
# Iptscrae Language Guide

**Version 1**

*Iptscrae Language Guide*, version 1

February, 2000



Printed in the USA.

# Preface

This manual describes how use to the Palace Iptscrae Language. ***Iptscrae*** (pig-latin for “script”) is a Forth-like interpreted language used exclusively by the Palace software. It possesses a full compliment of commonly-needed operators and functions, as well as over 100 Palace-specific commands, keywords and functions.

## Who should read this manual

This manuals is designed for all users, owners, and server operators that want to implement the advanced features the Iptscrae language provides. This document assumes you have read the section called *Your Cyborg.ipt File* in *The Palace Users Guide.*

In addition, you will want to study the default script files that come with the Palace software. There are two "script files" in the Palace system: the `Cyborg.ipt` (also called the ***cyborg script*** or ***user script***) which handles user-based actions, and a ***server script*** aka ***room script*** (also referred to as the ***Mansion Script*** due to its default contents) that defines your Palace site and handles room-based actions. The term ***server script*** will be used throughout this document to refer to this second file.

## Sections of this manual

This manual has the following sections:

- *An Introduction to Iptscrae*

  Introduces Iptscrae, and describes how to run it.

- *Iptscrae language Reference*

  Describes the Iptscrae language, including data elements and commands..

In addition, a number of appendices are included:

- *Quick Reference*

  A table of Iptscrae commands with brief summaries

- *Appendix A: Adding Machine exercise*

  Provides a sample Adding Machine script, describing step by step how to create this Iptscrae program

- *Appendix B: Code standards*

Describes Iptscrae data handling boundaries.

- *Appendix C: Palace Client plugin API*

  Describes how you can become a Palace development partner, creating your own plug-in to the Palace.

- *Index*

# Manual conventions

This manual has the following conventions:

- Technical terms appear in ***boldface italic*** in the text in their first appearance, usually accompanied by a definition.
- Graphical User Interface (GUI) elements, such as menu items and buttons, appear in **boldface**.
- References to other books, chapters, or sections are in *Italic*.

---

NOTE – Notes to the user look like this.

---

All code is printed in `Courier` font; commands and keywords are printed in `UPPERCASE`. This is not necessary as far as the script's operation is concerned - you can write commands in lowercase letters and use whatever font you want - but capitalization helps the code stand out from the data and makes it easier to read scripts written by others. For example:

```
"Welcome to " SERVERNAME & SAY
```

Placeholders for actual values are written in *`italics`*.

```
targetsName WHOPOS
selectedSpot 2 SETSPOTSTATE
```

Placeholders for string values always appear in double quotes (`" "`). These double quotes must remain in the code; do not remove them. For example:

```
"filename" MIDIPLAY
"This " "and " & "that." & tempString =
```

Symbols (variable names) begin with lowercase letters, although they may possess median capitals or underscores. Again; this is not necessary, but it makes your variables a little easier to find. For example:

```
myNewVariable
another_variable
```

Optional arguments (those that may be included or left out of the command) are enclosed in anglebrackets ( < > ). For example:

```
'banip <minutes> ip
```

Note that this goes against traditional "IBM-style" programming notation, which uses square brackets ( [ ] ) to indicate optional arguments. This is because Iptscrae uses square brackets to designate arrays.

Indentation indicates supporting code. All text supporting any ROOM, DOOR, SPOT, SCRIPT or ON *EVENT* keyword is indented, and the indent is decreased when the supporting code ends (i.e., after any ENDROOM, ENDDOOR, ENDSPOT or ENDSCRIPT keyword, and on the closing curly bracket of an ***atomlist*** or ***event handler***). This lets you tell at a glance where the code for each section starts and stops. For example:

```
ROOM
   ID 100
   NAME "Room Name"
   PICT "picture.gif"
   DOOR
      ID 1
      DEST 200
      OUTLINE 10,10 50,10 50,200 10,200
   ENDDOOR
   SPOT
      ID 2
      OUTLINE 200,50 400,50 400,200 200,200
      SCRIPT
         ON ENTER {
            "Hello!" SAY
         }
      ENDSCRIPT
   ENDSPOT
ENDROOM
```

## Other sources of information

Besides this *Language Guide* , you can use many other related manuals for administering your personal server, becoming an operator, and running The Palace client software. These manuals are all available from The Palace website section.

- *Server Guides*. These manuals tell you how to obtain, install, setup, and run your own Palace server. Macintosh, Unix, and Windows 95/98/NT versions are available.
- *Operator Guide*. Describes the responsibilities of an operator, and details the client's operator interface and commands.
- *User's Guide*. This manual describes how to use obtain, setup, and use the Palace client software. You will need the Palace client to perform owner/operator actions for your site. Both a Windows and Macintosh version are available.

In addition to online manuals, check out The Palace website for support, information, and online documentation at www.thepalace.com.

# An Introduction to Iptscrae

This section introduces Iptscrae:



## What is Iptscrae?

***Iptscrae*** is a Forth-like interpreted language used exclusively by the Palace software. It possesses a full compliment of commonly-needed operators and functions, as well as over 100 Palace-specific commands, keywords and functions.

Here is a sample Iptscrae script:

```
ON OUTCHAT {
   OCHAT GLOBAL
   CHATSTR OCHAT =
   {
      "coffee" "$1" GREPSUB WHOPOS ADDLOOSEPROP
   } CHATSTR "^buy (.*) coffee$" GREPSTR IF
}
```

# Reverse Polish notation

Unlike other scripting languages you may have seen, Iptscrae might appear rigidly formal and possesses almost no punctuation. Iptscrae's syntax also uses RPN ("Reverse Polish Notation"), a style of command structuring that may seem a little "backward" until you get used to it...

RPN uses what is called a ***postfix*** word ordering. In a sense, this is "backwards" from the usual *infix* word ordering that most of us are used to. For example, we are using *infix* word ordering when we say "two plus three." In *postfix* word ordering, it would be "two three plus." This is what the Palace software expects to see.

While saying "two three plus" might seem quite strange to you, to a computer it makes a good deal of sense. After all, the computer can't do anything with the operator until it possesses both operands; it can't perform an operation until it has all the data needed. In Iptscrae, as in many other programming languages, all data is stored on a *stack* until you pop it off and use it for something (see below for more information on how the stack operates). We have two pieces of data here — the number 2 and the number 3 — so in strictly logical terms, when we say "2 3 +", what we're really telling the machine is:

```
Put 2 on the stack.

Put 3 on the stack.

Add the two top things on the stack.

Place the result on the stack.
```

In plain English, you can think of this as placing the verb (action) at the end of the sentence or after the direct object. For example, instead of saying "I walk to the store," you'd say "I store walk."

Here are a few more examples of the differences between ***infix*** and ***postfix*** word ordering...

| Infix: | Postfix: |
|---|---|
| `2 + 3` | `2 3 +` |
| `(2 + 3) * 4` | `2 3 + 4 *` |
| `X = 4 - 3` | `4 3 - X =` |
| `printf("Howdy")` | `"Howdy" Say` |

# Scripting the stack

All Iptscrae functions put the results of some operation onto something called ***the stack.*** If you're not familiar with the use of a stack, it can be a tricky concept to get hold of. Here's one way to think about it:

Imagine a spring-loaded stack of dishes like they have in cafeterias. The spring is set so that only one dish is available at a time. When you pop one off the stack, the rest get pushed up so that a new one becomes available; conversely, if you put a dish on the stack, the ones beneath it get pushed down so that your new one is the only one available.

There you have your stack metaphor, and that's pretty much how it works; except that putting something on the stack is called "pushing" and taking something off the stack is called "popping." In fact, if you ever want to remove the top-most value on the stack, you can always issue a `POP` command, which discards the top-most value and makes the next one available. You can also swap the two top-most items by issuing a `SWAP` command. Pushing occurs automatically. To see this in action, type the following statements into your Palace client input window:

```
/ "one" "two" "three" SAY

/ "one" "two" "three" SWAP SAY

/ "one" "two" "three" POP SAY

/ "one" "two" "three" POP POP SAY
```

So now you can see that when we tell Iptscrae `" 2 3 + "`, the following steps occur:

1. Push 2 onto the stack
2. Push 3 onto the stack
3. Pop the top two items off the stack, add them, and push the result onto the stack

When it's done, the result (5 ) is sitting on the top of the stack, where we can easily get at it with another command.

# How to run Iptscrae

Depending on what you're trying to do, there are several ways to enter Iptscrae commands. "One-shot" commands can be typed directly into the Palace client input box, or more permanent changes can be made to the script by owners and operators in

authoring mode. For those interested in total control, any word processing program that can save as ASCII (text only) can also edit Iptscrae script files. The following sections explain how to do all of these things.

## Entering Iptscrae in the input box (“Slash Commands”)

A good way to get started with Iptscrae (or to test the functionality of individual commands) is to launch The Palace client, enter any room that allows user scripts, and type your commands straight into your client input window. To let the Palace software know that the coming text is a command (as opposed to mere speech), you must begin the line with a forward slash ( / ).

For example, if I type:

```
10 10 SETPOS
```

everyone in the room will see me say “`10 10 SETPOS`”. Not what I wanted. But if I type:

```
/ 10 10 SETPOS
```

the `SETPOS` command is executed instead, and my avatar moves to position `10 , 10` on the screen.

Like any complex system, the best way to get a feel for Iptscrae is to just jump in and start playing with it. The following examples will give you a good start. As you can see from the inclusion of the forward slash, these lines were intended to be typed into the Input Box in the Palace client. To paste them into scripts, remove the preceding slash.

```
/ "Hello" SAY
```

This says "Hello".

```
/ "My name is " USERNAME & SAY
```

This shows how to add two pieces of text together. `USERNAME` is a keyword that resolves to your name. The ampersand ( & ) is the concatenator; this operator takes the top two things off the stack and sticks them together into one big string variable, which it pushes back onto the stack. The `SAY` command takes this variable and sticks it in a cartoon balloon.

```
/ 2 2 + ITOA SAY
```

This one adds 2 and 2 together, converts the result to ASCII text (using the `ITOA` function), and displays the results in a cartoon balloon.

```
/ 2 3 * 4 + ITOA SAY
```

This one multiplies 2 and 3 together, then adds 4, then converts the answer from integer to ASCII and says the result.

```
/ "Hello" 10 40 SAYAT
```

This one says hello at the specified X (horizontal) Y (vertical) position on the screen. X cannot exceed 511. Y cannot exceed 383. This is because the screen dimensions of the Viewing Area are 512 by 384.

```
/ 100 RANDOM ITOA SAY
```

This one causes you to say a random number. Try it a few times; each time it will be different.

```
/ "Hello" 512 RANDOM 384 RANDOM SAYAT
```

This one says hello at a random position on the screen; try it a few times. `512 RANDOM` determines a random number from 0 to 511. `384 RANDOM` determines a random number from 0 to 383.

```
/ 10 10 SETPOS
```

This one moves your avatar to position 10, 10 on the screen. Try substituting different numbers.

```
/ 512 RANDOM 384 RANDOM SETPOS
```

This one moves you to a random position. Try it a few times.

By playing around with these simple examples you can get a feel for the flow of the language. Once you understand what these examples are doing, you can move on to creating your own, more complex, scripts.

## Iptscrae in Authoring Mode

When in ***authoring mode*** (select **Authoring Mode** from your client's **Operator** Menu as described in *The Palace Operators Guide*), you can edit the script of any spot in the current room.

1. Select the spot to be edited.
2. Double-click within its boundaries (alternately, you can select **Door Info** from the **Operator** Menu while the spot is selected). The **Door Info** window will appear.
3. Click **Script** (Macintosh) or **Edit Script** (Windows). You'll see the scripting window (shown in Windows version):


This window is little more than a large scrolling textbox. Any scripts already associated with the selected spot will appear here. By directly editing the text in this window and clicking **OK** (or selecting **Save** for Mac users), you cause the server to update the script file. The new script begins functioning almost immediately (this might take up to a minute on slower systems or under bad network conditions).

---

NOTE – The **Script Edit** window displays only the Iptscrae code, not the Room/Spot description code.

---

To see the scripting window in action, try the following simple example:

1. Create a new room and put a door in it.
2. Double-click on the door to bring up the **Door Info** window.
3. From this window, make the door's Type **Normal**. This turns it into a ***spot***.
4. Click **Script** to pull up the scripting window.
5. Now add a simple ***event handler*** (like this one, for instance…)

```
ON SELECT {
   "I selected the Spot!" SAY
}
```

6. Click **Ok** (or select **Save** for Macintosh) and turn authoring mode off.
7. Now test the script by clicking on your new spot.

## Iptscrae in ASCII: Editing script files

Using the scripting window in the Palace client doesn't provide you with access to everything. The only way to obtain 100% access to the contents of the script is to edit it directly, which can be done using any word processing program. The only stipulation

is that the word processor must be capable of saving as ASCII (“text only”), and fortunately, it’s hard to find one that isn’t. Simply open the script file and edit it as you would any other document; but be sure to make a backup copy first!

Never edit the server script while the server is running. This may cause overwrite errors, which can corrupt the file and/or cause your server to hang. Always save your server and make a backup before modifying your `peserver.pat` file.

When you have finished editing the script file, save it (remember: text only) and launch the server. If your script file is well-structured, you’ll see no error messages as the server sets up your Palace site. If you're a Windows or Macintosh user, the next thing you see will be the server interface itself. Congratulations; your new script has taken effect!

# Anatomy of a script file

Although to beginners Iptscrae might appear to be an irregular mass of confusing commands and oddly-ordered references, in fact it’s really quite a formal language, with a rigid and symmetrical structure all script files must adhere to. You got a glimpse of this structure while editing your own `cyborg.ipt` file (see *The Palace Users Guide*); you’ve probably also seen it in the scripting window while you were in authoring mode. The basic structure looks like this:

```
ON EVENT {
   command
}
ON EVENT {
   command
   {
      atomlist
   }
}
```

Basic Event Handling Structure

This example shows two ***event handlers***. The first possesses only a single ***command***, while the second possesses a ***command*** and an ***atomlist*** (see the following paragraphs). Note that an event handler always possesses a pair of curly brackets that denote its portion of the script. Any or all of the possible event handlers may appear here. Within each handler, any number of individual commands or atomlists may appear. The only limit is the total size of all scripts for the current room, which should not exceed 15.8 kilobytes in size (and that's an awful lot of Iptscrae).

Before moving on to the next level of the script structure, a note on terminology is necessary. The Palace community uses the word script in a number of different ways. While it is true that scripts are fractal in nature (i.e., small “scripts” can be put together to create larger things which might just as easily be called “scripts”), for purposes of this manual we need to differentiate these things semantically. The following terminology makes things a bit more clear:

- ***Scripts***

  Scripts are demarcated by the keywords `SCRIPT` and `ENDSCRIPT`; they are comprised of event handlers. Note that except for the absence of the keywords `SCRIPT` and `ENDSCRIPT`, your `Cyborg.ipt` is a special type of script (i.e., you could paste your entire Cyborg.ipt into the `SCRIPT... ENDSCRIPT` section of a single spot in your Palace; most of the commands would work just fine). Because both the Cyborg.ipt and the server script can be opened via word processor, edited individually and saved in ASCII format ("text only"), they are referred to as your ***script files***.

- ***Event handlers***

  Event handlers (or simply “handlers”) are demarcated by the keywords `ON EVENT` and a pair of curly brackets. They are comprised of individual commands and atomlists. There are ten event handlers in the current version of Iptscrae: `ON ALARM, ON ENTER, ON INCHAT, ON LEAVE, ON LOCK, ON MACRO0-ON MACRO9, ON OUTCHAT, ON SELECT, ON SIGNON` and `ON UNLOCK`.

- ***Commands and atomlists***

  A handler may contain a single Iptscrae command (such as `“Hello World!” SAY`), or a sequence of commands (like `“100 RANDOM 50 + tempVariable = tempVariable ITOA SAY”`). These commands are executed in linear order when the event handler containing them is triggered. For flow control reasons, however, it is sometimes desirable to break complex handlers up into smaller segments. These segments are called atomlists. Atomlists are demarcated by pairs of curly brackets occurring within event handlers. They are comprised of individual commands and (possibly) even smaller nested atomlists.

Another word for ***handler*** would be ***routine***. Another word for ***atomlist*** would be ***subroutine***.

When working with the `Cyborg.ipt` file, this is as much as you need to know; since there is only one Cyborg (you), all handlers in the script obviously refer to that entity. When working with the ***server script*** things get more complex, and two additional levels must be added to the structure we have just created. These two new levels appear *above* the level of “scripts” — they are ***doors/spots*** and ***rooms***.

Here’s how it works… Just as commands and atomlists are contained within handlers, handlers in the server script are found only within scripts (those clearly-marked `SCRIPT ... ENDSCRIPT` areas in the script file). ***Scripts*** are themselves contained within ***spots*** (one script per spot). Finally, ***spots*** (and their cousins, ***doors)*** are contained within ***rooms***. The following three bullet points, then, would be placed *above* the previous three:

- ***Script files***

  The server script is a script file (like `Cyborg.ipt,` it can be edited externally via word processor and saved as ASCII (text only). It contains ***server data*** (“preferences”) and rooms.

- ***Rooms***

  Rooms contain ***room data*** (some of which may be addressed via authoring mode, and some of which can only be viewed when editing the script file manually) and spots.

- ***Spots***

  Spots (including doors) are comprised of ***spot data*** (some of which may be addressed via authoring mode, and some of which can only be viewed when editing the script file manually) and scripts (which are demarcated by the keywords `SCRIPT ... ENDSCRIPT`, as explained above).

All elements higher than event handlers — scripts, doors, spots and rooms — are identified by special keywords indicating where they begin and end. These keywords are:

| Element | Begin | End |
|---|---|---|
| Room | `ROOM` | `ENDROOM` |
| Spot | `SPOT` | `ENDSPOT` |
| Door | `DOOR` | `ENDDOOR` |
| Script | `SCRIPT` | `ENDSCRIPT` |

Demarcation Keywords

Correct use of these keywords is crucial to the successful operation of the server script. The keywords are shown in *`BOLD CAPITALS`* in this example:

```
ROOM
   ; Room Data
   DOOR
      ; Door Data
   ENDDOOR
   SPOT
   ; Spot Data
      SCRIPT
   ON EVENT {
      Command
      Command
   }
   ON EVENT {
      Command
      {
         AtomList
      }
      {
         AtomList
      }
   }
      ENDSCRIPT
   ENDSPOT
ENDROOM
```

Basic Room Structure

In addition to the basic layout of keywords, this simple example also illustrates the types of "plurality" a script file can accommodate. Multiple commands and atomlists can be placed within a single handler, and multiple handlers can be placed within the SCRIPT ... ENDSCRIPT region of a single spot (there can only be one SCRIPT ... ENDSCRIPT region per spot, however).

Let's examine a sample room from the original "Mansion" Palace to see how this structure is put to use in real life. The following code was taken from the room called Heaven's Gate...

```
ROOM
   ID 777
   NAME "Heaven's Gate"
   PICT "Heaven.gif"
   ARTIST "Elaine Alderette"
   DOOR
      ID 1
      DEST 103
      OUTLINE 0,309  512,313  512,384  0,384
   ENDDOOR
   SPOT
      ID 2
      OUTLINE 35,35 96,35 94,76 34,77
      SCRIPT
         ON ENTER {
            "FazeIn" SOUND
            CLEARPROPS
            "Halo" DONPROP
            8 SETCOLOR
         }
      ENDSCRIPT
   ENDSPOT
   SPOT
      ID 3
      OUTLINE 387,24  444,21  437,57  390,51
      SCRIPT
         ON SELECT {
            "palace://mansion.thepalace.com:1313" NETGOTO
         }
      ENDSCRIPT
   ENDSPOT
ENDROOM
```

Example Room ("Heaven's Gate")

In this example, the room is constructed in nested levels of detail: Scripts reside within handlers, handlers reside within the areas indicated by SCRIPT and ENDSCRIPT keywords, SCRIPT ... ENDSCRIPT regions reside within doors or spots, which reside

within the Room itself. It is the rigidity of this structure that allows the server to quickly locate and execute the correct actions at the correct times; this is key to the power of the Palace.

---

NOTE – ON INDENTING: Although indenting your code is considered good programming practice and is definitely recommended for Iptscrae programmers, it is not strictly necessary, nor is it aided or enforced in any way by the software. In other words, it's up to you. But it's hard to imagine anything more confusing than unindented Iptscrae…

---

## Specifying room data

Every room in a server script begins with a block of data, only some of which is accessible via authoring mode. This "room data block" will contain some or all of the following commands:

```
ROOM
    ID number
    PRIVATE
    NOPAINTING
    NOCYBORGS
    HIDDEN
    NOGUESTS
    NAME "Room Name"
    PICT "picture.gif"
    ARTIST "Artist Name"
    PICTURE ID number NAME "another.gif" <TRANSCOLOR number> ENDPICTURE
```

### The Room Data Block

The Room block has the following components.

#### ID *number*

This line indicates the *roomID* assigned to this room in the script. Although this number may be assigned by whatever arbitrary means you wish, there must be a number assigned, and it must be unique to the room. In other words, it's perfectly fine to number rooms sequentially, or to skip numbers between rooms, or to put rooms in illogical order, but every room must possess an ID which is different from any other room's ID.

### PRIVATE

```
PRIVATE
```

This line, if it exists, hides the number of people in the room. Generally, when a user looks in the **Rooms** Window (via the **Options** Menu in the Palace client), each room is shown with the number of people currently in that room. Private rooms, on the other hand, possess a dash instead of a number. The only way to tell the number of users in a private room is to enter it. This line can be toggled in authoring mode via the **Private** option in the **Room Info** window.

### NOPAINTING

```
NOPAINTING
```

This line, if it exists, prohibits the use of all ***paint commands*** in the room. It is a good idea to use this command in rooms with a large amount of scripted activity (looping alarms, for example), as painting can seriously affect the speed of events in the room. This line can be toggled in authoring mode via the **No Painting** option in the **Room Info** window.

### NOCYBORGS

```
NOCYBORGS
```

This line, if it exists, prohibits user scripts (also known as “cyborg scripts”) in the room. It is a good idea to use this command in rooms with a large amount of scripted activity (for example, alarms), as well as in rooms where fairly large numbers of people are expected to gather, as user scripts can seriously affect the speed of events in the room. This line can be toggled in authoring mode via the **No User Scripts** option in the **Room Info** window.

### HIDDEN

```
HIDDEN
```

This line, if it exists, keeps the room from showing up in the **Rooms** Window. For all intents and purposes, the room doesn’t exist for non-operators (owners and operators still see all rooms in the Palace). Note that non-operators can still enter the room via a door or by using the `GOTOROOM` command if they know the roomID. This line can be toggled in authoring mode via the **Hidden** option in the **Room Info** window.

### NOGUESTS

```
NOGUESTS
```

This line, if it exists, prohibits Guests from entering the room (making the room into a “members only” room). This line can be toggled in authoring mode via the **No Guests** option in the **Room Info** window.

### NAME

```
NAME "Room Name"
```

This line indicates the name of the room as it will appear in the Status Bar (just beneath the Viewing Area in the Client interface) and when accessed by the `ROOMNAME` command. The room name can be changed in authoring mode via the **Room Info** window.

### PICT

```
PICT "picture.gif"
```

This line indicates the name of the .GIF image used as the background of the room. Whenever a user enters the room the Palace client checks to see if it possesses this graphic, and requests it from the server if necessary. The background graphic of a room can be changed in authoring mode via the **Room Info** window.

---

NOTE – All background graphics should be 512 by 384 pixels in size (since this is the size of the client's Viewing Area; larger images will not be seen in their entirety, and smaller images will not fill the screen). To avoid unwanted colorshifts, the Palace Palette should be used for all Palace art (this palette can be extracted from any of the images that come with the Palace software).

The Palace client for Macintosh version 2.3 can display .JPG files as well as .GIFs. To support older Macs and Windows users, however, GIF images must be still made available. When entering a room for which both file types exist, Macs version 2.3+ will request the JPG file only, while older Macs and Windows clients will request the GIF file only.

---

### ARTIST

```
ARTIST "Artist Name"
```

This line indicates the name of the artist whose work appears in the room. The name can be changed in authoring mode via the **Room Info** window.

### PICTURE ... ENDPICTURE

```
PICTURE ID number NAME "picture" ENDPICTURE
```

This line indicates the name of a graphic (`.GIF`) that will be used to animate a spot in the room. Whenever a user enters the room the client checks to see if it possesses all the graphics called for, and requests any graphics it needs from the server.

If additional graphics will be used in the room (i.e., anything besides the background image itself), each must possess its own `PICTURE ... ENDPICTURE` line. These pictures will appear, disappear and do their interactive stuff behind all avatars and props (the "foreground" layer) but in front of the room's "background" layer, in what is called the "midground" layer. Within this layer, midground pictures can be stacked and manipulated as desired.

Each picture in the midground must have a unique *ID* assigned to it. The ID of a picture can be any number at all — the order doesn't matter — but every graphic in the room must possess an ID that is different from the ID of any other graphic in the room.

### TRANSCOLOR

`TRANSCOLOR` *number*

This command, which may optionally be included in the `PICTURE ... ENDPICTURE` line, is used to inform the Mac client which of the picture's 256 colors (if any) should be displayed as ***transparent***, revealing whatever exists beneath. If this command is left out, the Mac client will use whatever transparency color the picture was originally created with. The Windows client will do this in any case.

A `TRANSCOLOR` command is automatically inserted into the `PICTURE ... ENDPICTURE` line whenever a new picture is assigned to a spot in authoring mode. The default *number* inserted is 0 (zero); i.e., by default, it is assumed that the graphic uses the color 0 as the transparent, or "alpha" channel. Mac-based operators may address this value directly via the **Door Info** window, and must set it correctly (or remove it via ASCII editing) so that other Mac users will see the transparent effects correctly. The following table explains how a sample picture will be displayed on each type of client under various circumstances; our sample picture uses color 114 as the transparent color.

| USER OS | TRANSCOLOR COMMAND (IN SERVER SCRIPT) | | |
|---|---|---|---|
| | `TRANSCOLOR 0` *(default inserted via authoring mode or by editing script manually)* | `TRANSCOLOR 114` *(set via Mac wizard in authoring mode or by editing script manually)* | <no `TRANSCOLOR` command> *(`TRANSCOLOR` can only be removed or avoided by editing the script manually)* |
| **Windows Client** | Right | Right | Right |
| **Mac Client** | Wrong *(color 114 will be visible, color 0 will be invisible)* | Right | Right |

---

NOTE – Windows-based operators: The Windows client does not allow operators to affect the `TRANSCOLOR` value via authoring mode, and as the table above indicates, it ignores the `TRANSCOLOR` command completely (using the actual alpha channel of the image instead). To make sure your Windows modifications will display properly on the Mac client, you have two choices. Either:

A. Create your GIFs using the color 0 (white) as the transparent color. If you have white regions in your picture that you want to appear visible (i.e., not transparent), use the alternate white (color number 215) for them. This method will allow you to assign new pictures to new spots in authoring mode later, without having to worry about the `TRANSCOLOR` setting, or

B. Edit your server script manually, using a word processor capable of saving as ASCII (text only). You may choose to replace all of your "`TRANSCOLOR 0`" commands with the appropriate numbers, or you can simply remove them entirely.

## Specifying spot data

Like rooms, every spot (including doors) begins with a block of special data, only some of which is accessible via the ***Scripting Window***. This "Spot Data Block" will contain some or all of the following commands:

```
SPOT
    ID number
    NAME "Spot Name"
    OUTLINE x1,y1 x2,y2 x3,y3 ...
    PICTS pictureID,xOffset,yOffset ... ENDPICTS
    SCRIPT
    ENDSCRIPT
```

The Spot Data Block

### ID

`ID number`

This line indicates the *spotID* assigned to this spot in the room. Although this number may be assigned by whatever arbitrary means you wish, it must be here, and it must be unique to the spot. Again, it's perfectly fine to number spots sequentially, or to skip numbers between spots, or to put spots in any illogical order you want, but every spot must possess an ID that is different from any other spot in the room.

### NAME

`NAME "Spot Name"`

This line indicates the name of the spot as it will appear if a `SPOTNAME` command is used. The name of a spot can be changed in authoring mode via the **Door Info** window.

### OUTLINE

`OUTLINE x1,y1  x2,y2  x3,y3  < ... >`

This line indicates the positions of all points (vertices) that determine the shape of the spot; each *x,y* specifies the position of a single point in the outline. Although the default doors created by selecting **New Door** from the **Operator** Menu always possess four points, by editing this line manually you can create doors and spots with any number of points greater than two (Mac users can set the number of points via the **Spot Info** window). As usual, all "x" values must be between 0 and 511, while all "y" values must be between 0 and 383.

### PICTS ... ENDPICTS

```
PICTS pictureID,offsetX,offsetY  < ... >  ENDPICTS
```

This line indicates the IDs and relative positions of the midground graphics that will be used to animate the spot. The data between the keywords is arranged in "triplets," each made up of three numbers (e.g.: "100,10,20" or "5,-25,0"). Each triplet describes one of the spot's *states:* the first triplet describes *state 0*, the second triplet describes *state 1*, the third describes *state 2*, and so on.

Each triplet — each *state* — is defined by a `pictureID`, an `xOffset`, and a `yOffset`. The *pictureID* specifies which graphic will be displayed when the spot enters the State. These IDs are the same as those specified in the `PICTURE ... ENDPICTURE` line of the *Room Data Block*. The *xOffset* indicates how far the graphic will be shifted to the left (negative) or right (positive), relative to the spot's actual location. The y*Offset* indicates how far the graphic will be shifted to upward (negative) or downward (positive), relative to the spot's actual location.

The graphics associated with a spot's states can be changed in authoring mode by selecting **Edit Pictures** from the **Door Info** window. The order in which these graphics are displayed in the Pictures List is the same as the "state order," i.e.: the first picture listed — "NONE" by default — represents *state 0*, the second is *state 1*, the third is *state 2*, and so on.

To change the X and Y offsets of a picture while in authoring mode:

1. Use `SETSPOTSTATE` to set the spot to the state you want to change.
2. Use `SETPICLOC` to move the picture relative to the spot's position.

### SCRIPT ... ENDSCRIPT

```
SCRIPT
  <handlers>
ENDSCRIPT
```

These lines (if they exist) indicate the beginning and end of a spot's script (as opposed to "Spot Data"). Everything appearing between the keywords `SCRIPT` and `ENDSCRIPT` is part of the script, and may also be accessed via the Scripting Window in authoring mode.

# Iptscrae Language Reference

This section is a reference to the Iptscrae language:



Iptscrae possesses over 100 specialized commands and keywords, as well as its own versions of many commonly-used operators and functions; each of these represents a wide range of interactive possibilities awaiting your imagination.

## Data types

The Palace is ***integer-based*** (meaning that it works in terms of whole numbers), but barring floating-point variables, the software can handle all basic data types: ***symbols*** (variables), ***numbers*** (integers), ***strings*** (string literals), ***atomlists*** (subroutines) and ***arrays***, as well as a number of special-case symbols and reserved keywords.

### Symbols (variable names)

Symbols must start with a letter, and may contain any combination of letters, numbers and the underscore. They may not contain any spaces, and have a maximum length of 31 characters. Examples:

```
x
plan9
my_really_big_variable
```

## Numbers ( integers)

Numbers in Iptscrae are stored as 32 bit sized integers. No floating-point is allowed. Numbers must be specified in decimal notation using the digits 0 through 9, with an optional leading ( - ) for negative numbers. Examples:

```
2
-32
4283748
```

## Strings (string literals)

String literals must be encased in double quotes, for Example "string”. If you need to include a quote symbol within a string, precede it with a backslash: `“these \”doublequotes\” are okay”`. You can create long strings by concatenating multiple strings together using the `&` operator (see *Operators* on page 81). Examples:

```
"Hello"

"A Flock " "of Words" &

"Suddenly, Fred shouted \"Look out!\" and hit the dirt."
```

## Arrays

An array is an ordered list of other Iptscrae data types. Arrays may be declared with the `ARRAY` command or by encasing the elements of the array in square brackets ([and]). You use the `GET` function to extract an item from an array, use the `PUT` command to insert an item into an array, and the `FOREACH` command to perform an operation on each item in an array. Arrays can be composed of different data types, including other arrays. Examples:

```
[ 100 200 300 ]
[ "Hello" "World" ]
[ 100 "Hello" [ 0 1 2 ] ]
5 7 9 ARRAY
```

## Atomlists

Atomlists are small Iptscrae scripts, or "subroutines." They can contain all other data types, including other atomlists. Some commands (such as `EXEC`, `IF`, `WHILE`, `ALARMEXEC` and `FOREACH`) operate *on* atomlists (rather than *in* them, as most other commands do). Atomlists must be encased in curly brackets ({and}). Examples:

```
{ 1 tempVar = }
{ "Howdy" SAY }
{
```

```
    23 firstVar =
    secondVar firstVar - deltaVar =
}
```

# Special-Case Symbols

## CHATSTR

CHATSTR is a reserved word and a special-case variable in Iptscrae. Whenever a script is executing in response to an INCHAT or OUTCHAT event, CHATSTR represents the chat text itself. This variable may be modified on the fly. In the case of an INCHAT event, this will change the text that ends up getting displayed on the screen. In the case of an OUTCHAT event, it will change the text that is sent to other users.

It is generally preferable to use OUTCHAT rather than INCHAT event handlers.

The following example shows how to make an effect that occurs whenever you speak (type) a key word or phrase, by applying an IF statement to CHATSTR. The whole thing resides in the OUTCHAT handler.

### Example

```
ON OUTCHAT {
    {
        "applause" SOUND
    } CHATSTR “!Thank you!” == IF
}
```

## \ *(The "Backslash" Character)*

The backslash has a special meaning in Iptscrae; when it appears within a character string, it indicates that the character immediately following it should be included within the string literally (i.e., as a printable character). It is most often used to indicate that a double quote should be printed as *part* of a string, rather than signifying the *end* of it (as it typically would). The backslash can be used with other control characters, as well as in GREPSTR regular expressions.

### Example 1 (a quote within a SAY command)

```
"The word he said was \"rosebud.\"" SAY
```

### Example 2 (a local whisper in a "sign balloon")

```
"@200,20\^Note to myself..." WHOME PRIVATEMSG
```

# Event handlers

***Events*** are the basic stimuli of the server, representing all the things your Palace can "watch for" and respond to. They include significant user actions such as entering and leaving rooms, clicking on doors and spots, talking and other basic activities, plus a special type of event called an ***alarm***.

For each event type, there exists an ***event handler***. Event handlers reside within the `SCRIPT... ENDSCRIPT` portions of a script file. The name of each event handler consists of the word `ON` followed by the name of the event it handles (i.e., `ON ENTER`, `ON LEAVE`, `ON SELECT`, and others). When an event occurs, the server script file is consulted to see whether there is a handler for that event in the current room. The client also checks the user's `Cyborg.ipt` file. If any appropriate event handlers are found, the scripts within them are executed immediately.

Note that not all handlers may be used in all objects; some may be applied only to doors or spots, others to cyborgs, and some to all three.

## ON ALARM

(Doors, Spots, Cyborgs)

An `ALARM` event occurs in response to the `SETALARM` command in a script. It can be used to schedule a periodic event, such as an animation, or to provide a delayed response. To trigger the following example handler, use the `SETALARM` command (see *SETALARM* on page 51).

### Example

```
ON ALARM {
   "I am alarmed!" SAY
}
```

## ON ENTER

(Doors, Spots, Cyborgs)

An `ENTER` event occurs when a user enters the room. Scripts in this handler can be used (among other things) to start animations (via `SETALARM`), initialize user-defined functions, start room behavior, generate automatic "hello" messages from the entering user, etc.

### Example

```
ON ENTER {
   "I have entered!" SAY
}
```

## ON INCHAT

(Doors, Spots, Cyborgs)

An INCHAT event is triggered in response to an incoming chat message; a better name for this handler might be ON HEAR. It is generally preferable to use the OUTCHAT handler instead of this one, because INCHAT events will be triggered by all user speech, user scripts, and any other "talking spots" in the room (very possibly flooding the server), whereas OUTCHAT events will only be triggered by users' deliberate speech. Scripts in the INCHAT handler can be used to modify the text of the incoming chat message via use of the CHATSTR variable.

### Example

```
ON INCHAT {
   "yes" SOUND
}
```

---

NOTE – Want to get killed for flooding your server? Make a spot in your Palace that does a SAY in the INCHAT handler. Better yet, make two of them in the same room. Then say something.

---

## ON LEAVE

(Doors, Spots, Cyborgs)

A LEAVE event occurs when a user leaves the room. Scripts in this handler will be executed in their entirety *before* the user actually departs.

### Example

```
ON LEAVE {
   "I am leaving!" SAY
}
```

## ON LOCK

(Lockable Doors)

A LOCK event occurs when a door becomes *locked*. The event is sent to the door itself. Scripts in this handler can be used to add additional behaviors to the door in question.

Example

```
ON LOCK {
   "The door is locked!" SAY
}
```

## ON MACROn for n=0 to n=9

(Cyborgs)

If the room allows cyborgs, this event runs when a script uses `ONMACROn` (where n is between 0 and 9) or the user uses an avatar selection 0-9

## ON OUTCHAT

(Doors, Spots, Cyborgs)

An `OUTCHAT` event is triggered in response to an outgoing chat message (when a user types something and presses the *Return* key). Scripts in the `OUTCHAT` handler can be used to modify the text of outgoing chat messages via use of the `CHATSTR` variable. A good example of an `OUTCHAT` script is the text message handler in *The Moor.*

### Example

```
ON OUTCHAT {
   {
      "Polo!" CHATSTR =
   } "Marco" CHATSTR == IF
}
```

## ON SELECT

(Doors, Spots)

A `SELECT` event occurs whenever a user clicks on a hotspot. Note that unless the hotspot possesses a `DONTMOVEHERE` command, the user's avatar will immediately move to the location clicked.

### Example

```
ON SELECT {
   "I selected the spot!" SAY
}
```

## ON SIGNON

(Cyborgs)

A `SIGNON` event is sent to each user as they sign on.

### Example

```
ON SIGNON {
   "I have signed on!" SAY
}
```

### ON UNLOCK

(Lockable Doors)

An `UNLOCK` event occurs when a door becomes unlocked. The event is sent to the door itself. Scripts in this handler can be used to add additional behaviors to the door in question.

#### Example

```
ON UNLOCK {
   "The door is unlocked!" SAY
}
```

# Commands and functions

Now that you know where scripts can be placed and what events they can react to, you'll probably want to know what kinds of things you can make them do. You can add action to your scripts by using the ***commands*** and ***functions*** described in this section.

Commands perform actions that directly affect the state of objects in the current room (users, doors, spots and props). Functions are similar to commands, but their concerns are data-oriented; a Function always leaves a value (some kind of data) on the top of the stack, so it can be accessed and manipulated by other commands and Functions. This is what we mean when we say that a Function "returns" a value: it places this value on the top of the stack. Most functions perform both "pops" and "pushes" in doing their job: for instance, the "plus" (+) operator pops the top two values off the stack, adds them, and pushes the sum onto the stack. When the operation ends there is *one* value — not three — on the stack.

For ease of use, the commands and functions have been divided into several categories, based upon the objects they affect and actions they perform:

- ***Cyborg Commands*** directly affect or refer to users, avatars and props. See *Cyborg commands and functions* on page 32.
- ***Spot Commands*** directly affect or refer to hotspots (both *doors* and *spots*). See *Spot commands and functions* on page 48.
- ***Paint Commands*** deal with the paint tools and painting on the screen. See *Paint commands and functions* on page 55.
- ***Sound Commands*** deal with WAV or MIDI files and their use. See *Sound commands and functions* on page 59.
- ***Flow Commands*** affect the logical flow (decision-making processes) of the program. *Flow commands and functions* on page 60.

- ***General Commands*** affect data and variables; usually on the stack, and perform other miscellaneous actions which don't fit easily into the other categories. See *General commands and functions* on page 64.

---

NOTE – Just because something is called a "cyborg command" doesn't mean that it can only be placed in the `Cyborg.IPT`. On the contrary, most commands will work just fine in either script file, in any type of object. Rather, the categories refer to the types of objects or actions that are ***manipulated*** by the command; what we might call "the subject objects."

---

The following listings describe all Iptscrae commands and functions.

# Cyborg commands and functions

## CHAT

`"`*`message`*`" CHAT`

This command displays the message in a cartoon balloon, as though the user typed it directly into the *Input Box*. It is identical to the `SAY` command.

### Example

```
"This is a sentence." CHAT
```

## CLEARPROPS

```
CLEARPROPS
```

This command removes all the props the user is wearing. A synonym is `NAKED`.

### Example

```
CLEARPROPS
```

## DOFFPROP

```
DOFFPROP
```

This command removes the last prop put on by the user.

### Related commands

DONPROP, DROPPROP and REMOVEPROP.

### Example

```
DOFFPROP
```

## DONPROP

```
propID DONPROP
"propName" DONPROP
```

This command (in either of its forms) adds a prop to the user's costume. The prop can be specified by ID# (preferable) or by Name.

### Related commands

DOFFPROP, DROPPROP and REMOVEPROP.

### Examples

```
1280 DONPROP
```

```
"BRBSIGN" DONPROP
```

## DROPPROP

*x y* DROPPROP

This command takes the last prop user put on and drops it into the floor (making it a loose prop). *x* and *y* specify where it will be dropped.

### Related commands

DOFFPROP, DROPPROP and REMOVEPROP.

### Example

```
512 RANDOM tempX =
384 RANDOM tempY =
tempX tempY DROPPROP
```

## GLOBALMSG

"*message*" GLOBALMSG

This command is available only to users with Operator privileges. It generates a message that everybody on the server sees. Use it sparingly.

### Example

```
"This is a Global Message." GLOBALMSG
```

## GOTOROOM

*roomID* GOTOROOM

This command is used by spots to navigate users to another room. You can find out the *roomID* by looking at the **Room Info** window, or by using a ROOMID command.

### Example

86 GOTOROOM

## GOTOURL

"*urlString*" GOTOURL

This command can be used to send users to other Palaces and Internet URLs. If you use a URL beginning with "palace://" the user will be connected to the Palace site specified (if possible); otherwise the user's system will attempt to go there via whatever application is normally associated with URLs of that type (web browsers, news readers, FTP utilities, etc.) Same as NETGOTO.

---

NOTE – If the URL begins with "palace://", it must be the only thing in the script in order to work with the Macintosh Client or The Palace Viewer.

---

### Examples

```
"palace://welcome.thepalace.com" GOTOURL
"http://www.thepalace.com" GOTOURL
```

## GOTOURLFRAME

"*url*" "*frame*" GOTOURLFRAME

This command can be used to send users to the url passed in the browser frame named "frame".

---

NOTE – Frame specification is effective in TPV only. The Macintosh and Windows clients use the default frame.

---

### Example

```
"http://www.thepalace.com" "myframe" GOTOURLFRAME
```

## HASPROP

"*propName*" HASPROP
propID HASPROP

This function pushes a 1 onto the stack if the user possesses the specified prop; otherwise it pushes a 0.

### Example

```
{
   "I am wearing the Ray Bans" SAY
} {
   "I am NOT wearing the Ray Bans" SAY
} "Ray Bans" HASPROP IFELSE
```

## INSPOT

*spotID* INSPOT

This function pushes a 1 onto the stack if the user's current location is within the spot indicated by *spotID*; otherwise it returns a 0. The following example assumes that the current room includes a spot with an ID of 1.

### Example

```
{
   "I'm in The Spot!" SAY
} {
   "I'm not in The Spot!" SAY
} 1 INSPOT IFELSE
```

## ISGOD

ISGOD

This function pushes a 1 onto the stack if the user running the script has owner-level access, otherwise it pushes a 0.

### Example

```
{
   "I am an Owner!" SAY
} {
   "I am not an Owner!" SAY
} ISGOD IFELSE
```

## ISGUEST

```
ISGUEST
```

This function pushes a 1 onto the stack if the user has guest access, otherwise it returns 0.

### Example

```
{
   "I am a Guest!" SAY
} {
   "I am not a Guest!" SAY
} ISGUEST IFELSE
```

## ISWIZARD

```
ISWIZARD
```

This function pushes a 1 onto the stack if the user has owner or operator-level access, otherwise it returns 0.

### Example

```
   "I am a operator!" SAY
} {
   "I am not a operator!" SAY
} ISWIZARD IFELSE
```

## KILLUSER

*userID* KILLUSER

This command “kills” (disconnects) the user with the specified userID#. If members aren't allowed to kill (which is typical of most Palace servers), this command won't work. In any case guests cannot use it. Note that to get *userID* it is necessary to use one of the following commands: ROOMUSER, WHOCHAT, WHOME or WHOTARGET. The following example shows you how to commit suicide in Iptscrae:

### Example

```
WHOME KILLUSER
```

## LOCALMSG

"*message*" LOCALMSG

This command generates a message that only the user executing the script sees. You can precede the message with @x,y to control its position.

### Example

```
"This is a LOCALMSG.  I am the only one who sees it." LOCALMSG
```

## MACRO

*number* MACRO

This command causes the user to don the specified ***macro*** (a "macro" corresponds to an "avatar" — a group of props that are all worn at the same time). If the user possesses a saved macro for the *number* used in the script, their avatar will instantly change to it. If an ON MACRO script exists in the user's Cyborg.IPT, it will be executed instead of the prop change.

### Example

```
16 RANDOM MACRO
```

## MOVE

*x y* MOVE

This command moves the user *x,y* pixels relative to the current position.

### Example 1 (move down and right)

```
5 5 MOVE
```

### Example 2 (move randomly)

```
11 RANDOM 5 - tempX =
11 RANDOM 5 - tempY =
tempX tempY MOVE
```

## NAKED

NAKED

This command removes all of a user's props. It is the same as CLEARPROPS.

### Example

```
NAKED
```

## NBRROOMUSERS

NBRROOMUSERS

This function returns the number of users currently in the room.

### Example

```
NBRROOMUSERS ITOA tempVar =
"NBRROOMUSERS = " tempVar & "." & SAY
```

## NBRUSERPROPS

NBRUSERPROPS

This function returns the number of props currently worn by the user.

### Example

```
NBRUSERPROPS ITOA tempVar =
"NBRUSERPROPS = " tempVar & "." & SAY
```

## NETGOTO

"*urlString*" NETGOTO

This command can be used to access other Palace servers or other Internet URLs. If you use a "palace://" URL, the user will be signed on to the Palace server indicated (if possible); otherwise the system will attempt to take the user there by some other means. Same as GOTOURL.

---

NOTE – If the URL begins with "palace://", it must be the only thing in the script in order to work with the Macintosh Client or The Palace Viewer.

---

### Example

```
"palace://welcome.thepalace.com" NETGOTO
"http://www.thepalace.com" NETGOTO
```

## POSX

POSX

This function returns the user's horizontal coordinate.

### Example

```
"My current POSX is " POSX ITOA & SAY
```

## POSY

POSY

This function returns the user's vertical coordinate.

### Example

```
"My current POSY is " POSY ITOA & SAY
```

## PRIVATEMSG

```
"message" userID PRIVATEMSG
```

This command generates a private message to another user. Note that to get *userID* it is necessary to use one of the following commands: ROOMUSER, WHOCHAT, WHOME or WHOTARGET.

### Example

```
"This is a PRIVATEMSG. I am whispering to myself." WHOME PRIVATEMSG
```

## REMOVEPROP

```
propID REMOVEPROP
"propName" REMOVEPROP
```

This command removes a prop from the user's costume. The prop can be specified by name or by *propID*. The following example removes the "Ray Bans" prop (if the user is wearing it).

### Related commands

DONPROP, DROPPROP and DOFFPROP.

### Example

```
{
   "Ray Bans" REMOVEPROP
} {
   "First I have to put on the Ray Bans!" SAY
} "Ray Bans" HASPROP IFELSE
```

## ROOMMSG

"*message*" ROOMMSG

This command generates a message that everyone in the room sees. Use it sparingly. You can precede the message with *@x,y* to control its position.

### Example 1

```
"This is a ROOMMSG.  Everyone in this room can see it." ROOMMSG
```

### Example 2

```
"@10,10 This is a ROOMMSG up in the corner.  Isn't that awesome?"
ROOMMSG
```

## ROOMUSER

*number* ROOMUSER

Every user on the server has a unique *userID* that stays the same as long as they remain connected, but at any given moment they also possess a "room user" number assigned to them by the room they're in. This function returns the *userID* of room user *number* in the current room.

### Example

```
WHOME ROOMUSER ITOA tempVar =
"I am currently ROOMUSER number " tempVar & "." & SAY
```

## SAY

"*message*" SAY

This command displays *message* as if the user typed it in directly. It is identical to the CHAT command.

### Example 1 (talking)

```
"I am saying something!" SAY
```

### Example 2 (thinking)

```
":I am thinking something!" SAY
```

### Example 3 (shouting)

```
"!I am shouting something!" SAY
```

### Example 4 (sign)

```
"^This is a sign!" SAY
```

### Example 5 (positioning)

```
"@10,10 Now I’m saying something way up here!" SAY
```

## SETCOLOR

*number* SETCOLOR

This command sets the user’s face color to one of 16 colors. If the "tinted balloon" preference is checked, this command also contols the color of the word balloon. The specified *number* must be an integer from 0 to 15. The possible colors are numbered by dividing the spectrum into 16 equal steps, as follows:

| | |
|---|---|
| #0 Red | #1 Orange |
| #2 Orange / Yellow | #3 Yellow |
| #4 Yellow / Green | #5 Light Green |
| #6 Green | #7 Green / Cyan |
| #8 Cyan | #9 Light Blue |
| #10 Medium Blue | #11 Dark Blue / Purple |
| #12 Purple | #13 Magenta |
| #14 Magenta / Pink | #15 Pink |

### Example

```
16 RANDOM SETCOLOR
```

## SETFACE

*number* SETFACE

This commands sets the user's face to one of the 13 built-in faces (props are not removed, however). The specified *number* must be an integer from 0 to 12.s

| | |
|---|---|
| #0 Eyes Closed (sleeping or blushing) | #1 Smile |
| #2 Look Down (nodding) | #3 Talking |
| #4 Wink Left | #5 Normal |
| #6 Wink Right | #7 Tilt Left (shaking head) |
| #8 Look Up (nodding) | #9 Tilt Right (shaking head) |
| #10 Sad | #11 Blotto |
| #12 Angry | |

### Example

```
13 RANDOM SETFACE
```

## SETPOS

*x y* SETPOS

This command immediately moves the user to position *x y* in the *Viewing Area*. *x* must be an integer from 0 to 511. *y* must be an integer from 0 to 383.

### Example

```
10 10 SETPOS
```

## SETPROPS

[ *propArray* ] SETPROPS

This command acts like a ***macro***, causing the user to immediately don all props listed in [ propArray ]. Props may be listed either by Name or by ID#

### Example

```
[ "Ray Bans" "daisy" "Wine Bottle" ] SETPROPS
```

Like all arrays, *[ propArray ]* must be enclosed in square brackets *( [ ] )*. Also, prop names, being strings, must be enclosed in double quotes (").

## SOUND

```
"fileName" SOUND
```

This command plays the sound file *filename*. Sounds are WAV files, saved without the .WAV extension, and reside on the client in `\Palace\Media\YourPalaceName\Sounds`.

### Example 1 (play specified sound)

```
"Applause" SOUND
```

### Example 2 (play random sound)

```
6 RANDOM tempVar =
[ "Yes" "No" "Fazein" "Applause" "Boom" "Crunch" ] tempVar GET SOUND
```

## SUSRMSG

```
"message" SUSRMSG
```

This command generates a message that all owners and operators will see, no matter where they are on the server. Use it sparingly.

### Example

```
"This is an SUSRMSG from " USERNAME & SUSRMSG
```

## TOPPROP

```
TOPPROP
```

This function returns the *propID* of the last prop the user put on. If the user is "naked" it returns 0 (zero). The following example shows you how to scatter all your currently-worn props.

### Example

```
{ 400 RANDOM 300 RANDOM DROPPROP } { TOPPROP } WHILE
```

## USERNAME

```
USERNAME
```

This function returns the user's `User Name` as specified in the **Preferences** dialog. You can't change a user name from a script.

### Example

```
"Hello, my name is " USERNAME & "!" & SAY
```

## USERPROP

```
number USERPROP
```

This function returns the *propID* of one of the props currently worn by the user. *number* is a number from 0 to 8 indicating which prop you want to identify (note that this refers to the order they were *donned* in, not necessarily the order they *appear* in). You can determine the number of props currently worn by using the `NBRUSERPROPS` command, as illustrated in the following example.

### Example

```
NBRUSERPROPS RANDOM whichProp =
whichProp USERPROP ITOA propIdent =
whichProp ITOA " USERPROP = " & propIdent & "." & SAY
```

## WHOCHAT

```
WHOCHAT
```

This function returns the *userID* of the user who invoked an `INCHAT` event.

### Example

```
WHOCHAT ITOA tempVar =
"The WHOCHAT command returns " tempVar & "." & SAY
```

## WHOME

```
WHOME
```

This function returns the user's own *userID*.

### Example

```
WHOME ITOA tempVar =
"The WHOME command returns " tempVar & "." & SAY
```

## WHONAME

*userID* WHONAME

This function returns the *User Name* of the specified user. Note that to get *userID* it is necessary to use one of the following commands: ROOMUSER, WHOCHAT, WHOME or WHOTARGET. The following example causes you to say the name of room user 0 (zero) in the current room (that's *you*, if you're the only person in the room at the moment!)

### Example

```
0 ROOMUSER WHONAME SAY
```

## WHOPOS

"*name*" WHOPOS
*userID* WHOPOS

This function (in either of its forms) returns the current *x,y* position of the user. Note that *x* is placed on the stack before *y*, which means that y is ready to be retrieved from the stack first. To reverse their positions so they can be used in their typical order (X, then Y), use the SWAP function.

### Example

```
WHOME WHOPOS SWAP ITOA tempY = ITOA tempX =
"WHOME WHOPOS returns '" tempX & "' '" & tempY & "'." & SAY
```

## WHOTARGET

WHOTARGET

This function pushes the *userID* of the person you have selected for private chat (i.e., Whisper Mode or ESP) or zero if you have not selected a target.

### Example

```
WHOTARGET USERNAME tempVar =
{
    "WHOTARGET USERNAME returns '" tempVar & "'." & SAY
```

```
} {
   "I must select someone in order to use the WHOTARGET Command." SAY
} tempVar <> "" IFELSE
```

# Spot commands and functions

## DOORIDX

*number* DOORIDX

This function returns the *ID* of the door indicated by *number*. The following example causes the user to leave through a random door:

### Example

```
NBRDOORS RANDOM DOORIDX SELECT
```

### Related functions

NBRDOORS, SELECT.

## GETSPOTSTATE

*spotID* GETSPOTSTATE

This function returns the current state of the specified hotspot or door. The following example uses NBRSPOTS and SPOTIDX as well as GETSPOTSTATE to determine the state of a random door or spot in the current room.

### Example

```
NBRSPOTS RANDOM tempVar =
"The state of spot number " tempVar ITOA & " (" & tempVar SPOTIDX
SPOTNAME & ") is " & tempVar SPOTIDX GETSPOTSTATE ITOA & SAY
```

## ISLOCKED

*doorID* ISLOCKED

This function returns a 1 if the indicated door is locked, otherwise it returns a 0. The following example uses NBRDOORS and DOORIDX to determine the state of a randomly-selected door in the current room.

### Example

```
NBRDOORS RANDOM doorNumber =
{
   "Door number " doorNumber ITOA & " is locked." SAY
} {
   "Door number " doorNumber ITOA & " is unlocked." SAY
} doorNumber DOORIDX ISLOCKED IFELSE
```

## LOCK

*doorID* LOCK

This command is used by deadbolts (or doorknobs) to lock doors. Its counterpart is the UNLOCK command. The following example assumes there is a lockable door with an ID of 1 in the current room.

### Example

```
1 LOCK
```

## ME

ME

When a spot or door is executing the script, this function pushes its *ID*.

### Example

```
" \"ME SPOTNAME\" returns \"" ME SPOTNAME & "\"." & SAY
```

## NBRDOORS

```
NBRDOORS
```

This function returns the number of doors in the room. This number may be less than or equal to the number returned by `NBRSPOTS` (because all doors are spots, but not all spots are doors).

### Related commands

`DOORIDX`

### Example

```
" \"NBRDOORS\" returns \"" NBRDOORS ITOA & "\"." & SAY
```

## NBRSPOTS

```
NBRSPOTS
```

This function returns the number of spots (including doors) in the room.

### Example

```
" \"NBRSPOTS\" returns \"" NBRSPOTS ITOA & "\"." & SAY
```

## SELECT

*`spotID`* `SELECT`

This command “clicks” the spot specified by *`spotID`*. If the spot has an `ON SELECT` handler, the script will be executed just as though the user had selected it physically. The following example assumes there is a spot with an ID of 1 in the current room. To see it work, put an `ON SELECT` handler in this spot that does something noticeable.

### Example

```
1 SELECT
```

## SETALARM

*futureTicks spotID* SETALARM

This command is used to schedule an ALARM event in the future. It can be used to create animations and other interesting activity. The user's subjective duration of a "tick" depends on the speed of both the client and server as well as the network load at the moment, but is about $1/60^{th}$ of a second. The following example assumes there is a spot with an ID of 1 in the current room. To see it work, put an ON ALARM handler in this spot that does something noticeable (see *Handlers* earlier in this document).

### Example

```
300 1 SETALARM
```

## SETLOC

*x y spotID* SETLOC

This command is used to move a spot or door, relative to its current position. It is functionally equivalent to selecting the spot or door while in authoring mode and dragging it to the new position. Note that this command is only accessible to owners and operators; i.e. it will not be executed unless the user is in owners or operator mode. For this reason, it is much more useful as an authoring command than as a scripted command. The following example assumes that you are in owners or operator mode, and that there is a spot with an ID of 1 in the current room.

### Example

```
10 10 1 SETLOC
```

## SETPICLOC

*x y spotID* SETPICLOC

This command is used to change the *x* and *y* offsets of a picture associated with spot *spotID* (these are the second and third numbers in the "triplets" appearing between PICTS and ENDPICTS). Note that only a single picture is affected, corresponding to the current *state* of the spot — any pictures associated with other states of the same spot will remain unchanged. Note also that this command is only accessible to owners and operators; i.e., it will not be executed unless the user is in owners or operator mode. For this reason, it is much more useful as an authoring command than as a scripted command. In fact, the SETPICLOC command provides the only way to change a picture's offset without editing the server script, and this makes it very useful for "fine-tuning" the placement of a particularly tricky graphic. The following example assumes that you are in owners or operator mode, and that there is a spot with an ID of 1 (and at least one picture) in the current room.

### Example

```
10 10 1 SETPICLOC
```

To see this command in action, launch the Palace server using the "Mansion" script and try this simple experiment:

1. Launch your client to access your server.
2. From your client, enter the room called "The Study" and enter **Operator mode** (from the **Options** menu).
3. If the secret bookshelf-door isn't already open, say "open sesame" to flip the spot's state and display the "open" graphic.
4. Type the following command into the *Input Box* (100 is the ID of the magical door):

   ```
   /-50 -50 100 SETPICLOC
   ```

5. You will see the graphic suddenly jump to a very "wrong" location. Try saying "close sesame" and "open sesame" a few times; you'll see that you have "permanently" changed the position of the graphic associated with the "open" state.
6. To return the graphic to its original position, type:

   ```
   /54 -21 100 SETPICLOC
   ```

## SETSPOTSTATE

*state spotID* SETSPOTSTATE

This command changes the state of a spot for all users currently in the room. For multi-state hotspots, this can be used to create animation effects. The following example assumes that the current room contains a spot with an ID of 3 which possesses three states (0, 1 and 2); the script will advance the spot to the next of these three states by using an `IFELSE` command. Try executing it several times in a row.

### Example

```
{
   0 3 SETSPOTSTATE
} {
   3 GETSPOTSTATE 1 + 3 SETSPOTSTATE
} 2 3 GETSPOTSTATE == IFELSE
```

## SETSPOTSTATELOCAL

*state spotID* SETSPOTSTATELOCAL

This command functions just like SETSPOTSTATE, except that only the person executing the script will actually see the new state occur. Because this command does its work locally (i.e., on the client computer only), it changes the spot's state much more quickly than the non-local version. For this reason, this is the preferred way to do animations and effects that don't need to sync up exactly for all users. The following example assumes that the current room contains a spot with an ID of 3 that possesses three states (0, 1 and 2). The difference between this example and the preceding one (SETSPOTSTATE) is that in this case, the user who executes the script will be the only one who sees the spot change.

### Example

```
{
   0 3 SETSPOTSTATELOCAL
} {
   3 GETSPOTSTATE 1 + 3 SETSPOTSTATELOCAL
} 2 3 GETSPOTSTATE == IFELSE
```

## SHOWLOOSEPROPS

SHOWLOOSEPROPS

This command creates a list in the Log Window, providing the propID and location of all loose props in the room. This is useful, for example, if you want to write a script that automatically places chess pieces on a chess board: In authoring mode, determining the exact X and Y positions to place all these props by hand would be a tedious task. Instead of doing this the hard way, you can simply place the props in the desired positions on the screen, type /SHOWLOOSEPROPS into the client input box, and copy the listing from the *Log Window*. This command may also be executed from within a script. The listing in the Log Window will follow the format shown below:

```
1009 188 120 ADDLOOSEPROP
1013 108 178 ADDLOOSEPROP
1018 162 185 ADDLOOSEPROP
```

### Example

```
SHOWLOOSEPROPS
```

## SPOTDEST

*spotID* SPOTDEST

This function returns the DEST (destination) of the spot or door specified by *spotID*. Note that *Normal* spots may possess DEST fields, although unlike *Passages*, they require a scripted GOTOROOM in the ONSELECT handler to send the user there when selected. The following example assumes that the current room contains a door with an ID of 1, for which a DEST has been set:

### Example

```
1 SPOTDEST ITOA tempVar =
"Door number 1 leads to Room number " tempVar & SAY
```

---

NOTE – You might find it odd that a normal spot can contain a *DEST* it doesn't use, but consider this: if you place an integer value into a spot's *DEST* field (which may require editing the server script manually), you can then use *SPOTDEST* to refer to it, effectively providing a "room-level constant" (and you can do this for each normal spot in the room). Palace designers are always looking for places to store data without using globals or incurring too much memory overhead; this is one of 'em.

---

## SPOTNAME

*spotID* SPOTNAME

This function returns the name of the spot (or door) specified by *spotID*. The following example assumes that there is a spot (or door) with an ID of 1 in the current room, and that it has a name. The following example determines the names of all spots in the current room, and prints its output to the *Log Window*.

### Example

```
0 tempVar =
{
   "Spot " tempVar ITOA & "'s name is \"" & tempVar SPOTIDX SPOTNAME &
"\"." & LOGMSG
   tempVar ++
} { NBRSPOTS tempVar > } WHILE
```

### SPOTIDX

*number* SPOTIDX

This function returns the *spotID* of the spot specified by *number*. The following example determines the IDs of all spots in the current room, and prints its output to the *Log Window*.

#### Example

```
0 tempVar =
{
    "Spot " tempVar ITOA & "'s ID is " & tempVar SPOTIDX ITOA & LOGMSG
    tempVar ++
} { NBRSPOTS tempVar > } WHILE
```

### UNLOCK

*doorID* UNLOCK

This command is used by *Deadbolts* (BOLT commands) to unlock doors. Its counterpart is the LOCK command. The following example assumes that there is a lockable door with an ID of 1 in the current room.

#### Example

```
1 UNLOCK
```

## Paint commands and functions

Paint Commands always operate in the foreground layer of the Viewing Area; that is to say, "in front of" all graphics in the midground layer.

### LINE

*x1 y1 x2 y2* LINE

This command draws a line from point *x1,y1* to point *x2,y2*. The line is drawn in the current PENSIZE and PENCOLOR. The following example draws a line from the upper left corner of the Palace client viewing area to the user who triggered it.

#### Example

```
0 0 POSX POSY LINE
```

## LINETO

*x y* LINETO

This command draws a line from the current PENPOS to a point *x,y* away from the current PENPOS. The line is drawn in the current PENSIZE and PENCOLOR. The following example draws a diagonal line that goes 100 pixels to the right and 50 pixels upward, starting from the pen's current position.

### Example

```
100 -50 LINETO
```

## PAINTCLEAR

PAINTCLEAR

This command erases all painting/drawing from the screen, regardless of who put it there. You can do the same thing by double-clicking on the Detonator in the Painting Window.

### Example

```
PAINTCLEAR
```

## PAINTUNDO

PAINTUNDO

This command erases the last painting/drawing command or action performed. You can do the same thing by clicking once on the Detonator in the Painting Window.

### Example

```
PAINTUNDO
```

## PENBACK

```
PENBACK
```

This command moves the pen to the “back” of the foreground layer: any painting commands or actions subsequently performed will appear behind all avatars in the room (but they'll still be in front of any graphics in the midground layer). Any paint already on the screen is not affected. Note that you can do the same thing by clicking on the Layerer in the Painting Window.

### Example

```
PENBACK
```

## PENCOLOR

```
r g b PENCOLOR
```

This command sets the color of the pen: any painting commands or actions subsequently performed will appear in the specified color. You can do the same thing with the Palette in the Painting Window. The three arguments *r*, *g* and *b* represent the relative amounts of red, green and blue in the color, on a scale of 0 to 255 (where 0 0 0 yields black and 255 255 255 yields white). The following example sets the pen color randomly.

### Example

```
255 RANDOM tempR =
255 RANDOM tempG =
255 RANDOM tempB =
tempR tempG tempB PENCOLOR
```

## PENFRONT

```
PENFRONT
```

This command moves the pen to the “front” of the foreground layer: any painting commands or actions subsequently performed will appear in front of all avatars in the room (in the closest possible position to the user's face). Paint already on the screen is not affected. You can do the same thing by clicking on the Layerer in the Painting Window.

### Example

```
PENFRONT
```

## PENPOS

*x y* PENPOS

This command moves the pen to position *x y* on the screen, without drawing anything. The following example moves the pen to the user's position.

### Example

```
POSX POSY PENPOS
```

## PENSIZE

*number* PENSIZE

This command sets the pixel width of all lines drawn by the pen to *number* (an integer from 1 to 9): any painting commands or actions subsequently performed will create lines of this width. Paint already on the screen is not affected. You can do the same thing with the Line Sizer in the Painting Window. The following example paints a gradually-widening line across the Viewing Area.

### Example

```
30 150 PENPOS
1 PENSIZE
50 0 LINETO
2 PENSIZE
50 0 LINETO
3 PENSIZE
50 0 LINETO
4 PENSIZE
50 0 LINETO
5 PENSIZE
50 0 LINETO
6 PENSIZE
50 0 LINETO
7 PENSIZE
50 0 LINETO
8 PENSIZE
50 0 LINETO
9 PENSIZE
50 0 LINETO
```

### PENTO

*x y* PENTO

This command moves the pen to a position *x y* relative to the current PENPOS, without drawing anything. The following example draws a line 100 pixels long, moves the pen via PENTO, and continues drawing.

#### Example

```
0 150 100 150 LINE
50 50 PENTO
100 0 LINETO
```

## Sound commands and functions

Prior to version 2.0 of the Palace client, audio files could not be sent across the network. For WAV or MIDI files to be heard, they had to exist on the user's hard disk, in the Sounds folder. A few users are still running around with this limitation, and sounds should therefore be made available via a Web Page, public FTP directory, or some other means.

Version 2.0 and greater allows clients to receive sounds as downloads from the server. To be sent out, the audio files in question must be placed in the Pictures folder on the server's computer.

### MIDIPLAY

"*fileName*" MIDIPLAY

This command causes the MIDI file "*fileName*" to be played. The following example assumes that there is a MIDI file called "testme.mid" in the /Palace/Media/*YourPalaceName*/Sounds folder.

#### Example

```
"testme.mid" MIDIPLAY
```

### MIDISTOP

MIDISTOP

This command causes the currently-playing MIDI file to immediately stop. (PC only)

#### Example

```
MIDISTOP
```

### SOUND

```
"fileName" SOUND
```

This command causes the file "*fileName*" to be played for all users in the room. It is functionally identical to typing ")*filename*" SAY into the *Input Box*.

#### Example

```
"teehee" SOUND

"Song.midi" SOUND
```

## Flow commands and functions

### ALARMEXEC

```
{ atomlist } ticks ALARMEXEC
```

This command schedules an *atomlist* to be executed at a pre-specified time (after so many "ticks" have elapsed). The user's subjective duration of a "tick" depends on the speed of both the client and server as well as network load at the moment, but is considered to be $1/60^{th}$ of a second. The following example waits ten seconds before finishing.

#### Example

```
"Don't you hate..." SAY
{ "waiting?" SAY } 600 ALARMEXEC
```

### BREAK

```
BREAK
```

This command breaks out of a WHILE or FOREACH loop. The following example sets up a FOREACH loop causing a sentence to be spoken one word at a time, but halts after the fourth word due to a BREAK command.

### Example

```
0 tempVar =
{
   tempStr =
   tempVar ++
   {
      tempStr SAY
   } {
      BREAK
   } 5 tempVar > IFELSE
} [ "I" "will" "never" "finish" "speaking" "this" "sentence" ] FOREACH
```

## EXEC

*atomlist* EXEC

This command executes an atomlist. It can be used in combination with the DEF command (see below) to execute a "user-defined function." Note that unless the function was defined in the same handler, it must be made GLOBAL.

### Example

```
{ "Hello world!" SAY } definedFunction =
definedFunction EXEC
```

## EXIT

EXIT

This command stops the currently-running script. It is useful for breaking out of looping errors that might otherwise flood the server or lock up the client. The following example bounces you around the screen randomly. It would continue to do so forever, except for the imbedded EXIT command.

The following script is likely get you killed for flooding if a death penalty exists on the server where it is executed. It is recommended that you turn the death penalty for flooding OFF before attempting to use this script.

### Example

```
400 150 SETPOS
{
   51 RANDOM 25 - tempX =
   51 RANDOM 25 - tempY =
   tempX tempY MOVE
   { EXIT } POSX 256 < IF
} { 1 } WHILE
```

## FOREACH

```
{ atomlist } [ array ] FOREACH
```

This command executes *atomlist* once for each item in *array*. Before executing the atomlist, each item in the array is pushed onto the stack. The atomlist should be something that pops these items off the stack and does something with them, as the following example indicates.

### Example

```
{ SAY } [ "Ready" "Steady" "Go!" ] FOREACH
```

## IF

```
{ atomlist } condition IF
```

This command can be used to create a conditional statement: if the condition evaluates to TRUE (non-zero), *atomlist* will be executed. If the condition evaluates to FALSE ("0") it will not. Any *operator (*or logical series of operators) may be used to describe the condition being checked for (see the *Operators* section). The following example rolls a pair of imaginary dice, looking for a lucky total of 7.

### Example

```
6 RANDOM 1 + tempVar =
6 RANDOM 1 + tempVar + tempVar =
"I rolled a " tempVar ITOA & SAY
{ "I'm a winner!" SAY } tempVar 7  == IF
```

## IFELSE

```
{ trueAtomList } { falseAtomList } condition IFELSE
```

This command can be used to create mutually-exclusive conditional statements: if the condition evaluates to TRUE (non-zero), the *trueatomlist* will be executed. Otherwise, the *falseatomlist* will be executed. Warning: a very common Iptscrae bug is to use IF when you really mean IFELSE. The following example randomly determines two numbers from 1 to 100 and compares them.

### Example

```
100 RANDOM 1 + tempVar1 =
100 RANDOM 1 + tempVar2 =
{
   tempVar1 ITOA "is less than or equal to " & tempVar2 ITOA & SAY
} {
   tempVar1 ITOA "is greater than " & tempVar2 ITOA & SAY
} tempVar1 tempVar2 <= IFELSE
```

## RETURN

This command breaks out of an atomlist.

### Example

```
{
   "This line will be executed" SAY
   RETURN
   "This line will not" SAY
}
```

## SETALARM

*futureTicks spotID* SETALARM

This command schedules the ALARM event for the spot *spotID*; this event will occur *futureTicks* in the future. A "tick" is $1/60^{th}$ of a second. The following example assumes that there is a spot with an ID of 1 in the current room, and that this spot possesses an ON ALARM handler. When executed, the code will cause the spot's ON ALARM handler to be triggered ten seconds later.

### Example

```
600 1 SETALARM
```

## WHILE

{ *atomlist* } { *condition* } WHILE

This command creates a loop in which *atomlist* will continue iterating until *condition* evaluates to TRUE (non-zero). Any *operator* (or logical series of operators) may be used to describe the condition being checked for. The following example will continue counting until *tempVar* equals 5.

### Example

```
{
   tempVar 1 + tempVar =
   tempVar ITOA SAY
} { tempVar 5 < } WHILE
```

## General commands and functions

### ; <*comment*>

```
; comment
```

A semicolon (;) at the beginning of a line tells the program to ignore everything up to the next carriage return; it is used to insert comments into your scripts. Commenting your code is considered good programming practice in general, and comes in especially handy when you return to a script you haven't looked at in a long time. Note that comments placed outside of the script proper (i.e. outside of the `SCRIPT...ENDSCRIPT` block) will not be saved by the server.

### Example

```
"This line will be executed" SAY
; "This line will not" SAY
```

### ADDLOOSEPROP

```
propID x y ADDLOOSEPROP
"propName" x y ADDLOOSEPROP
```

This command adds a loose prop to the *Viewing Area*. The prop can be specified by *propID* or by *propName*. In the first case (*propID*), you must either know the *propID* already or use a command to retrieve it (`TOPPROP` or `USERPROP` will accomplish this). In the second case (*propName*), remember to place the name of the prop in quotes. Note that the prop specified must exist in either the client's propfile or the server's propfile; otherwise the command will have no effect.

### Example 1 (by *propName*)

```
"halo" 100 200 ADDLOOSEPROP
```

### Example 2 (by *propID*)

```
1016 100 200 ADDLOOSEPROP
```

### Example 3 (duplicating a worn prop)

```
"halo" DONPROP
TOPPROP 100 200 ADDLOOSEPROP
```

## ARRAY

*number* ARRAY

This command creates an *array* containing *number* elements. This array will contain zeros when first created. Data may be stored via the PUT command. The following example creates an empty array of ten elements and names it "myArray."

### Example

```
10 ARRAY myArray =
```

## ATOI

"*string*" ATOI

This function (“Ascii TO Integer”) converts a character string to a number. Strings – even numerals spoken as text strings — must be converted to integers before you can do math with them. The following example causes any integer spoken (all by itself) to be multiplied by ten before it appears in the user's cartoon balloon.

### Related commands

ITOA

### Example

```
{
   CHATSTR ATOI yourNumber =
   yourNumber 10 * myNumber =
   myNumber ITOA CHATSTR =
} CHATSTR "^[0-9]$" SUBSTR == IF
```

## BEEP

```
BEEP
```

This commands causes the system beep sound to be heard on the user's computer. The following example causes this sound to be heard whenever the user says "beep" (even if it's imbedded in another word, or capitalized).

### Example

```
{
   BEEP
} CHATSTR LOWERCASE "beep" SUBSTR == IF
```

## CLEARLOOSEPROPS

```
CLEARLOOSEPROPS
```

This command clears all loose props from the room. The following example clears the room of loose props whenever the user says "be gone" (without the quotes).

### Example

```
{
   CLEARLOOSEPROPS
} CHATSTR "be gone" == IF
```

## CLIENTTYPE

```
CLIENTTYPE
```

This command pushes "WINDOWS32", "MAC68K", "MACPPC", "TPV", or "unknown" onto the stack, depending on which client is running the script. The following example tells the user which client he/she is using whenever the user asks "which client".

### Example

```
{
CLIENTTYPE SAY
} CHATSTR "which client" == IF
```

## DATETIME

```
DATETIME
```

This function returns the number of seconds that have passed since January 1st, 1970 (Pacific Standard Time). Translating this number to a Julian date is left as an exercise for the reader (it's tough, but quite do-able).

### Example

```
"The current DATETIME is " DATETIME ITOA & SAY
```

## DEF

```
{ atomlist } symbol DEF
```

This command is used to create your own custom functions. Note that *symbol* must be declared GLOBAL if you want it to be recognized by any event handlers other than the one it's defined in; it will also have to be declared GLOBAL there (i.e., in the other handlers). As long as you adhere to this rule, your function can be executed in any room in your Palace.

### Example 1 (defined and executed within the same handler)

```
{
   "@50,50! " USERNAME & " has entered the room!" & SAY
} myFunction DEF
myFunction EXEC
```

### Example 2 (defined ON ENTER, executed ON SELECT)

```
ON ENTER {
   myFunction GLOBAL
   {
      "@50,50! " USERNAME & " is the greatest!" & SAY
   } myFunction DEF
}

ON SELECT {
   myFunction GLOBAL
   myFunction EXEC
}
```

## DELAY

*number* DELAY

This command causes a delay affecting *all activity on the client* — events, alarms, queued commands and even prop animations — for the duration specified by *number*. Delay times are measured in *ticks* (1/60 of a second) Note that SETALARM and ALARMEXEC are preferred since they don't lock up all processes on the client, although use of the DELAY command might be appropriate in a game, or as a penalty for breaking some house rule. The following example suggests one possible use:

### Example

```
ON ALARM {
   foulMouthFlag GLOBAL
   {
      USERNAME ", you have been flagged for swearing. You have been
sentenced to 30 seconds of dead time.  If you persist you will be kicked
off the server." & LOCALMSG
      1800 DELAY
   } foulMouthFlag 1 == IF
```

## DIMROOM

*number* DIMROOM

This command allows you to "dim the lights" in the room, decreasing the luminance of all visible graphics and props. The natural state of a room is 100% lit. By specifying an integer (*number*) lower than 100 and higher than 0, you can set the lighting to any desired percentage. Note that if *number* equals 100 or 0 (zero), the room will be made 100% lit again. The following example fades the lights down and then brings them back up again.

### Example

```
{
   lightingNow =
   lightingNow DIMROOM
} [ 90 80 70 60 50 40 30 20 10 0 10 20 30 40 50 60 70 80 90 100 ]
FOREACH
```

## DUP

```
DUP
```

This command duplicates the top element on the stack. The following example shows an easy way to multiply an expression by itself; in this case, ( x + 1 ) * ( x + 1 ).

### Example

```
x 1 + DUP *
```

## GET

```
array index GET
```

This function gets item number *index* from *array*. Note that the elements of an array are numbered from 0 (zero) to (number of elements minus 1). The array may be specified directly (element by element) or by reference to its *symbol* (name).

### Related commands

```
PUT
```

### Example 1 (referring to array directly)

```
[ "alpha" "beta" "gamma" "delta" "epsilon" ] 5 RANDOM GET tempStr =
tempStr SAY
```

### Example 2 (referring to array by its *Symbol*)

```
[ "alpha" "beta" "gamma" "delta" "epsilon" ] myArray =
myArray 5 RANDOM GET tempStr =
tempStr SAY
```

## GLOBAL

*symbol* GLOBAL

This command declares *symbol* as a global variable, which allows it to be shared among event handlers. *The GLOBAL command must used in EVERY event handler and ALARMEXEC in which the global symbol is used, even in the same room.* It is good practice to declare your globals as soon as you enter the handler in which they will be used; this makes it easy to remember which ones you need and what you were doing.

### Example

```
ON ENTER {
   first_variable GLOBAL
   other_variable GLOBAL
   "Hello" first_variable =
   "World" other_variable =
   60 ME SETALARM
}
ON ALARM {
   first_variable GLOBAL
   other_variable GLOBAL
   first_variable SAY
   other_variable SAY
}
```

***About nested globals.*** GLOBAL is an executable command. After it is executed, the variable will operate as a global value for the rest of the script.

For non-programmers, the following examples should make all of this a bit more clear; compare the level of indentation at which the *myGlobal GLOBAL* statement appears in each example.

### Example 1 (this works):

```
{ myGlobal GLOBAL "I rolled a " myGlobal ITOA & SAY } 30 ALARMEXEC
myGlobal GLOBAL
6 RANDOM 1 + myGlobal =
```

### Example 2 (this doesn't):

```
{ "I rolled a " myGlobal ITOA & SAY myGlobal GLOBAL } 30 ALARMEXEC
myGlobal GLOBAL
6 RANDOM 1 + myGlobal =
```

## GREPSTR

*string* "*pattern*" GREPSTR

This function performs a case-sensitive search for the specified *pattern* within the specified *string*, and returns true (1) if the pattern is found. It may be placed in the INCHAT or OUTCHAT handler to operate directly on CHATSTR. Note that this command uses UNIX grep-style syntax; i.e., any character matches itself, unless it is one of the following special characters:

| Character | Matches |
|---|---|
| . | Any character |
| \ | The character following it |
| [<*set*>] | One of the characters in the set, for example:<br>[aeiou] matches any vowel (except y)<br>[A-Za-z] matches any alphabetic character<br>[^0-9] matches anything *but* the characters 0-9 |
| * | Any pattern followed by *matches *zero or more* instances of the pattern. |
| + | Same as * except it matches *one or more* instances of the pattern. |
| () | Used to tag sub-expressions that can be referred to in a GREPSUB command or subsequently, using the special symbols $1 through $9. |
| ^ | A pattern beginning with ^ must start at the beginning of the line. |
| $ | A pattern ending with $ must match to the end of the line. |

Special GREPSTR Characters

### Example 1 (using IF to check for existence of *string*)

```
{
   "I hate $1" GREPSUB ROOMMSG
} CHATSTR LOWERCASE "^i like (.*)$" GREPSTR == IF
```

### Example 2 (using WHILE to check for all instances of *string*)

```
{
   "$1darn$2" GREPSUB CHATSTR =
} { CHATSTR LOWERCASE "(.*)damn(.*)" GREPSTR } WHILE
```

NOTE – These examples use the special Symbols *$1* and *$2*, allowing the GREPSUB command to use the text picked up by the wildcards *(.*)* in the GREPSTR command. Up to nine such symbols may be used in a single GREPSTR-GREPSUB structure ($1 through $9). For more information on regular expressions in general, see *Mastering Regular Expressions* by Jeffrey Friedl. Copyright 1997, O'Reilly and Associates

## GREPSUB

"*replacementPattern*" GREPSUB

This function is executed in conjunction with a GREPSTR command: it locates specified spaces within a string, and fills them with any text that was "captured" by the GREPSTR. The replacement pattern uses the special *Symbols* $1 through $9 to refer to these captured character strings.

Example (the "Elmer Fudd" script from "The Moor"):

```
{
   "$1w$2" GREPSUB CHATSTR =
} { CHATSTR "(.*)[lr]([aeiouy][^ .].*)" GREPSTR } WHILE
```

## IPTVERSION

IPTVERSION

This command pushes the current Iptscrae version number (currently 1) into the stack. The following example would tell another user which version of Iptscrae you are currently using.

### Example

```
"I'm currently using Iptscrae version" IPTVERSION ITOA & SAY
```

## ITOA

```
ITOA
```

This function (“Integer TO Ascii”) takes a numeric variable from the top of the stack, converts it to a character string, and places it back on the stack. Numerals must be converted to character strings before you can text-based commands (such as SAY) on them. For instance,

```
WHOME SAY
```

fails, since the `WHOME` function puts an integer (your *`userID`*) on the stack, while the `SAY` command is looking for a character string. The example below shows how you can use ITOA to remedy this.

### Example

```
WHOME ITOA SAY
```

### Related commands

ATOI

## LAUNCHAPP

```
appName LAUNCHAPP
```

This command tells the Palace software to look for the program called *`appName`* in the user's *PlugIns* folder, and launch it. It is used to launch Palace-compatible games and other software known as "Palace Plugins." The *PalacePresents Viewer*, which is distributed with the Palace User Software client, is a good example. Note that the full name of the program must be used (including file extensions, if any exist). The full path does *not* need to be specified, as this command applies to the *Plugins* folder only.

### Example

```
ON ENTER {
   "PalacePresents Viewer.dll" LAUNCHAPP
}
```

## LENGTH

*array* LENGTH

This function returns the number of elements in *array*. The array may be specified directly (element by element) or by reference to its *symbol* (name).

### Example 1 (referring to array directly)

```
[ "alpha" "beta" "gamma" "delta" "epsilon" ] LENGTH ITOA SAY
```

### Example 2 (referring to array by its *Symbol*)

```
[ "alpha" "beta" "gamma" "delta" "epsilon" ] myArray =
myArray LENGTH ITOA SAY
```

## LOGMSG

"*message*" LOGMSG

This command causes *message* to appear in the user's *Log Window*. Like CHAT, SAY and other message-related commands, it deals with character strings rather than integers. This command is primarily useful for debugging, since many users keep their Log Windows closed (guests and new users may be completely unaware that this window exists at all).

### Example

```
"This is a message in your Log Window." LOGMSG
```

## LOWERCASE

"*string*" LOWERCASE

This function converts a character string to lowercase.

### Example

```
"I WANT TO SHOUT, BUT I CAN'T!" LOWERCASE SAY
```

## MOUSEPOS

```
MOUSEPOS
```

This function returns the current X (horizontal) and Y (vertical) coordinates of the cursor. The X coordinate is put on the stack first, then the Y coordinate; they must be retrieved separately. This means you'll need two ITOA commands to get the mouse position, not just one. If you want get them in the traditional order (X, then Y), issue a `SWAP` command before getting them from the stack. The example below shows how to do this, sending the output to the Log Window.

### Example

```
MOUSEPOS SWAP ITOA LOGMSG ITOA LOGMSG
```

## OVER

```
OVER
```

This command is the same as `1 PICK`. See the `PICK` command below for a full description.

## PICK

```
PICK
```

The command `n Pick` reaches down *n* stack items and copies that item to the top of the stack. `0 PICK` is the same as `DUP`, and `1 PICK` is the same as `OVER`.

## POP

```
POP
```

This command pops the top element off the stack and discards it.

### Example

```
"none" "one" "two" "three" "four"
POP POP POP POP
SAY
```

## PUT

*data array index* PUT

This command is used to put a *data* element into an *array*, in the position indicated by *index*. If the data is a string (as opposed to an integer), it must be encased in double quotes. Note that the elements of an array are numbered from 0 (zero) to (number of elements minus 1). The following example creates an array of letters, places the word "foo" in a random position within it, and prints the results in the *Log Window*.

### Related commands

GET

### Example

```
[ "a" "b" "c" "d" "e" ] myArray =
"foo" myArray 5 RANDOM PUT
{ LOGMSG } myArray FOREACH
```

## RANDOM

*number* RANDOM

This function puts a random integer on the stack, from 0 to (*number* minus 1). The following example shows you how to roll a die (i.e., how to generate a random integer from 1 to 6):

### Example

```
6 RANDOM 1 + tempVar =
"I rolled a " tempVar ITOA & "!" & SAY
```

## ROOMID

ROOMID

This function returns the ID of the current room (as an integer).

### Example

```
"The ID of this room is \"" ROOMID ITOA & ".\"" & SAY
```

## ROOMNAME

```
ROOMNAME
```

This function returns the name of the current room.

### Example

```
"The name of this ROOM is \"" ROOMNAME & ".\"" & SAY
```

## SAYAT

```
"message" x y SAYAT
```

This command causes *message* to appear as though it was spoken from position *x,y*. This is also known as "spoofing."

### Example

```
512 RANDOM tempX =
384 RANDOM tempY =
tempX ITOA " by " & tempY ITOA tempStr =
tempStr tempX tempY SAYAT
```

## SERVERNAME

```
SERVERNAME
```

This is the name of the server as specified in the Server **Preferences** dialog. You can't change the *servername* from a script.

### Example

```
"Hello and welcome to " SERVERNAME & "!" & LOCALMSG
```

## STACKDEPTH

```
STACKDEPTH
```

This command pushes the number of items on the stack to the top of the stack.

## STATUSMSG

```
"message" STATUSMSG
```

This command causes *message* to be displayed in the status bar (just above the Input Box) on the Macintosh and Windows clients. On The Palace Viewer, this command causes *message* to be displayed in the center of the **Grapic** window. It can be annoying; use it sparingly.

### Example

```
"What a cool STATUSMSG!" STATUSMSG
```

## STRINDEX

```
"str" "sp" -- off
```

This command pushes the offset of the string "sp" in "str" or pushes -1 if "sp" does not appear in "str".

## STRLEN

```
"str" -- len
```

This command pushes the length of the string to the top of the stack.

## STRTOATOM

```
"string" STRTOATOM
```

This command turns a character string into an executable *atomlist*.

### Example

```
"WHOME WHONAME SAY" STRTOATOM EXEC
```

## SUBSTR

*string "stringpattern"* SUBSTR

This function searches *string* for *stringpattern* (not case-sensitive) and returns 1 if it is found; otherwise it returns a 0.

### Example

```
ON OUTCHAT {
   CHATSTR LOWERCASE tempStr =
   {
      "The letters ae appeared in that sentence." CHATSTR =
   } {
      "The letters ae did not appear in that sentence." CHATSTR =
   } tempStr "ae" SUBSTR IFELSE
}
```

## SUBSTRING

"str" off len -- "f"

This command pushes the substring of "str" at offset off for length len. Negative values of len mean the rest of the string at offset off. Negative values of offset are an error. The example below says "I like roses".

### Example

```
"I like violets" 0 7 SUBSTRING "roses" & SAY
```

## SWAP

SWAP

This command swaps the top two elements on the stack.

### Example

```
MOUSEPOS
 "The current MOUSEPOS is " ITOA & " (Y) " & ITOA & "(X)." & SAY
MOUSEPOS
 "But if I do a SWAP I get " SWAP ITOA & " (X) followed by " & ITOA &
"(Y)." & SAY
```

## TICKS

```
TICKS
```

This function returns the current time (on the client) in *ticks*. The user's subjective duration of a "tick" depends on the speed of both the client and server as well as the network load at the moment, but is considered to be about $1/60^{th}$ of a second.

### Example

```
"Current TICKS = " TICKS ITOA & "." & SAY
```

## TOPTYPE

```
TOPTYPE
```

This command pushes a number indicating the type of the top item on the stack (the top item remains on the stack). The codes are

0 - Internal Error/unknown/stack empty
1 - number
2 - symbol (variable name)
3 - AtomList
4 - String
5 - ArrayMark (a [ character)
6 - Array

## UPPERCASE

```
"string" UPPERCASE
```

This function converts *string* to uppercase. The following example causes everything to be spoken that way (in Cyborg.IPT it will operate on everything the user says; in a room script it will operate on everything said by anyone in the room).

### Example

```
ON OUTCHAT {
   CHATSTR UPPERCASE CHATSTR =
}
```

## VARTYPE

```
VARTYPE
```

This command is like TOPTYPE, unless the top item on the stack is a symbol (variable). If the top item is a symbol, the type of the current value of the variable is pushed. For variable types, see TOPTYPE on page 80

### WHOPOS

```
"userName" WHOPOS
userID WHOPOS
```

This function (in either of its forms) returns the `x,y` position of the user specified. The following example causes the user to speak his/her coordinates.

#### Example

```
WHOME WHOPOS
ITOA tempY = ITOA tempX =
"My current WHOPOS is " tempX & " by " & tempY & "." & SAY
```

# Operators

***Operators*** are functions that perform traditional mathematical and logical operations. If you want to add, subtract, multiply, divide, or perform higher math tricks, you'll need these. You'll also need to use operators for setting and comparing the values of ***symbols*** and other data; the most commonly-used operators are "equal" ( == ) and "let equal" ( = ). All operators make use of the ***stack***, where they deposit the results of their calculations.

## Standard Operators

***Standard operators*** allow you to perform all basic mathematical operations in RPN format. This section describes the syntax and use of all standard operators.

Each of the following examples is preceded by the equivalent statement in infix format (in parentheses).

### *valueA valueB +*

Adds the two values and pushes the result onto the stack. If the two values are strings, this operator concatenates the two strings and pushes the result onto the stack.

#### Example ( 2 + 3 )

```
2 3 +
```

### *valueA valueB -*

Subtracts *valueb* from *valuea* and pushes the result onto the stack.

Example ( 3 - 2 )

```
3 2 -
```

### *valueA valueB **

Multiplies the two values and pushes the result onto the stack.

Example ( 2 * 3 )

```
2 3 *
```

### *valueA valueB /*

Divides *valueA* by *valueB* and pushes the (integer) result onto the stack. The remainder is discarded (e.g., the example below yields a result of “1”).

Example ( 3 / 2 , discard remainder)

```
3 2 /
```

### *valueA valueB %*

Divides *valuea* by *valueb* and pushes the remainder (modulo) onto the stack. The result itself is discarded (e.g., the example below yields a result of “5”).

Example ( 3 / 2 , keep remainder only)

```
3 2 %
```

### *value value ==*

Pushes 1 onto the stack if the two values are equal, 0 otherwise. If the two values are strings, this operator does a case-insensitive string comparison. The example below returns 0 (false).

Example ( 2 = 3 )

```
2 3 ==
```

### *value value* !=
### *value value* <>

These two operators are synonymous. They push 1 onto the stack if the two specified values are not equal, otherwise they return 0. The examples below both return 1 (true). Values specified for this function may be either integers or strings. These operators are case-insensitive when comparing strings.

Example 1 ( 2 <> 3 )

```
2 3 <>
```

Example 2 ( "a" != "b" )

```
"a""b" !=
```

### *valueA valueB* <

Pushes 1 onto the stack if *valueA* is less than *valueB*; otherwise it returns a 0. Values specified for this function may be either integers or strings. This operator is case-insensitive when comparing strings. The example below returns a 1 (true).

Example ( 2 < 3 )

```
2 3 <
```

### *valueA valueB* >

Pushes 1 onto the stack if *valueA* is greater than *valueB*; otherwise it returns a 0. Values specified for this function may be either integers or strings. This operator is case-insensitive when comparing strings. The example below returns a 0 (false).

Example ( 2 > 3 )

```
2 3 >
```

### *valueA valueB* <=

Pushes 1 onto the stack if *valueA* is less than or equal to *valueB*; otherwise it returns a 0. Values specified for this function may be either integers or strings. This operator is case-insensitive when comparing strings. The example below returns a 1 (true).

Example ( 2 <= 3 )

```
2 3 <=
```

### *valueA valueB* >=

Pushes 1 onto the stack if *valueA* is greater than or equal to *valueB*; otherwise it returns a 0. Values specified for this function may be either integers or strings. This operator is case-insensitive when comparing strings. The example below returns a 0 (false).

#### Example ( 2 >= 3 )

```
2 3 >=
```

### *valueA valueB* AND

Pushes 1 onto the stack if the two values are both true (non-zero), otherwise it returns a 0. The example below returns a 0 (false).

#### Example (0 AND 1)

```
0 1 AND
```

### *valueA valueB* OR

Pushes 1 onto the stack if either of the two values is true (non-zero), otherwise it returns a 0. The example below returns a 1 (true).

#### Example ( 0 OR 1 )

```
0 1 OR
```

### *value* NOT
### value !

These two operators are synonymous. They push the logical inverse of *value* onto the stack (e.g., 1 if value is equal to zero, 0 otherwise). The `NOT` of 0 is 1. The `NOT` of 1 (or any non-zero integer) is 0.

#### Example ( NOT 1 )

```
1 NOT
```

### *value* SINE

Pushes the trigonometric sine of *value* multiplied by 1000. Remeber that *value* is in degrees.

### Example (sine 30)

```
30 SINE ITOA SAY
```

### *value* COSINE

Pushes the trigonometric cosine of*value* multiplied by 1000. Remember that *value* is in degrees.

### Example (cosine 30)

```
30 COSINE ITOA SAY
```

### *value* TANGENT

Pushes the trigonometric tangent of *value* multiplied by 1000. Remember that *value* is in degrees.

### Example (tangent 45)

```
45 TANGENT ITOA SAY
```

### *value* SQUAREROOT

Pushes the integer part for the square root of *value*. The folowing example displays the message "4".

### Example (squareroot 20)

```
20 SQUAREROOT ITOA SAY
```

### *string1 string2* &

Concatenates *string1* and *string2*, and pushes the result onto the stack. The example below creates a complete sentence out of two parts.

#### Example ( “Are we having fun yet?” )

```
"Are we " "having fun yet?" &
```

# Assignment Operators

Assignment operators are shortcuts for commonly-used sets of functions. For example, to add 4 to X, you could say:

```
x 4 + x =
```

This works just fine (“take X and 4 and add them, and then let X equal that”), but using the += assignment operator is easier, because it allows you to combine the addition and the let equal operations into a single function that does exactly the same thing:

```
4 x +=
```

Assignment operators exist for all basic math operations (addition, subtraction, multiplication, integer and modulo division), and for the common operations of incrementing and decrementing (adding or subtracting 1). So instead of saying

```
x 1 + x =
```

you can say

```
x ++
```

The following entries explain the syntax and effects of all the assignment operators.

### *value symbol* +=

Adds `value` to `symbol` and assigns the total to `symbol`.

#### Example (let x = x + 3)

```
3 x +=
```

In this operation, you may also specify a string as the value. In this case the operation appends the string "a" to the string variable x, then assigns the result to x.

#### Example (let x=x+"a")

```
"a" x +=
```

### *value symbol* -=

Subtracts *value* from *symbol* and assigns the total to *symbol*.

Example (let x = x - 3)

```
3 x -=
```

### *value symbol* *=

Multiplies *value* by *symbol* and assigns the total to *symbol*.

Example (let x = x * 3)

```
3 x *=
```

### *value symbol* /=

Divides *symbol* by *value* and assigns the total (integer) to *symbol*.

Example (let x = x / 3, rounded down)

```
3 x /=
```

### *value symbol* %=

Divides *symbol* by *value* and assigns the integer remainder (modulo) to *symbol*.

Example (let x = the remainder of x / 3)

```
3 x %=
```

### *symbol* ++

Adds 1 to the value of *symbol*.

Example (let x = x + 1)

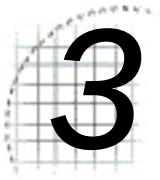
```
x ++
```

### *symbol* --

Subtracts 1 from the value of *symbol*.

Example (let x = x - 1)

*x* - -



# Quick Reference

The following table is a brief summary of the Iptscrae commands and their meanings.

| Word | Stack | Synopsis |
|---|---|---|
| ! | a -- bool | Pushes 1 if a is zero, else 0 |
| != | a b -- bool | Pushes 1 if a is zero, else 0 |
| # | -- | Rest of line is a comment |
| % | a b -- m | Pushes the remainder (modulus) of a/b |
| %= | v sym -- | Divides contents of sym by v and stores remainder in sym |
| & | "a" "b" -- "ab" | Pushes the concatenation of a and b |
| * | a b -- p | Pushes the product of a*b |
| *= | v sym -- | Multiplies v by the contents of sym and stores result in sym |
| + | a b -- s | Pushes the sum of a+b |
| + | "a" "b" -- "ab" | Pushes the concatenation of a and b |
| ++ | sym -- | Adds 1 to the contents of sym and stores result in sym |
| += | v sym -- | Adds v to the contents of sym and stores result in sym |
| += | "s" sym -- | Appends "s" to the string value of sym and stores result in sym |
| - | a b -- d | Pushes the difference a-b |

| Word | Stack | Synopsis |
|---|---|---|
| -- | sym -- | Subtracts 1 from the contents of sym and stores result in sym |
| -= | v sym -- | Subtracts v from the contents of sym and stores result in sym |
| / | a b -- id | Pushes the integer dividend of a/b |
| /= | v sym -- | Divides contents of sym by v and stores the result in sym |
| ; | -- | Rest of line is a comment |
| < | a b -- bool | Pushes 1 if a less than b, else 0 |
| < | "a" "b" -- bool | Pushes 1 if a less than b, else 0 |
| <= | a b -- bool | Pushes 1 if a less than or equal b, else 0 |
| <= | "a" "b" -- bool | Pushes 1 if a less than or equal b, else 0 |
| <> | a b -- bool | Pushes 1 if a not equal b, else 0 |
| <> | "a" "b" -- bool | Pushes 1 if a not equal b, else 0 |
| = | v sym -- | Stores v in location sym |
| == | a b -- bool | Pushes 1 if a equals b, else 0 |
| == | "a" "b" -- bool | Pushes 1 if a equals b, else 0 |
| > | a b -- bool | Pushes 1 if a greater than b, else 0 |
| > | "a" "b" -- bool | Pushes 1 if a greater than b, else 0 |
| >= | a b -- bool | Pushes 1 if a greater than or equal b, else 0 |
| >= | "a" "b" -- bool | Pushes 1 if a greater than or equal b, else 0 |
| ADDLOOSEPROP | propID x y -- | Place propID at x,y |
| ADDLOOSEPROP | "name" x y -- | Place prop name at x,y |
| ALARMEXEC | {al} ticks -- | Run atomlist al after ticks 1/60 seconds |
| AND | a b -- bool | Pushes 1 if both a and b are non-zero |
| ARRAY | n -- [ar] | Allocate array ar of n elements |
| ATOI | "str" -- n | Converts str to an integer or zero |
| BEEP | -- | Sound system beep |

| Word | Stack | Synopsis |
|---|---|---|
| BREAK | -- | Exit from a WHILE or FOREACH loop |
| CHAT | | deprecated, use SAY |
| CLEARLOOSEPROPS | -- | Clear all loose props in room |
| CLEARPROPS | | deprecated, use NAKED |
| CLIENTTYPE | -- "type" | Pushes the type of the client, e.g. "TPV" |
| COSINE | degrees -- sin | Pushes cosine(degrees)*1000 |
| DATETIME | -- t | Pushes seconds since 1/1/1970 |
| DEF | {al} sym -- | Define atomlist al as symbol sym |
| DELAY | n -- | Stop client for n seconds |
| DEST | -- roomID | Pushes destination roomID of the ME door or 0 for Cyborg |
| DIMROOM | n -- | Dim room to n% of fully lit |
| DOFFPROP | -- | Removes last-worn prop |
| DONPROP | propID -- | Add prop propID |
| DONPROP | "name" -- | Add prop name |
| DOORIDX | n -- doorID | Pushes doorID of door number n |
| DROPPROP | x y -- | Put last-worn prop at x,y |
| DUP | n -- n n | Duplicate top of stack |
| EXEC | {al} -- | Execute atomlist al |
| EXIT | -- | Stop the currently executing script |
| FOREACH | {al} [a] -- | Run al for each element of a |
| GET | [a] n -- v | Pushes element n from array a onto stack |
| GETSPOTSTATE | spotID -- n | Pushes the state n of spotID |
| GLOBAL | sym -- | Declares symbol sym to be global scope |
| GLOBALMSG | "msg" -- | Sends msg to everyone on the server |
| GOTOROOM | roomID -- | Moves user to room roomID |
| GOTOURL | "url" -- | Moves user or browser to url |
| GOTOURLFRAME | "url" "frame" -- | Moves user or browser to url |

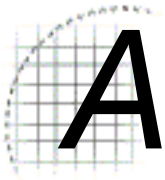
| Word | Stack | Synopsis |
|---|---|---|
| GREPSTR | "s" "p" -- bool | Greps s for pattern p, pushes 1 if found else 0 |
| GREPSUB | "rep" -- "s" | Replaces values in rep from GREPSTR and pushes result |
| HASPROP | propID -- bool | Pushes 1 if user has prop, else 0 |
| HASPROP | "name" -- bool | Pushes 1 if user has prop, else 0 |
| ID | -- id | Pushes spotID/doorID executing script or 0 for Cyborg |
| IF | {al} bool -- | Run al if bool is not zero |
| IFELSE | {tal} {fal} bool -- | If bool not zero run tal, otherwise fal |
| INSPOT | spotID -- bool | Pushes 1 if user in within spot spotID, else 0 |
| IPTVERSION | -- ver | Pushes the version of the Iptscrae language supported |
| ISGOD | -- bool | Pushes 1 if user is an owner, else 0 |
| ISGUEST | -- bool | Pushes 1 if user is guest, else 0 |
| ISLOCKED | doorID -- bool | Pushes 1 if doorID is locked |
| ISWIZARD | -- bool | Pushes 1 if user is an operator or owner, else 0 |
| ITOA | n -- "S" | Converts n to a string |
| KILLUSER | userID -- | Forces user userID off server |
| LENGTH | [a] -- n | Pushes the number of elements in a |
| LINE | x1 y1 x2 y2 -- | Draws from absolute x1,y1 to x2,y2 |
| LINETO | x y -- | Draws from penpos relative x,y |
| LOCALMSG | "msg" -- | Sends msg to user running script (only) |
| LOCK | doorID -- | Locks doorID |
| LOGMSG | "msg" -- | Puts msg in client log |
| LOWERCASE | "S" -- "s" | Converts upper case in S to lower case |
| MACRO | number -- | Runs user's avatar macro number |
| ME | -- id | Pushes spotID/doorID executing script or 0 for Cyborg |

| Word | Stack | Synopsis |
|---|---|---|
| MIDIPLAY | "fn" -- | Plays MIDI file fn. |
| MIDISTOP | -- | Stops the currently playing MIDI |
| MOUSEPOS | -- | x y Pushes the current mouse x,y |
| MOVE | x y -- | Moves the user relative x,y from current position |
| NAKED | -- | Clears all props from user |
| NBRDOORS | -- n | Pushes number of doors in room |
| NBRROOMUSERS | -- n | Pushes number of users in room |
| NBRSPOTS | -- n | Pushes number of spots in room |
| NBRUSERPROPS | -- n | Pushes number of props being worn |
| NETGOTO | "url" -- | same as GOTOURL |
| NOT | a -- bool | Pushes 1 if a is zero, else 0 |
| OR | a b -- bool | Pushes 1 if either a or b is non-zero |
| OVER | --item | Pushes a copy of the next to top of the stack |
| PAINTCLEAR | -- | Clears all painting |
| PAINTUNDO | -- | Erases last painting command |
| PENBACK | -- | Moves pen behind avatars |
| PENCOLOR | r g b -- | Sets pen to red/green/blue |
| PENFRONT | -- | Moves pen in front of avatars |
| PENPOS | x y -- | Moves pen to x,y w/o drawing |
| PENSIZE | n -- | Sets size of pen to n (1-9) |
| PENTO | x y -- | Moves pen to relative x,y w/o drawing |
| PICK | n -- item | Pushes a copy of the item n down on the stack |
| POP | n -- | Removes top stack element |
| POSX | -- x | Pushes user's X coordinate |
| POSY | -- y | Pushes user's Y coordinate |
| PUT | d [a] n -- | Stores d in element n of array a |
| PRIVATEMSG | "msg" userID -- | Sends private msg to userID |

| Word | Stack | Synopsis |
|---|---|---|
| RANDOM | n -- v | Pushes a random number between 0 and n-1 |
| REMOVEPROP | propID -- | Removes prop propID |
| REMOVEPROP | "name" -- | Removes prop name |
| RETURN | -- | Break out of an atomlist |
| ROOMID | -- n | Pushes the current roomID |
| ROOMMSG | "msg" -- | Sends msg to everyone in the room |
| ROOMNAME | -- "name" | Pushes the current room name |
| ROOMUSER | n -- userID | Pushes userID of nth user in room |
| SAY | "msg" -- | Makes user speak msg |
| SAYAT | "msg" x y -- | Cause msg to appear at x,y |
| SELECT | spotID -- | Causes ON SELECT handler of spotID to run |
| SERVERNAME | -- "name" | Pushes the current server name |
| SETALARM | ticks spotID -- | Runs ON ALARM for spotID in ticks time |
| SETCOLOR | n -- | Sets roundhead to color 0-15 |
| SETFACE | n -- | Sets roundhead expression to 0-12 |
| SETLOC | x y spotID -- | Moves spotID to x,y if god/wiz |
| SETPICLOC | x y spotID -- | Moves current state pic for spotID relative x,y if wiz/god |
| SETPOS | x y -- | Moves user to absolute x,y |
| SETPROPS | [props] -- | Causes user to wear props |
| SETSPOTSTATE | n spotID -- | Sets state of spotID to n for all in room |
| SETSPOTSTATELOCAL | n spotID -- | Sets state of spotID to n for user only |
| SHOWLOOSEPROPS | -- | Lists loose props locations in log |
| SINE | degrees -- sin | Pushes sine(degrees)*1000 |
| SOUND | "fn" -- | Plays sound fn |
| SPOTDEST | spotID -- n | Pushes room number spot leads to |
| SPOTNAME | spotID -- "name" | Pushes name of spot spotID |

| Word | Stack | Synopsis |
|---|---|---|
| SPOTIDX | n -- spotID | Returns ID of the nth spot in room |
| SQUAREROOT | n -- sqrt | Pushes the square root of n |
| STACKDEPTH | -- n | Pushes the number of items on the stack |
| STATUSMSG | "msg" -- | Puts msg in client status window |
| STRINDEX | "str" "sp" -- off | Pushes the offset of sp in str or -1 |
| STRLEN | "str" -- len | Pushes the length of str |
| STRTOATOM | "str" -- {al} | Compiles str into atomlist |
| SUBSTR | "str" "sp" -- bool | Search str for sp, push 1 if found, else 0. Case independent. |
| SUBSTRING | "str" off len -- "f" | Pushes the substring of str at offset off for length len |
| SUSRMSG | "msg" -- | Sends msg to owner/operator via page |
| SWAP | a b -- b a | Swap top two stack elements |
| TANGENT | degrees -- sin | Pushes tangent(degrees)*1000 |
| TICKS | -- tick | Push current client time in 1/60 seconds |
| TOPPROP | -- propID | Pushes the propID of the top prop |
| TOPTYPE | -- type | Pushes the type of the top item on the stack without removing it. Types are: 0 - Error/unknown/stack empty, 1 - number, 2 - variable, 3 - AtomList, 4 - String, 5 - ArrayMark, 6 - Array |
| UNLOCK | doorID -- | Unlocks door doorID |
| UPPERCASE | "str" -- "STR" | Converts lower case str letters to upper case |
| USERID | -- userID | Pushes my userID |
| USERNAME | -- "name" | Pushes the user's screen name |
| USERPROP | n -- propID | Pushes ID of the nth worn prop |
| VARTYPE | -- type | Like TOPTYPE, but if the top item is a variable pushes the type of it's value |

| Word | Stack | Synopsis |
|---|---|---|
| WHILE | {al} {test} -- | Runs al as long as test is non-zero |
| WHOCHAT | -- userID | Pushes userID in INCHAT handler |
| WHOME | -- userID | Pushes my userID |
| WHONAME | userID -- "name" | Pushes screen name of userID |
| WHOPOS | userID -- x y | Pushes x,y of user userID |
| WHOPOS | "name" -- x y | Pushes x,y of user name |
| WHOTARGET | -- userID | Pushes userID of whisper/esp target |



# Adding Machine Exercise

The following exercise provides a step-by-step look at the creation of a script called "Adding Machine," which performs addition problems. With this script in place, whenever you type in something like "673 plus 897" your avatar will say "673 plus 897 equals 1570", and at last your parents will see the point behind all that tuition…

The *Adding Machine* script will sit in the `OUTCHAT` handler of some spot or cyborg, waiting to hear the magic word "plus" in a `CHATSTR`. Since most of our utterances won't include this word, we can tell right away that the active part of our script will have to be placed within an `IF` command:

```
ON OUTCHAT {
   <the active part of the script>
   } CHATSTR "^(.*) plus (.*)$" GREPSTR IF
```

The two wildcards in the above sentence – these things: (.*) – tell `GREPSTR` to grab everything spoken in their places (i.e., outside of the blank spaces surrounding the word "plus"). `GREPSTR` then saves them. Thanks to this function, we can now use a `GREPSUB` command to get them back, and assign them to the special symbols `$1` and `$2` as you will see in the active part of our script, which we can now turn to.

Since `$1` and `$2` are strings, we need to convert them to integers before we can do math with them. But because we're going to want their string values later (when we speak the answer), we'll create a couple new symbols, and convert *those* into numbers using the `ATOI` command:

```
"$1" GREPSUB FIRSTNUMBER =
"$2" GREPSUB SECONDNUMBER =
FIRSTNUMBER ATOI FIRSTNUMBER =
SECONDNUMBER ATOI SECONDNUMBER =
```

So far so good. `FIRSTNUMBER` and `SECONDNUMBER` are now symbols corresponding to the integer values of these numbers, while `$1` and `$2` still store the string versions (note that we had to use `GREPSUB` to get at these values). Now that we have our integers, we can do the addition operation itself, sending our output to a third symbol we'll call "`TOTAL`":

```
FIRSTNUMBER SECONDNUMBER + TOTAL =
```

As you'll see in the following paragraphs, the first part of this line ("`FIRSTNUMBER SECONDNUMBER +`") uses the `+` operator to put the sum we need onto the stack. The remaining part ("`TOTAL =`") grabs this number and assigns it to a symbol. Now we have `TOTAL`, but it's an integer! Before we can speak it out loud, we have to change it into a character string with the `ITOA` command:

```
TOTAL ITOA TOTAL =
```

At last we're ready to speak. We'll need to use another `GREPSUB` command to replace `$1` and `$2` just as before, then we'll slap a simple sentence together with a couple `&` operators (see below) and stick our `TOTAL` onto the end of that sentence. Finally we'll replace the original `CHATSTR` with this whole contraption, so that no one will ever see the original text that triggered our script. Here's the magical line that performs all that trickery:

```
"$1 plus $2" GREPSUB " equals " & TOTAL & CHATSTR =
```

That's it! When you put it all together, you get a nifty little script that can easily be modified to perform any mathematical operation.

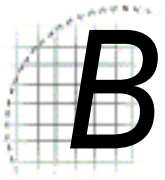
```
; "ADDING MACHINE" script
; Put this in your OUTCHAT handler
{
   "$1" GREPSUB FIRSTNUMBER =
   "$2" GREPSUB SECONDNUMBER =
   FIRSTNUMBER ATOI FIRSTNUMBER =
   SECONDNUMBER ATOI SECONDNUMBER =
   FIRSTNUMBER SECONDNUMBER + TOTAL =
   TOTAL ITOA TOTAL =
   "$1 plus $2" GREPSUB " equals " & TOTAL & CHATSTR =
} CHATSTR "^(.*) plus (.*)$" GREPSTR IF
```

### The Adding Machine Script

Congratulations! If you've performed this exercise carefully and have made liberal use of the *Language Reference* section, you're probably beginning to get a feel for the language and its unusual ways. You might even be getting very specific ideas about what you'd like to do with it, and how it might work. Are the words "handler" and "atomlist" actually making sense to you? Are you feeling at ease with such backward-looking formulae as `"2 3 + total ="`?

If you've reached that point, you just might want to set aside some time and brainstorm out a full design; you're ready to become one with the Iptscrae, and enter the weird world of the Palace owners. That's right — you can now say that you're about as advanced an Iptscrae programmer as anyone — so get out there and do it!

And don't forget to invite us to your grand opening!

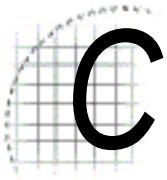


# B Code Limitations

This section describes the current Iptscrae implementation limits. No matter how cleverly you assign variable names and swap graphic elements, there are some data handling boundaries hard-coded into the software. In general, The Palace Viewer version of Iptscrae is limited only by the memory available in the browser, so these limits only apply to The Palace User Software clients for Windows and Macintosh. Note that client and server limites may override those noted below.

- The maximum length of a string generated by GREPSUB is 16384 bytes.
- The maximum length of a compiled GREPSTR regular expression is 1024 bytes.
- The maximum length of a compound string (one created by concatenation or SUBSTRING) is limited only by client storage.
- The maximum number of variables in a single handler is 64.
- The maximum number of nested AtomLists (IF, WHILE, EXEC etc.) is 64.
- The maximum number of alarms that can be queued at one time is 64.
- The maximum number of data items that can be held on the stack is 256.
- The range of spot states is -32768 to 32767.
- The maximum number of array elements is 256.
- The maximum number of global variables is limited only by client storage.

Of all the above limitations, the one you'll really want to look out for is the *stack limit* -- if more than 256 data items get placed on the stack, some of them will “fall off,” becoming lost forever. In general, however, these code limitations really shouldn't be too much of a problem for you -- unless the goal is to do truly complex, site-wide programming (like tracking the states of multiple users and objects from room to room, for instance). Although they require great determination and programming fluency, such tasks are indeed possible in Iptscrae -- and are left as exercises for the reader.



# The Palace Client Plugin API

The Palace client software has an open architecture that makes it possible for other programs, such as games, to plug-in. If you're interested in finding out more in joining the developer's program, please go the Partner's page on The Palace website (`www.thepalace.com`).

# Index

## T



## U



## V



## W