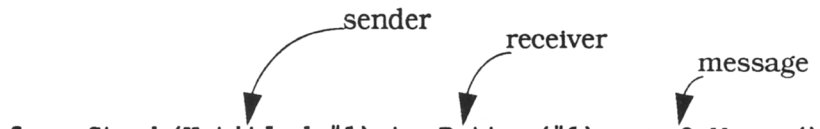
Finding Out About Messages

It's possible to print all messages on the console. See "Logging Messages" on page 3-33 to find out how to switch this feature on. As you use the system, you will see all the messages being generated.




```
from Stack(Untitled-#1) to Button(#1) --> OnMouse()
from Button(#1) to Button(#1) --> Action(0)
from Button(#1) to Stack(Message) --> Init((Hello
world!))
```

Figure 166

Part of the message log that is printed on the console when a button is pressed.

Writing Scripts

HyperLook objects know about certain messages. In this section examples are given of how to use the most common ones.

See also Appendix B, "Methods" for an overview of the most important HyperLook methods which are discussed in this chapter.

The Action Method

```
/Action        % value --
```

The **Action** method is called when the *action* of the object should be performed. This always happens after the user has changed the state of the object. It happens, for example, when a button is pressed or when the value of a slider is changed

The most common use of the **Action** method is to define the action of a button. Try this out by creating a button and giving it the following script[1]:

```
/Action {      % number -- number(ignored)
   beep
} def
```

When you press the button, the **Action** is executed and the **beep** operator causes the keyboard to beep.

Here is an example of how to use the **Action** message of a slider. Create a slider and give it the following script:

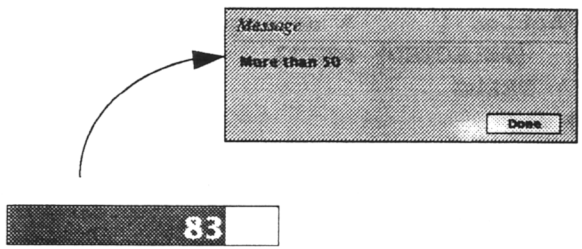
```
/Action {      % nr --
   50 gt {
        (More than 50) ShowMessage
   } {
        (Less than or equal to 50) ShowMessage
   } ifelse
} def
```

1. The arguments and results of the **Action** method can be ignored safely.

The **ShowMessage** function shows a pop up message window containing the argument string. When you have applied the script try, changing the value of the slider.

Figure 167

A slider in action.



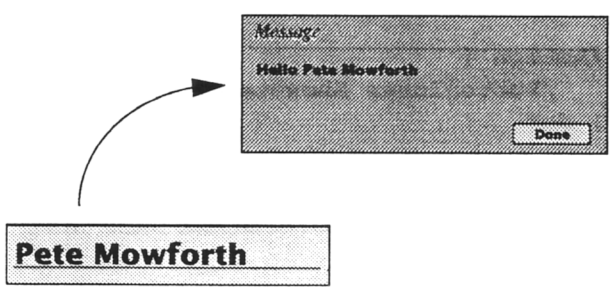
The **Action** message of text and field objects is called with a string as an argument. Here is an example of an **Action** method for a field object:

```
/Action {      % string --
   (Hello ) exch append ShowMessage
} def
```

When you type a string into the field and hit return, the **Action** method is called, with the string that you typed as an argument, so a message is shown.

Figure 168

A field in action.



The following script can be used for any object. Try assigning it to a PullDown object. It uses the **DEBUG** function to print a debugging message on the console.

```
/Action {      % string --
   (SELECTED) DEBUG
   DEBUG
} def
```

When you select an item from the pulldown menu a debug message is printed on the console:

```
DEBUG: SELECTED
DEBUG: two
```

You can use the **DEBUG** function whenever you want to print debugging information.

Manipulating Stacks

Now that you know how to program a button action, it is easy to show a stack when a button is clicked. Let's say you want to show the ButtonIdeas stack when button is clicked:

```
/Action {
   /ButtonIdeas ShowStack
} def
```

The **ShowStack** function loads a stack (if it is not already loaded) and shows it. You can hide the ButtonIdeas stack using the **HideStack** function:

```
/Action {
   /ButtonIdeas HideStack
} def
```

If you want to be really adventurous, you can hide the current stack. The variable **MyStack** returns the current stack:

```
/Action {
   MyStack HideStack
} def
```


See "Opening Stacks" on page 3-8 on how to open your stack again.

Sending Messages

The **Send** operator is used to send a message to an object. Sending messages is explained in more detail in "Messages" on page 7-12. It takes the following arguments:

```
/Send          % arg-array message target --
```

Create a button and give it the following script:

```
/Action {      % number -- number(ignored)
   [] /IconifyStack Self Send
} def
```

When you press the button it will send a message called **IconifyStack** to itself. Because a button does not know how to handle it, the message is passed on to its parent. The stack is iconified because the message is handled by the stack.

Messages can be sent directly to the stack or client by using the **MyStack** or **MyClient** variables as a target. There are various other ways in which messages can be sent:

```
[] /Message Self Send     % to the object itself
[] /Message MyStack Send  % directly to the stack
[] /Message ParentSend    % to the parent of the object
[] /Message ClientSend    % directly to the client
[] /Message /name Send    % to a named object
```

The target can be specified as a stack name or object name. Here is an example of a button script that iconifies the system stack.

```
/Action {
   [] /IconifyStack /system Send
} def
```

You can also send message between objects on the same stack. Create a slider and call it **Foo**. A button on the same stack can now send a message to the slider as follows:

```
/Action {
   [50] /SetValue /Foo Send
} def
```

This sends a **SetValue** message to the slider called **Foo**. The message has one argument, the number **50**.

Addressing Objects

Every stack name in the HyperLook system must be unique. Every object on a stack must have a name which is unique within that stack. This makes the addressing of objects very easy[1]:

```
/system FindStack DEBUG
DEBUG: instance(class(Stack))
```

Finds the stack named **system**.

The **FindObject** function finds an object on a stack. It takes a stack name and an object name as an argument. It provides you with an unambiguous way to address an object:

```
/system /Exit FindObject DEBUG
DEBUG: instance(class(Button))
```

This finds the object called **Exit** on the stack called **system**.

The **Send** primitive uses the **LookupObject** function to address objects. This function first tries to find the object on the current stack. If that fails it assumes that you are addressing a stack. For example:

```
/system LookupObject DEBUG
DEBUG: instance(class(Stack))
```

1. The output of **DEBUG** will appear on the console.

In this case it returned a stack. Addressing an object with **LookupObject** is also possible. For example:

```
/CardNr LookupObject DEBUG
DEBUG: instance(class(Field))
```

Here is a button script that address an object unambiguously using **FindObject**:

```
/Action {
   [] /Press /system /Exit FindObject Send
} def
```

The **Press** method only applies to buttons. It highlights the button and performs the button action just as if the user had pressed it.

Linking Buttons to Cards

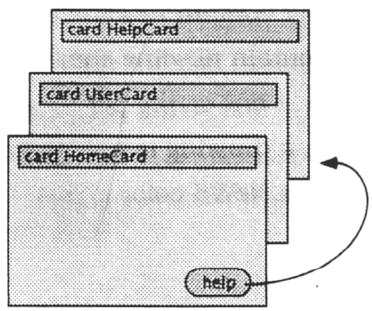
Now that you know how to send messages and how to address objects, it is easy to link a button to a card. Here is the script of a "next card" button. When the button is pressed the next card is shown:

```
/Action {
   [] /GoNextCard ParentSend
} def
```

The following button script is an example of how a button can go to a particular card of a stack. Create a new card and call it **HelpCard**.

Figure 169

A button can be linked to any card of the stack.



You can now create a button (on another card) that shows the card named **HelpCard**.

```
/Action {
   [/HelpCard] /GotoCard ParentSend
} def
```

You can also open a stack at a particular card. This is done by first sending a **GotoCard** message to the stack and then showing the stack using **ShowStack**.

To try this out, assign the following script to a button. It will open the **ButtonIdeas** stack at card number 2 (make sure the stack is hidden first).

```
/Action {
   [2] /GotoCard /ButtonIdeas Send
   /ButtonIdeas ShowStack
} def
```

The SetValue Method

Every object has a **SetValue** method. The value of the object depends on its type. The value of a text object is a string, while the value of a slider is a number. Here a a list of the value types for the HyperLook objects:

- **Button** -- *integer* -- the state of the button.
- **Slider** -- *integer* -- the value of the slider.
- **Field** -- *string* -- doesn't contain newline characters.
- **Text** -- *string* -- may contain newline characters.
- **List** -- *array-of-strings* -- one string per item.
- **DrawTool** -- *drawing* -- a drawing is a dictionary object.
- **ColorSelect** -- *color* -- a NeWS color object.

This button script sets the value of a text object called **MyText**.

```
/Action {
   [(Hi There,\nhave a nice day!)]
   /SetValue /MyText Send
} def
```

Note that the **\n** in the string inserts a newline in the text (see the Adobe PostScript manual).

Below is an example of a script for a slider. Every time the slider is changed by the user, it sends a **SetValue** message to a slider called **MyBrother**, keeping it up to date with the same value:

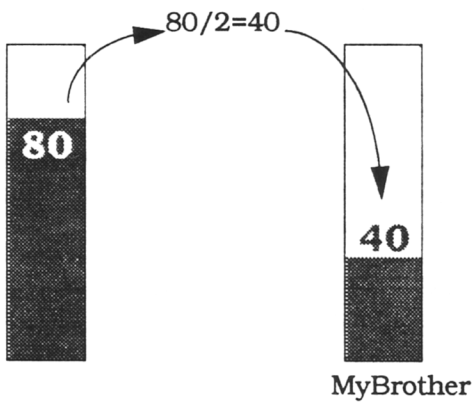
```
/Action {  % nr --
   [exch] /SetValue /MyBrother Send
} def
```

Alternatively you can change the value before setting the other slider. In this example the value is halved:

```
/Action {  % nr --
   [exch 2 idiv] /SetValue /MyBrother Send
} def
```

Figure 170

The value of the right slider is half the value of the left slider.



Getting the Value of an Object

You may have noticed that the **Send** operator does not return values. That means that you can't use it to access the value of an object. You must use the **send**[1] operator to access the value of an object's variable.

The definition of the send operator is as follows:

```
/send      % ...args message object -- results...
```

Note that the arguments are not in an array but simply on the stack. Also notice that the **send** operator can return results.

In the following example a slider accesses the value of another slider in the **Action** method:

```
/Action {
   /Value /MyBrother LookupObject send SetValue
} def
```

It first uses the **LookupObject** function to address the other slider before it gets the value of the slider using **send**. It then uses the **SetValue** method to set its own value to be equal to the value of the other slider.

The result is that the slider does not let the user change its value. It always resets itself to the value of the other slider.

The OnHelp Method

Section "Defining Help" on page 5-29 explains how to assign help text to an object. It is also possible to perform an action when the user presses the Help key over an object.

When this happens, an **OnHelp** message is sent to the appropriate object. This object can respond with any action it likes. For example:

```
/OnHelp {      % --
   (Sorry, no help today...) ShowMessage
} def
```

1. Notice the difference in capitalization between **Send** and **send**.

Pressing the **Help** key over an object with this script will show the message stack instead of the default help stack.

The Show and Hide Methods

Sending a **Hide** message to an object makes it invisible. Sending a **Show** message shows it again. To illustrate this, you need to create a check box and another object of arbitrary type called **Foo**.

Give the checkbox the following script:

```
/Action {       % state --
   0 eq {
       [] /Hide /Foo Send
   } {
       [] /Show /Foo Send
   } ifelse
} def
```

Switching the checkbox on shows the **Foo** object, and switching the checkbox off hides the **Foo** object.

The OnOpen and OnClose Methods

The **OnOpen** method of an object is called when the object first becomes visible. The **OnClose** method of an object is called when the object is removed from the screen.

You can use the **OnOpen** method to perform some initialization that may be needed. It can be used, for example, to give the object an initial value.

The following script for a field object sets the initial value of the field to today's date, this can be useful in database applications:

```
/OnOpen {       % --
   SystemDate SetValue
} def
```

The **SystemDate** operator returns a string containing today's date. The use of the OnClose methods is illustrated in the following script:

```
/OnClose {     % --
   (I'm leaving now!) ShowMessage
} def
```

You can use it for any object. It shows a message as soon as the object disappears of the screen.

Defining Your Own Methods

So far only system defined methods were discussed. You are not restricted to using these alone. You can add new methods to objects yourself.

Here is a script for a slider object that defines a method called **SetRootValue** for the object:

```
/SetRootValue {    % nr --
   sqrt SetValue
} def
```

The new method sets the value of the slider to the square root of a number. You can invoke this methods by sending a **SetRootValue** message to the object. For example, a button script may contain the following if the slider is called **Foo**:

```
/Action {      % --
   [25] /SetRootValue /Foo Send
} def
```

Using IncludeScript

You may find that your script gets too long to edit efficiently with the script properties editor. You can store the script in a file which you can edit with your favorite editor.

To do this, you must create a file containing the script in a resource directory. Usually this would be the same directory as the one where you save the stack containing the object. Now put the following in the script of the object:

```
(myfile.script) IncludeScript
```

This will read the object's script from the file **myfile.script** when you press the **Apply** button on the script property stack. You must apply the script every time you've made changes to the file.

Remember to keep the object and the script file together. The script file must be accessible when the object is loaded. The object will be loaded incorrectly if the script cannot be found.

Scripting Examples

This section contains a simple example and two more advanced examples.

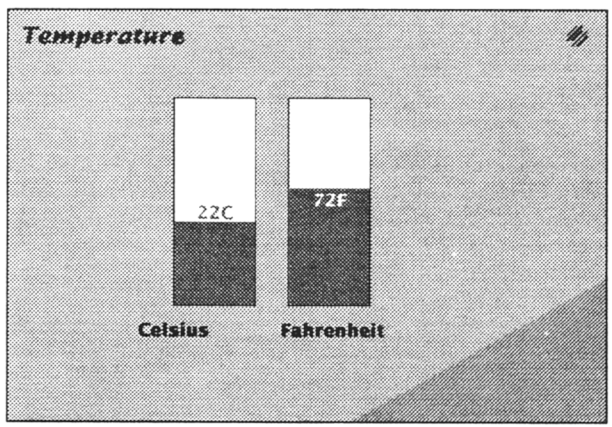
Fahrenheit and Celsius

In this example two sliders are programmed to represent temperatures in Fahrenheit and Celsius. When either of the sliders is changed the other slider's value is updated to the corresponding value.

For this example you need to create a stack containing two vertical bar sliders. Name the one on the left **Celsius**, the one on the right **Fahrenheit**. Set the range of the **Celsius** slider to be **-100** to **100**. The range of the **Fahrenheit** slider should be the same.

Figure 171

The Temperature example stack.



Assign the following script to the **Celsius** slider:

```
/ConvertValue {% val -- str
   (%C) sprintf
} def

/Action {% val --
   [exch 1.8 mul 32 add] /SetValue /Fahrenheit Send
} def
```

The **ConvertValue** method lets you modify the string that is displayed in the slider. The **sprintf**[1] operator is used to convert the integer value to a string and append a **C** for Celsius. The action method sets the value of the **Fahrenheit** slider to **C*1.8+32**.

The script of the **Fahrenheit** slider is very similar. It sets the value of the **Celsius** slider to **(F-32)/1.8**:

```
/ConvertValue {% val -- str
   (%F) sprintf
} def

/Action {% val --
   [exch 32 sub 1.8 div] /SetValue /Celsius Send
} def
```

You can now modify one slider, and the other slider will be automatically updated to the corresponding value.

Let's make a small modification to the **SetValue** method so that the updating of the sliders happens immediately instead of when you release the mouse. Add this definition of **SetValue** to both scripts.

```
/SetValue {% val --
   round dup Value ne {
       /SetValue super send
       [Value] /Action Self Send
   } {pop} ifelse
} def
```

The **SetValue** method is called continuously when you are changing the value of the slider. It now sends an **Action** method to itself to update the other slider every time its own value changes.

1. See the NeWS programmer's manual.

Let's make one last change to **SetValue** method of the sliders, to make them change color when the temperature is below freezing. Change the SetValue method in the script of the **Celsius** slider to:

```
/Freezing  0 def
/Red       1 .5 .5 rgbcolor def
/Blue      .5 .6 1 rgbcolor def

/SetValue {% val --
   round dup Value ne {
       Value Freezing gt {Red} {Blue} ifelse
       /ValueColor exch def
       [Value] /Action Self Send
   } {pop} ifelse
} def
```

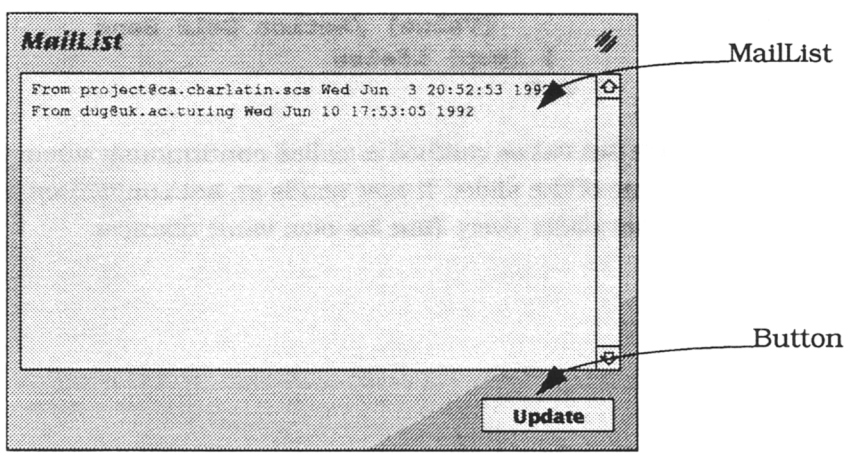
Change the script of the **Fahrenheit** slider accordingly (you need to change the **Freezing** variable to **32** instead of **0**).

Listing Mail Messages

In this more advanced example, a button is programmed to read the output produced by the Unix program called **from**. This standard Unix program lists the messages in your mail box.

Figure 172

The MailList example stack.



For this example you need to create a stack containing a button labelled **Update** and a list object called **MailList**.

Now give the **Update** button the following (rather horrible) script:

```
/Action {
   [(from) pipe pop {
       dup 256 string readline {exch} {
           pop pop exit
       } ifelse
   } loop]
   [exch] /SetValue /MailList Send
} def
```

The script uses the **pipe** operator to fork a **from** process. It then proceeds by reading one line of output until the end of file is reached. The resulting strings are left on the stack to form an array of messages which is then assigned to the **MailList** object using the **SetValue** method.

If you press the **Update** button the list of incoming mail messages is updated from the output of the **from** program.

Animated Slider

In this advanced example the **OnOpen** and **OnClose** methods are used to start and stop a timer which is called at regular intervals to increment the value of a slider. Create a bar slider and give it the following script:

```
/Timer null def
/Delay ScrollThresh def
/OnOpen {      % --
   /Timer {
       Value 1 add MaxValue mod SetValue
   } Delay StartTimer def
} def
/OnClose {     % --
   Timer StopTimer
   /Timer null def
} def
```

The **ScrollThresh** variable used to initialize the **Delay** variable. It is done this way because the time format varies between versions of OpenWindows.

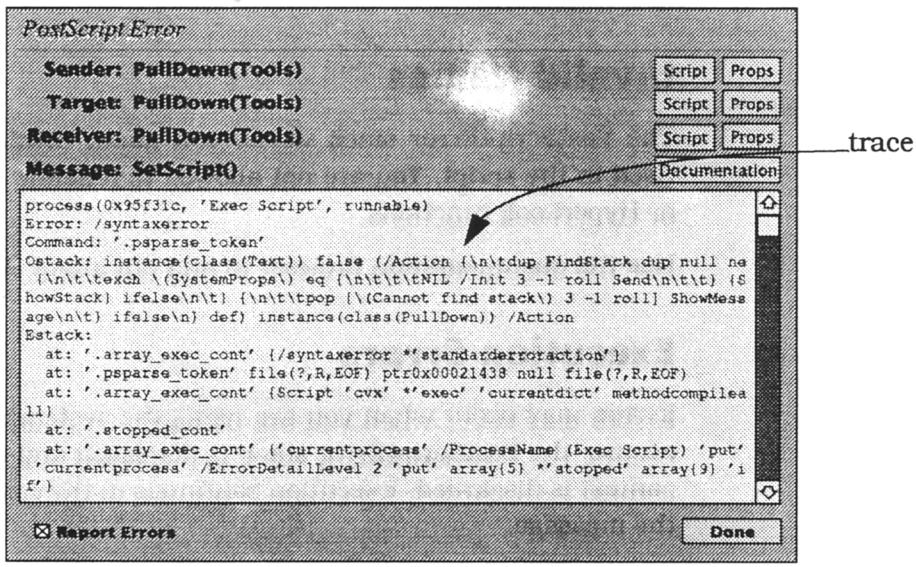
In the OnOpen method, a timer is started using **`StartTimer`**, which increments the value of the slider periodically. The **`OnClose`** method stops the timer.

Dealing with Errors

When the script is applied to the object, it is evaluated by the PostScript interpreter. If an error occurs when the script is applied, the PostScript error stack is shown (see also "Report Errors" on page 3-33).

Figure 173

The PostScriptError stack.



Pay attention to the trace when an error occurs[1]. It tells you the name of error and the name of the operator that caused the error. The **Receiver** field indicates in which context the error occurred.

Before you can apply the script correctly, the error must be fixed. So press **Done** on the PostScript error stack and edit the script until the problem is solved.

The properties or the script of either the **Sender**, the **Target** or the **Receiver** of the message can be shown by pressing the corresponding **Props** or **Script** button.

1. See the NeWS Programmer's Manual for more information.

Syntax Errors

A common error, when writing scripts, is to leave "{" and "}" or "(" and ")" brackets unbalanced, this results in a **syntaxerror**. When this error occurs, press **Done**. Fix the unbalanced brackets in the script before applying it again.

Invalid Names

The PostScriptError stack warns you about invalid names which are used in the script. You are not allowed to redefine PostScript operators or HyperLook functions.

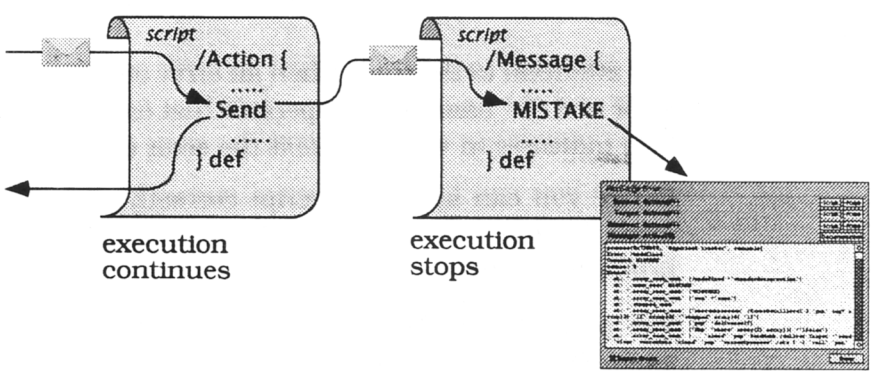
Use the documentation browser to find out about HyperLook functions.

Execution Errors

Errors may occur when you are using the system. As soon as an error occurs, the PostScriptError stack is displayed and the current message context is discarded. Execution continues in the context of the sender of the message.

Figure 174

The sender of a message is not affected when an error occurs at the receiving end.



The PostScript execution trace can help you find the cause of the problem. Only the most recent error is displayed if multiple errors occur. Switch the **Report Errors** check box off to stop errors from being displayed in the PostScriptError stack.

Unix Scripting Utilities

HyperLook provides some Unix utilities which give you control over objects from the Unix shell.

The HyperLook PostScript Shell

The **hlpsh** program (see **hlpsh(1)** on page C-6) lets you execute PostScript commands in the HyperLook environment. For example, type the following in a Unix shell:

```
hlpsh -i
Welcome to HyperLook
(Hello World!) ShowMessage
^D
```

This uses the **ShowMessage** function to display a string in the message window.

You can use the **hlpsh** program to create new HyperLook Unix commands. The following **sh** program iconifies all HyperLook stacks:

```
#!/bin/sh
hlpsh << EOF
StackDict {
   [] /IconifyStack 3 -1 roll Send
   pop
} forall
EOF
```

Sending Messages from the Shell

The **hlsend** program (see **hlsend(1)** on page C-7) is used to send message from a Unix shell to HyperLook objects. The format of the command is:

hlsend *object message args....*

Client Programming

8

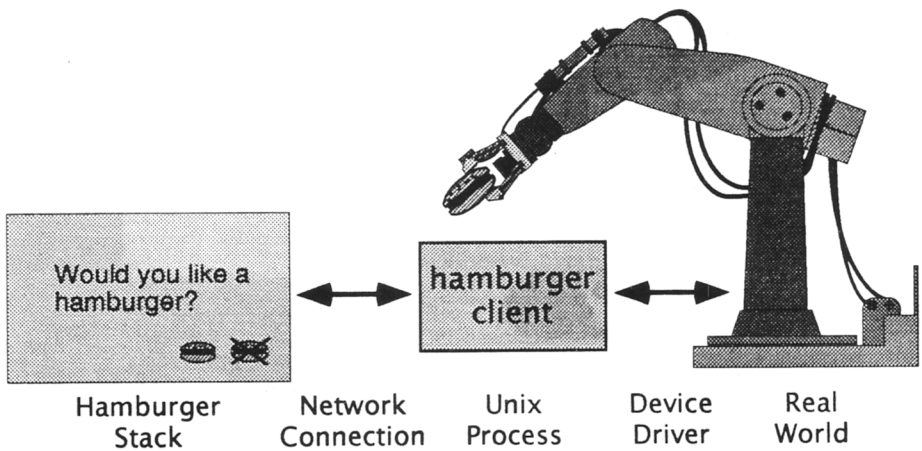
The HyperLook Client Interface

HyperLook objects live in the OpenWindows server. The HyperLook client interface allows Unix processes (called "clients") to connect to OpenWindows via the network, and communicate with HyperLook objects.

Clients can send messages to HyperLook stacks and objects. A client can also receive and respond to messages received from HyperLook. The way messages are passed to the user is discussed in "Messages" on page 7-12.

Figure 175

Interfacing HyperLook to the real world.



High Level Message Passing

HyperLook clients communicate at a high level by passing messages and arguments back and forth to HyperLook. This insulates your application from the look and feel, so you can easily reconfigure the user interface of your application without modifying the client, even while it's running.

Clients need not worry about details such as layout, drawing, and mouse tracking. They need only know about the high level messages which are relevant to the task at hand.

Because of this clean separation between HyperLook and the application, you can create a highly interactive graphical user interface before writing a line of code. The user interface will always be easy to change later on. In fact, the same client can run with any number of different user interfaces.

Client interface programming is very easy. There are only a few routines that you need to know about. Making an application respond to or send a message only requires a few lines of code.

Appendix D, "Library functions" contains a complete list of all client interface library routines.

Client Interface Overview

Below is the skeleton of a HyperLook client program[1]. It is explained in more detail in the rest of this section.

```
#include <hyperlook.h>

int handle_message(msg, argc, argv)
hl_any *msg, *argv[];
int argc;
{
    hl_send0("StackName:ObjectName","Message",NULL);
    return 1;
}

main()
{
    if (!hl_start("ClientName")) {
        fprintf(stderr,"Cannot connect to HyperLook!\n");
        exit(1);
    }
    hl_path(NULL);
    hl_connect("StackName");
    hl_show("StackName");
    hl_register(handle_message,
            "StackName:ObjectName","Message");
    hl_listen(hl_forever);
    hl_stop();
}
```

Initializing the Client Interface

Before communicating with HyperLook, a client must connect to the OpenWindows server and initialize the interface by calling the function `hl_start()`.

1. The source to this example can be found in "demos/client/skeleton.c" in the HyperLook directory.

The function **hl_path()** is called to tell HyperLook where to find the application's resources (see "The Resource Manager" on page 3-23). Passing **NULL** means that the resources are found in the directory where the client was started.

Before the client exits, it calls **hl_stop()** to clean up and disconnect from HyperLook.

Connecting to Stacks

To receive messages from a HyperLook stack, the client must be connected to it. A stack is connected to a client using **hl_connect()**. The **hl_show()** function displays the stack *and* connects it to the client.

Any number of stacks can be connected to one client, but each stack can have at most one client.

Receiving Messages from HyperLook

In order to receive messages from HyperLook, a client must register message patterns and handlers with **hl_register()**. A handler is called when a message matching a requested pattern is sent to the client.

In order to process messages, you call **hl_listen()**. Any messages that can't be handled by stacks to which you're connected are sent from HyperLook to the client, and are matched against its registered message patterns. If any patterns match, the appropriate handler functions are called with the message arguments, otherwise unmatched messages are ignored.

When a message is received, the handlers for the message are called in the same order as they were registered. Only the handlers that match the message and object name are called. Each handler has a chance to examine the message and arguments and decide whether or not to handle it.

If a handler returns `0` the next matching handler is called. If a handler returns `1`, then no other handlers are called. If a handler returns `-1`, no other handlers are called and **hl_listen()** returns.

Sending Messages to HyperLook

Clients can send messages to HyperLook objects by calling **hl_send0()**. This function takes a name identifying the target object, a message name, and any number of message arguments (terminated by **NULL**).

The message is sent from the client to HyperLook, and it is delivered to the appropriate object using HyperLook message passing (See "Messages" on page 7-12).

The message arguments passed to the C handler functions and **hl_send0()** are **hl_any** structures. These structures represent PostScript data structures such as names, strings, numbers and arrays. The HyperLook client interface provides a number of functions to create these data structures in C.

PostScript Data Structures in C

The **hl_any** data structure is declared in **hyperlook.h** as follows (see **hl_any(3)** on page D-3 for a more detailed description)[1]:

```
typedef enum {
    hl_number_type,
    hl_boolean_type,
    hl_string_type,
    hl_name_type,
    hl_array_type,
    hl_null_type,
} hl_any_type;

typedef struct hl_any {
    hl_any_type type;
    short len;
    union {
        double number;
        int boolean;
        char *string;
        char *name;
        struct {
            int argc;
            struct hl_any **argv;
        } array;
    } u;
} hl_any;
```

This data structure is used to represent the PostScript messages and arguments that are received from HyperLook. The same data structure is used to construct arguments to messages which are sent back to HyperLook.

There are a number of convenient library routines that let you create **hl_any** data structures. Using these routines you can create the same data structures that you can in PostScript. This lets you use PostScript methods (described Chapter 7, "Scripting") from C.

1. The complete include file is found in "include/hyperlook.h" in the HyperLook directory.

The Hamburger Client

This section gives a complete example of how to program the HyperLook client interface. It explains the client interface in detail[1].

The application is *very* simple. It is a program to ask whether you would like a hamburger. The application exits when you accept the offer.

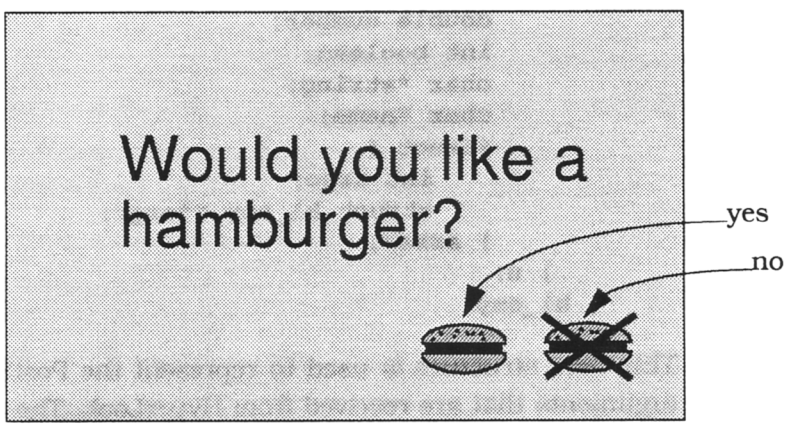
Creating the Hamburger Stack

A client needs to connect to a stack to receive messages, so the first thing to do is to build a stack (see Chapter 5, "Editing Stacks").

Figure 176 show a picture of the stack called **Hamburger**. It has a text object asking the question, and two drawing buttons. The left button is named **yes** and the right is named **no**.

Figure 176

The Hamburger stack.



1. The source to this example can be found in "demo/client/hamburger.c" in the HyperLook directory.

The Hamburger Code

Here is the complete listing of the Hamburger client program.

```
#include <hyperlook.h>

int no()
{
    hl_send0("Message", "Init",
        hl_new_string("Please reconsider my offer!"),NULL);
    return 1;
}

int yes()
{
    hl_send0("Message", "Init",
        hl_new_string("Have a nice day!"),NULL);
    return -1;
}

main(argc, argv)
int argc;
char *argv[];
{
    hl_verbose = 1;

    if (!hl_start(argv[0])) {
        fprintf(stderr,"Cannot connect to HyperLook!\n");
        exit(1);
    }
    hl_path(NULL);
    hl_connect("Hamburger");
    hl_show("Hamburger");

    hl_register(no,"Hamburger:no","Action");
    hl_register(yes,"Hamburger:yes","Action");
    hl_listen(hl_forever);

    hl_hide("Hamburger");
    hl_stop();
}
```

Called when the **no** button is pressed. It shows a message window.

Called when the **yes** button is pressed. It returns **-1** so **hl_listen** will terminate.

Set **hl_verbose** so that messages are printed on **stderr**.

Connect to HyperLook.

The **Hamburger** stack is loaded from the current directory.

Process messages until **yes()** returns **-1**, then clean up.

The Hamburger Code Explained

The first thing that the client must do is to initialize of the interface. It calls **hl_start()** and passes it the name of the client. This client will be known by that name to HyperLook.

It then adds the current directory to the client resource path, by calling **hl_path()**. HyperLook can now find the **Hamburger** stack which is located in the same directory as the program.

Next, it connects to the stack, and shows it. It is not necessary to call **hl_connect()** because **hl_show()** also connects to the stack. Now the **Hamburger** stack is displayed.

Before calling **hl_listen()** to process incoming message, the client registers two message handlers, **yes()** and **no()**, which will be called when the corresponding buttons are pressed. Finally it calls **hl_listen()** to process incoming messages.

When you click one of the buttons, a message named **Action** is sent to the button. In this example, neither the buttons, the card, the background, nor the stack have **Action** handlers. So the **Action** message is passed on to the client, and the corresponding message handler is called.

The handler sends a message to the HyperLook **Message** stack, telling it to display a string to the user. When the user finally presses the **yes** button, the **yes()** message handler is called. This handler returns **-1** which causes **hl_listen()** to terminate.

Finally, the client calls **hl_hide()** to remove the stack from the screen, followed by **hl_stop()** to clean up and break the connection with HyperLook.

HyperLook Include File

```
#include <hyperlook.h>
```

In order to use the HyperLook client interface, you must include the "hyperlook.h" include file. You'll find it in the "include" sub directory of the HyperLook directory.

Debugging

```
hl_verbose = 1;
```

Setting the **hl_verbose** flag causes diagnostics to be written to the client's **stderr** file. The diagnostics include a log of all messages sent to the client. A typical message looks like this:

```
hl_dispatch: message(Hamburger:Button(no),Action,[0])
```

The above means that an **Action** message is received with a single argument, the number zero. The target of the message is a **Button** object named **no** on the **Hamburger** stack.

Initialization

```
if (!hl_start(argv[0])) {
    fprintf(stderr,"Cannot connect to HyperLook!\n");
    exit(1);
}
```

This initializes the client interface, connects to OpenWindows server, and loads HyperLook if necessary. The argument is the name of the program, which identifies the client to HyperLook. If **hl_start()** returns 0 it wasn't possible to connect with HyperLook.

```
hl_path(NULL);
```

HyperLook needs to know where to find the client's resources (the Hamburger stack in this case), so **hl_path()** is called with an argument of **NULL**, to add the current directory to the client resource path. The argument to **hl_path()** can also be a string with an absolute directory name.

```
hl_connect("Hamburger");
```

This connects the client to the Hamburger stack, loading it from the resource directory if necessary. Once a client is connected to a stack, any messages the stack can't handle are sent to that client.

```
hl_show("Hamburger");
```

This shows the Hamburger stack on the screen. It also implicitly connects the client to the stack, if it's not already connected[1].

Registration

```
hl_register(no,"Hamburger:no","Action");
hl_register(yes,"Hamburger:yes","Action");
```

The above code registers the **no()** and **yes()** message handlers, so they're called whenever an **Action** message is received from the **yes** or **no** buttons on the **Hamburger** stack.

The message address is specified as "**stackname:objectname**". In this example the stack is called Hamburger and the buttons are called **no** and **yes**.

Notification

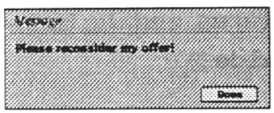
```
hl_listen(hl_forever);
```

Calling **hl_listen()** enters the HyperLook notifier, which processes incoming messages and calls the appropriate handlers. **hl_listen()** exits when a handler returns **-1**.

The argument specifies how long the notifier should wait before calling the timeout handler. In this case we aren't interested in a timeout, so we pass **hl_forever**, so the timeout handler[2] is never called.

1. Strictly speaking, it is not necessary to call **hl_connect** in this example.
2. Use the **hl_register_timeout()** function to register a timeout handler.

Message Handlers

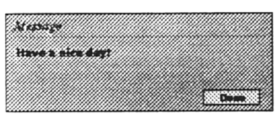



```
int no()
{
   hl_send0("Message", "Init",
    hl_new_string("Please reconsider my offer!"),NULL);
   return 1;
}
```

This message handler is called in response to an **Action** message, when the **no** button (with the picture of the crossed out hamburger) is pressed. It doesn't care about the message arguments, so they're not declared. It sends an **Init** message to the HyperLook **Message** stack (part of the HyperLook system), with one argument: a string to display to the user.

The message arguments passed to **hl_send0()** are of type **hl_any** (which is what **hl_new_string()** returns). A small message window is shown displaying the string. The **Message** stack is a facility provided by HyperLook to display messages.

The C message handlers return an integer to let **hl_listen()** know whether the message was handled. The **no** message handler returns **1**, which means that it was able to deal with the message and that no other handlers should be considered. Had it returned **0**, other matching handlers would have been given a chance.




```
int yes()
{
   hl_send0("Message", "Init",
       hl_new_string("Have a nice day!"),NULL);
   return -1;
}
```

The **yes** handler is similar to the **no** handler, except for its return value, which is **-1**. This causes **hl_listen()** to stop processing messages and return to its caller.

Cleanup

```
hl_hide("Hamburger");
```

Once the user presses the **yes** button, **hl_listen** returns, and the Hamburger stack is removed from the screen using **hl_hide()**.

```
hl_stop();
```

Calling **hl_stop** disconnects the client from the OpenWindows server and terminates the HyperLook connection.

Compiling the Hamburger

The source to this example is found in "**demo/client/hamburger.c**". You can compile the program (if you have write permission in this directory) by typing:

```
make hamburger
```

You can run the program, once the program has been successfully compiled, by typing:

```
hamburger
```

The program is compiled as follows. First **hamburger.c** is compiled to **hamburger.o**:

```
cc -c hamburger.c -I$HLHOME/include \
       -I$OPENWINHOME/include/wire \
       -I$OPENWINHOME/include/NeWS \
       -I$OPENWINHOME/include
```

The **-I** flags are used to specify the directories where the C preprocessor should find the include files. Then the program is linked with the following command:

```
cc -o hamburger hamburger.o -L$HLHOME/lib -lhl
```

This creates a binary called **hamburger**. The **-L** and **-l** flags are needed to link the HyperLook client interface library into the program.

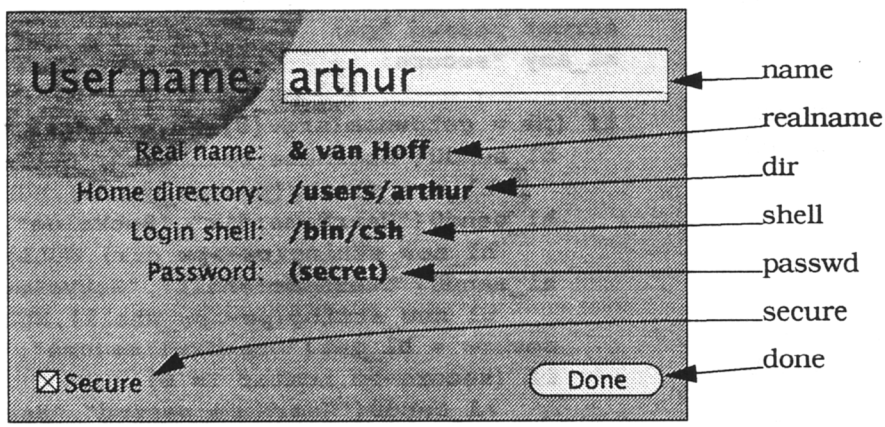
The UserName Client

This section discusses a more sophisticated example. The program provides a simple interface which lets you access information from the Unix password database[1].

When you enter a user name followed by return, the information about the user is displayed in the stack. The **getpwnam(3V)** Unix function call is used to access the information from the Unix password database.

Figure 177

The username stack.



The password is shown encrypted. If the **Secure** checkbox is marked, it should show the string "*(secret)*" instead of the encrypted password.

The program exits when you press the **Done** button.

The stack is called **UserName**, and it contains several fields and a button. The **name** field at the top right is the only editable one. The other fields are called **realname**, **dir**, **shell**, and **passwd**. The checkbox is called **secure**. The button is called **done**.

1. The source to this example can be found in "demo/client/username.c" in the HyperLook directory.

The UserName Code

The complete username program is listed below.

```
#include <pwd.h>
#include <hyperlook.h>

int request(msg,argc,argv)
hl_any *msg, *argv[];
int argc;
{
    struct passwd *pw;
    hl_any *secure;

    if (pw = getpwnam(argv[0]->u.string)) {
        hl_send0("UserName:realname","SetValue",
            hl_new_string(pw->pw_gecos),NULL);
        hl_send0("UserName:dir","SetValue",
            hl_new_string(pw->pw_dir),NULL);
        hl_send0("UserName:shell","SetValue",
            hl_new_string(pw->pw_shell),NULL);
        secure = hl_get("UserName:secure","Value");
        if (secure->u.number != 0)
            hl_send0("UserName:passwd","SetValue",
                hl_new_string("(secret)"),NULL);
        else
            hl_send0("UserName:passwd","SetValue",
                hl_new_string(pw->pw_passwd),NULL);
        hl_free(secure);
    } else {
        hl_send0("UserName:realname","SetValue",
            hl_new_string("-- not found --"),NULL);
        hl_send0("UserName:dir","SetValue",
            hl_new_string(""),NULL);
        hl_send0("UserName:shell","SetValue",
            hl_new_string(""),NULL);
        hl_send0("UserName:passwd","SetValue",
            hl_new_string(""),NULL);
    }
    hl_send0("UserName:name","SelectAll",NULL);
    return 1;
}
```

This function is called when the user has entered a name.

Call **getpwnam** to get the user info.

Show the password if it can be disclosed.

Clear the fields if the user is not found.

Select the name that the user typed.

This function is called when the **Done** button is pressed.

```
int done()
{
    return -1;
}

main(argc,argv)
int argc;
char *argv[];
{
    hl_verbose = 1;

    if (!hl_start(argv[0])) {
        fprintf(stderr,"Cannot connect to HyperLook!\n");
        exit(1);
    }

    hl_path(NULL);
    hl_show("UserName");

    hl_register(request,ANY,"Action");
    hl_register(done,"UserName:done","Action");
    hl_listen(hl_forever);

    hl_hide("UserName");
    hl_stop();
}
```

Connect to HyperLook.

Show the stack.

Register message handlers.

The UserName Code Explained

The structure of the program is very similar to the previous example. The client shows the UserName stack (automatically connecting to it), and registers two message handlers.

The **request()** handler is called when a **Action** message is received from the **name** field. The handler uses the string argument of the message and looks up the user name in the password database.

If the name is found, **SetValue** messages are sent to the data fields of the **UserName** stack.

Before the **passwd** field is set, the value of the **Secure** checkbox is examined to determine whether the encrypted password should be displayed. The program uses the **hl_get()** routine to get the value of the checkbox.

The program exists when an **Action** message is received from the **done** button.

Message Handler Arguments

When **hl_listen()** calls a handler, it passes three arguments to the handler.

```
int request(msg,argc,argv)
hl_any *msg, *argv[];
int argc;
{
   /* message handler */
}
```

The first argument, **msg**, is an **hl_any** data structure which contains the complete message that **hl_listen()** received from HyperLook. The **argv** argument is an array of length **argc**, containing pointers to **hl_any** data structures, which are the arguments to the message.

The **request()** handler is called in response to an **Action** message from a **field** object. It has one argument, the string that the user typed. The string data can be accessed as follows:

```
argv[0]->u.string
```

Compiling the UserName client

The source to this example is found in **demo/client/username.c**. You can compile the program (if you have write permission in this directory) by typing:

```
make username
```

You can run the program, once the program has been successfully compiled, by typing:

```
username
```


A

Glossary

Background

A place holder for components on a stack. Every card has at least one BackGround. A BackGround can be shared by several cards. Therefore objects on a background can be visible on more than one card. The parent of a BackGround is the stack.

Button

A user interface component that performs an action when it is pressed. Buttons can have different styles, for example: push button, checkbox, drawing button, etc.

Card

A card is like a page in a book. Every stack contains at least one card. Only one card of a stack can be visible at any time. You can flip between cards like turning the pages of a book. The parent of a card is a BackGround.

Checkbox

A checkbox is a user interface component that can either be *on* or *off*. It is on when it is crossed. It is just another style of button.

Class

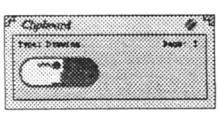
A class contains the code describing an object type. Classes inherit functionality from their super classes.

Client

A client is a Unix process that is connected to HyperLook. It can communicate with stacks and objects by sending and receiving messages.

Client Interface

A set of C routines which let you write client programs that communicate with HyperLook.



Clipboard

A place holder for data when it is moved between stacks and applications. It can contain graphics, text, images, etc.

Color: 

ColorSelect

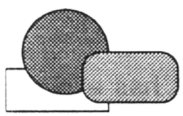
A user interface component that lets you select a color from a color pallet.

Component

A user interface object such as a button, field or slider.

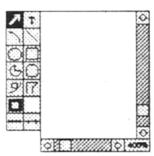
Desktop

The OpenWindows user interface metaphor for the screen. It contains windows and icons to represent files and applications.



Drawing

A HyperLook PostScript graphics object that can be edited using the HyperLook graphics editor. It can be translated to Encapsulated PostScript format.

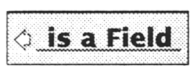


DrawTool

A user interface component that lets you edit graphics interactively. It is HyperLook graphics editor in a single object.

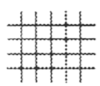
Encapsulated PostScript

A standard format which is used when PostScript files are transported between applications. Most desktop publishing and graphics editing applications know how to import and export this format.



Field

A user interface component that lets you edit a single line of text.

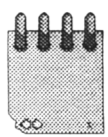


Grid

A grid defines the accuracy at which editing operations are performed.

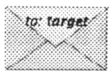
Group

A group is a set of graphical objects which can be manipulated as if they are a single object.



Icon

A small graphical representation of a stack or an application.



Message

Messages are passed between HyperLook objects and clients as a means of communication.

Method

A method is a function which is defined in a class or script. The function is executed in the context of the object.

NeWS

Network extensible Windowing System. This is the name for the PostScript interpreter which is part of OpenWindows.

Object Oriented

A way to structure programs and data. Each object has a class which defines its functionality. Classes are organized in a hierarchical structure that allows for inheritance of functionality.

OPENLOOK®

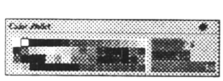
A user interface "look and feel" using lots of bevelled edges[1]. This is the default "look and feel" of OpenWindows.

OpenWindows

The environment in which HyperLook runs. It manages the windows and icons which you normally see on the screen.

Operator

A PostScript function.



Pallet

A set of things from which you can choose. For example, a color pallet.

Polygon

A polygon is a graphical object made of straight lines only.

PostScript

An interpreted language that was especially designed for graphics.

1. OPENLOOK is a registered trademark of Unix systems Laboratories Incorporated.

PullDown

A button with a menu associated to it. When the button is pressed, the menu is shown and you can select an option.

Resource

A file used by HyperLook. Resources are accessed using the resource manager.

Script

A script contains code that customizes an object. Each HyperLook object can have a script. Scripts contain method and variable definitions that can redefine and extend the default behavior of objects.

Scrollbar

A scrollbar is a user interface component that is used to view data. Its value reflects which data is visible.

Slider

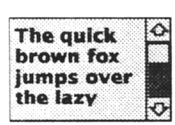
A slider is a user interface component that lets you select a numeric value from a range.

Spline

A spline is a smooth polygon.

Stack

A HyperLook window. It contains components, cards and backgrounds.

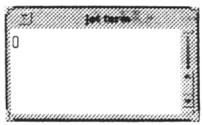


Text

A text object is a component that lets you edit multiple lines of text. It is a simple text editor.

Unix

The operating system which runs on Sun computers. It is the software that supports all programs on your computer.



Window

An area of the screen usually dedicated to a single application. Windows can be moved and resized. You can turn them into icons if they take up too much screen space.

Zoom

The magnification at which graphics are displayed. Zooming *in* means that the graphics are enlarged. Zooming *out* means that they get smaller.

B

Methods and Variables

HyperLook Methods and Variables

This Appendix briefly mentions the most important functions, methods and variables which are used by HyperLook. The list is by no means complete. Use the documentation browser ("The Documentation Browser" on page 7-10) to examine the full list of documented methods and variables.

These methods are defined in the HyperLook class and are always available.

```
/DEBUG              % anything --
/ShowMessage        % string|array-of-strings --

/ShowStack          % stack|stackname --
/HideStack          % stack|stackname--

/FindStack          % stackname -- stack|null
/FindObject         % stackname objectname -- object|null
/LookupObject       % name -- object|stack|null

/Send               % array-of-arg message target --
/ParentSend         % array-of-arg message --
/ClientSend         % array-of-arg message --
```

Stack Methods

These methods can only be used in the script of a stack, or sent to a stack.

```
/Move              % x y --
/ToFront           % --
/ToBack            % --
/IconifyStack      % --
/DeIconifyStack    % --
/FlipIconifyStack  % --

/GoHomeCard        % --
/GoLastCard        % --
/GoNextCard        % --
/GoPreviousCard    % --
/GotoCard          % number|name --
```

Card Object Methods and Variables

These variables can be referenced in the script of every card object, e.g. Buttons, Sliders etc:

```
/Value          % -- value
/Width          % -- number
/Height         % -- number
/Color          % -- color
/FillColor      % -- color
/FontName       % -- name
/FontSize       % -- number

/MyStack        % -- stack
/MyClient       % -- file
/MyCanvas       % -- canvas
```

These methods can be used in the script of every card object, e.g. Buttons, Sliders etc:

```
/Draw           % --
/Damage         % --

/Move           % x y --
/Reshape        % x y width height --
/Show           % --
/Hide           % --

/OnAscii        % char --
/OnMouse        % --
/OnHelp         % --
/OnOpen         % --
/OnClose        % --
```

Button Methods

```
/Action         % number --
/SetValue       % number --
/Path           % --
/Press          % --
```


Field Methods

```
/Action       % string --
/SetValue     % string --
/Selection    % -- string
/SelectAll    % --
```

Text Methods

```
/Action       % string --
/WordAction   % string --
/SetValue     % string --
/Selection    % -- string
/SelectAll    % --
```

List Methods

```
/Action       % array-of-strings --
/SetValue     % array-of-strings --
/Select       % array-of-strings --
/Selected     % -- array-of-strings
```

PullDown Methods

```
/Action       % string --
/SetValue     % string|array-of-strings --
```

Slider Methods

```
/Action       % number --
/SetValue     % number --
/SetRange     % min val max --
/ConvertValue % number -- string
```

DrawTool Methods

```
/Action       % drawing --
/SetValue     % drawing --
/GetValue     % -- drawing
```