECONDARY
Displays Hyperbole menu item help when item is selected.

When pressed at the end of a line but not the end of a buffer:
PRIMARY
Scrolls up according to value of smart-scroll-proportional. If smart-scroll-proportional is nil (the default) or if point is on the top window line, scrolls up (forward) a windowful. Otherwise, tries to bring current line to top of window. Leaves point at end of line and returns t if scrolled, nil if not.
SECONDARY
Scrolls down according to value of smart-scroll-proportional. If smart-scroll-proportional is nil (the default) or if point is on the bottom window line, scrolls down (backward) a windowful. Otherwise, tries to bring current line to bottom of window. Leaves point at end of line and returns t if scrolled, nil if not.

When pressed on a Hyperbole button:
PRIMARY
Activates button.
SECONDARY
Displays help for button, typically a summary of its attributes.

When pressed on a Smart Menu item:
PRIMARY
Activates item.
SECONDARY
Displays help for item.

```
When pressed at the end of a Help buffer:
  PRIMARY or SECONDARY
    Restores screen to configuration prior to help display.


When pressed within an OO-Browser listing window:
  PRIMARY
     (1) in a blank buffer or at the end of a buffer, browser help
         information is displayed in the viewer window;
     (2) at the beginning of a (non-single char) class name, the classes
         ancestors are listed;
     (3) at the end of an entry line, scrolls listing up;
     (4) on the '...', following a class name, point is moved to the class
         descendency expansion;
     (5) before a feature name, the implementor classes of the name are listed
     (6) anywhere else on an entry line, the source is displayed for editing.
  SECONDARY
     (1) in a blank buffer, a selection list of buffer files is displayed;
     (2) at the beginning of a (non-single char) entry, the classes descendants
         are listed;
     (3) at the end of an entry line, scrolls listing down;
     (4) on the '...', following a class name, point is moved to the class
         expansion;
     (5) anywhere else on a class entry line, lists the class' features;
     (6) anywhere else on a feature entry line, lists the feature's
         implementors;
     (7) on a blank line following all entries, the current listing buffer
         is exited.


When pressed within an OO-Browser Command Help Menu buffer:
  PRIMARY
     Executes an OO-Browser command whose key binding is at point.
  SECONDARY
     Displays help for an OO-Browser command whose key binding is at point.


When pressed on an identifier within an OO-Browser source file:
  PRIMARY
     Tries to display identifier definition.
  SECONDARY
     Not applicable.
```

```
When pressed within a C code source file:
  PRIMARY
    Jumps to the definition of selected C construct:
    (1) on a '#include' statement, the include file is displayed;
    (2) on a C identifier, the identifier definition is displayed,
        assuming the identifier is found within an 'etags' generated tag file
        in the current directory or any of its ancestor directories.
    (3) if 'smart-c-use-lib-man' is non-nil, the C identifier is
        recognized as a library symbol, and a man page is found for the
        identifier, then the man page is displayed.
  SECONDARY
    Jumps to the next tag matching an identifier at point.


When pressed on a Lisp symbol within a Lisp code buffer:
  PRIMARY
    Jumps to the definition of any selected Lisp construct.
    If on an Emacs Lisp require, load, or autoload clause and 'find-library'
    from load-library package by Hallvard Furuseth <hallvard@ifi.uio.no> has
    been loaded, jumps to library source, if possible.
  SECONDARY
    Jumps to the next tag matching an identifier at point or if using the
    "wtags" package and identifier is an Emacs Lisp symbol, then displays
    documentation for the symbol.


When the OO-Browser has been loaded and the press is within a C++ buffer:
  PRIMARY or SECONDARY
    Jumps to the definition of selected C++ construct via OO-Browser support.
    (1) on a '#include' statement, the include file is displayed;
    (2) within a method declaration, its definition is displayed;
    (3) on a class name, the class definition is shown.


When pressed within an occur-mode or moccur-mode buffer:
  PRIMARY or SECONDARY
    Jumps to the source buffer and line of the current occurrence.
```

```
When pressed within a calendar-mode buffer:
  PRIMARY
    (1) at the end of the buffer, the calendar is scrolled forward 3 months;
    (2) to the left of any dates on a calendar line, the calendar is scrolled
        backward 3 months;
    (3) on a date, the diary entries for the date, if any, are displayed.
  SECONDARY
    (1) at the end of the buffer, the calendar is scrolled backward 3 months;
    (2) to the left of any dates on a calendar line, the calendar is scrolled
        forward 3 months;
    (3) anywhere else, all dates with marking diary entries are marked in the
        calendar window.


When pressed within a man page apropos buffer:
  PRIMARY
    (1) on a UNIX man apropos entry, the man page for that entry is
        displayed in another window;
    (2) on or after the last line, the buffer in the other window is
        scrolled up one screen.
  SECONDARY
    (1) on a UNIX man apropos entry, the man page for that entry is
        displayed in another window;
    (2) on or after the last line, the buffer in the other window is
        scrolled down one screen.


If Smart Menu package has been loaded and 'smart-key-always-display-menu' is
non-nil:
  PRIMARY or SECONDARY
    Pops up a window with a Smart Menu of commands.
    Menu displayed is selected by (smart-menu-choose-menu).
```

If pressed within an outline-mode buffer or when 'selective-display' is non-nil:
  PRIMARY
    Collapses, expands, and moves outline entries.
    (1) after an outline heading has been cut via the smart-key, then paste
        the cut heading at point;
    (2) at the end of buffer, show all buffer text
    (3) at the beginning of a heading line, cut the headings subtree from the
        buffer;
    (4) on a header line but not at the beginning or end, if headings
        subtree is hidden then show it, otherwise hide it;
    (5) anywhere else, scroll up one screen.
  SECONDARY
    (1) after an outline heading has been cut via the smart-key, allow
        multiple pastes throughout the buffer (last paste should be done
        with smart key, not meta smart key);
    (2) at the end of buffer, hide all bodies in buffer;
    (3) at the beginning of a heading line, cut the current heading (sans
        subtree) from the buffer;
    (4) on a header line but not at the beginning or end, if heading body is
        hidden then show it, otherwise hide it;
    (5) anywhere else, scroll down one screen.

If pressed within an Info manual node:
  PRIMARY
    (1) the first line of an Info Menu Entry or Cross Reference, the desired
        node is found;
    (2) the Up,Next,or Previous entries of a Node Header (first line),
        the desired node is found;
    (3) the File entry of a Node Header (first line),
        the 'Top' node within that file is found;
    (4) at the end of the current node, the Next node is found (this will
        descend subtrees if the function 'Info-global-next' is bound);
    (5) anywhere else (e.g. at the end of a line), the current node entry is
        scrolled up one screen
  SECONDARY
    (1) the first line of an Info Menu Entry or Cross Reference, the desired
        node is found;
    (2) the Up,Next,or Previous entries of a Node Header (first line),
        the last node in the history list is found;
    (3) the File entry of a Node Header (first line),
        the 'DIR' root-level node is found;
    (4) at the end of the current node, the Previous node is found (this will
        return from subtrees if the function 'Info-global-prev is bound);
    (5) anywhere else (e.g. at the end of a line), the current node entry is
        scrolled down one screen

```
If pressed within a Hyperbole-supported mail reader, 'hmail:reader', or mail
summary mode, 'hmail:lister', buffer at:
  PRIMARY
    (1) a msg buffer, within the first line or at the end of a message,
        the next undeleted message is displayed;
    (2) a msg buffer within the first line of an Info cross reference, the
        reference is followed;
    (3) anywhere else in a msg buffer, the window is scrolled up one screen;
    (4) a msg summary buffer on a header entry, the message corresponding to
        the header is displayed in the msg window;
    (5) a msg summary buffer, on or after the last line, the messages marked
        for deletion are expunged.
  SECONDARY
    (1) a msg buffer, within the first line or at the end of a message,
        the previous undeleted message is displayed;
    (2) a msg buffer within the first line of an Info cross reference, the
        reference is followed;
    (3) anywhere else in a msg buffer, the window is scrolled down one
        screen;
    (4) a msg summary buffer on a header entry, the message corresponding to
        the header is marked as deleted;
    (5) a msg summary buffer, on or after the last line, all messages are
        marked undeleted.
```

```
If pressed within a GNUS listing of newsgroups buffer at:
  PRIMARY
    (1) a GNUS-GROUP line, that newsgroup is read;
    (2) to the left of any GNUS-GROUP line, on any of the whitespace, the
        current group is unsubscribed or resubscribed;
    (3) at the end of the GNUS-GROUP buffer, after all lines, checks for new
        news.
  SECONDARY
    (1) a GNUS-GROUP line, that newsgroup is read;
    (2) to the left of any GNUS-GROUP line, on any of the whitespace, the
        user is prompted for a group name to subscribe or unsubscribe to;
    (3) at the end of the GNUS-GROUP buffer, after all lines, quits from the
        newsreader.
```

```
If pressed within a GNUS newsreader subject listing buffer at:
  PRIMARY
    (1) a GNUS-SUBJECT line, that article is read, marked deleted, and
        scrolled forward;
    (2) at the end of the GNUS-SUBJECT buffer, the next undeleted article
        is read or the next group is entered.
  SECONDARY
    (1) a GNUS-SUBJECT line, that article is read and scrolled backward;
    (2) at the end of the GNUS-SUBJECT buffer, the subject is exited, the
        user is returned to group mode.
```

If pressed within a GNUS newsreader article buffer at:
PRIMARY
(1) the first line or end of an article, the next unread message is displayed;
(2) the first line of an Info cross reference, the reference is followed;
(3) anywhere else, the window is scrolled up one screen.
SECONDARY
(1) the first line or end of an article, the previous message is displayed;
(2) the first line of an Info cross reference, the reference is followed;
(3) anywhere else, the window is scrolled down one screen.

If pressed within a listing of buffers (Buffer-menu-mode):
PRIMARY
(1) on the first column of an entry, the selected buffer is marked for display;
(2) on the second column of an entry, the selected buffer is marked to be saved;
(3) anywhere else within an entry line, all saves and deletes are done, and selected buffers are displayed, including the one just clicked on (if in the OO-Browser, only the selected buffer is displayed);
(4) on or after the last line in the buffer, all saves and deletes are done.
SECONDARY
(1) on the first or second column of an entry, the selected buffer is unmarked for display and for saving or deletion;
(2) anywhere else within an entry line, the selected buffer is marked for deletion;
(3) on or after the last line in the buffer, all display, save, and delete marks on all entries are undone.

If pressed within a dired-mode buffer:
PRIMARY
(1) within an entry line, the selected file/directory is displayed for editing;
(2) on or after the last line in the buffer, if any deletes are to be performed, they are executed after user verification, otherwise, this dired invocation is quit.
SECONDARY
(1) on a '~' character, all backup files in the directory are marked for deletion;
(2) on a '#' character, all auto-save files in the directory are marked for deletion;
(3) anywhere else within an entry line, the selected buffer is marked for deletion;
(4) on or after the last line in the buffer, all delete marks on all entries are undone.

```
If pressed with a buffer in View mode:
  PRIMARY
     Scrolls buffer forward a windowful.
  SECONDARY
     Scrolls buffer backward a windowful.


If pressed on a cross reference within a man page entry section labeled
NAME, SEE ALSO, or PACKAGES USED, or within a man page C routine
specification (see 'smart-man-c-routine-ref') and man page buffer
has either an attached file or else a 'man-path' local variable containing
its pathname:
  PRIMARY or SECONDARY
     Displays man page or code source for cross reference.


If pressed within the wrolo match display buffer:
  PRIMARY
     (1) At the start of a rolodex match entry, the entry is edited in the
         other window;
     (2) anywhere else within an entry line, toggles between displaying only
         the selected rolodex item and displaying all matches.
  SECONDARY
     (1) At the start of a rolodex match entry, the entry is edited in the
         other window;
     (2) anywhere else within an entry line, toggles between displaying only
         the selected rolodex item and displaying all matches.
```

Appendix C Suggestion or Bug Reporting

See the 'README' file in the Hyperbole distribution directory for complete details on Hyperbole mailing lists and how to subscribe.

If you find any errors in Hyperbole's operation or documentation, feel free to report them to the Hyperbole discussion list: hyperbole@cs.brown.edu. Be sure to use the Msg/Compose-Hypb-Mail menu item whenever you send a message to the mail list since it will insert important system version information for you.

If you use Hyperbole mail or news support, see Section 4.6.7 [Buttons in Mail], page 27, a click with your primary Smart Key on the Hyperbole mail list address will insert a description of your Hyperbole configuration information into your outgoing message, so that you do not have to type it. This is useful when composing a reply for the Hyperbole mail list. Otherwise, be sure to include your Emacs, Hyperbole and window system versions in your message. Your Hyperbole version number can be found in the top-level Hyperbole menu.

Please use your subject line to state the position that your message takes on the topic that it addresses, e.g. send "Subject: Basic bug in top-level Hyperbole menu." rather than "Subject: Hyperbole bug". This simple rule makes all e-mail communication much easier.

If you have suggestions on how to improve Hyperbole, send them to the same address. Here are some issues you might address:

What did you like and dislike about the system?

What kinds of tasks, if any, does it seem to help you with?

What did you think of the Emacs-based user interface?

How was the Hyperbole Manual and other documentation?

Was the setup trivial, average or hard?

What areas of Hyperbole would you like to see expanded/added?

How does it compare to other hypertext tools you have used?

Was it easy or difficult to create your own types? Why?

Did you get any use out of the external system encapsulations?

Appendix D Questions and Answers

How can I change the Smart Mouse Key bindings?

Since the Smart Mouse Keys are set up for use under five different Emacs configurations, there is no easy way to provide user level customization. Therefore, to change the bindings you must edit the `sm-mouse-setup` and `sm-mouse-get-bindings` functions in the '`hmouse-key.el`' file.

The *smart-key-alist* and *smart-key-mouse-alist* variable settings in '`hui-mouse.el`' and '`hui-window.el`' must be altered if you want to change what the Smart Keys do in particular contexts. You should then update the Smart Key summary documentation in the file, '`hmouse-doc.txt`', and potentially the same summary in this manual.

Missing Action Types

What if someone sends a mail message with a button for which I do not have the action type? Or a button whose link referent I can't access?

You receive an error that an action type is not defined or a link referent is not accessible/readable if you try to use the button. This is hardly different than trying to get through a locked door without a key; you try the doorknob, find that it is locked, and then realize that you need to take a different approach or else give up.

Like all communication, people need to coordinate, which usually requires an iterative process. If you get a mail message with a button for which you don't have the action type, you mail the sender and request it.

How can I modify a number of global buttons in succession?

Rather than typing the name for each, it is quicker to jump to the global button file and edit the buttons there as you would any explicit buttons. By default, the ButFile/PersonalFile menu item takes you to the file where global buttons are saved.

Why is all the button data scattered across directories?

When you think of a hyper-space that you depend on every day, you don't want to have a single point of failure make you incapable of doing work. With Hyperbole, if some directories become unavailable for a particular time (e.g. the filesystems on which they reside are dismounted) you can still work elsewhere with minimal effect. We believe this to be a compelling factor to leave the design with external button data storage.

This design also permits the potential addition of buttons to read-only media.

Why are action types defined separately from their implicit button types?

Any category of button can make use of an action type. Some action types are useful as behavior definitions for a variety of button categories, so all action types are defined separately to give them independence from those types which apply them.

For implicit button types that require a lot of code, it is useful to add a module that includes the implicit button type definition, its action type definition and supporting code.

Appendix E Ongoing Work

This appendix is included for a number of reasons:

- to better allow you to assess whether to work with Hyperbole by providing sketches of possible additions (only minimal work is done on Hyperbole at the present time, however);
- to direct further development effort towards known needs;
- and to acknowledge known weaknesses in the current system.

Note that due to a lack of volunteers to further develop Hyperbole, much of this work may not be done. So if you want to see these features, encourage qualified people to volunteer.

Button Copying, Killing, and Yanking

There is as yet no means of transferring buttons among buffers. We realize this is a critical need. Users should be able to manipulate text with embedded buttons in ordinary ways. Some of the Emacs Lisp functions that must be augmented to support this behavior include:

```
(append-next-kill)
(copy-region-as-kill)
(delete-region) ;; a built-in function
(get-register)
(insert-buffer-substring)   ;; a built-in function
(insert-file-contents)
(kill-append)
(kill-region)
(rotate-yank-pointer)
(set-register)
(view-register)
(yank)
(yank-pop)
(undo)
```

Hyperbole Views

These will permit rapid perspective shifting when studying and manipulating bodies of text. They will also allow fine control over the presentation of information displayed by action types.

Trails

Trails are an extension to the basic history mechanism presently offered by Hyperbole. Trails will allow a user to capture, edit and store a specific sequence and set of views of information for later replay by other users. Conditional branching may also be supported.

Storage of button data within button source files.

The current design choice of storing buttons external to the source file was made under the assumption that people should be able to look at files that contain Hyperbole buttons with any standard editor or tool and not be bothered by the ugly button data (since they won't be able to utilize the buttons anyway, they don't need to see or have access to them).

In many contexts, embedding the button data within the source files may be a better choice, so a provision which would allow selection of either configuration may be added. Here are some of the PROs and CONs of both design choices:

```
                      POSITIVE                        NEGATIVE

     Bdata in src file
                      Documents can stand alone.      All edit operators have
                      Normal file operations apply.   to account for file
                                                      structure and hide
                      Simplifies creation and         internal components.
                      facility expansion for
                      structured and multi-media
                      files.

     Bdata external to src file
                      Files can be displayed and      Currently, bdata for
                      printed exactly as they look.   whole directory is
                      No special display formatting   locked when any bdata
                      is necessary.                   entry is locked.

                      Button-based searches and
                      database-type lookup operations
                      need only search one file
                      per directory.
```

NLS/AUGMENT-type File Structuring Being Investigated

NLS/AUGMENT treated files as a hierarchical set of nodes, for example, each paragraph would be treated as a node, with its own address. This would aid greatly in idea structuring and cross-referencing. Don't expect anything soon on this front, but keep in mind that we are interested in hearing about any recent related work.

Forms-based Interfaces

This will allow one to create buttons more flexibly. For example, button attributes could be given in any order. Entry of long code sequences, quick note taking and cross-referencing would also be made easier.

Collaboration Support

From the early stages of Hyperbole design, collaborative work environments have been considered. A simple facility has demonstrated broadcast of button activations to a number of workstations on a local area network, so that one user can lead others around an information space, as during an online design review. (This facility is not yet ready for release.) We shall do some work in specific collaborative mechanisms, but also expect that others who concentrate in collaborative work will provide more extensive capabilities.

Appendix F References

[AkMcYo88]

Akscyn, R. M., D. L. McCracken and E. A. Yoder. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. *Communications of the ACM*, Vol. 31, No. 7, July 1988, pp. 820-835.

[Bro87] Brown, P. J. Turning Ideas into Products: The Guide System. *Proceedings of Hypertext '87*, November 13-15, 1987, Chapel Hill, NC. ACM: NY, NY, pp. 33-40.

[Con87] Conklin, Jeff. Hypertext: An Introduction and Survey. *IEEE Computer*, Vol. 20, No. 9, September 1987, pp. 17-41.

[Eng68] Engelbart, D., and W. English. A research center for augmenting human intellect. *Proceedings of the Fall Joint Computer Conference*, 33, 1, AFIPS Press: Montvale, NJ, 1968, pp. 395-410.

[Eng84a] Engelbart, D. C. Authorship Provisions in Augment. *Proceedings of the 1984 COMPCON Conference (COMPCON '84 Digest)*, February 27-March 1, 1984, San Francisco, CA. IEEE Computer Society Press, Spring, 1984. 465-472. (OAD,2250,)

[Eng84b] Engelbart, D. C. Collaboration Support Provisions in Augment. *Proceedings of the AFIPS Office Automation Conference (OAC '84 Digest)*, February, 1984, Los Angeles, CA, 1984. 51-58. (OAD,2221,)

[Fos88] Foss, C. L. Effective Browsing in Hypertext Systems. *Proceedings of the Conference on User-Oriented Content-Based Text and Image Handling (RIAO 88)*, March 21-24, MIT, Cambridge MA. Centre de Hautes Etudes Internationales d'Informatique Documentaire, 1988, pp. 82-98.

[GaSmMe86]

Garrett, N., K. E. Smith and N. Meyrowitz. Intermedia: Issues, Strategies, and Tactics in the Design of a Hypermedia Document System. *Computer-Supported Cooperative Work (CSCW '86) Proceedings*, December 3-5, Austin, TX, 1986, pp. 163-174.

[HaMoTr87]

Halasz, F. G., T. P. Moran and R. H. Trigg. NoteCards in a Nutshell. *Proceedings of the CHI and GI '87 Conference on Human Factors in Computing Systems*, Toronto, J. M. Carroll and P. P. Tanner, (editors), ACM: NY, NY, April 1987, pp. 45-52.

[Har88] Harvey, G. *Understanding HyperCard.* Alameda, CA: SYBEX, Inc., 1988.

[KaCaLoLa92]

Kaplan, S., A. M. Carroll, C. Love and D. M. LaLiberte. *Epoch 4.0 Manual.* Department of Computer Science, University of Illinois, Urbana-Champaign, March 1992.

[KaKaBeLaDr90]
Kaplan, S. J., M. D. Kapor, E. J. Belove, R. A. Landsman, and T. R. Drake. AGENDA: A personal Information Manager. *Communications of the ACM*, No. 33, July 1990, pp. 105-116.

[Nel87a] Nelson, T. H. *Computer Lib/Dream Machines.* MicroSoft Press, Redmond, WA, 1987.

[Nel87b] Nelson, T. H. *Literary Machines, Edition 87.1.* Available from the Distributors, 702 South Michigan, South Bend, IN 46618, 1987.

[NoDr86] Norman, D. A. and S. W. Draper, editors. *User Centered System Design.* Lawrence Erlbaum Associates: Hillsdale, New Jersey, 1986.

[Shn82] Shneiderman, B. The future of interactive systems and the emergence of direct manipulation. *Behavior and Information Technology*, Vol. 1, 1982, pp. 237-256.

[Sta87] Stallman, R. *GNU Emacs Manual.* Free Software Foundation, Cambridge: MA, March 1987.

[Tri86] Trigg, R., L. Suchman, and F. Halasz. Supporting collaboration in NoteCards. *Proceedings of the CSCW '86 Conference*, Austin, TX, December 1986, pp. 147-153.

[TrMoHa87]
Trigg, R. H., T. P. Moran and F. G. Halasz. Adaptability and Tailorability in NoteCards. *Proceedings of INTERACT '87*, Stuttgart, West Germany, September 1987.

[Wei92] Weiner, B. *PIEmail: A Personalized Information Environment Mail Tool.* Department of Computer Science Masters Project, Brown University: Providence, RI, May 10, 1992.

[YaHaMeDr88]
Yankelovich, N., B. J. Haan, N. Meyrowitz and S. M. Drucker. Intermedia: The Concept and the Construction of a Seamless Information Environment. *IEEE Computer*, Vol. 21, No. 1, January 1988, pp. 81-96.

[YoAkMc89]
Yoder, E. A., R. M. Akscyn and D. L. McCracken. Collaboration in KMS, A Shared Hypermedia System. *Proceedings of the 1989 ACM Conference on Human Factors in Computer Systems (CHI '89)*, April 30-May 4, 1989, Austin, TX, ACM: NY,NY, 1989, pp. 37-42.

Key Binding Index

C

M

Q

Code and File Index

M

R

S

W

Concept Index

A

B

C

D

E

F

G

H

I

L

Short Contents

Table of Contents