HyperNeWS1.4 Reference Manual

The Turing Institute

September 1990

HyperNeWS was written by Arthur A. van Hoff. The graphical design was masterminded by Frank Bollen. The manual was written from the contributions of Danny Pearce, Jim Rudolf, Arthur van Hoff, Suhayya Abu-Hakima, Sunil Sharma and Tim Niblett. Special thanks to Douglas Scoular for help with distribution of HyperNeWS.

We would also like to thank all the users of HyperNeWS, who have provided useful feedback.

The General Manager
The Turing Institute
George House
36 North Hanover Street
Glasgow G1 2AD
United Kingdom
Tel: +44 41 552 6400
Fax: +44 41 552 2985
E-mail: newsdev@turing.ac.uk

Contents

List of Figures

1 Introduction

1.1 Contents of this Document

Welcome to the HyperNeWS Reference Manual. The purpose of this document is to give you more detailed and advanced information on HyperNeWS. A companion document, the HyperNeWS User Guide, gives more general information that is aimed at the beginning user of HyperNeWS. This Reference Manual consists of:

- Section 1, "Introduction," the section you are reading now.
- Section 2, "The C Interface," talks about writing C client applications for your stacks.
- Section 3, "The Prolog Interface," similarly discusses interfacing HyperNeWS stacks with Prolog programs.
- Section 4, "The Lisp Interface," demonstrates how a Lisp client can communicate with a stack.
- Section 5, "Utilities," describes the `drawps`, `dvi2ps`, and `plot2` utilities that come with HyperNeWS.
- Section 6, "Stacks with Application Interfaces," discusses how to access some existing system stacks from PostScript.
- Section 7, "Customizing Canvas Objects," gives some basic guidelines to follow when you are adding methods to canvases, which are the most commonly modified objects.
- Section 8, "Creating New Objects," shows you how to create a new object type that is based upon an existing object type.
- Section 9, "System Methods and Instance Variables," is a fairly exhaustive list of the methods and instance variables for the HyperNeWS classes.

At some point you may notice that this document is remarkably similar to the documentation online under HyperNeWS. There is good reason for this, as both versions are generated from the same source. So if you are using HyperNeWS and you don't have this guide nearby, you can find the same information in the Hyperman stack on the system. References to other sections or figures (such as, "See Section 8") will appear in red online, and double-clicking on one will take you to that particular part of the manual.

HyperNeWS version 1.4 has been placed on various anonymous FTP sites in the United States and Europe. It is also available via mail server in the United

Kingdom. For further information, please contact the Turing Institute at the e-mail or postal address listed on page 2.

HyperNeWS is the copyright of The Turing Institute. It is free to be used and modified for academic, research, and personal endeavors. A brief acknowledgement is required in any research paper or other publication where this software has made a significant contribution. The Turing Institute offers hardcopy manuals, HyperNeWS on quarter-inch cartridge tape (for those without network access), and consulting services. HyperNeWS may also be licensed for use in commercial products. Please contact the Turing Institute for prices and conditions.

1.2 Changes in Release 1.4

HyperNeWS1.4 has undergone some noticeable and some not-so-noticeable changes since release 1.3. Here is a short list of those changes:

- The new release requires OpenWindows2.0. In the past we have tried to maintain HyperNeWS on the available environments as long as it was practical. In this case, OpenWindows2.0 uses the olwm window manager, and Sun is no longer supporting pswm which was used in OpenWindows1.0. There are enough differences between pswm and olwm that it was not practical for us to attempt to support both versions of OpenWindows.

- The functionality of GoodNeWS has been merged with HyperNeWS. Now you will see just one style of terminal emulator, and exiting HyperNeWS can be a one-step process unlike release 1.3 where you had to exit HyperNeWS and GoodNeWS in separate steps.

- The installation process and the filesystem structure have changed. A configure script should help to make installation easier, and there is now only 1 *make install* that needs to be done, versus two in release 1.3. You can specify where you want the binaries, libraries, and fonts to go. The directories are no longer required to be something like *bin-sun4-xnews*. The filesystem has been split into a source/binary style, so you can remove the source tree after installation, and the files required to run HyperNeWS will have been installed in the binary tree.

- Some GoodNeWS applications have been ported to stacks. These include the performance monitor and the clipboard. The Clipboard stack is a multi-card clipboard that can hold 16 different pieces of information at once. The user's interface is described in the Users Guide, and the programmer's interface is described in Section 6. As of yet, we have not ported the LaTeX previewer, *dvinews*.

- The close button in the upper-right corner of some stacks will now iconify the stack. The icons are themselves small stacks, and you move them around like you would any stack, using the AdjustButton. Click on the icon once with the PointButton to open the application again.

- Every stack now has a resizable attribute, which is initially set to true. Along with that are values that let you specify the minimum allowed width and height of the stack when a stack is resizable. If you try to make a stack smaller than it is allowed, the rubber band box will not get any smaller than its minimum allowed size.

- When GoodNeWS went away, so went the GoodNeWS background menu. In a step towards better integration with OpenWindows, you will get the OpenWindows menu when you click the MenuButton over the background if the system stack is in browse mode. If the system stack is in edit mode, you'll see the edit menu as before.

- To exit HyperNeWS, you have two choices. If you select *Exit* from the OpenWindows menu, you will exit the entire window system. But if you click on the *Exit HyperNeWS* system button, you will leave HyperNeWS but remain in OpenWindows. This is very convenient if you are developing an application and you want to restart HyperNeWS without having to restart OpenWindows too.

- The documentation has been split into two parts, the User Guide and the Reference Manual. The former presents introductory topics, while the latter has more advanced material. It is not possible to generate a hardcopy version of the online manual, but they are available from the Turing Institute at minimal cost.

- The process of saving stacks and drawings as ASCII files has been changed so that lines will not exceed 80 characters and the order in which objects are saved will be consistent. In the past, some long lines would be truncated if you tried to mail a stack. These changes will make it possible to control stack development using SCCS. Unless you actually look at the saved stack file, you probably won't notice this change.

- The way Canvas objects are drawn and updated has been made more straightforward. This may affect your existing stacks that have canvases. See Section 7 for a description of how Canvas object scripts should be structured.

- All stacks have been modified so they will look better when HyperNeWS is viewed on a monochrome display.

- There are a few small bugs and quirks to be aware of:

  - HyperDraw may have trouble importing some EPSF drawings.
  - HyperNeWS will not exhibit blinding speed on a 3/50 with only 4MB. In fact, the terminal emulator may give you fits.
  - OpenWindows2.0 beeps once as it is starting up. This is normal. It does not mean something has gone wrong.
  - Sometimes the cursor, represented as a HyperNeWS hand or an OpenWindows arrow, does not go away when you exit OpenWindows. It will remain on your screen until you reboot or restart OpenWindows.
  - Sun has acknowledged a problem with input focus when you have both NeWS and X applications running. So even if you have follow-mouse for an OpenWindows focus setting, you may sometimes have to click in the window to get the focus to change when you switch between X and NeWS applications.
  - Depending upon your hardware, you may get a message when you start OpenWindows2.0 saying something about needing a GFX revision of SunOS in order to work properly. We haven't noticed any unusual behavior when this message appears.

- Despite the few glitches mentioned above, numerous bugs have been fixed, many of which were found by users. Keep those comments and suggestions coming!

2 The C Interface

2.1 Basics

The C interface to HyperNeWS allows users to connect C processes to HyperNeWS stacks. Each stack has exactly one client, although one client can be connected to several stacks. In principle, this works in the same way as the connection of C processes to the standard X/NeWS environment. Because HyperNeWS operates at a higher level than the standard toolkits and performs more work in PostScript, it is possible to run interfaces at a higher level of abstraction with little or no graphics work being done by C. All that the C process needs to know is the names and functionality of the HyperNeWS objects it is interfacing.

Each C process connects to one or more HyperNeWS stacks. It is possible to find out which processes are connected by looking at the SystemInformation stack. When HyperNeWS starts up, the `hnserver` process is attached to the system stack. This process provides timing events and error beeps for all HyperNeWS objects.

The client interface with HyperNeWS is implemented using Unix sockets. This low-level interface is implemented in C, so any language that has a standard C interface can be linked to the low-level routines. The client program sends short PostScript programs over this socket to the server where the code is executed. The client may in fact be running on a separate machine, since sockets work across a network. Communication must be established at two levels:

1. A client program must establish low-level communication with HyperNeWS over a socket. This enables messages to be sent and received. However, this level is too general and access must be constrained.

2. Once connected at a low level, the client must establish communication with each required stack.

The sending and receiving of messages to and from HyperNeWS requires some knowledge of PostScript. A set of higher-level routines have been implemented to make this communication simpler. These routines are available in Prolog and Lisp, as well as C. These routines form a set of common functions for program communication. However, if some desired functionality is not available, you are free to augment the basic set by adding your own routines.

When a client exists, it is treated as just another object. Therefore, communication between the client and the stack (or objects on the stack) is done via message passing. From the PostScript side there are two ways in which a message can be sent to the stack's client:

1. An object on the stack or the stack itself can send the message directly to the client. For example, to send the message `/foo` with an integer argument, the following code segment would be used:


```
[10] /foo ClientSend
```

2. A message can also reach the client as a result of message *percolation*. If a message is sent to an object that cannot handle the message itself, it will pass the message to the object above it in the message hierarchy. For instance, if a message `/foo` is sent to a PullDown object but it doesn't understand the message, it will pass the message to its parent in the message hierarchy, which is the card object that the menu resides on. Similarly, if the card cannot handle the message, it will be passed to the background, the stack, and finally the client if one exists.

On the client side, messages are sent and received using a specialized data structure defined in `hypernews.h`. The structure provides a type-secure interface which catches most programming errors. The main technical problem with the message passing is its asynchronous nature. Messages can be sent from HyperNeWS at arbitrary times, outside the control of the client. These messages are queued. Sometimes the client sends a message, for example a request for a value, for which it requires an immediate answer. The C interface provides a synchronization mechanism which allows this type of synchronization to take place. The details of the mechanism are described in Section 2.3.6

2.2 Representing HyperNeWS Objects in C

In order for a client program to access a stack or objects on the stack, it must be able to reference the objects. HyperNeWS introduces a new data type called `hn_any` that can be used to represent HyperNeWS objects in the C world. Actually, a C variable of type `hn_any` can be used to represent HyperNeWS objects and the different PostScript types, with the exception of dictionaries:

```
typedef struct hn_any {
       hn_any_type     type;
       int             refcount;
       union   {
              double                number;
              int                   boolean;
              char                  *string;   /* null terminated */
              char                  *name;     /* null terminated */
              struct {
                     struct hn_any  *target;   /* object */
                     char           *name;     /* string */
                     struct hn_any  *args;     /* array of arguments */
              } message;
              struct {
                     int            argc;
                     struct hn_any  **argv;    /* array of hn_any */
              } array;
```



```
                struct {
                        char            *stack;     /* stack name (or null) */
                        hn_type         type;       /* object type nr */
                        char            *name;      /* object name */
                } object;
                /* nulltype has no value */
        } u;
} hn_any;
```

Apart from built-in PostScript objects such as numbers and strings, there are two HyperNeWS types, the message and the object. The data carried by these types (i.e. their values) is stored in the `hn_any` type.

As can be seen from the figure, data is tagged with its type, which allows maximum flexibility in the user interface and also permits type checking on data at run-time. This is done with the `type` field, which can represent `null`, numbers, booleans, strings, names, arrays, messages, or HyperNeWS objects:

```
typedef enum {
        hn_number_type,
        hn_boolean_type,
        hn_string_type,
        hn_name_type,
        hn_array_type,
        hn_null_type,
        hn_message_type,
        hn_object_type
} hn_any_type;
```

Be aware that the field `any->u.number` is used to pass both integers and reals between C and HyperNeWS. Since the field is defined as `double`, it is advised that the receiver of a message with an integer argument coerce the argument. That is, use `cvi` in PostScript, or do something like the following in C:

```
hn_any  *any;
int     i;
i = (int) any->u.number;
```

The `message` field contains three subfields:

target

An object representing the sender of a message from HyperNeWS, or the target of a message to HyperNeWS.

name

An object representing the name of the message sent. From C this might be `SetValue` for example.

args

An array of objects representing the arguments of the message.

The `array` structure contains two fields, a count of the number of elements in the array (`argc`) and the array of arguments (`argv`).

The `object` field contains three arguments:

stack

This is a string representing the stack on which the object lives. If the object is a stack, this field is the null object.

type

This field has type `hn_type`. It is an enumerated type, listing the HyperNeWS object types:

```
typedef enum {
        hn_Stack = 1,
        hn_BackGround,
        hn_Card,
        hn_Button,
        hn_EditText,
        hn_Slider,
        hn_PullDown,
        hn_ColorSelect,
        hn_Canvas
} hn_type;
```

name

This is the name of the object.

2.3 Writing a Client Program

2.3.1 Basics

A number of steps must be taken by a client process in order to communicate with a stack:

1. Initialize the HyperNeWS environment.
2. Define C variables for the stack and any other objects with which the client will communicate.
3. Define the messages that the client can expect to receive from the HyperNeWS stack.

4. Establish a communication link with the stack(s).
5. Interact with the stack. This might be an event loop that waits for messages from the stack, or it might be processing done in the client with the results sent to the stack as messages.

Now we'll cover each of these steps in more detail.

2.3.2 Initialize the Environment

Once your client program begins, it must initialize a few things and connect to the HyperNeWS environment. Those steps are accomplished with the following command:

```
hn_start(progname);
```

where `progname` is the name of the client program.

2.3.3 Define Objects

Most likely, your client program is going to want to reference a stack and some of the objects on the stack. In order to do so, your program needs to have a way to refer to these objects. The code:

```
hn_any *stack;
stack = hn_new_object((char *)NULL,hn_Stack,"MyStack");
```

will return in `stack` a pointer to a stack called MyStack. Likewise, the code segment:

```
hn_any  *button1;
button1 = hn_new_object("MyStack",hn_Button,"CloseButton");
```

will return a pointer to a button called *CloseButton* located on MyStack. The arguments to `hn_new_object` are the name of the stack the object resides on (or `NULL` if the object is itself a stack), the type of the object, and the name of the object. As you'll recall, the object types were shown earlier in `hn_any_type`.

2.3.4 Define Messages

In most cases, communication will be two-way; that is, messages will be sent from the stack to the client, and vice versa. The client must know what messages to expect, so those messages have to be *registered*. Along with knowing the names of the messages, the client needs to know what object sent the message, and how many arguments (if any) to expect when that message is received. This makes it possible to have messages with the same name dealt with by different routines, based upon what object sent the message.

Messages are registered in the following way:


```
hn_register(myproc,
      hn_new_message(button1,"Pressed",hn_new_array_body(0)));
```

The arguments for the `hn_register()` routine are the routine to be called when a message is received, and the definition of the message itself. Routines like `myproc()` are called *handlers*. They are usually not invoked from within the client program, but instead are invoked when a certain message is received by the client. The arguments for the `hn_new_message()` routine are the sender of the message, the message sent, and an array body big enough to hold the arguments that will be passed with the message. In this case, no arguments are passed with the *Pressed* message, so an array body of size 0 is specified. The result of all this is that `myproc()` will be called every time the *CloseButton* on MyStack is pressed.

2.3.5 Connect to the Stack

Now that all the necessary objects and messages have been defined, the client needs to actually form a connection with the appropriate stack. This can be done with the command:

```
hn_connect(stack);
```

which will make a connection between the client and stack called MyStack defined above. The client is now able to exchange messages with the stack.

2.3.6 Interact with the Stack

At this point, you are ready to communicate with the stack. The most common type of interaction is the event loop, where the client waits for button clicks or menu selections from the user, and performs certain functions accordingly. HyperNeWS provides an event loop mechanism using the following call:

```
hn_listen(timeout);
```

where if a message is not received in `timeout` milliseconds, a timeout message will be sent to the client. (See Section 2.3.7 for a more complete explanation.)

The `hn_listen()` routine is meant for asynchronous message sending to the client. For applications that require sychronous communication, the `hn_get()` and `hn_put()` routines can be used to read/write values to/from a stack. For example:

```
hn_any  *slider,*value;
value = hn_get(slider,"Value");
```

will return the `Value` instance variable from the object pointed to by `slider`.

While the client must register messages that it expects to receive from the stack, no such registration is required for sending messages from a stack to a client. The code to send a message to a stack might look like this:

```
hn_any  *args,*prompt;
prompt = hn_new_string("Press Cancel or OK");
args = hn_new_array(prompt,NULL);
hn_send(dialog,"SetValue",args);
```

The `hn_new_string()` routine returns a HyperNeWS object that represents a string. The string is then made the first element of a HyperNeWS argument array, and passed to an object called `dialog`. All message arguments must be passed within an array, not unlike the process of passing message arguments in PostScript.

2.3.7 Messages Generated by HyperNeWS

There are a number of messages generated by HyperNeWS that are sent to clients automatically under certain circumstances. The client can choose to register any of the messages or ignore them. Those messages are:

hn_quit

> This message is sent to the client when HyperNeWS is exiting. The client should be prepared to clean up and exit upon receipt of this message.

hn_timeout

> In Section 2.3.6, `hn_listen()`was called with a timeout value. If a message is not sent to the client within the timeout period, this message will be sent to the stack. A possible use for this message is in a client that is a timer process and wants to periodically update a dial on the stack. Setting a 20 second timeout value would cause `hn_timeout` to be sent to the client 3 times a minute, after which the client could update the dial. If you never expect to handle this message, you can disable it by using `hn_forever` as the timeout value when you call `hn_listen()`.

Connected

> When the client executes `hn_connect()`, one side effect is that a *Connect* message is sent from the client to the stack. Once the stack has received this message, and a successful connection has been made, the stack will reply with the *Connected* message.

Disconnected

> If a stack receives a *Connect* message from a client that it is already connected to, it will send a *Disconnected* message to the client before it sends a *Connected* message.

2.3.8 Handler Routines

The first argument of the `hn_register()` call is the handler routine that will get control when a particular message is received by the client. The call may include arguments that have been passed from HyperNeWS. The first argument passed to a handler is the object sent from HyperNeWS. If this object is a message, the second and third arguments of the handler are the number of message arguments (`int`), and a vector of arguments (`hn_any *[]`).

All handler routines should return an integer argument when they finish. If it returns 0, other handlers will be considered. If it returns 1, no other handler can be invoked. If a message matches the requirements for multiple handlers and the handlers return 0 so that others can be considered, they will be invoked in the order that they are registered using `hn_register()`.

The following is an example that shows how to extract arguments from a message once the handler has been invoked:

```
int myhandle2(any,argc,argv)
hn_any  *any,*argv[];
int     argc;
{
        int i;
        char *buff;

        i = (int) argv[0]->u.number;
        buff = argv[1]->u.string;
        return(1);
}
```

In this example, the `myhandle2()` handler expects to get 2 arguments: an integer and a string. The message itself is passed in `any`, but the message arguments are stored in `argv[]`. The argument needs to be coerced to an integer, since looking at the `hn_any` structure shows that `u.number` is actually of type `double`.

2.4 C Interface Routines

Following are the definitions for the HyperNeWS C interface routines, listed alphabetically. The return value for routines that have an integer result is 1 on normal completion and 0 in case of error, unless otherwise specified.

hn_connect int hn_connect (hn_any *stack)

Connect to stack `stack`. All future unhandled events from this stack will subsequently be sent to this client, as well as those originating from `ClientSend` commands in HyperNeWS.

hn_error void **hn_error** (char *str)

Print the string `str` on the standard error channel.

hn_exists int **hn_exists** (hn_any *object)

Returns 1 if the object exists, 0 otherwise.

hn_free void **hn_free** (hn_any *object)

Unallocate storage of previously created HyperNeWS objects. This routine is recursive, so passing `hn_free()` an array of `hn_any` objects will free all of them.

hn_get hn_any * **hn_get** (hn_any *object, char *instvar)

Get an instance variable of an object. This procedure uses the synchronization mechanism described in Section 2.3.6. If a slider has been created within the C interface then the following call could be used to get the (integer) value of the object:

```
hn_any *slide,*val;
val = hn_get(slide,"Value");
```

ERRORS

`NULL` is returned if the object is invalid, or there is some problem with the HyperNeWS link.

hn_hide int **hn_hide** (hn_any *object)

Hide the object in HyperNeWS.

hn_listen int **hn_listen** (int delay)

Listen for input from HyperNeWS. When input is received, a list of registered handlers is checked, and the appropriate handler(s) called. If the delay (in milliseconds) is exceeded without a message being received, an `hn_timeout` message is sent to the client. By default this is the `hn_any` object representing the PostScript name `hn_forever`.

ERRORS

If there is no message queue on startup, 0 is returned. If there is an I/O error, `EOF` is returned.

hn_new_array hn_any * **hn_new_array** (hn_any *obj1, ...,*objn, NULL))

Create an array with instantiated arguments. The list of arguments is terminated with the null object.

hn_new_array_body hn_any * **hn_new_array_body** (int length)

Create a new array body with the specified number of arguments. The elements of the array are initialized to `NULL` .

hn_new_name hn_any * **hn_new_name** (char *string)

Return a HyperNeWS object that represents a PostScript name.

hn_new_number hn_any * **hn_new_number** (double n)

Return a HyperNeWS object that represents a number.

hn_new_message hn_any * **hn_new_message** (hn_any *target, char *name, hn_any *args)

Return a HyperNeWS object that represents a message.

hn_new_object hn_any * **hn_new_object** (char *stack, hn_type type, char *name)

Create a new HyperNeWS object on the stack with the given type and name. This object is is created only on the C side of the interface; it will not implicitly create the object on the stack.

hn_new_string hn_any * **hn_new_string** (char *string)

Create a HyperNeWS object that represents a string.

hn_put int **hn_put** (hn_any *object, char *instvar, hn_any *value)

Set the instance variable of an object. If a slider has been created within the C interface then its value could be set to `10` by the following calls:

```
hn_any *ivalue,*slide;
ivalue = hn_new_number((double)10);
hn_put(slide,"Value",ivalue);
```

ERRORS

`NULL` is returned if the object is invalid, or if there is some problem with the HyperNeWS link. Note that no type checking is done on the value against the instance variable; it is possible to cause errors within HyperNeWS if, for example, an integer instance variable is given a string value.

hn_register void **hn_register** (int (*)() handler, hn_any *match)

Register a handling routine to be called when a message has been matched. The procedure will be called by `hn_listen()` when the input from HyperNeWS matches the `match` argument. Any part of the `match` data structure can be set to `ANY` which will match anything. Other handlers will be considered.

hn_rename hn_any * **hn_rename** (hn_any *object, char *name)

Rename the object. Stacks cannot be renamed. The object is returned. This is necessary, because previous references to this object will now be invalid. This procedure uses the synchronization mechanism of Section 2.3.6.

hn_send int **hn_send** (hn_any *target, char *message, hn_any *args)

Sends the message, with the given arguments to the object.

hn_set_drawing int **hn_set_drawing** (hn_any *object, char *file)

Set the drawing of button `object` to be the HyperDraw drawing contained in file `file`.

ERRORS

No checking is done to determine whether the file exists or has a drawing in it.

hn_show int **hn_show** (hn_any *object)

Show the object in HyperNeWS.

hn_show_error int **hn_show_error** (hn_any *text)

Show an error message in the HyperNeWS ErrorMessage stack. The `text` argument should be an array of HyperNeWS strings.

hn_stack_copy hn_any * **hn_stack_copy** (hn_any *stack)

Make a copy of stack `stack` . The new stack object is returned.

hn_start int **hn_start** (char *programname)

Connect the client to HyperNeWS. The `programname` variable is used within HyperNeWS to provide a name for the client. Initialization is performed and a connection is established with HyperNeWS.

hn_stop int **hn_stop** ()

Disconnect the client from HyperNeWS. The value 1 is always returned.

2.5 A Simple Example

2.5.1 Basics

We show a simple example of how to establish communication with a HyperNeWS stack. The stack is named `Stack1`. It contains a menu and a slider, and there is very simple one-way communication between them, via the client. When you choose a menu item, that value will be passed to the client, and then the client will tell the slider to set itself to half that value. Of course this application is simple enough that you could send messages directly from the PullDown object to the Slider object without a client, but it makes for a reasonable example.

The C code, the stacks, and the Makefile for the two examples shown here are located in `$HNHOME/share/examples`.

2.5.2 The Stack

This example will consist of two parts, the first part being the stack. To create the stack, follow these steps:

1. Create a new stack template using the *Create New Stack* system button.
2. Create a slider and edit its info stack. Change its name to *TheSlider* and unset the *Scrollbar* checkbox so that it will show its value. Set the minimum and maximum values to be something reasonable like 0 and 100.
3. Create a menu and edit its info stack. Change the name to *TheMenu*, change the values to something like 0, 80, 120, and 210 and be sure to press the *Set Menu* button. Then press the *Script* button and enter this code:

   ```
   /Action {
          cvi [exch] /DoWhatsRight ClientSend
   } def
   ```

   What this method will do is convert the menu item you selected from a string into an integer and pass it as an argument of the message `DoWhatsRight` which is sent to the client.
4. Finally, save the stack as Stack1.

2.5.3 The C Client Code

This is the second part of the combined stack/client application. Look at the routine `client1.c` located in `$HNHOME/share/examples`:


```
#include <stdio.h>
#include <hypernews.h>

hn_any *mystack,*mymenu,*myslider;

int myhandler(any,argc,argv)
hn_any *any,*argv[];
int argc;
{
int myint;
hn_any *mynum,*myargs;

myint = argv[0]->u.number;
mynum = hn_new_number((double) myint/2);
myargs = hn_new_array(mynum,NULL);
hn_send(myslider,"SetValue",myargs);
hn_free(myargs);
return 1;
}

void quit()
{
hn_stop();
}

main(argc,argv)
int argc;
char *argv[];
{
if (!hn_start(argv[0])) {
fprintf(stderr,"client1: hn_start() failed.\n");
exit(1);
}

/* Define the interface objects */
mystack = hn_new_object((char *) NULL,hn_Stack,"Stack1");
mymenu = hn_new_object("Stack1",hn_PullDown,"MyMenu");
myslider = hn_new_object("Stack1",hn_Slider,"MySlider");

/* Register the messages */
hn_register(myhandler,
hn_new_message(mymenu,"DoWhatsRight",hn_new_array_body(1)));
hn_register(quit,hn_quit);

/* Connect to the stack */
if (!hn_connect(mystack)) {
fprintf(stderr,"client1: hn_connect() failed.\n");
exit(1);
}

/* Wait for messages */
```

```
hn_listen(hn_forever);

exit(0);
}
```

First we'll talk about the main routine, and then we'll look at the handlers. The first thing that is done in `main()` is to call `hn_start()`, passing it the name of the program. Then the stack objects that will be referenced from the client are defined using `hn_new_object()`. Next, the handler `myhandler()` is registered so that it will be called whenever the `DoWhatsRight` message is received, and we should expect an argument to be passed with it. When HyperNeWS exits, the client will be sent the `hn_quit` message, which doesn't have to be created using `hn_new_message()` because it is a message that HyperNeWS already knows about. Next we connect to the stack that we have defined above, and we enter a listening loop, waiting for messages from the stack.

Regarding the handling routines: When `myhandler()` gets control, it expects to find one argument in the argument list. It then creates a new HyperNeWS object which is the passed integer divided by two. The new object is then placed in an array which is the argument list. Argument passing in HyperNeWS C clients is similar to C in argument passing, in that all arguments are passed in a null-terminated array. Next, `hn_send()` is used to send the `SetValue` message to the Slider object in the stack. The `hn_free()` call is not required, but it is good practice to free up large chunks of storage after you are done with them. Note that `hn_free()` is recursive, so passing `myargs` will also free `mynum`. Finally, we return a 1, saying that no other handlers should be considered with this particular message.

The `quit()` handler is called upon receipt of the `hn_quit` message. All it does is call `hn_stop()`, which will exit the listening loop and close the HyperNeWS connection. In the next example you'll see that you can simply specify `hn_stop()` when registering the `hn_quit` message.

2.5.4 Executing the Stack

Once the program is compiled, it can be connected to the Stack1 stack by opening the stack from within HyperNeWS and then running the `client1` program from a terminal window. The slider will then respond every time a menu item is selected. Remember that you can also see all the messages that are being passed by enabling logging in the SystemInformation stack.

Typically though, it is more convenient to start the C process automatically when the stack is first opened. This can be done by adding the following code to the stack's script:

```
/OnInit {
      HOME (/client1) append
```

```
        dup FileExists {
                runprogram
        } {
                pop [(Can't start client: ~/client1)] ShowError
        } ifelse
} def
```

This assumes that the binary is located in your home directory. It will check to see if the binary exists, and if so, it will run the client as a new Unix process.

2.6 Another Example

2.6.1 Basics

In this second example we'll show a little more complex client that actually uses the timeout capabilities and passes more than one argument. Again, this example is *completely* useless, but it does demonstrate how to do a few things that are useful. The stack will have a menu as it had before, but this time selecting an item will send 2 arguments to the stack: the number selected multiplied by 2, and the number itself. This will be sent to the client, and the client will send a move command to the stack with these 2 numbers as its X and Y coordinates. An added feature will move the stack slightly if no menu item is chosen within 5 seconds.

2.6.2 The Stack

We'll create this stack like so:

1. Create a new stack, as before.
2. Create a new PullDown object. Give it smaller numbers, such as 0, 20, 40, 60, and create a script like this:

   ```
   /Action {
           [exch cvi dup 2 mul exch] /MoveIt ParentSend
   } def
   ```

   so that you will send the desired X,Y coordinate of the lower-left corner of the stack, with the X value being twice the Y value.
3. Save this stack as Creeper.

Note that in this example we used `ParentSend` instead of `ClientSend` to send the message to the client. In this case the message will be passed to the card, background, stack, and then client, since only the client will be able to handle the `MoveIt` message.

2.6.3 The C Client Code

The C code that will communicate with the Creeper stack is shown below:

```
#include <stdio.h>
#include <hypernews.h>

hn_any *mystack;

int myhandler(any,argc,argv)
hn_any *any,*argv[];
int argc;
{
int xint,yint;
hn_any *xany,*yany,*myargs;

xint = argv[0]->u.number;
yint = argv[1]->u.number;
xany = hn_new_number((double) xint);
yany = hn_new_number((double) yint);
myargs = hn_new_array(xany,yany,NULL);
hn_send(mystack,"Move",myargs);
hn_free(myargs);
return 1;
}

int creep()
{
hn_any *myargs,*xany,*yany;
int xint,yint;

xany = hn_get(mystack,"ScreenX");
yany = hn_get(mystack,"ScreenY");
xint = xany->u.number + 40;
yint = yany->u.number + 40;
myargs = hn_new_array(hn_new_number((double) xint),
hn_new_number((double) yint),NULL);
hn_send(mystack,"Move",myargs);
hn_free(myargs);
return 1;
}

int debug(any,argc,argv)
hn_any *any,*argv[];
int argc;
{
fprintf(stderr,"Message received by client: /%s\n",
any->u.message.name);
return 0;
}
```



```
main(argc,argv)
int argc;
char **argv;
{
if (!hn_start(argv[0])) {
fprintf(stderr,"creeper: hn_start() failed.\n");
exit(1);
}

/* Define the interface objects */
mystack = hn_new_object((char *) NULL,hn_Stack,"Creeper");

/* Register the messages */
hn_register(debug,hn_new_message(ANY,ANY,ANY));
hn_register(myhandler,
hn_new_message(ANY,"MoveIt",hn_new_array_body(2)));
hn_register(creep,hn_timeout);
hn_register(hn_stop,hn_quit);

/* Connect to the stack */
hn_connect(mystack);

/* Wait for messages */
hn_listen(5000);

exit(0);
}
```

Once again, we'll walk through the code. In `main()`, we only identify the stack itself, and have the stack manage communication with any objects residing on the stack. Both this style and the style used in the first example are valid. It is a matter of personal taste which one you prefer. The only other differences in the main routine are some of the handlers that are registered. The `debug()` routine is useful to see what messages are sent to the stack. It will accept any message, and since it returns a 0, HyperNeWS will see if it matches any other handling routines. The `myhandler()` routine gets invoked when `MoveIt` is sent to the client, and 2 arguments are passed. The operation of this handler is similar to the handler in the prior example. The `creep()`handler is called if no messages are sent to the stack within 5000 milliseconds (or 5 seconds). It uses synchronous communication to get the current origin of the stack, stored in the stack instance variables `ScreenX` and `ScreenY`. The coordinates are incremented by 40, and then the `Move` message is sent to the stack, which will make it slowly creep to the upper-right corner of the screen. Finally, the `hn_stop()`routine that will terminate the listening loop is used as a direct argument to `hn_register()`, so that once the `hn_quit` message is received by the client, the HyperNeWS connection will be broken and control in the client will continue on the statement following the `hn_listen()` call.

3 The Prolog Interface

3.1 Basics

This section describes the interface between Quintus Prolog and HyperNeWS. Predicates exist to send PostScript code to HyperNeWS and interpret messages received. An event loop and message handler are provided to allow the programmer a template for interpreting messages. A simple example program is given in Section 3.12. This interface is based on the C interface described in Section 3.5.

3.2 Getting Prolog Started

The necessary interface routines are loaded into Prolog as follows:

```
:- ensure_loaded('<$HNHOME>/intf/prolog/interface.pl').
```

where `$HNHOME` is the environment variable for the HyperNeWS home directory on your system. It should be expanded to an absolute path name for use with the `ensure_loaded` predicate. These routines already exist in a compiled form as a saved Prolog state, which can be started with the command:

```
hnprolog
```

The following routines are used to start, stop and run the HyperNeWS interface:

hn_start

This command starts the HyperNeWS interface. If HyperNeWS is already running, Prolog connects to it. Otherwise it will start HyperNeWS. Do not attempt to make a saved state of your Prolog program after having called this routine. This routine gives the client program the default name `prolog`.

ERRORS

Fails if the communication cannot be made successfully with HyperNeWS.

SEE ALSO

hn_start/1

hn_start(+Name)

Performs the same function as hn_start/0 except that the name of the client program is supplied as the atom *Name*.

ERRORS

Fails if the communication cannot be made successfully with HyperNeWS.

SEE ALSO

hn_start/0

hn_stop

This routine disconnects Prolog from HyperNeWS. However, it does not kill HyperNeWS. It is safe to make a saved state of your Prolog program after calling this routine.

hn_run

This routine receives messages from HyperNeWS within an event loop. It calls the routine hn_read/1 to wait for user input then hn_message/3 to process the message received.

SEE ALSO

hn_run/1

hn_run(+TimeOut)

This routine receives messages from HyperNeWS within an event loop. It calls the routine hn_read/2 supplying an integer timeout value (in micro seconds). If no user input has occurred within the specified period, the atom *timeout* is returned. Otherwise a message from a HyperNeWS object will be present. It then calls hn_message/3 to process the message received.

SEE ALSO

hn_run/0

hn_message(+Object, +Message, +Args)

You should supply clauses for this routine if you use the standard control loop hn_run/0 . It is called by the control loop after data has been read from HyperNeWS. The arguments (see Section 3.4) are returned from HyperNeWS through hn_read/1 . Your clauses should allow messages to be handled by matching on object, message name, or argument values.

SEE ALSO

hn_run/0 hn_run/1

After establishing low-level communication using hn_start/0 or hn_start/1 , the program must be connected to a HyperNeWS stack before any communication can occur. This can be performed explicitly through the hn_connect/1 predicate, or implicitly when a stack is opened using the hn_show/1 predicate. This allows communication to occur regardless of whether a stack is visible or not.

3.3 Data Type Conversion

To allow effective communication between a Prolog client and the PostScript HyperNeWS program, data must be converted into a suitable representation on each side. Table 1shows how data types are equated between the two processes.

Note that the functor name <objtype> can take one of the following atomic forms: background, card, button, edittext, slider, pulldown, colorselect or canvas.

HyperNeWS type	Prolog equivalent	
booleantype	atom	true/false
nulltype	atom	null
integertype	integer	value
floattype	float	value
nametype	atom	name
stringtype	term	string('Text')
arraytype	list	[...]
message	term	message(Object,Message,Args)
Object (dicttype)	term	stack('StackName')
Object (dicttype)	term	'StackName':<objtype>('ObjName')
Message (nametype)	atom	messagename
Args (arraytype)	list	[args]

Table 1: Prolog data typeconversion.

3.4 Responding to Messages

3.4.1 HyperNeWS Data Format

The predicates hn_run/0 and hn_run/1 are two examples of control loops both of which call hn_read/1 to wait indefinitely for an event, before returning the data from HyperNeWS. Data is returned in the form of a prolog term (message/3) with three arguments as shown:

```
message(Object,Message,Args)
```

The user defined predicate hn_message/3 is then called (if it exists) to process the arguments of the term. If you use one of these control loops, you must supply clauses for hn_message/3 that allow matching of the HyperNeWS messages with appropriate actions. The format of the three arguments is given below:

Object

Can take two different forms: if the message has come directly from a stack, then the object will be `stack(StackName)` where *StackName* is the name of the HyperNeWS stack. Otherwise the object has a two part name: `StackName:hntype(Object)` where *Object* is the name of a HyperNeWS object, and *hntype* may be one of the following atoms: *background*, *card*, *edittext*, *button*, *slider*, *pulldown*, *colorselect* , or *canvas*.

Message

The name of the message, which must be an atom.

Args

Message arguments are returned as elements in a Prolog list. See Section 3.3 for details of how HyperNeWS data objects are represented in Prolog.

3.4.2 Examples

This section illustrates some examples of HyperNeWS messages, and what the corresponding Prolog format of the message would look like:

1. The stack `Ex` issues the following message directly to the client:

   ```
   [(hi there client) 2 3] /Foo ClientSend
   ```

   The resulting Prolog data would take the form:

   ```
   message(stack('Ex'),'Foo',[string('hi there client'),2,3])
   ```


2. The push button `push1` within the stack `New` is clicked, causing an `Action` message to be routed through the HyperNeWS hierarchy and on to the client. Prolog would see this data in the form:

   ```
   message('New':button(push1),'Action',[0])
   ```

3. From the same stack, double-clicking on the word "Danny" in the EditText object `p25` sends a `WordAction` message. This would be seen from Prolog as:

   ```
   message('New':edittext(p25),'WordAction',[string('Danny')])
   ```

3.5 Control Routines

The following routines are used for general control:

hn_flush_input

Flush pending input from HyperNeWS.

ERRORS

Fails if the connection between Prolog and HyperNeWS is broken.

SEE ALSO

hn_flush/0

hn_flush

Flush pending output to HyperNeWS.

ERRORS

Fails if the connection between Prolog and HyperNeWS is broken.

SEE ALSO

hn_flush_input/0

hn_ok

Checks the status of the connection between Prolog and HyperNeWS. It returns `true` if HyperNeWS is still running and the connection is good.

hn_verbose(+Flag)

Turns verbose message information on or off. Call with the argument `on` to turn on, or `off` to turn off. Verbose is initially turned off.

hn_clear

The Prolog routines use a stack structure to communicate their arguments and results to and from HyperNeWS. This predicate clears the communication stack.

3.6 High-Level I/O

The following routines define the high-level I/O facilities available from Prolog.

hn_read(-Data)

Reads data from HyperNeWS. If no data is present, the routine blocks until input is available to be read. Data is returned as the Prolog term `message(Object,Message,Args)`. `Object`, `Message` and `Args` are defined in Section 3.4.

SEE ALSO

hn_read/2

hn_read(-Data,+Delay)

Reads data from HyperNeWS. If no data is present, the routine waits for `Delay` microseconds. If no input is received within this time period, the atom `timeout` is returned. Otherwise data is returned as the Prolog term `message(Object,Message,Args)`. `Object`, `Message` and `Args` are defined in Section 3.4.

ERRORS

Fails if the argument `Delay` is not an integer.

SEE ALSO

hn_read/1

hn_send(+Object,+Message,+Args)

Sends the message `Message` to the HyperNeWS object `Object` with the list of arguments given in `Args`. Details of the format of the arguments are given in Section 3.4.

ERRORS

Arguments must be correctly typed otherwise the predicate fails.

SEE ALSO

hn_read/1 hn_read/2

hn_ps(+PostScript)

Sends the given PostScript code to the NeWS server.

hn_write(+Data)

Determines the type of the data before writing it to HyperNeWS in a suitable format. See Section 3.3 for details of data type conversion between Prolog and HyperNeWS.

3.7 Editing Objects

The routines hn_get/3 and hn_put/3 allow object instance variables to be set and retrieved from HyperNeWS. See Section 9.2 for a complete list of available object variables organized by object class. The list details for each variable its name, type and default value. Note that using the class hierarchy, an object inherits the instance variables from its parent classes.

hn_get(+Object,+Var,-Value)

> Returns in `Value` the current value of the instance variable `Var` from the HyperNeWS object `Object`. See Section 9.2 for details of which variables are appropriate for each object class.
>
> ERRORS
>
> > Fails if the object does not exist, or the variable is not an instance variable within the object's class.
>
> SEE ALSO
>
> > hn_put/3

hn_put(+Object,+Var,+Value)

> Sets the instance variable `Var` within the HyperNeWS object `Object` to `Value`. The instance variable must be capable of being set. See Section 9.2 for details of which variables are appropriate for each object class.
>
> ERRORS
>
> > Fails if the object does not exist, if the variable is not an instance variable within the object's class, or if the value given is not of the correct type.
>
> SEE ALSO
>
> > hn_get/3

hn_set_text(+Stack:edittext(Object),+Text)

Sets the EditText object on the given stack to the value of the text by sending a `Setvalue` message. The `Text` argument should take one of the following forms:

1. An atom. Useful for setting the object to a single line of text.
2. A list of `string(TextLine)` terms. Each `TextLine` argument holds a single line of text as an atom.
3. The term `text(TextList)`, where `TextList` is a list of atoms, one per line of text.

ERRORS

Fails if the object does not exist, or if the text is not in the format described above.

SEE ALSO

hn_get_text/2

hn_get_text(+Stack:edittext(Object),-Text)

Returns the current value of the EditText object on the given stack. `Text` is returned as a list of `string(TextLine)` terms.

ERRORS

Fails if the object does not exist.

SEE ALSO

hn_set_text/2

3.8 General Operations on Objects

hn_show(+Object)

Show the given object. If the object is a stack, the client is first connected. No action is taken if the object is already visible.

ERRORS

Fails if the given object does not exist.

SEE ALSO

hn_hide/1 hn_connect/1

hn_hide(+Object)

Hide the given object. The connection is not broken if the object is a stack. No action is taken if the object is already hidden.

ERRORS

Fails if the object does not exist.

SEE ALSO

hn_show/1

hn_connect(+stack(Stack))

Connect the client program to the given stack.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_show/1

hn_exists(+Object)

Returns true if the given object exists.

hn_update(+Object)

Updates the object (the object will be redrawn).

ERRORS

Fails if the object does not exist.

hn_lock_update(+stack(Stack))

Lock all updates for the given stack. Updates will be performed when the stack is unlocked. No action is taken if the stack is already locked.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_unlock_update/1

hn_unlock_update(+stack(Stack))

Unlock all updates for the given stack. Any changes to the stack since it was locked will be performed. No action is taken if the stack is already unlocked.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_lock_update/1

hn_stack_copy(+stack(Stack),-stack(NewStack)

Create a new copy of the given stack, returning its unique name as the `NewStack` argument.

ERRORS

Fails if the stack does not exist.

hn_save(+stack(Stack))

Save the stack using its current name.

ERRORS

Fails if the stack does not exist.

hn_save_as(+stack(Stack),Name)

Give the stack a new name and save it.

ERRORS

Fails if the stack does not exist, or if the name is not atomic.

hn_rename(+Stack:Object,+Name,-Stack:NewObject)

Rename the object to the atom `NewStack`. Stacks cannot be renamed using this predicate. To rename a stack use the hn_save/1 or hn_save_as/2 routine.

ERRORS

Fails if the stack or object does not exist, or if the name is not atomic.

hn_new_object(+stack(Stack),+Name)

Add a new object to the stack taken from the NewObjects stack. The supplied name must be the name of an existing object on that stack. This routine calls hn_menu/3 to select the object from the *New* menu.

ERRORS

Fails if the stack does not exist, or if the name is not atomic.

hn_start_edit(+stack(Stack))

Puts the stack into edit mode.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_stop_edit/1

hn_stop_edit(+stack(Stack))

Puts the stack into browse mode.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_start_edit/1

hn_select(+Stack:CardObject)

Selects the given card object.

ERRORS

Fails if the object does not exist, or if the object is not a card object.

SEE ALSO

hn_start_edit/1

hn_show_error(+Text)

Shows the error stack with text taken from the argument.

hn_confirm(+Text,-Answer)

Opens the Confirm stack, displaying the text message. It waits for user input before returning the answer `yes` or `no`.

hn_set_drawing(+Stack:button(Name),+File)

Copies a drawing from the file into the button.

3.9 Card Routines

hn_home_card(+stack(Stack))

Go to the home card of the given stack.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_next_card/1 hn_previous_card/1 hn_last_card/1

hn_next_card(+stack(Stack))

Go to the next card of the given stack.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_home_card/1 hn_previous_card/1 hn_last_card/1

hn_previous_card(+stack(Stack))

Go to the previous card of the given stack.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_home_card/1 hn_next_card/1 hn_last_card/1 hn_last_card(+stack(Stack))

Go to the last card of the given stack.

ERRORS

Fails if the stack does not exist.

SEE ALSO

hn_home_card/1 hn_next_card/1 hn_previous_card/1

hn_goto_card(+Stack:card(CardName))

Go to the named card on the given stack.

ERRORS

Fails if the stack or card does not exist.

SEE ALSO

hn_goto_card/2

hn_goto_card(+Stack,+Card)

Go to the named card on the given stack. The argument `Card` may either be an integer card number, or atomic card name.

ERRORS

Fails if the stack or card does not exist.

SEE ALSO

hn_goto_card/1

3.10 Resize and Move Operations

hn_move(+Object,+[X,Y])

Move the object to coordinate location [X,Y]. Coordinates are given in screen pixels. If the object is a stack, the coordinates are relative to the lower-left corner of the screen. Card object coordinates are relative to the lower-left corner of the stack they are on. Card and background objects cannot be moved.

ERRORS

Fails if the object does not exist.

SEE ALSO

hn_size/2 hn_reshape/2

hn_size(+Object,+[W,H])

Resize the object to new width W and height H. Sizes are given in screen pixels.

ERRORS

Fails if the object does not exist.

SEE ALSO

hn_move/2 hn_reshape/2

hn_reshape(+Object,+[X,Y,W,H])

Resize the object to new width W and height H and also move the object to position [X,Y]. Dimensions are given in screen pixels, and coordinates are relative to the lower-left corner of the stack or screen, as with the move operation.

ERRORS

Fails if the object does not exist.

SEE ALSO

hn_move/2 hn_size/2

3.11 Menu-based Operations

These routines send messages to objects as if they had come from a HyperNeWS menu selection. They are useful as you do not need to know the name of an object before sending a message.

hn_menu(+stack(Stack),+Message,+Args)

Sends the message to the selected object or to the current card.

ERRORS

Fails if the stack does not exist.

hn_focus_menu(+stack(Stack),+Message,+Args)

Sends the message to the current focus.

ERRORS

Fails if the stack does not exist.

hn_stack_menu(+stack(Stack),+Message,+Args)

Sends the message to the stack.

ERRORS

Fails if the stack does not exist.

3.12 A Sample Program

This program starts communication with HyperNeWS, opens the ExampleObjects stack, then waits for a message, and displays the name of the object, message and arguments. Although simple, it follows a basic structure that is a useful template for most applications; that of recognizing messages from HyperNeWS and taking some action.

```
go :-
      hn_start,
      hn_show(stack('ExampleObjects')),
      hn_run.

hn_message(stack(Stack),Message,Args) :-
      write('Stack: '),
      write(Stack),nl,
      write('Message: '),
      write(Message),nl,
      write('Args: '),
      write(Args),nl,nl.

hn_message(Stack:Object,Message,Args) :-
      write('Stack: '),
      write(Stack),
      write(' Object: '),
      write(Object),nl,
      write('Message: '),
      write(Message),nl,
      write('Args: '),
      write(Args),nl,nl.

% catch anything else
hn_message(_,_,_).
```

Example output from running the program, clicking on various HyperNeWS objects, looks like this:

```
yes
| ?- go.
Stack: ExampleObjects
Message: Connected
Args: []

Stack: ExampleObjects Object: button(pushbutton)
Message: Action
Args: [0]

Stack: ExampleObjects Object: button(Checkbox)
Message: Action
Args: [1]
```



```
Stack: ExampleObjects Object: edittext(textbox)
Message: WordAction
Args: [string(scrollable)]

Stack: ExampleObjects Object: slider(#10)
Message: Action
Args: [36]

Stack: ExampleObjects Object: colorselect(#9)
Message: Action
Args: [[0.54,0.54,0.54,0.54]]

Stack: ExampleObjects Object: canvas(#11)
Message: OnEnter
Args: []
```

4 The Lisp Interface

4.1 Basics

HyperNeWS provides a number of object types, such as buttons, sliders, menus etc. These can be used to design an interface to a program written in C, Prolog or Common Lisp. The program is referred to as a "client" of the HyperNeWS server. All interface objects used by a client exist on a "stack" to which the client must "connect" before it can access the objects.

All objects communicate with each other and the client using messages. The client can read or set the value associated with a particular object at any time by sending an appropriate message to the object. The client can also be put into "listen" mode where it waits until a message arrives from one of the interface objects.

The client must specify the messages it is interested in. An event loop and message handler are provided to give the programmer a template for interpreting messages. Below, details of the Lisp interface to HyperNeWS are given.

There are four stages in using a HyperNeWS stack from Lisp:

1. Declaration of the interface objects.
2. Registration of functions, if any, to be called in response to messages arriving from the interface.
3. Starting HyperNews and connecting to a stack.
4. Sending and receiving messages.

4.2 Declaring an Object

The following types of objects are defined for use in an interface: Stack, Background, Card, Button, Slider, EditText, PullDown, ColorSelect, and Canvas. More than one of each object type may be used in a stack, provided they all have unique names. Furthermore, each object used must be declared in the client. This is done using the function `hnl_interface_object` described below. The value returned by this function serves as a reference to the object, and must be retained as the value of a local variable. So for example, if the program is to connect to a stack called `lisptest`, a local variable, say `mystack`, may be used as follows:

```
(setq mystack (hnl_interface_object NIL 'stack "lisptest"))
```


The first argument to `hnl_interface_object` is the name of the stack on which the object exists. This is not applicable to the parent stack itself, hence the `NIL` as first argument when declaring a stack. Each interface object on the stack is declared in a similar fashion. For example, to use a button named `button1` in the interface, this may be declared (using a local variable, say `mybutton`) as:

```
(setq mybutton (hnl_interface_object mystack 'button "button1"))
```

hnl_interface_object

hnl_interface_object (Stack-Object Type Name)

Declare a new interface object to be used by the client. The value returned must be used in all future references to the object, for example in all functions requiring an Object as an argument.

`Stack-Object` is of type `Object`. It should be `NIL` when declaring a stack. Otherwise it should be a value obtained from a previous call to `hnl_interface_object` defining the parent stack.

`Type` is of type `Symbol`, and is one of: `stack`, `background`, `card`, `button`, `slider`, `edittext`, `pulldown`, `colorselect`, or `canvas`.

`Name` is of type `String` and is the name of the object as specified on the HyperNeWS stack.

Syntax Examples:

```
(setq mystack (hnl_interface_object NIL
'stack "lisptest"))
(setq mybutton (hnl_interface_object mystack
'button "button1"))
```

4.3 Registering Message Handlers

It is possible to have the client program operate in a "listen" mode, essentially executing an infinite wait-for-message-then-process-message loop. If this is to be used, it is neccesary to specify the function to be called for each kind of message the client is interested in. Each such function is referred to as a *handler*, and is registered using the function `hnl_set_handler`. For instance, given the button `mybutton` declared above, we might want the function `button-handler` to be called each time the button is pressed.

The function `hnl_set_handler` would be used to register this as follows:

```
(hnl_set_handler mybutton "Action" 'button-handler)
```

This specifies that the function `button-handler` is to be called each time a message named `Action` is received from the object declared as `mybutton`. The entire message is passed on to the function as its argument. The function `button-handler` would be defined elsewhere as a normal Lisp function with one argument, for example, as:

```
(defun button-handler (msg &aux sender msg-name value)
   (setq sender (first msg)
         msg-name (second msg)
         value (third msg))
   (format t "~%Button-handler: Button ~A pressed, value = ~S~%"
   (hnl_object_name sender) value))
```

After all handlers are set, it is necessary to call `hnl_run` to set the client into listen mode. The Lisp interface calls the specified function each time a corresponding message type arrives. Messages which do not match any registered handler are ignored.

hnl_set_handler — hnl_set_handler (Object Message-Name Function)

Register an interest of the client, and specify the name of the function to be called when a message matching this arrives. Registration of all handlers should be preceeded by a call to `hnl_clear_interests`. The `Object` variable is of type `Object` and is the sender of the message; a value obtained from a call to `hnl_interface_object`. `Message-Name` is of type `String` and is the name of the message received. The `Function` variable is the name of the function to be called when the message arrives, and is of type `Symbol`.

hnl_clear_interests

hnl_clear_interests ()

All registered interests (i.e., message templates) and associated templates are stored in the variable `*hnl_interests*`. This function sets the variable to `NIL`.

4.4 Starting Hypernews and Connecting to a Stack

The client program must ensure that the HyperNeWS server is running, and inform it that a new client is starting. This is done using the function `hnl_start`.

The client program must establish a connection with the appropriate stack, i.e. the one containing all the required interface objects. This is done by a call to `hnl_connect`. Finally, to ensure that the stack is displayed, the client may execute `hnl_show`.

hnl_start

hnl_start (&optional Client-Name)

The `Client-Name` is a string, which defaults to “lisp”.

hnl_connect

hnl_connect (Stack)

The `Stack` is a string or object to which we want to connect. If the stack has already been declared as an object this object can be given instead of the name of the string, which defaults to “lisp”.

hnl_show

hnl_show (Object)

The `Object` is an object returned by `hnl_interface_object`. It will be shown.

4.5 Sending and Receiving Messages

Functions for sending and receiving general messages are:

```
hnl_read
hnl_send
```

Since setting and getting values of objects is the most common operation, two general functions to do this are available. These are:

```
hnl_get_value
hnl_set_value
```

The following functions are provided for reading and setting parameter (or instance variable) values of HyperNeWS objects. See Section X.4 for a complete list of available object variables organized by object class.

```
hnl_get
hnl_put
```

The use of `hnl_get` and `hnl_put` should, however, be avoided.

A call to the function `hnl_run` puts the client in an infinite loop, waiting for messages from the interface. This is essentially making repeated calls to hnl_read. Additionally, as messages arrive, appropriate functions are called to handle them. A call to abort from one of the handlers terminates the client program. Finally, the function `hnl_ps` can be used to send PostScript code to be executed by the NeWS server.

hnl_read hnl_read (&optional Timeout)

Read data from HyperNeWS. If no data is present, wait for `Timeout` milliseconds. If no input is received within this time period, the string "timeout" is returned. `Timeout` is an integer variable. If omitted, it is set to infinity.

hnl_send

hnl_send (Object Message Args)

Send a message to an object with a list of arguments. `Object` is of type Object, as obtained from `hnl_interface_object`. `Message` is a string, and the name of the message to be sent. `Args` is a list of arguments.

Syntax Examples:

```
(hnl_send myslider "SetValue" '(23))
(hnl_send myslider "HalveValue" NIL)
```

hnl_get_value

hnl_get_value (Object)

Returns the value of the specified object. For sliders, buttons and check boxes this is a single integer value. For EditText objects, the returned value is a list of strings, successive strings giving the text on each line of the object. For PullDown objects, the returned value is a list of all the items (as strings) in the menu. For ColorSelect objects, the value returned is a list of values, representing the red, green, blue and gray values of the color respectively. `Object` is of type Object, and is a value obtained from `hnl_interface_object`.

Syntax Examples:

```
(setq slider-value (hnl_get_value myslider))
(setq first-line (car (hnl_get_value mytext)))
(setq button-state (hnl_get_value mybutton))
```

hnl_set_value

hnl_set_value (Object Value)

Set a new value for a particular object. `Object` is of type Object, and is a value obtained from `hnl_interface_object`.

Syntax Examples:

```
(hnl_set_value myslider 34)
(hnl_set_value mytext '("line1" "line2" "line3"))
(hnl_set_value mytext "only on one line")
(hnl_set_value mybutton 1)
```

hnl_get hnl_get (Object Var)

Return the current value of the instance variable `Var`. `Object` is of type Object, as obtained from `hnl_interface_object`. `Var` is a string, and the name of the instance variable.

hnl_put hnl_put (Object Var Value)

Set the instance variable `Var` within the HyperNeWS object Object to `Value`. The instance variable must be capable of being set.

Syntax Examples:

```
(hnl_put myslider "Value" 23)
(hnl_put mytext "Value" '("line1" "line2"))
```

hnl_run hnl_run (&optional Timeout)

Enter an event loop. This should only be called if message handlers have been registered in the client. Receive messages within the event loop and call an appropriate handler, if registered. If no timeout is specified, the system waits until user input occurs at the interface. If specified, the string "timeout" is returned if no user input occurs within the specified period. The `Timeout` variable is the waiting period in milliseconds.

hnl_ps hnl_ps (string)

Send a string of PostScript code to HyperNeWS.

4.6 Object Query Operations

hnl_object_type hnl_object_type (Object)

Returns the type of an object as a string. The value returned will be one of "stack" "background" "card", "button", "slider", "pulldown", "edittext", etc.

hnl_object_name hnl_object_name (Object)

Return the name of an object (as defined on the HyperNeWS stack), as a string.

hnl_object_stack Object hnl_object_stack (Object)

Return the stack object which is the parent of the object. To find the name of the stack, one must use `hnl_object_name`.

hnl_object_parts List hnl_object_parts (Object)

Returns three values: the stack name, type, and object name of the argument. All three values returned are strings.

hnl_message List hnl_message (Message)

Returns three values: the `Sending-Object` of the message (i.e., the object which sent the message), the `Name` of the message, and the `Arguments` accompanying the message.

4.7 General Control Routines

hnl_flush_input hnl_flush_input ()

Flush pending input from HyperNeWS.

hnl_flush hnl_flush ()

Flush pending input to HyperNeWS.

hnl_ok hnl_ok ()

Checks the status of the connection between Lisp and HyperNeWS. It returns T if HyperNeWS is still running and the connection is good.

4.8 General Operations on Objects

hnl_hide hnl_hide (Object)

Hide the given object. No action is taken if the object is already hidden.

hnl_exists hnl_exists (Object)

Returns T if the given object exists.

hnl_update hnl_update (Object)

Update the object — the object will be redrawn.

hnl_lock_update hnl_lock_update (StackObject)

Lock all updates for the given stack. Updates will be performed when the stack is unlocked. No action is taken if the stack is already locked.

hnl_unlock_update hnl_unlock_update (StackObject)

Unlock all updates for the given stack. Any changes to the stack since it was locked will be performed. No action is taken if the stack is already unlocked.

hnl_stack_copy Object hnl_stack_copy (OldStack)

Create a new copy of the given stack, returning its unique name. The value returned is a reference to the new stack. This value must be used in all future references to the object.

hnl_save hnl_save (Stack)

Save the stack using its current name.

hnl_save_as hnl_save_as (Stack String)

Save the stack using the string provided as its name.

hnl_rename hnl_rename (Object String)

Rename the object to `String`. Stacks cannot be renamed using this function. To rename a stack use `hnl_save` or `hnl_save_as`. The value returned by `hnl_rename` should be used in all future references to the stack.

hnl_new_object hnl_new_object (Stack String)

Add a new object to the stack. The supplied name must not be the name of an existing object on that stack. This function calls `hnl_menu` to select the object from a menu.

hnl_start_edit hnl_start_edit (Stack)

Put the stack into edit mode.

hnl_stop_edit hnl_stop_edit (Stack)

Put the stack into browse mode.

hnl_select hnl_select (Object)

Select the given card object.

hnl_show_error hnl_show_error (ListofStrings)

Shows the error stack with an EditText object containing the list `ListofStrings`. The message should be a list of strings to be displayed on successive lines of the error stack. If only one line is to be displayed, argument may be a simple string of text.

hnl_confirm hnl_confirm (ListofStrings)

Open the Confirm stack, displaying the text message. It waits for user input before returning the value T if the user answers "yes" and NIL if the user answers "no".

hnl_set_drawing hnl_set_drawing (Object FileName)

Copy a drawing from the file into the button. The button now takes the shape of the drawing, created using the drawing tool, and saved in the file.

4.9 Card Routines

hnl_home_card hnl_home_card (StackObject)

Go to the home card of the given stack.

hnl_next_card hnl_next_card (StackObject)

Go to the next card of the given stack.

hnl_previous_card hnl_previous_card (StackObject)

Go to the previous card of the given stack.

hnl_last_card hnl_last_card (StackObject)

Go to the last card of the given stack.

hnl_goto_card hnl_goto_card (Object &optional Id)

Go to the named card of the given stack. `Object` is either a card, in which case the second argument is ignored, or the stack on which the desired card lives. In the latter case, the second argument is used to identify the card. This is either the name of the card, or its index on the stack.

4.10 Resize and Move Operations

hnl_move hnl_move (Object XY-list)

Move the object to the coordinate location specified. Coordinates are given in screen pixels. If the object is a stack, the coordinates are relative to the lower-left corner of the screen. Card object coordinates are relative to the lower-left corner of the stack they reside on. Card and background objects cannot be moved.

hnl_size hnl_size (Object XY-list)

Resize the object to the new width and height specified (in pixels).

hnl_reshape hnl_reshape (Object XYWH-list)

Move an object to a new location and resize the object to the new width and height specified (in pixels).

4.11 A Simple Example

The following piece of code is a simple program illustrating the use of the various kinds of objects. A number of message handlers are specified. These are called when matching messages arrive.

```
(defun run-test (&aux mystack mymenu mytext)
  (hnl_clear_interests)
  (setq mystack (hnl_interface_object nil 'stack "lisptest"))
  (setq mymenu (hnl_interface_object mystack 'pulldown "m1"))
  (setq mytext (hnl_interface_object mystack 'edittext "t1"))
  (setq myslider (hnl_interface_object mystack 'slider "s1"))
  (setq mybutton (hnl_interface_object mystack 'button "b1"))
  (setq mystop (hnl_interface_object mystack 'button "StopButton"))
  (setq mycolor (hnl_interface_object mystack 'colorselect "cs1"))
  (setq mycheck (hnl_interface_object mystack 'button "cb1"))


  (hnl_set_handler mymenu "Action" 'my-menu-handler)
  (hnl_set_handler mytext "WordAction" 'my-text-handler)
  (hnl_set_handler myslider "Action" 'my-slider-handler)
  (hnl_set_handler mybutton "Action" 'my-button-handler)
  (hnl_set_handler mycolor "Action" 'my-color-handler)
  (hnl_set_handler mycheck "Action" 'my-check-handler)
  (hnl_set_handler mystop NIL 'my-abort)
  (hnl_set_handler nil nil 'print-message)

  (hnl_start)
  (hnl_connect mystack)
  (hnl_show mystack)
  (hnl_run)
  )

(defun my-menu-handler (selection)
  (format t "~%Menu-handler: received selection ~S~%"
  (third selection)))

(defun my-text-handler (selection)
  (format t "~%Text-handler: Double Click on ~S~%"
  (third selection)))

(defun my-slider-handler (value)
  (format t "~%Slider-handler: Got new value - ~S~%"
  (third value)))

(defun my-button-handler (value)
  (format t "~%Button-handler: Button pressed, value = ~S~%"
  (third value)))

(defun my-color-handler (value)
  (format t "~%Color-handler: New Color = ~S~%"
```



```
  (third value)))

(defun my-check-handler (value)
  (format t "~%Check-box-handler: Box is now ~A~%"
  (if (= (third value) 0) "NOT CHECKED!" "CHECKED!")))

(defun my-abort (x)
  (abort))

(defun print-message (msg &aux stack type name sender mess-name args)
  (multiple-value-setq (sender mess-name args)
                       (hnl_message msg))
  (multiple-value-setq (stack type name)
       (hnl_object_parts sender))
  (format t "~%Received Message: ~A:~A:~A ~S ~S~%"
  stack type name mess-name args))
```

5 Utilities

5.1 drawps

The `drawps` utility converts HyperNeWS drawings into PostScript format. The input is read from `stdin` if a file is not specified, and the output is sent to `stdout`. The syntax of the command is:

```
drawps [-latex] [-news] [-rgb] [-pict] [-scale factor] [file]
```

where the options are:

-latex

Generate PostScript that can be used as the argument to a LaTeX `special` command.

-news

Generate PostScript that is compatible with NeWS.

-rgb

This option will take more care in converting color drawings into shades of gray. Useful when sending drawings to a laser printer.

-pict

Do not include images in the PostScript output. This option can be used to avoid the creation of large PostScript files for previewing.

-scale factor

Scale the drawing by `factor`.

5.2 dvi2ps

The `dvi2ps` utility converts `dvi` files into PostScript format. The input is read from `filename` and the output is sent to `stdout`. The syntax of the command is:

```
dvi2ps [-f n] [-t n] filename
```

where the most common options are:

-f n

Generate PostScript beginning at page `n`.

-t n

Generate PostScript up to and including page `n`.

5.3 plot2

The `plot2` program generates a HyperNeWS drawing representing a graph from a set of data points. The format of the input file is fairly straightforward, and is explained with the help of examples which can be found in the directory `$HNHOME/share/examples/plot2data`. The graph in Figure 1 was generated from the sample file `exam.3.data` located in the directory mentioned above. Your graph can be one of the types: line, thinline, bars, dots, smalldots, cross, or blocks. You can also specify the title width, height, labels, etc. of the graph. The output of `plot2` is a HyperNeWS drawing that you can load (and edit) in HyperDraw.

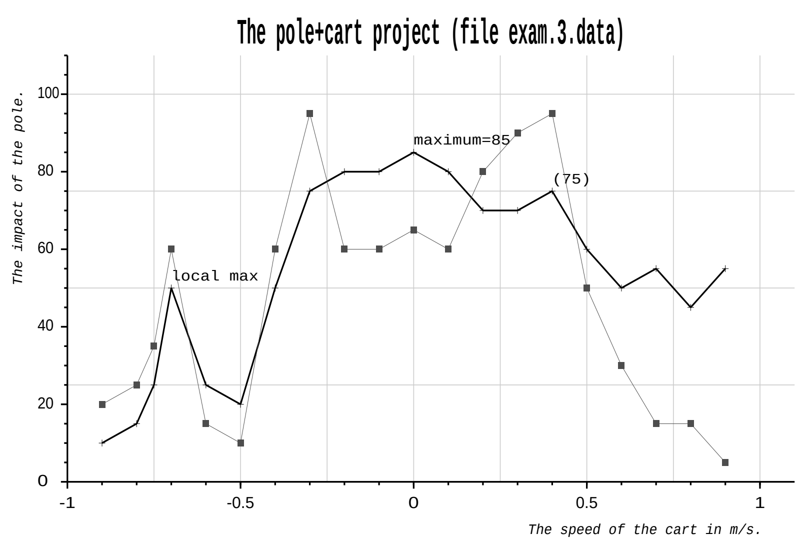

Figure 1: Plot2 output from data file `exam.3.data`.

6 Stacks with Application Interfaces

6.1 Basics

A few of the HyperNeWS system stacks are capable of sending and receiving messages from stacks you create. Their messages and capabilities are described in this section.

6.2 The Clipboard Stack

The Clipboard stack is a multi-card stack that is introduced in the User Guide. Besides manipulating the stack using the menu or its shortcuts, it is also possible to use the clipboard from within your stacks. Navigation between cards is done with the usual card navigation messages such as `/GoNextCard`.

To put information into the current card of the Clipboard stack, use the *Put* command. Its syntax is:

```
[obj] /PutFoo /Clipboard Send
```

where *obj* is the object to be cut/copied to the clipboard, and *Foo* is one of the following object types:

Drawing

The argument is a HyperNeWS drawing.

HNObject

The argument is a HyperNeWS object.

Image

The argument is a Sun raster image or a HIPS image.

Text

The argument is a character string.

If you want to cut/copy a type that isn't listed above, use the variation:

```
[obj drawproc type] /PutAny /Clipboard Send
```

where *obj* is the object to be cut/copied to the clipboard, *drawproc* is a procedure that will draw the object in the clipboard (or *nullproc*), and *type* is the name of the object type that will be displayed at the top left of the clipboard.

Since message passing in HyperNeWS cannot return values, you get the value of the clipboard by sending the clipboard a message and waiting for the clipboard to send a message back to you. The syntax for this is:

```
[arg] /Get /Clipboard Send
```

where *arg* is either a name, or an array of names, of types you are willing to accept. The type names that can be sent by themselves or as an element of an array are the same as those listed above for putting objects into the clipboard.

In addition, if the clipboard contains an object of some other type, or if you are willing to accept the clipboard regardless of the object's type, the argument `/Any` can be used by itself. That is, using this argument, the clipboard will send a reply message with data as long as the current card of the clipboard is not empty. The `/Any` argument does not make sense as an element of an array, so it should not be used in that manner.

The messages you may expect to receive from the Clipboard stack are:

/ClipboardFooData

Foo will be the type of data you requested, and the data is returned as an argument.

/ClipboardEmpty

The clipboard was empty. A NIL argument is returned.

/ClipboardWrongType

The type of the data in the current card of the clipboard is not the type (or one of the types) you requested. A NIL argument is returned. This reply is not possible if you sent `/Any` as the argument.

6.3 The Dialog Stacks

6.3.1 Basics

The dialog stacks provide an easy way to do basic and consistent I/O with the user. The NoteDialog shows a prompt or message you provide, with a button to continue. The ConfirmDialog has two buttons, so the user can choose between two options. The InputDialog shows an input field for the user to type into, with the ability to cancel if she chooses.

Each variety of stack takes an argument, `arg`, which may be either a character string or a dictionary. If it is a dictionary, a number of options can be changed. If the argument is a string, then it is the message to be displayed, and the options will take on their default values.

6.3.2 The NoteDialog Stack

The note dialog offers an easy way to display a message with a single pushbutton the user can press to continue. The syntax for this stack is:

```
[arg] /Init /NoteDialog Send
```

The dictionary keys and their defaults are:

/Message

The string to be displayed. This value has no default, so it must be provided.

/Title

The string to show at the top of the dialog. This might be the name of the application. The default is “Note”.

/Button

The string to be shown in the pushbutton. The default is “Continue”.

/Reply

The message that should be returned when the pushbutton has been pressed. The default is `/NoteReply`.

/Sender

The object that the reply should be sent to. The default is the object that sent `/Init` to the NoteDialog stack.

/ScreenX

The X coordinate where the left edge of the stack should be positioned. The default is 25, to line up the button with the system buttons.

/ScreenY

The Y coordinate where the bottom edge of the stack should be positioned. The default is 550.

/Beep

A boolean that states if the stack should beep when it is opened. The default is false.

6.3.3 The ConfirmDialog Stack

The confirm dialog offers an easy way to display a question and get a yes or no response from the user via a pushbutton. The syntax for this stack is:

```
[arg] /Init /ConfirmDialog Send
```

The dictionary keys and their defaults are:

/Message

The string to be displayed. This value has no default, so it must be provided.

/Title

The string to show at the top of the dialog. This might be the name of the application. The default is “Confirm”.

/Button

A 2-element array with the strings to be shown in the pushbuttons, with the top button listed first. The defaults are “Yes” and “No”.

/Reply

The message that should be returned when a pushbutton has been pressed. It may either be a message name, or a 2-element array of message names that correspond to the button labels. If this value is a single name, the message is returned with a string argument representing the label of the pressed button. If this argument is an array, the NIL argument will be returned with the appropriate message. The single default message is `/ConfirmReply`.

/Sender

The object that the reply should be sent to. The default is the object that sent `/Init` to the ConfirmDialog stack.

/ScreenX

The X coordinate where the left edge of the stack should be positioned. The default is 25, to line up the buttons with the system buttons.

/ScreenY

The Y coordinate where the bottom edge of the stack should be positioned. The default is 550.

/Beep

A boolean that states if the stack should beep when it is opened. The default is false.

6.3.4 The InputDialog Stack

The input dialog displays a dialog that has an input field for the user to type into, and a pushbutton typically used to cancel the operation. A carriage return completes a successful operation. The syntax for this stack is:

```
[arg] /Init /InputDialog Send
```


The dictionary keys and their defaults are:

/Message

The string to be displayed. This value has no default, so it must be provided.

/Title

The string to show at the top of the dialog. This might be the name of the application. The default is "Input".

/Button

A string to be shown in the pushbutton. The default is "Cancel".

/Reply

The message that should be returned when the pushbutton or the carriage return has been pressed. It may either be a message name or a 2-element array of message names that correspond to the success or failure of the input operation, respectively. A success occurs when there is text in the input field and the carriage return is pressed. A failure occurs if the carriage return key is pressed with no text in the input field, or all cases when the button is pressed. The text is returned for successful cases, and a nullstring is returned in all failing cases. The single default message is `/InputReply`.

/DefaultInput

A string to be initially displayed in the input field. The default is a nullstring.

/Sender

The object that the reply should be sent to. The default is the object that sent `/Init` to the InputDialog stack.

/ScreenX

The X coordinate where the left edge of the stack should be positioned. The default is 25, to line up the buttons with the system buttons.

/ScreenY

The Y coordinate where the bottom edge of the stack should be positioned. The default is 550.

/Beep

A boolean that states if the stack should beep when it is opened. The default is false.

7 Customizing Canvas Objects

7.1 Basics

While all HyperNeWS objects have a certain default behavior, you can modify that behavior by adding scripts to an object. You may have already seen examples of modifying behavior using the `Action` routine. Here we'll go through an example that shows how to modify that behavior. And since the Canvas object has so little default behavior to begin with, it makes for a good example.

When working with a canvas, or any other object that may have a graphic representation, you need to worry about being able to redraw it after the object has been damaged, such as when a window that was obscuring it is moved. Therefore, the data needed to redraw the canvas should be maintained within the object. If the data changes, then the canvas should be redrawn to reflect the changes in the object.

7.2 Events

Back in the Users Guide, we talked about events that are sent to different objects, and the fact that canvas objects receive a set of messages that other objects don't see. These include `OnEnter` and `OnExit`, sent when the cursor enters and exits the canvas, respectively. The canvas also receives the messages `OnLeft` (disguised as `OnMouse` in other card object types), `OnMiddle`, and `OnRight`. After any of the latter three messages has been received, you can determine where the mouse was clicked by referencing the `LastX` and `LastY` values, like so:

```
/OnLeft {
       LastY Height 2 div lt LastX Width 2 div lt and {
              % click was in lower-left quarter of the canvas
       } if
} def
```

Other routines that you will most likely want to include in your canvas scripts are:

Draw

The routine that will redraw your objects on the canvas.

OnUpdate

An `OnUpdate` message is sent to a stack whenever a part of it has been damaged. That is, if the entire screen is redrawn, or if a window that was covering your stack is closed or sent to the back, an `OnUpdate` message will notify you that you need to redraw all or part of your stack. However, you typically don't supply an `OnUpdate` routine yourself. By default, the message will redraw the canvas in its fill color, and then outline the canvas in its stroke color.

OnStackSize

If your stack is resizable, this message will be sent to all card objects on the current card when the stack is resized. What you'll want to consider is how the object should be moved or resized when this occurs. The `Position` message in class CardObject can be used to place your card objects in certain places.

7.3 An Example

As an example, we'll use the stack called CanDemo, shown in Figure 2. The stack itself is located in `$HNHOME/share/examples`, and the script for the Canvas object is shown here:

```
/Things     [] def          % An array of things to display
/Border     20 def          % Size of border
/InColor    [0 1 0 0] def   % Color of internal area

% Clear the canvas and empty the /Things array
/Clear {
    /Things [] def
    Draw
} def

% Draw the canvas
/Draw {
    NeedsDraw? {
        DrawBack
        DrawThings
    } if
} def

% Draw the background of the canvas
/DrawBack {
    gsave
        SetCanvas
        InColor SetColor
        Border dup dup 2 mul
        Width 1 index sub Height 3 -1 roll sub
        rectpath fill
    grestore
} def

% Draw the things in the /Things array
/DrawThings {
    gsave
```



```
        SetCanvas
        Color SetColor
        Things {aload pop DrawThing} forall
    grestore
} def

% Draw one of the things
/DrawThing { % x y --
    % Choose one (or more) of these
    10 0 360 arc fill
    %25 10 rectpath fill
    %moveto (NeWS is gNaRLy) cshow
} def

% Respond to a PointButton click
/OnLeft { % --
    LastX LastY 2 copy DrawThing
    [[4 2 roll]] Things exch append
    /Things exch def
} def

% Respond to an AdjustButton click
/OnMiddle { % --
    NIL /StackOnMiddle MyStack Send
} def
```

The CanDemo stack contains a Canvas object that has a white border and a green center. It behaves the following way: If you click the PointButton over the

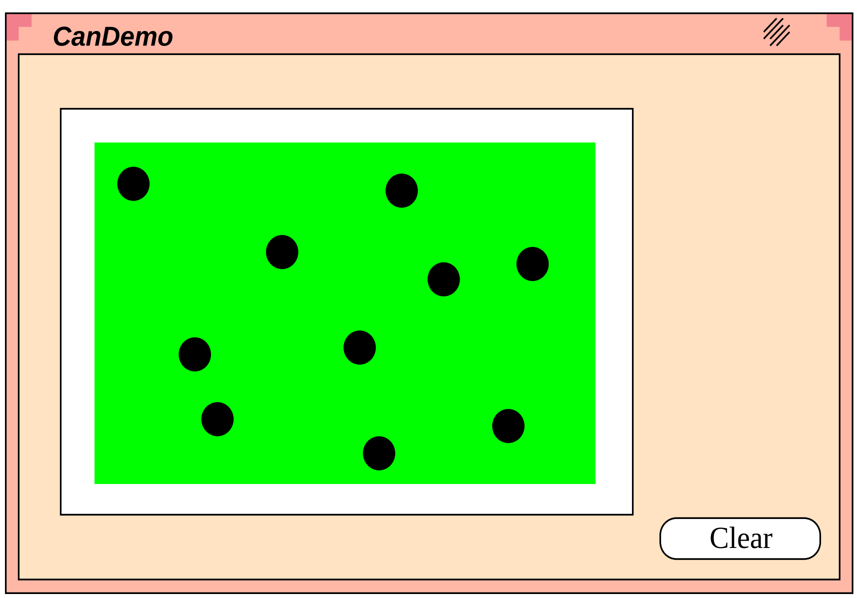

Figure 2: The CanDemo stack.

canvas, it will draw a dot. If you click on the *Clear* button, it will erase all the dots. Very elementary. Now let's look at the code itself.

As we mentioned before, the stack needs to keep a data structure with enough data to redraw the image in the canvas. The `Coords` array will hold 2-element arrays of X,Y coordinates where dots should be drawn. Each time a dot is added, the 2-element array for that dot is appended onto the existing array. When *Clear* is pressed, the array is reset to a length of zero.

The `Clear` routine gets invoked when the *Clear* button is pressed. It resets the coordinate array and sends itself an `OnUpdate` message. In most cases, a canvas wouldn't send itself this message. But we need to make sure the outer area of the canvas is redrawn (with the canvas' fill color) after *Clear* is pressed, in case some dots are outside the green area that `DrawBack` redraws.

The `Draw` routine will be invoked when HyperNeWS when it has determined that the canvas has been damaged. The routine checks the `NeedsDraw?` flag to save us the trouble of redrawing the canvas if it isn't currently visible. Then our two routines are called that will draw our green background and all the dots on the canvas.

The `DrawBack` routine sets the canvas object to be the current canvas, sets our green color to be the current color, then subtracts the border from the size of the canvas and draws a green rectangle. This constitutes our simple background.

`DrawThings` will repeatedly call the `DrawThing` routine for every point in the `Coords` array.

When passed the X,Y coordinates of a thing, the `DrawThing` routine will draw a thing at that location. We first set the color to the stroke color defined in the CanvasInfo stack. By using `Color` and `FillColor` when possible, you give the user the flexibility to easily change those colors without delving into the stack itself. Then we provide three different *things* that can be drawn. Choose which one you want to use by uncommenting it. If you want to draw more than one, be sure to include `2 copy` where necessary so the X,Y coordinates are available for each of the things being drawn.

The main user interaction of this demo is that the stack should intercept PointButton clicks and place a thing at that location. When the left mouse button is pressed, a message will be sent to the canvas that invokes the `OnLeft` routine. It finds the coordinates of the last event from `LastX` and `LastY`, draws a thing at that point, and then concatenates the coordinates on the end of the `Coords` array used for updating the canvas.

Finally, we want the AdjustButton to behave the same for this stack as it does by default for other stacks. So when we receive an `OnMiddle` message, we turn around and sent a `StackOnMiddle` message to the stack, which will move the stack around the screen as long as the AdjustButton is held down.

7.4 Design Considerations

By breaking up the function of the script into modular routines, it is easy to make changes. For instance, to draw a different object, you only have to change the `DrawThing` routine. If you wanted to draw some other figure over the green background, you could simply add a new routine called `DrawFig` that draws the figure, and place a call to that routine between the two existing calls to draw routines in `Draw`.

You should also think carefully about where you want to place the `gsave-grestore` pairs. Though it is always safer to include more than you need, they are relatively time-consuming operations, and unrestrained use of them may cause a noticeable performance degradation for complex drawings.

8 Creating New Objects

8.1 Basics

In this section we'll describe the steps to take when you create your own objects. We will illustrate with a dial object (see Figure 3), which is located in `$HNHOME/Stacks/Misc` if you want to take a look at the code itself. This object is complicated in that it contains many parameters which affect the object's display. The work that we have to do falls into three steps:

1. Create a script which displays the object and implements the necessary mouse interaction.

2. Provide a mechanism for setting the values of the display parameters.

3. Provide a mechanism for saving the values of display parameters, so that copies of the dial object can be saved and reloaded in the same way as any other HyperNeWS object.

8.2 Creating the Object Script

In a simple sense every HyperNeWS object with a user-defined script is a new object created by the user. However, the behavior defined by the script does not usually alter the display. In creating a dial object we must alter both visual display and the mouse behavior of the object. To create a button with a different look, programming is not necessary. You just draw the new shape with the HyperDraw stack and paste the drawing into an existing button. When the dynamic behavior

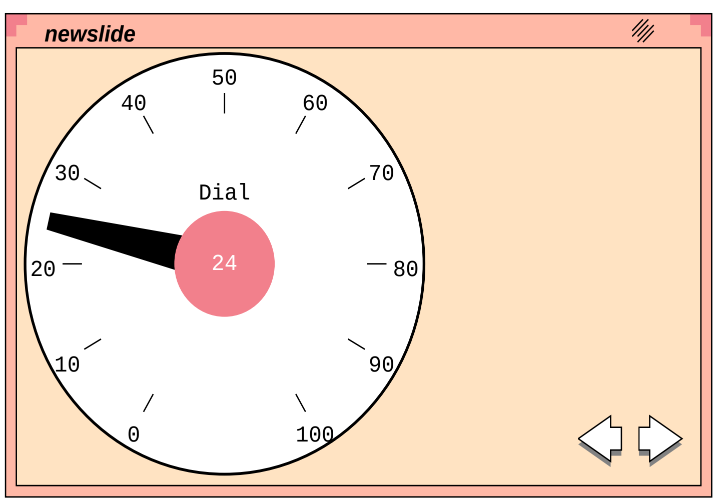

Figure 3: A HyperNeWS Dial object.

has to be changed, programming is necessary and this is the subject we now address.

We have modified a standard slider for the dial, since this object contains a parameter (*Value*) which can be used to display the value of the dial. Note that when modifying the script, we don't do the editing in the standard text item of the script stack, as this is not suitable for editing large amounts of text. Instead we create a file (`dial.script`) which is loaded from the object's script. This allows a standard text editor to be used.

For the slider object two of the standard methods are over-written. These are the `Draw` and `OnMouseBar` methods, which provide for display and interaction, respectively. In addition a new method to be called from external objects, `UpdateValue`, has been added. The reason for this is discussed below.

The `Draw` method is called whenever an object is redisplayed. As long as you can program in PostScript this is straightforward. It should be kept in mind that you are responsible for setting up the appropriate canvas and graphics state for the object to be drawn, as HyperNeWS does not provide these facilities. The reason for setting up the canvas is that the drawing might be requested by an object on another stack, in which case the active process will have its current canvas set incorrectly. The graphics state is, by convention, set to a reasonable value by objects when they finish drawing, so only minor changes such as setting the appropriate clipping path need to be done. A reasonable way to proceed is to issue the following instructions at the beginning of a draw method:

```
...
SetCanvas % set the canvas
gsave
    InitMatrix newpath 0 0 Width Height rectpath Clip
    ....
grestore
```

This ensures that any changes to color, linewidth, etc. are localized.

Very much the same idea holds for the `OnMouseBar` procedure. Since this is dynamic, and you wish to access events, the following ideas may come in useful. First, objects use the HyperNeWS mechanism for accessing events. This means that the last event has the name `LastEvent` and its X and Y coordinates can be accessed from the variables `LastX` and `LastY`. Second, the following is a general way to handle sequences of events during dragging:

```
currentcanvas dup createoverlay
    matrix currentmatrix exch setcanvas setmatrix
    LastX LastY {
        erasepage
        .... % Your specific handler
    } GetAnimated waitprocess pop
```



```
    erasepage
    matrix currentmatrix exch setcanvas setmatrix
```

You just insert the drawing commands you wish in the middle of this procedure.

The reason for including the new method `UpdateValue` is that for the dial object, a change in value should not involve a complete redraw as happens with the standard `SetValue` procedure, as the complexity of the object causes excessive flicker when updated. The `UpdateValue`method just redraws the pointer of the dial, which produces a nice effect.

8.3 Saving Variable Values

The sample dial object has two variables, `BossColor` and `PointColor` which alter the color of the central boss and pointer respectively, and which we wish to save with the object, either when it is written with its stack to disk, or when we copy the object. If we don't, we will always get the default colors when we load the stack even if we save it with the new colors, because the script is executed when the stack is loaded, and it initializes the colors to the values chosen when the script was created. The `LocalInstances` mechanism in HyperNeWS gives us this capability. If an object's script contains an array with this name, the keys defined within it will be saved and copied with the object. In the future, these values will be used instead of the default values. For example, in our situation we could have the following line in the dial object script:

```
/LocalInstances [/BossColor /PointColor] def
```

Be sure to initialize the variables in the first place, so that they are valid in case they are never modified.

8.4 Setting Values

To edit the dial object it is best to create an info stack for it. The info stack for dial is shown in Figure 4.You should look at the scripts in this stack to see how it works. A stack like this can be made by copying one of the standard info stacks, and changing the `Info` method of the new object. The method for the dial is:

```
(--
Show info on the dial.)
/Info { % --
    NIL /Init /DialInfo Send
} mdef
```

Notice the use of the documenting string and `mdef`. This allows the object to be included in the internal documentation. It is not necessary to include this. Instead, the following definition would have been sufficient:

```
/Info {
    NIL /Init /DialInfo Send
} def
```

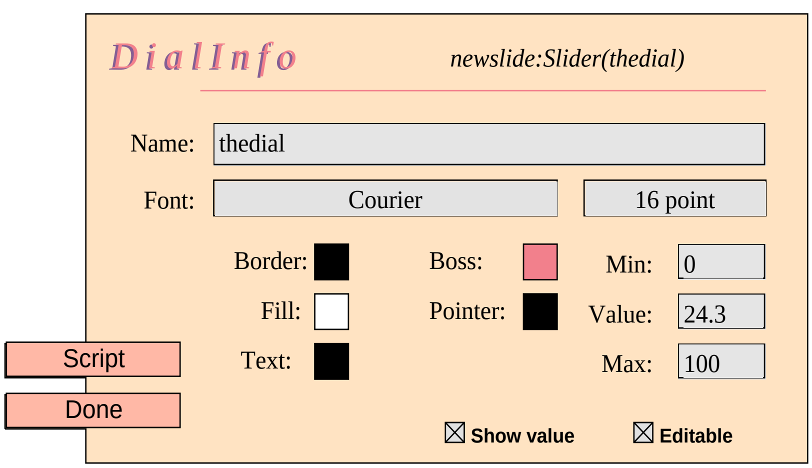

Figure 4: The DialInfo stack.

9 System Methods and Instance Variables

9.1 Methods by Object Type

9.1.1 RootObject Methods

destroy – destroy –

Destroy the object.

new – new –

Create an object.

9.1.2 GraphObject Methods

InvalidScriptNames listofnames InvalidScriptNames –

Warn of redefinition of protected names.

ScriptInfo – ScriptInfo –

Show the ScriptInfo stack for this object.

SetScript text SetScript –

Change the script of the object.

9.1.3 BackGround Methods

BackInfo – BackInfo –

Show information on this object.

Draw – Draw –

Draw all the objects it contains.

Hide – Hide –

Ignored.

NewObj typename NewObj –

Create a new object. The typename should either be a classname or the name of an object on the 'NewObjects' stack.

Show – Show –

Ignored.

9.1.4 Stack Methods

CloseStack – CloseStack –

Hide the stack. If the stack has been updated the user will be asked for confirmation.

Connect – Connect –

Connect a client to the stack. This should only be sent from a client to a stack.

CopyDraw – CopyDraw –

Create a drawing of the stack and put it on the clipboard.

CopyShape – CopyShape –

Copy the shape of this stack to the clipboard.

DeleteCard – DeleteCard –

Delete the current card. You can't delete all cards in a stack.

Draw – Draw –

Draw the stack and its contents.

FindCard string FindCard –

Find the next card containing a text item containing the search string.

FindString – FindString –

Show the FindString stack to ask the user for a string to find.

GoHomeCard – GoHomeCard –

Go to the home (first) card of the stack.

GoLastCard – GoLastCard –

Go to the last card of the stack.

GoNextCard – GoNextCard –

Go to the next card of the stack.

GoPreviousCard – GoPreviousCard –

Go to the previous card of this stack.

GotoCard card/cardname/cardnumber GotoCard –

Go to the indicated card.

Hide – Hide –

Hide the stack.

IconifyStack – IconifyStack –

Iconify the stack. Use an icon stack if it can be found. Otherwise create a default icon.

Move x y Move –

Move the stack to a new location.

NewCard – NewCard –

Insert a new card just after the current card, and make it the current card.

OnMouse – OnMouse –

Send the stack to the foreground. See also 'ToFront'.

OnMouseEnter – OnMouseEnter –

The mouse has entered the stack.

OnMouseExit – OnMouseExit –

The mouse has exited the stack.

OnMoveObj – OnMoveObj –

Interactively move the stack window.

OnSizeObj – OnSizeObj –

Interactively resize the stack window if 'Resizable?' is true. The user will resize the stack by its nearest corner.

PasteObj – PasteObj –

Paste the clipboard contents into the stack if it is a card.

PasteShape – PasteShape –

Paste the clipboard contents into the stack shape if it is a drawing.

Redisplay – Redisplay –

Redisplay the entire stack.

Reshape	x y width height **Reshape** – Change the size and location of the stack.
Revert	– **Revert** – Revert the stack to the last version saved.
Save	– **Save** – Save the stack using its current name.
SaveAs	– **SaveAs** – Ask the user for a new name for the stack and save it.
SaveStack	name **SaveStack** – Give the stack a new name and save it. Nothing happens when the name is invalid.
SelectObject	object **SelectObject** – Select an object.
Show	– **Show** – Show the stack.
Size	width height **Size** – Change the size of the stack.
StackInfo	– **StackInfo** – Get information on this stack.
StartEditMode	– **StartEditMode** – Start edit mode.
StopEditMode	– **StopEditMode** – End edit mode and continue in browsing mode.
SwitchEditMode	– **SwitchEditMode** – Switch between edit mode and browsing mode.
ToBack	– **ToBack** – Send the stack to the background.
ToFront	– **ToFront** – Send the stack to the foreground.

9.1.5 Card Methods

BackInfo – BackInfo –

Passed on to its parent (a background).

CardInfo – CardInfo –

Show information about this card.

CopyCard – CopyCard –

Copy this card to the clipboard.

NewBack – NewBack –

Create a new background for this card.

PasteObj – PasteObj –

Paste the clipboard contents onto this card if it is a card object.

9.1.6 CardObject Methods

CardInfo – CardInfo –

Passed to the current card.

ClearObj – ClearObj –

Delete the object.

CopyDraw – CopyDraw –

Copy an object as a drawing to the clipboard.

CopyObj – CopyObj –

Copy the object to the clipboard.

CutObj – CutObj –

Cut an object. Equivalent to CopyObj followed by ClearObj.

Draw – Draw –

Draw the object.

GenDraw – GenDraw –

Generate a drawing of the object and put it in the global variable 'TheDrawing'.

Hide – Hide –

Hide the object.

Move x y Move –

Change the position of the object.

OnStackSize – OnStackSize –

Called when the stack size changes. Usually calls the routine 'Position' to relocate the object.

Position fromleft frombottom fromright fromtop Position –

Change the position and size of the object relative to the edges of the stack. 'null' means not relevant, and 'same' means keep the same distance as before.

Reshape x y width height Reshape –

Change the size and location of the object.

Show – Show –

Show the object.

Size width height Size –

Change the size of the object.

ToBack – ToBack –

Send the object to the back.

ToBackGround – ToBackGround –

Send the object to the background.

ToCard – ToCard –

Send the object to the card.

ToFront – ToFront –

Send the object to the front.

ToStack – ToStack –

Send the object to the stack.

Update – Update –

Update the object. Make sure that all objects below and above are updated as well.

9.1.7 Button Methods

BestSize – BestSize –

Change the size of the button to fit the drawing. This only applies to drawing-type buttons.

Draw – Draw –

Draw the button.

DrawHilite value DrawHilite –

Draw the hilite of the button. This applies only to check boxes.

DrawPushButton boolean DrawPushButton –

Draws a push button. It the boolean argument is false the button is drawn inverted.

Info – Info –

Show info on the button.

OnMouse – OnMouse –

Called when the user clicks the left mouse button in this button. An 'Action' message is send to the button when the mouse click is successful.

PasteObj – PasteObj –

Paste the clipboard contents into the button if it is a drawing. If the clipboard does not contain a drawing, the message is passed on to the current card of the stack.

Path – Path –

Set the current path to the path of the button

SetValue nr/boolean SetValue –

Change the value of the button. This does not apply to push-buttons. A boolean value is translated into 0 or 1.

9.1.8 EditText Methods

Clear – Clear –

Clear the currently selected text.

Copy – Copy –

Copy the currently selected text onto the clipboard.

Cut – Cut –

Cut text. Equivalent to 'Copy' followed by 'Clear'.

EditInsertString string EditInsertString –

Insert a string in the text at the current cursor position.

FocusCursor – FocusCursor –

Scroll the text (if needed) so that the cursor becomes visible.

Info – Info –

Show info on the text.

Paste – Paste –

Paste text from the clipboard into the object. If no text is found on the clipboard the message is passed on to the currentcard.

SelectAll – SelectAll –

Select all of the text.

SelectDown – SelectDown –

Move the cursor down in the text.

SelectLeft – SelectLeft –

Move the cursor left in the text.

SelectLine linenumber SelectLine –

Select a line of text.

SelectRight – SelectRight –

Move the cursor right in the text.

SelectUp – SelectUp –

Move the cursor up in the text.

SetEditTop linenumber SetEditTop –

Set the top line of the text. The first line of text is line number 0.

SetScroll boolean **SetScroll** –

Change the scrollbar of the text. If the argument is true a scrollbar will be added to the text. It will be removed if the argument is false.

SetValue text/name/string **SetValue** –

Set the value of the text if the argument is text. It must be an array of strings (remember to use enough brackets).

SortValue – **SortValue** –

Sort the text lines in ascending order.

Write string **Write** –

Append a string to the end of the text.

WriteLn string **WriteLn** –

Append a line to the end of the text.

9.1.9 Slider Methods

ConvertValue value **ConvertValue** string

Convert the integer value to a string for display.

Draw – **Draw** –

Draw the slider.

Info – **Info** –

Show info on the slider.

OnSize – **OnSize** –

Called when the size of the slider changes. The slider will become 'vertical' when the height exceeds the width.

SetRange minvalue value maxvalue **SetRange** –

Change the range of the slider. The minvalue must be less or equal to the maxvalue. The value must lie between the min and max values.

SetValue value **SetValue** –

Change the value of the slider. The value must lie between the minvalue and maxvalue of the slider. See 'SetRange'.

9.1.10 PullDown Methods

Draw – Draw –

Draw the object.

DrawValue – DrawValue –

Draw the label of the object. Usually called after a 'SetValue'.

Info – Info –

Show info on the object.

SetValue name/string/text SetValue –

If the argument is a name or a string the label of the object is changed (only if the string occurs in the menu). If the argument is text, the menu is changed.

9.1.11 ColorSelect Methods

Draw – Draw –

Draw the object.

DrawValue – DrawValue –

Draw the value (the selected color).

Info – Info –

Show info on the object.

SetValue color SetValue –

Change the value. A color is a 4-element array defining the red, green, blue and gray component of a color.

9.1.12 Canvas Methods

Draw – Draw –

A dummy routine that is usually over-ridden by the canvas object.

Info – Info –

Show info on the canvas.

OnClose – OnClose –

Hide the canvas.

OnOpen – OnOpen –

This method creates the canvas. It is called when the canvas is first shown.

OnUpdate – OnUpdate –

Redraw the background and send 'Draw' to the canvas. This message is generated by the system when the canvas has been damaged.

ToBack – ToBack –

Ignored.

ToFront – ToFront –

Ignored.

9.2 Instance Variables by Object Type

9.2.1 RootObject Instances

ObjectClass

The class dictionary of the object's class.

TYPE: DICT DEFAULT: dictionary[6/10]

ObjectName

The name of the object.

TYPE: NULL DEFAULT: null

ObjectParent

The parent of the object.

TYPE: NULL DEFAULT: null

ObjectType

The type of the object (class name).

TYPE: NAME DEFAULT: /RootObject

Self

The object itself.

TYPE: NULL DEFAULT: null

9.2.2 GraphObject Instances

Height

The height of the object in its own coordinate system.

TYPE: INTEGER DEFAULT: 0

LocalDict

The object dictionary. Can be set by the user and is saved with all its values.

TYPE: DICT DEFAULT: dictionary[0/32]

Matrix

The matrix which defines the coordinate system in which the object is displayed. Before using, it is concatenated to the stack's matrix.

TYPE: ARRAY DEFAULT: [1 0 0 1 0 0]

MyStack

A reference to the stack to which this object belongs. For stacks, this is identical to 'Self'.

TYPE: NULL DEFAULT: null

Script

The script of the object. A textual version of user-defined methods.

TYPE: ARRAY DEFAULT: [()]

Width

The width of the object in its own coordinate system.

TYPE: INTEGER DEFAULT: 0

9.2.3 BackGround Instances

Objects

A list of objects. The front-most object appears last in the list.

TYPE: ARRAY DEFAULT: []

Showing?

A flag indicating whether it is currently visible on the stack.

TYPE: BOOLEAN DEFAULT: false

9.2.4 Stack Instances

CardList

The ordered list of cards in the stack. The first element of the list is the home card of the stack.

TYPE: ARRAY DEFAULT: []

Cdate

The creation date of the stack.

TYPE: STRING DEFAULT: (?)

Creator

The login name of the stack's creator.

TYPE: STRING DEFAULT: (newsdev)

CurrentAscii

The object which currently holds the focus for characters typed in the stack.

TYPE: NULL DEFAULT: null

CurrentBack

The current background of the stack.

TYPE: NULL DEFAULT: null

CurrentCard

The current card of the stack.

TYPE: NULL DEFAULT: null

CurrentObj

The currently selected object of the stack (if any).

TYPE: NULL DEFAULT: null

Drag

True if the user is allowed to drag the stack.

TYPE: BOOLEAN DEFAULT: false

EditGrid

The size of the stacks edit grid.

TYPE: INTEGER DEFAULT: 5

EditMode?

True if the stack is in edit mode.

TYPE: BOOLEAN DEFAULT: false

Editable?

True if the stack is editable.

TYPE: BOOLEAN DEFAULT: true

LoadFileName

The file from which the stack was loaded.

TYPE: NULL DEFAULT: null

Mdate

The modification date of the stack.

TYPE: STRING DEFAULT: (?)

MinHeight

If 'Resizable?' is true, then the minimum allowed height for the stack.

TYPE: INTEGER DEFAULT: 300

MinWidth

If 'Resizable?' is true, then the minimum allowed width for the stack.

TYPE: INTEGER DEFAULT: 400

MyCanvas

The canvas of the stack window. This canvas is created as soon as the stack is loaded.

TYPE: NULL DEFAULT: null

ObjectDict

The dictionary which relates object names to actual objects. Every object of the stack must be defined in this dictionary under its current name.

TYPE: NULL DEFAULT: null

ObjectNr

The last used object number for objects with no name.

TYPE: INTEGER DEFAULT: 0

Resizable?

True if the stack can be resized.

TYPE: BOOLEAN DEFAULT: false

Retained

True if the stack's canvas is in Retained mode.

TYPE: BOOLEAN DEFAULT: false

SaveBehind

True if the stack's canvas is in SaveBehind mode.

TYPE: BOOLEAN DEFAULT: false

ScreenX

The current X position of the stack on the screen.

TYPE: INTEGER DEFAULT: 100

ScreenY

The current Y position of the stack on the screen.

TYPE: INTEGER DEFAULT: 100

Showing?

True if the stack is visible.

TYPE: BOOLEAN DEFAULT: false

StackPath

An executable array which, when executed, sets the current path to the shape of the stack. It is derived from the stack shape.

TYPE: NULL DEFAULT: null

StackShape

A drawing which describes the shape and background of the stack.

TYPE: NULL DEFAULT: null

Untitled?

True if the stack is untitled.

TYPE: BOOLEAN DEFAULT: true

UpdateLock

True if updates are currently locked.

TYPE: INTEGER DEFAULT: 0

Updated?

True if the stack has been updated.

TYPE: BOOLEAN DEFAULT: false

9.2.5 Card Instances

9.2.6 CardObject Instances

Color

The line/stroke/text color of the object.

TYPE: ARRAY DEFAULT: [0 0 0 0]

FillColor

The color used to fill the object.

TYPE: ARRAY DEFAULT: [1 1 1 1]

Focus?

A flag which is true if the object currently holds the focus for characters typed in the stack.

TYPE: BOOLEAN DEFAULT: false

FontName

The name of the font which is used to display text inside the object.

TYPE: NAME DEFAULT: /Times-Roman

FontSize

The size of the font (in points) which is used to display text inside the object.

TYPE: INTEGER DEFAULT: 18

Label

The label of the object (if any).

TYPE: STRING DEFAULT: ()

Value

The value of the object. Its meaning depends on the type of the object.

TYPE: NULL DEFAULT: null

Visible?

A flag which indicates whether the object is visible or not.

TYPE: BOOLEAN DEFAULT: false

9.2.7 Button Instances

ButtonType

The type of button: /Push, /Drawing, /Check, or /Transparent.

TYPE: NAME DEFAULT: /Push

ButtonWidth

Size of the checkbox button.

TYPE: INTEGER DEFAULT: 14

DrawingData

The drawing shown by the button when it is a drawing button.

TYPE: NULL DEFAULT: null

Round

Rounding of pushbuttons.

TYPE: INTEGER DEFAULT: 10

Value

State of the button.

TYPE: INTEGER DEFAULT: 0

9.2.8 EditText Instances

Adjust

Adjustment of the text: /Left, /Right, or /Center.

TYPE: NAME DEFAULT: /Left

Cursor?

True if the cursor is visible.

TYPE: BOOLEAN DEFAULT: true

EditBox?

True if there is a box around the text.

TYPE: BOOLEAN DEFAULT: true

EditCurLn

The screen line on which the current selection starts.

TYPE: INTEGER DEFAULT: 0

EditCurPos

The postition in the screen line of the start of the current selection.

TYPE: INTEGER DEFAULT: 0

EditEndLn

The screen line on which the current selection ends.

TYPE: INTEGER DEFAULT: 0

EditEndPos

The postition in the screen line of the end of the current selection.

TYPE: INTEGER DEFAULT: 0

EditLines

The number of lines of text when the text is shown.

TYPE: INTEGER DEFAULT: 0

EditMap

A map into the text which describes for each screen line the start offset in the text line, the length of the screen line, and the text line number (in that order).

TYPE: ARRAY DEFAULT: [[0 0 0]]

EditOneLine?

True if typing a return character causes the 'Action' message to be sent.

TYPE: BOOLEAN DEFAULT: false

EditTopLn

Number of the top line in the text.

TYPE: INTEGER DEFAULT: 0

EditTrans?

True if the text is transparent.

TYPE: BOOLEAN DEFAULT: false

Editable?

True if the user is allowed to edit the text.

TYPE: BOOLEAN DEFAULT: true

Scroll?

True if the text has a scrollbar.

TYPE: BOOLEAN DEFAULT: false

ScrollColor

The color of the scroll bar.

TYPE: ARRAY DEFAULT: [0.6 0.6 0.6 0.6]

ScrollEnd

The end value of the scroll bar.

TYPE: INTEGER DEFAULT: 0

ScrollFill

The fill color of the scroll bar.

TYPE: ARRAY DEFAULT: [1 1 1 1]

ScrollPage

The length of scrolled text page.

TYPE: INTEGER DEFAULT: 10

ScrollStart

The start value of the scroll bar.

TYPE: INTEGER DEFAULT: 0

ScrollStroke

The stroke color of the scroll bar.

TYPE: ARRAY DEFAULT: [0 0 0 0]

ScrollValue

The value of the scroll bar.

TYPE: INTEGER DEFAULT: 0

Value

The actual text.

TYPE: ARRAY DEFAULT: [()]

9.2.9 Slider Instances

Box?

True if there is a box around the slider.

TYPE: BOOLEAN DEFAULT: true

Editable?

True if the slider can be changed by the user.

TYPE: BOOLEAN DEFAULT: true

InActiveColor

The fill color when the slider is inactive.

TYPE: ARRAY DEFAULT: [0.7 0.7 0.7 0.7]

MaxValue

The maximum value of the slider.

TYPE: INTEGER DEFAULT: 100

MinValue

The minimum value of the slider.

TYPE: INTEGER DEFAULT: 10

ShowValue?

True if the value of the slider is shown.

TYPE: BOOLEAN DEFAULT: true

SliderType

The type of the slider: /Scroll or /Bar.

TYPE: NAME DEFAULT: /Scroll

Value

The value of the slider.

TYPE: INTEGER DEFAULT: 80

ValueColor

The color of the sliders value part.

TYPE: ARRAY DEFAULT: [0.5 0.5 0.8 0]

Vertical?

True if the slider's height is greater than its width.

TYPE: BOOLEAN DEFAULT: false

9.2.10 PullDown Instances

SetLabel?

True if the label is to be changed when the user makes a selection.

TYPE: BOOLEAN DEFAULT: true

ShadowColor

The color of the shadow.

TYPE: ARRAY DEFAULT: [0.6 0.6 0.6 0.6]

ShadowSize

The size of the shadow.

TYPE: INTEGER DEFAULT: 3

Value

A list of menu items. A single '-' will appear as a horizontal line.

TYPE: ARRAY DEFAULT: [(one) (two) (-) (three)]

9.2.11 ColorSelect Instances

Value

The currently selected color.

TYPE: ARRAY DEFAULT: [1 0.7 0.7 0.5]

9.2.12 Canvas Instances

Cursor

The cursor of the canvas: /ptr, /hand1, /hand2, /hourg, or /xhair.

TYPE: NAME DEFAULT: /ptr

MyCanvas

The actual NeWS canvas.

TYPE: NULL DEFAULT: null

Retained

True if the canvas is to be retained.

TYPE: BOOLEAN DEFAULT: false

SaveBehind

True if the canvas is to be in SaveBehind mode.

TYPE: BOOLEAN DEFAULT: false

10 Index

