The
Ghu
Ghuide

(A manual for God's Handy Utility)
by
Chip Morningstar

Lucasfilm Ltd. Games Division
March 9, 1987

Introduction

This document is a user's manual and reference guide for **Ghu**, a utility for Habitat™ operators on the Stratus system. This is a preliminary draft and will no doubt evolve as both the tool and its users become more sophisticated.

What It Is

Ghu is meant to be a full-featured operator's utility for the Habitat system. (It is descended from an earlier utility named **Twiddle**, which you may have heard of or even used.) It is intended to be run by Quantum and Lucasfilm operational staff from a conventional terminal connected to the Stratus host. **Ghu** has lots of ''features''; however, we will try to keep everything as simple as we can. We anticipate that additional features will be added as our knowledge of what we need develops.

Ghu is a command-line-oriented utility. It reads command lines from the terminal and executes them. (It is not designed as a visual, CRT-oriented program since it must be usable over something like Telenet.) Using **Ghu** commands, you can examine, modify or add to any of the various databases that the Habitat system uses. It lets you do this in a very general and powerful way that (hopefully) will keep the amount of work you must do to a minimum. For more advanced uses, **Ghu** allows you to define macros and command loops to turn common or repetitive procedures into single commands.

The Databases

There are three primary databases that Habitat uses, together with a number of more minor ones. Each of the three primary databases describes one of the major elements of the Habitat world: regions, objects and avatars. The lesser databases contain information about teleport addresses, text for book and paper objects, requests made to the oracle, and so on. They will be described in more detail in the sections below that the discuss the commands that deal with them.

Things and Types

Each region, object and avatar in the Habitat world has a *global ID number* that identifies its place in its respective database. In many of the commands described below, you need to provide one of these global ID numbers for some entity in one of the databases. Since avatar #145733901 is not the same as region #145733901, you need to indicate, in addition to the global ID number, what sort of thing it is that you are referring to. Many of the command descriptions below will refer to something called a *thing* (in italics, just like that), which is an expression that refers to an object, region or avatar. It has the following form:

t globalID

where *t* is ''type specifier'' that is one of the three letters `a` (for avatar), `o` (for object), or `r` (for region); and *globalID* is a global ID number. For example,

```
r 145733901
```

refers to region #145733901, while

```
o 47
```

refers to object #47.

In practice, usually you can leave the type specifier out, in which case a default type will be chosen depending on the particular command being executed. In the command descriptions that follow, we will tell you what the default type associated with each command is (the defaults are chosen to be the most common types you would expect to use with each command).

You may also apply the `a` type to strings, allowing you to designate avatars by name. You do this by entering an expression of the form:

```
a 'name'
```

For example

```
a 'chip'
```

designates the avatar named "chip" (that's me!) while

```
a 'spblives'
```

designates the avatar named "spblives". In contexts where a string value is not allowed but an avatar number *is* allowed, **Ghu** will automatically try to treat a string value as an avatar name.

Values and Expressions

Note: this is fairly advanced stuff. If you are just using **Ghu** casually you can skip ahead to the next section, **The Commands**.

The above description of *things* and types referred to a "global ID number". In the examples that were given, we actually used a number, just as you would expect. However, wherever a **Ghu** command calls for the use of a numeric value of any sort (or a string value or a bit value, for that matter), you can use a symbolic name or an expression instead. Furthermore, you can enter numbers in hexadecimal, octal, quaternary or binary bases instead of the normal decimal, if for some reason you wish to do so.

In general, the command descriptions below will refer to a *value*. A *value* can be a number, a string, a bitstring, a symbolic name (which you create using the `name` command), or an expression. Each of these will be described in what follows.

Numbers

A number may be specified in a variety of different formats. For the sake of eliminating arguments about which is "better" (and at the expense of some minor complication) we have chosen to adopt both the Unix-derived representations that we use at Lucasfilm and the PL/1 derived representation used by Quantum, since both can be handled at the same time without significant ambiguity.

- Decimal — a decimal number is represented by a sequence of decimal digits (i.e., `0` through `9`). The only restriction is that the first digit may not be a `0`. Examples: `47  3  123786432333`
- Octal — an octal number is represented by a sequence of octal digits (i.e., `0` through `7`), the first of which *must* be a `0` digit. Examples: `0377  022  0  047`

  Alternatively, a sequence of octal digits enclosed in apostrophes and followed by `b3` may be used. Examples: `'377'b3  '22'b3  '0'b3  '47'b3`
- Hexadecimal — a hex number is represented by a sequence of hex digits (i.e., `0` through `9` and `a` through `f`) preceded by `0x` or `0X`. Examples: `0xf000  0x80  0X10FF  0x0`

  Note that case is not significant in the hex digits `a` through `f` (i.e., `a` is the same as `A`). Alternatively, a sequence of hex digits enclosed in apostrophes and followed by `b4` may be used. Examples: `'f000'b4  '80'b4  '10ff'b4  '0'b4`
- Binary — a binary number is represented by a sequence of binary digits (i.e., `0` and `1`) preceded by `0b` or `0B`. Examples: `0b10101010  0B111  0b10001`

- Quarters — a quarter is represented by a sequence of quaternary digits (i.e., `0` through `3`) preceded by `0q` or `0Q`. Examples: `0q1230  0q2020  0Q0013`

  Alternatively, a series of quaternary digits enclosed in apostrophes and followed by `b2` may be used. Examples: `'1230'b2  '2020'b2`

Bit-strings

A bit-string is represented by a series of binary digits (i.e., `0` and `1`) enclosed in apostrophes and followed by the letter `b`. Examples: `'1011101'b  '1'b  '0'b  '1111'b`

A bit-string is *not* the same as an ordinary number specified in a binary format, as described above under **Numbers**.

Strings

A string is represented by a series of characters enclosed in apostrophes or quotation marks (e.g., `"hello world."` or `'this is a string'`). Non-printing and special characters may be embedded in the string by using the backslash character (“`\`”) as an escape followed by an indicator character (we are borrowing this convention from Unix). The escape characters include, for example:

`\n`	newline
`\t`	horizontal tab
`\b`	backspace
`\\`	backslash
`\'`	apostrophe
`\"`	quote
`\^`*c*	CONTROL-*c* (where *c* is any character).
`\`*ddd*	arbitrary byte (where *ddd* is one, two or three octal digits).
`\x`*hh*	arbitrary byte (where *hh* is one or two hexadecimal digits).

Additionally, a variety of escape codes that represent the Habitat sign formatting characters are recognized. A complete list of all the escape codes is given in **Appendix A** at the end of this document.

Names

A symbolic name is represented as a sequence of alphanumeric characters, the first of which may not be a decimal digit. Names may also contain the characters underscore (“`_`”) and dollar sign (“`$`”). The case of alphabetic characters in symbolic names *is* significant (i.e., `foo`, `FOO` and `FoO` are *not* the same name). The value of a symbolic name in an expression will be either the value defined for that name using the `name` or `set` commands, or, if the name is the name of a field of the current-default-thing (see the description of the current-default-thing under **General Database Handling Commands** below), then the value will be the current value of that field.

Special Names

A few symbolic names are pre-defined by **Ghu** and have a special meaning all their own.

`today` —
: is a number that represents the system time code of today’s date (i.e., the time as of midnight).

`now` —
: is a number that is the system time code for whatever time it is when the expression containing it is evaluated.

`last_textid` —
: is always set to the text id of the most recently written text database entry.

`page_count` —
: is always set to the page count of the most recently read or written text database entry.

`user_name` —
is a string that is the Stratus user name of the user running **Ghu**.

`general_access` —
is a number that is TRUE (i.e., 1) if the user running **Ghu** is on the "general access" list and FALSE (i.e., 0) if not. (See **Access Control** below.)

`random` —
is a random number. It is a different random number every time it is used.

`index` —
is used only inside integer `for` loops (see the *for* command below) and is the current value of the loop counter.

`contents` —
may be used as an array field of any container object, and represents the contents of that object. For example

```
o 15446.contents(3)
```

represents the object that is contained in slot 3 of object #15446. (See the discussion of fields and arrays below.)

Expressions

In general, any place where a *value* is called for an expression may be given. There is a variety of expression operators that you can use.

Arithmetic operators

expression `+` *expression*
expression `-` *expression*
expression `*` *expression*
expression `/` *expression*
`-` *expression*

These correspond to addition, subtraction, multiplication, division and negation respectively. The addition operator ("`+`") is also the string concatenation operator, i.e., "adding" two strings concatenates them. For example,

```
"foo" + "bar"
```

is equal to

```
"foobar"
```

Logical operators

expression `&&` *expression*
expression `and` *expression*
expression `||` *expression*
expression `or` *expression*
`!` *expression*
`not` *expression*

Logical values are used in **Ghu**'s conditional statement, `if`. A logical value is represented in **Ghu** by an ordinary number. A value of `0` is interpreted as FALSE. Any other value is interpreted as TRUE. The logical operators for AND, OR and NOT are each represented in two forms: as symbolic operators and as keywords. Use whichever form you feel most comfortable with (the symbolic operators are derived from the programming language **C**).

Comparison operators

expression `<` *expression*
expression `lt` *expression*
expression `<=` *expression*

expression `leq` *expression*
expression `==` *expression*
expression `eq` *expression*
expression `!=` *expression*
expression `neq` *expression*
expression `>=` *expression*
expression `geq` *expression*
expression `>` *expression*
expression `gt` *expression*

The comparison operators may be used to compare the values of numbers, strings, or bit-strings. As with the logical operators, there are two forms for each comparison, one symbolic and the other a keyword. The comparisons supported are less-than, less-than-or-equal, equal, not-equal, greater-than-or-equal, and greater-than. The result of a comparison will be either `0` or `1`, representing FALSE or TRUE respectively.

Bit operators

expression `&` *expression*
expression `|` *expression*
expression `^` *expression*
`~` *expression*
`^` *expression*

The bit operators work on bit-strings and numbers. The operators shown are (respectively) AND, OR, XOR, NOT and NOT. There are two forms of the NOT operator, one derived from **C** and the other from PL/1.

Test operator

`?` *expression*

is used to determine what sort of thing *expression* is. Its value is a number that encodes *expression*'s type: `0` means it is a region, `1` means it is an avatar, `2` means it is an object, and any other value means it is none of these.

Field selection

The database entries for objects, avatars and regions have various fields, each of which is named. You can extract the value from a field using the "`.`" operator:

expression `.` *fieldname*

where *expression* is the global ID of the thing you are interested in and *fieldname* is the name of the field you want. For example:

```
a 435679123.x
```

refers to the x-coordinate field of avatar #435679123. Additionally, some fields are *arrays* of values, and these arrays may be indexed:

expression1 `(` *expression2* `)`

where the *expression1* identifies the thing and field in question and *expression2* is the array index. For example:

```
o 634989100.nitty_bits(5)
```

refers to the fifth element of object #634989100's `nitty_bits` field.

Other stuff

In addition to the operators, there are a few more things that need to be mentioned regarding expressions.

All of the operators described above have the sort of precedences you would expect them to have (e.g., multiplication and division before addition and subtraction). The exact operator precedences are listed in **Appendix B**. Parentheses:

`(` *expression* `)`

may be used at any point to override the normal precedences of the operators.

Note that since there is only one string operator, concatenation, most expressions involving strings per se are rather silly (you *can* treat objects' string fields as arrays of characters and index them, however). In arithmetic expressions, we treat a single character string as a number whose value is the value of the 8-bit ASCII code for the character in the string. Similarly, two-character strings represent 16-bit numbers and so on. We do *not* adopt the nearly useless PL/1 convention of trying to convert the string containing the characters `'4'` and `'7'` into the number 47.

The Commands

The specifics of the various commands will be described shortly. First, some general information:

The command names have been chosen so that they are unique in the first few characters. Single letter abbreviations are often possible. However, any unambiguous initial substring of a command name is an acceptable abbreviation.

In general, commands are terminated by the end of the line, but multiple commands may be given on a line separated by semicolons ("`;`"). For example, the line:

```
display r 5001; contents r 5001; quit
```

is equivalent to entering each of the three commands separately on its own line.

A single command may be spread over multiple lines by ending all but the final line with a backslash ("`\`"). For example:

```
add class_teleport = r 5000, 40, 150,\
0, 0, 200,\
0, '1'b, "Foon St"
```

is the same as

```
add class_teleport = r 5000, 40, 150, 0, 0, 200, 0, '1'b, "Foon St"
```

Usually this is only worth bothering with for very long command lines, which are relatively rare.

Any command may be followed by

`into` *filename*

in which case the printed output generated by the command is written to the given file instead of to the terminal. For example:

```
contents r 5000 into dumpfile.out
```

would output a list of the contents of region #5000 into the file "`dumpfile.out`".

A comment on notation: Each of the command descriptions that follow begin with a summary of the syntax of the command. Things that are to be entered literally, such as keywords and delimiter characters, will be written in a `typewriter-like typeface such as this`. Things which you are supposed to substitute an expression or value of your own will be written in an *italic typeface such as this*. Things which are optional will be enclosed with funny brackets ⟦ like this. ⟧ Sometimes you will also see things enclosed in brackets ⟦ like this ⟧* which means that the thing within the brackets may be repeated zero or more times.

Important Basic Commands

These two commands should be all you need to know to at merely survive using **Ghu**.

help

`help` ⟦ *command* ⟧

If no *command* is given, `help` prints out a list of all the available commands along with a quick one-liner description of each. If a *command* is given, it prints an explanation of that command. For example:

```
help display
```

prints an explanation of the `display` command.

quit

```
quit
```

Exits **Ghu** and returns to the Stratus system command level.

General Database Handling Commands

These commands often refer to something called the "current-default-thing". The current-default-thing is a thing (object, avatar or region) that **Ghu** keeps in memory. Various commands implicitly or explicitly change which thing is the current default, and other commands operate on the current-default-thing unless explicitly told otherwise. This should become clearer as you read the command descriptions themselves.

add

`add` [*class* [`=` *value* [`,` *value*]*]]

Creates a new thing of the given class (note: regions have class 0 and avatars have class 1). You can specify the class and the values of the new thing's property fields (in order) directly on the command line. Alternatively, if any of these are omitted **Ghu** will prompt you for them. The new thing becomes the current-default-thing. For an example of the use of the `add` command, see the discussion on page 6 of line continuation using backslashes.

contents

`contents` *value* [`full`]

The `contents` command prints a list of the things contained by the thing specified by *value* (a region by default). Each thing is listed on a line with its global ID and the values of its fundamental properties (x-y position, orientation, etc.). If the keyword `full` is given, it also lists the contents of each container inside *value*, and each container inside those, and so on. For example:

```
contents r 8100
```

would print a list of the things contained in region #8100, whereas

```
contents r 8100 full
```

would print not only the things contained by region #8100, but any things contained by those things, and any things contained by *those* things, and so on.

display

`display` [*value*] [`.`]

The `display` command displays the thing indicated by *value*. If value designates an object, avatar or region, it makes the indicated thing the current-default-thing and then prints the values of all of its fields. If *value* is not a thing (e.g., if it is simply a string or a number) *value* itself is printed out. If *value* is omitted, the current-default-thing is displayed. If the `display` command is ended with a period, **Ghu** withholds printing a newline after displaying the value. For example:

```
display r 9164
```

would print out a description of region #9164. Whereas

```
display o 17007.container
```

would print out a description of the region, object or avatar that contains object #17007 (compare this to the similar example given under the description of the `value` command above).

find

`find` *value*

Prints out the chain of containership from the thing indicated by *value* (an object by default, though it may also be an avatar) to the outermost containing region, listing the global ID and class of each container in the chain. For example, the command

```
find o 17659
```

might result in output of the form

```
Object 17659 (flashlight) currently in object 1005131
Object 1005131 (box) currently in avatar 255538382
Avatar 255538382 (chip) currently in region 9132
```

get

`get` *value*

The `get` command makes the thing indicated by *value* the current-default-thing. In this respect it operates like the `display` command, except that it doesn't print out a description of the thing. By default, *value* is an object, though it may also be a region or an avatar. For example:

```
get a 453124989
```

would make avatar #453124989 the current-default-thing, while

```
get 1609
```

would get object #1609.

set

`set` *lvalue* `=` *value* [`!`]

Sets *lvalue* to have the value *value*. *lvalue* must be one of three things: (1) a symbolic name or (2) the name of a field of the current-default-thing or (3) an expression designating a field of some other *thing*. If it is a field, then *value* must have a type that matches **Ghu**'s internal definition for that field (see the descriptions of the `class` and `define` commands below). If *lvalue* designates a field of some thing other than the current-default-thing, the thing designated is first made the current-default-thing before the field's value is set. If the command is terminated with an exclamation point ("`!`"), it immediately writes the new version of the thing changed to the database as if an `update` command had been issued. For example

```
set foobar = 47
```

would set the value of the symbolic name `foobar` to be the number `47`.

```
set a 'fred'.region = 9173
```

would set the value of the avatar named "fred"'s `region` field to region #9173. The avatar named "fred" would also become the current-default-thing. In contrast

```
set a 'fred'.region = 9173!
```

would do all of the above *and* write the changed avatar record back out to the database file. Note that you cannot set the special pre-defined field `contents`. For example,

```
set a 'fred'.contents(3) = o 47
```

will not work!

update

`update`

Writes the current-default-thing out to its associated database.

value

`value` *value* **[.]**

Simply prints out the value *value*. If the `value` command is ended with a period, **Ghu** withholds printing a newline after printing the value (this can be useful in writing macros that want to display data in formats of the macros' choosing). Examples:

```
value 42+5
```

would print out

```
47
```

whereas

```
value o 17007.container
```

might print out

```
r 9164
```

The sequence of commands

```
value "The container is " .
value o 10024.container .
value " and it is of class " .
value o 10024.container.class
```

would print out something like:

```
The container is o 17007 and it is of class 13
```

If *value* is the name of a macro, the definition of the macro will be printed.

Specialized Database Handling Commands

list

`list` **[** *value1* **] [** `--` **] [** *value2* **]**

Prints a list of the existing regions in the range from *value1* to *value2* (inclusive). If *value1* is omitted, it defaults to the first region in the database. If *value2* is omitted, it defaults to the last region in the database. If only one *value* is given and the `--` is omitted, only the one region specified is listed. If both *value*s are omitted the `--` may be omitted as well. Omitting both *value*s means that the entire region database is to be listed. For each region, the `list` command prints a line containing the region ID number, those of its four neighbors (or -1 if there is no neighbor in a particular direction), and the region's orientation. For example:

`list 5`	lists region number 5
`list 5--10`	lists regions 5 through 10
`list 5--`	lists from region 5 to the end of the database
`list --10`	lists from the start of the database to region 10
`list --`	lists all regions
`list`	also lists all regions

thus, the command

```
list 9100--9103
```

would result in output looking something like

```
REGION          WEST       NORTH          EAST       SOUTH
  9100            -1        9101            -1          -1
  9101            -1        9102          9113        9100
  9102            -1          -1            -1        9101
  9103            -1          -1          9108          -1
```

newturf

`newturf` *value*

Adds the region number specified by *value* to Habitat's internal list of regions available to be allocated as turfs for new users.

nuke

`nuke` *value1* [`--` *value2*]

Removes the specified regions (and any objects contained within them) from the database. The regions are specified as in the `list` command, except that any range of regions must have explicit start and end numbers. For example

```
nuke 500--999
```

would delete all the regions with ID's in the range 500 to 999. If there are avatars in the regions being `nuke`'d, it will ask if you wish to delete them. Note that if you don't delete them, you had better move them to regions that still exist before allowing any of their associated players to enter Habitat.

oracle

`oracle` *value* [`dump`] [`since` *date*]

or

`oracle all` [`dump`] [`since` *date*]

Prints the oracle question database entries associated with the oracle object identified by *value*. Alternatively, the keyword `all` may be given, in which case the entries for all oracle objects are printed. If the keyword `dump` is given, it simply prints the entries in sequence, listing the oracle ID, the time and date of the question, the avatar asking the question, and the text of the question itself. If the keyword `dump` is omitted, it prints each question individually and then prompts for an action to take. The allowed actions are:

`quit` — stop printing questions and return to **Ghu** command level.

`delete` — remove this question from the database and display the next one.

`answer` [*filename*] — send a Habitat mail message to the avatar asking the question. **Ghu** will prompt the terminal for the text of the message, or, if *filename* is given, the text will be taken from that file. After sending this message the question is then removed from the database.

`next` — print the next question.

For convenience, abbreviations to any of these answers are allowed. Furthermore, simply entering a carriage return in response to the action prompt will be taken as a `next` action.

You can limit the questions by date by specifying `since` *date*, where *date* is either a numeric time-code or a string that will be interpreted as a standard VOS date/time string, i.e., `"87-12-22 17:43"`.

pages

`pages` *value1* `=` *value2*

The `pages` command truncates the page count of a document. *value1* is taken as a text entry ID. *value2* is taken as a page count. The page count of the document indicated by *value1* is set to be no more that the value indicated by *value2*. In other words, any additional pages exceeding the page count are deleted from the document. For example, if document #1009 is a seven page document, the command

```
pages 1009 = 5
```

would truncate it to five pages, discarding pages 6 and 7.

read

`read` ⟦ *value1* ⟦ `,` *value2* ⟧ ⟧

The `read` command prints out the text of a document. If *value1* designates an object, it must be a paper, book or plaque object, and that object's text entry is printed. Otherwise, *value1* should be a plain number, which will be taken as the text database ID number of a text entry to be printed. *value2*, if given, specifies a page number within the chosen text entry to print. If *value2* is omitted, all pages of the chosen entry are printed.

readmail

`readmail` *value*

Will display mail messages waiting for the avatar designated by *value*. It prints each mail message individually and the prompts for an action to take. The allowed actions are:

`quit` — stop printing mail messages and return to **Ghu** command level.

`delete` — remove this message from the mail database and display the next one.

`answer` ⟦ *filename* ⟧ — send a reply mail message to the avatar who send the message just displayed. **Ghu** will prompt the terminal for the text of the message, or, if *filename* is given, the text will be taken from that file.

`next` — print the next message.

For convenience, abbreviations to any of these answers are allowed. Furthermore, simply entering a carriage return in response to the action prompt will be taken as a `next` action.

For example

```
readmail 'chip'
```

would read any Habitat mail waiting for Chip.

remturf

`remturf` *value*

Removes the region number specified by *value* from Habitat's internal list of regions available to be allocated as turfs to new users. If *value* is not on this list, **Ghu** will give you an error message.

sendmail

`sendmail` *value1* ⟦ `from` *value2* ⟧ ⟦ `file` *filename*⟧

Sends mail to the avatar designated by *value1*. *value2*, if given, designates the avatar from whom the mail is supposed to be sent. If *value2* is omitted, the mail will be from "The Oracle". Ordinarily, the text of the message will be taken from the terminal, just as when entering text for a piece of paper using the `write` command. If a *filename* is given, the text will be taken from that file instead. For example

```
sendmail 'chip'
```

would send a mail message to Chip from The Oracle. On the other hand,

```
sendmail 'chip' from 'spblives'
```

would send a mail message to Chip from SPBlives. The command

```
sendmail 'chip' from 'spblives' file form_letter
```

would send the message with the text taken from the file "form_letter".

teleport

`teleport` *value1* [`=` *value2*]

value1 is taken as a teleport address (and so it should be a string value). If *value2* is not given, the `teleport` command prints out the teleport database entry associated with the given address (**Ghu** will automatically add the area code if you don't explicitly give one). For example

```
teleport 'Bleem St Cross'
```

might output

```
Teleport  "pop-bleemstcross    " --> o 2344
```

Also, when simply displaying a teleport entry, you can give an address of the form

string`*`

and **Ghu** will display all teleports with address that start with the string *string*. For example:

```
teleport 'Bleem*'
```

might display this

```
Teleport  "pop-bleemstcross    " --> o 2344
Teleport  "pop-bleemstnend     " --> o 2226
```

value2, if given, may be a region ID or the object ID of a teleport booth or elevator object. If *value2* is given, the `teleport` command sets the teleport database entry associated with the address *value1* to point to the region indicated by *value2*. If *value2* identifies a teleport booth or elevator object, the ID of region containing the booth or elevator is used. For example,

```
teleport 'secretaddress' = o 2344
```

would make the teleport address "`secretaddress`" (or "`Secret Address`", since case and whitespace are squeezed out in use) a synonym for "`bleemstcross`".

titles

`titles` [*value1*] [`--`] [*value2*]

Prints the titles of documents in the text database for documents in the range from *value1* to *value2*. If *value1* is omitted, it defaults to 0. If *value2* is omitted it defaults to 100,000 (this way you won't see peoples' mail).

turfs

```
turfs
```

Prints a list of all the region numbers in Habitat's internal list of regions available to be allocated as turfs to new users.

write

`write` [*value1* [`,` *value2*]] [`file` *filename*]

The `write` command is the opposite of the `read` command. *value1* and *value2* are taken as a document identifier and page number, respectively, just as in the `read` command. The `write` command, however, *writes* the text of the indicated document. If a *filename* is given, the text is taken from that file. Otherwise, **Ghu** prompts the terminal for the new text.

There is a limit of 16 lines per page with no more than 40 characters per line allowed. All the standard `Ghu` backslash ('\') escapes are recognized, in order to allow you to put Habitat graphic characters into documents; each such escape sequence, of course, only counts as one character towards the 40 character limit. Page breaks may be indicated by lines beginning with the pound-sign ('#') character. These lines do not go into the text database and do not count towards the 16 line per page limit. When entering text from the

terminal, the end of the text is indicated by a line consisting of a lone period character (‘`.`’). When reading the text from a file, the end of text is indicated by the end of the input file.

If a page number is given, pages in the indicated document are replaced starting with the given page number. As many pages as you provide text for will be replaced, while remaining pages that were already in the document will not be touched. For example, suppose document #1009 contains 7 pages and you entered the command

```
write 1009, 3
```

and then provided two pages worth of text. The two pages starting with page 3 (i.e., pages 3 and 4) would be replaced, while pages 1 and 2 and 5 through 7 would be left unchanged.

If no page number is given, the entire document is replaced by the new text you provide. Thus, in the above example, if you had instead entered the command

```
write 1009
```

and then provided the same two pages worth of text, the new document #1009 would be a two page document containing just the two new pages you entered. The old contents of pages 1 through 7 would be deleted.

Note that the format of the output from the `read` command matches the format of the input expected by the `write` command. Thus you can extract the text from one document and put it into another using a temporary file as an intermediary. For example:

```
read 2000 into tempfile
write 2500 file tempfile
```

would copy the text from document 2000 into document 2500. Using a Stratus file outside of **Ghu** in this fashion is also useful if you need to do extensive editing on a document, e.g.,

```
read 2000 into tempfile
quit
edit tempfile using Stratus editor
ghu
write 2000 file tempfile
```

Note that the input and output formats used by `read` and `write` are set up so that this will work even with multi-page documents.

Class Definition Handling Commands

Ghu maintains an internal table that describes the names and data types of the fields of each class of object in the system (it also describes the fields of avatars and regions). This table is saved away in a secret file that **Ghu** reads every time it starts up. **Ghu** provides commands that let you examine and update this table. (In general, you should never change this table yourself, unless you are making changes to the Habitat software also, such as adding a new object class.)

class

```
class value
```

Prints out the current class definition for class number *value*. Note that the class definitions for ordinary objects do not include the fields that are common to all objects. The definition of *these* fields may be displayed by entering a class value of `-1`. For example, entering the command

```
class 74
```

in the current system would give you the following output:

```
---- Class 74 (teleport) ----
  state:                  bin15 = 0
  take:                   bin31 = 0
  address(20):            character = "                    "
```

define

```
define value string
    fielddefinitions
enddefine
```

Adds or changes one of **Ghu**'s internal class definitions. *value* indicates the class number of the class being defined. *string* indicates the name to be given the class (this is the name that is printed out in contents lists and so on). *fielddefinitions* are a list of field definitions, one for each field of the class. Note that regions are class `0`, avatars are class `1`, and objects have class numbers greater than `1`. Also, the class definitions for ordinary objects do not include the fields that are common to all objects. These are defined by the "pseudo-class" number `-1`.

The field definitions themselves are a series of lines, each with the following form:

⟦ `#` ⟧ *name* ⟦ `(` *value1* `)` ⟧ `:` *type* ⟦ `=` *value2* ⟧

where *name* is the name of the field and *type* is the field's data type (the allowed types will be given momentarily). *value1*, if given, indicates the dimension of an array or string field. It should be a number. *value2*, if given, indicates a default initial value for the field. This value will be used when new instances of this class are created (e.g., using the `add` command). If the field is an array field, *value2* may actually be a list of values separated by commas, one value for each entry in the array. The "`#`" character at the start of the line, if given, indicates that this is an "invisible" field. Such fields are not displayed when the `display` command displays a thing (though invisible fields can be displayed by asking for them explicitly).

The allowed data types for fields are the following:

`character` — A character or character string value. The dimension indicates the length of the string.

`bin15` — A bin(15) value, i.e., a 2-byte integer.

`bin31` — A bin(31) value, i.e., a 4-byte integer.

`bit` — A bit or bit-string value. The dimension, if given, indicates the length (in bits) of the bit-string.

`words` — A character or character string, in which each character is placed in its own 2-byte word (this is required for any character string which is to be transmitted to the Commodore 64). The dimension indicates the length (in words/characters) of the string.

`regid` — A region ID number.

`objid` — An object ID number.

`avaid` — An avatar ID number.

`fatword` — A 2-byte integer value in which each byte is placed in its own 2-byte word (this is required for any word value that is to be transmitted to the Commodore 64).

`entity` — A database ID of some sort. It is stored as a 4-byte ID number together with a 2-byte code that indicates whether it is an object, avatar or region ID number.

`varstring` — A PL/1 style variable length character string. The dimension, if given, indicates the maximum length of the string.

`byte` — A 1-byte integer value.

For example, the following command would `define` the display case object as it is currently constituted:

```
define 22 'display case'
  open_flags:      bin15 = 3
  key:             fatword = 0
  owner:           avaid = 0
  locked(5):       bit = 0, 0, 0, 0, 0
# filler(11):      bit
enddefine
```

note that the unused "filler" bits are flagged as invisible, since they are ordinarily of no interest.

Power User Commands

These commands enable you to create macros and command files and to tell **Ghu** to perform operations repetitively or conditionally. Although everyone will probably make use of macros that are defined for them, most of these facilities are for more advanced **Ghu** users.

alias

`alias` [*name* [`=` *string*]]

Defines *name* to be a command alias (abbreviation) for the string *string*. Whenever *name* is found at the beginning of a command line it is textually replace by *string*. If *string* is omitted, the `alias` command prints out the current alias definition for *name*. If both *name* and *string* are omitted, the `alias` command prints out a list of all currently defined command aliases. For example

```
alias odump = "oracle all dump"
```

would define the alias `odump` to be a command that would give a complete dump of the oracle question database.

allmacros

`allmacros`

Prints out the names and definitions of all macros currently defined. See the discussion of the `macro` command, below.

allnames

`allnames`

Prints a list of all the symbolic names currently defined, with their values and associated types. See the discussion of the `name` command below.

break

`break`

The `break` command provides a convenient way to exit from a loop or a macro. The `break` command simply aborts the innermost containing `for` loop or macro call. For example:

```
for all o
    if (class == box)
            value "Hey!  I found a box, id=".
            value ident
            break
    endif
endfor
```

This will look through the object database until it finds the first box object, at which point it will print a message and stop.

execute

`execute` *filename*

Reads and executes **Ghu** commands from the indicated file, resuming command input from the terminal upon reaching the end of the file or encountering an error in one of the commands in it. For example

```
execute mymacros.ghu
```

would execute the commands in the file "`mymacros.ghu`". One warning about `execute`: the reading and execution of the commands in the given file will only take place after all commands on the line containing the `execute` command have been processed. Thus the command line

```
display foo; execute myfyle; display bar
```

would first display the value of `foo`, then set up the file "`myfyle`" to be executed, then display the value of `bar`, and only *then* read and execute whatever is in "`myfyle`". Thus you must be careful in your use of the `execute` command. In particular, if the macro `verify_user` were to be defined in the file "`macros.ghu`", then the following would generate an error:

```
execute macros.ghu; verify_user
```

since `verify_user` will not be defined until after the whole line is processed. Putting these two commands on separate lines is the right way to handle this situation. Similarly,

```
execute some_stuff; quit
```

will process the `quit` command (and thus exit **Ghu**) *before* the file "`some_stuff`" is ever read. For the same reason, you cannot use the `execute` command in a command string passed to **Ghu** from VOS command level using the `-command` option (described below in the section **The Ghu Command Line**).

for

`for` *value1* `to` *value2*
    commands
`endfor`

or

`for all` *type*
    commands
`endfor`

`forr all` *type* `in` *value*
    commands
`endfor`

The `for` command provides a looping facility. There are three forms of the `for` command. The first form loops from *value1* to *value2*. These values may be object, avatar or region identifiers or they may be plain integer numbers. If they are object, avatar or region identifiers, then they must both be identifiers of the same type (e.g., both are region identifiers). In the case of the loop bounds being object, avatar or region identifiers, **Ghu** executes the commands *commands* once for each object, avatar or region in the given range, with the current-default-thing set to each object, avatar or region in the range in turn. If the loop bounds are plain integer numbers, then **Ghu** executes *commands* once for each value in the range, with the special symbolic name `index` set to this value each time through.

The second form of the `for` command allows you to loop through all objects, avatars, or regions in the database. *type* should be one of the type specifiers `a`, `o` or `r`.

The third form of the `for` command allows you to loop through all objects or avatars contained in a particular region or container. *type* should be one of the type specifiers `a` or `o`. *value* should be the identifier of a region, avatar or container object.

Examples:

```
for all a
    value screen_name
endfor
```

prints the screen name of every avatar in the database.

```
for r 1000 to r 5000
    display
```

```
endfor
```

displays all the regions in the range 1000 to 5000.

```
for 1 to 20
    value index
endfor
```

will print the numbers from 1 to 20 (a trivial example).

```
for all o in r 9164
    value class
endfor
```

will print the class numbers of all the objects in region 9164.

if

```
if value1
    commands1
[ elseif value2
    commands2 ]*
[ else
    commands3 ]
endif
```

The `if` command provides a conditional command execution facility. If *value1* evaluates to TRUE (i.e., non-zero), the commands *commands1* are executed. Else if *value2* evaluates to TRUE, then the commands *commands2* are executed. Otherwise, the commands *commands3* are executed. There may be any number of `elseif` clause (or none at all). The final `else` clause may be omitted as well. For example

```
if region == turf
    value "The avatar is already home"
else
    region = turf!
    value "The avatar has been moved home"
endif
```

would make sure that the default thing (assumed to be an avatar) is home, printing out an appropriate message in either case.

macro

```
macro name
    commands
endmacro
```

Defines the macro *name* as the series of **Ghu** commands given by *commands*. The symbol *name* may then be used as a **Ghu** command whose function is to perform whatever actions would be performed by *commands*.

When macros are called, you can pass arguments to them. In the text of the macro itself you can refer to the variable names `$1`, `$2`, `$3`, etc., which refer to the first, second, third, etc. arguments of the macro call. Here is an example:

```
macro reset
    get a $1
    set region = turf
    set bank_balance = $2
    update
endmacro
```

This macro will move an avatar back to his turf and set his bank balance to a specified amount. You would call it with a command such as this:

```
reset 'chip', 100
```

or

```
reset 145322919, 500
```

The first example would move the avatar named "chip" home and set his bank balance to 100 tokens, while the second example would move avatar #145322919 home and set his bank balance to 500 tokens.

All macros defined using the `macro` command are saved in **Ghu**'s environment file when you quit, and are read in again when you start **Ghu** up again. Thus, any macro you create will be available for use anytime you run **Ghu** in the future. See the section below on **The Ghu Command Line** for more details about environment files.

name

`name` *name* `=` *value*

Creates the global symbolic name *name* with the type and value specified by *value*. For example

```
name library = r 9164
```

would define the name `library` and give it the value `r 9164`.

All names defined using the `name` command are saved in **Ghu**'s environment file when you quit, and are read in again when you start **Ghu** up again. Thus, any value you name using the `name` command will be available for use anytime you run **Ghu** in the future. See the section below on **The Ghu Command Line** for more details about environment files.

Certain names are predefined by **Ghu** automatically. These include all the class numbers, represented by symbols such as `teleport` or `box`, in addition to the special symbols such as `now` and `random` described earlier in the section on expressions.

start_logging

`start_logging` *filename* [`continue`]

Starts logging all **Ghu** output to the file *filename*. This logging is done using the Stratus logging facility, and is a global modification to your Stratus environment. In particular, if you start logging within **Ghu** and then exit, terminal output will continue to be logged in *filename* outside of **Ghu** as well.

If the `continue` keyword is given, the log file is appended to rather than written afresh.

stop_loggin

`stop_logging`

Stops any logging that had previously been started by a `start_logging` command.

unname

`unname` *name*

Deletes the symbolic name *name* from **Ghu**'s symbol table and removes it from the environment file. In effect, this command undoes the work of the `name` or `macro` commands.

Obscure Lucasfilm Tools Commands

These commands are used in conjunction with several of the region creation and editing utilities we use at Lucasfilm. The Stratus user probably does not need to be concerned with these commands, but we are documenting them in the interest of completeness.

cook

`cook` *filename*

Translates the file full of "raw" format object and region definitions named by *filename* into a collection of new entries in the region and object databases.

There is an internal array that **Ghu** allocates which limits the maximum size of the "raw" file that the `cook` command can handle. By default, this limit is set at 10,000 regions and objects (total). If you need to cook a larger "raw" file, you can set the array size larger by using the `-cooksize` option on the **Ghu** command line (see below).

griddle

`griddle` *value* [`full`]

The `griddle` command dumps the object, region or avatar specified by *value* in "Griddle" format. ("Griddle" format is a special text format for describing database entries. It is used by some of the region creation utilities at Lucasfilm.) If `full` is specified, it dumps not only the thing specified but any things contained within it, and any things contained within *those*, and so on.

raw

`raw` *value* [`full`]

The `raw` command dumps the object, region or avatar specified by *value* in "raw" format. ("raw" format is a special text format for describing database entries. It is used by some of the region creation utilities at Lucasfilm.) If `full` is specified, it dumps not only the thing specified but any things contained within it, and any things contained within *those*, and so on.

The Ghu Command Line

The **Ghu** command issued from VOS command level on the Stratus has the following form:

`ghu` [*databasedir*] [`-command` *commandstring*] [`-env` *environmentfile*]
[`-no_automacro`] [`-cooksize` *size*]

where *databasedir* is the pathname of a directory in which the various database files for a running Habitat system will be found. If omitted it defaults to whatever the directory containing the current operational system is. *(Note: for the time being it defaults to a* **Ghu** *test directory. You have to enter the name of the production system directory yourself.)*

commandstring is an optional string of **Ghu** commands that will be parsed like an ordinary **Ghu** command line. If the `-command` option is used, the commands in *commandstring* will be executed and then **Ghu** will exit. If `-command` is not given, **Ghu** will come up in interactive mode and start reading command lines from the terminal until a `quit` command or an end-of-file is encountered.

environmentfile is the name of a file from which to read (and into which to store) the **Ghu** environment. If omitted it defaults (for the time being) to `#d010>lucas>toolbox>default.env`. We recommend that anyone doing any serious work with **Ghu** keep their own environment file. Stored in the environment file are symbolic name and macro definitions. Whenever you define a symbolic name using the `name` command or a macro using the `macro` command, it is saved in the environment file. When you start up a **Ghu** session, this file is read in automatically and all the names and macros are available to you.

When an environment file is read (either the default file or the one specified with the `-env` option), **Ghu** looks in the file for a macro named `automacro`. If it finds such a macro, it executes it before starting to read commands from the user. The execution of the `automacro` can be circumvented by supplying the `-no_automacro` option on the `Ghu` command line. However, you must have general access (see the next section) to be allowed to use this option.

The default size of the internal array used by the `cook` command (see above) may be adjusted using the `-cooksize` option. The default is 10,000, but it is sometimes necessary to set it to a larger value when

"cooking" huge realms.

Access Control

`Ghu` provides a simple but powerful mechanism for controlling access to the various Habitat databases and to the manipulations to these databases that `Ghu` enables. There are two categories of access that `Ghu` recognizes, which we call "general access" and "restricted access". A user is allowed general access if his or her Stratus account name is listed in the access control file, `#d010>lucas>toolbox>ghu_access`. If the user name is not in this file, then **Ghu** assumes restricted access.

A general access user may do any of the operations described in this document. A restricted access user may only issue macro calls or execute the `quit` command. Note that in order for a restricted access user to be able to do anything useful, this mechanism requires that there be macros pre-defined for him or her to use. These macros come from an environment file that is set up for the user by someone else with general access. Access to environment files is regulated via the `automacro` mechanism described in the previous section. Typically, an environment file will contain an `automacro` that looks something like this:

```
macro automacro
    if user_name != 'joe.user' and !general_access
        value 'Hey, you are not the right guy!'
        quit
    endif
endmacro
```

This macro will throw the user off unless he has user name `joe.user` or he is a privileged (general access) account. Typically, environment files should be set up containing macros to perform various special but limited sets of operations. For example, someone who was responsible for publishing the *Rant* would be given macros for this purpose.

Using Ghu With The Real Database

This section discusses the current Habitat databases in some detail. The information given here describes the database records of the significant object classes, including any important constraints on the values allowed in the various fields. Eventually, **Ghu** will attempt to enforce these constraints automagically. For the time being, however, it only performs the most minimal checks on the data you give it, so you need this information to help you avoid breaking things.

In the descriptions that follow, we will describe each field using the syntax of the `define` command, described above. This will provide the name, dimension, data type, and so on. We will follow this with descriptive text giving other details about the field.

The Basic Object

These are the fields of the "basic object". All objects have these fields, regardless of class. Some objects also have additional, class-specific fields, which will be discussed later.

```
# ident:          objid
```

This is the global ID number of the object. DO NOT CHANGE THIS VALUE UNDER ANY CIRCUMSTANCES.

```
# class:          bin31
```

This is the class of the object. This is always a number in the range 2 to 255. Not all of the numbers in this range are currently used, however, and the `class` field should always have a value that corresponds to an existing class number. In general, you should never change the value of this field, except in the most specialized of circumstances. If you need to change the class of an existing object, consult your local Habitat guru.

```
  container:      entity
```

This is the ID of the thing containing this object. All objects are contained by something. This ID should correspond to an actual object, avatar or region somewhere in the database, with the following exception: `r 99` (region #99) is the "container" for deleted objects. That is, objects which are destroyed are not actually

removed from the database. Instead, they are moved to region #99. There is no region #99, so if you try to display it you will get an error message from **Ghu**. However, you *can* look at its contents with the `contents` command (amazing, huh?) and see the list of deleted objects (this will be a *long* list!).

`x:` `bin15`

This is the horizontal position of the object on the screen. It should be a positive number less than 160 (sometimes a larger value will be used to place an object "off stage", but this should not be done in regular practice). Furthermore, this value *must* be a multiple of 4. A value of 0 corresponds to the left-hand edge of the screen, while 156 (the maximum allowed value in general) corresponds to the right-hand edge. The x-coordinate is only relevant if the object is contained by a region. If the object is contained by an avatar or another object, then the x-coordinate is ignored.

`y:` `bin15`

This is (in general) the vertical position of the object on the screen. However, the interpretation of this field's value is a little bit complicated.

If the object is contained by a region, the `y` field indicates the vertical screen position of the object. The allowed vertical screen positions are in the range from 0 to 127, with 0 indicating the bottom of the screen and 127 indicating the top (actually, the top of the Habitat graphics window, not the top of the screen itself). However, objects may be placed in either the *foreground* or the *background*. Background objects are not redrawn every frame, so it is advantageous, in terms of frame rate, to have as many of the objects in the region as possible be in the background. However, objects that move around cannot be placed in the background, and avatars cannot walk behind background objects. Thus a lamp post on a street corner, for example, should be placed in the foreground. Background objects are given `y` values in the range 0 to 127 corresponding directly to their vertical screen positions. Foreground objects are given `y` values in the range 128 to 255, corresponding to their vertical screen positions plus 128 (i.e., 128 is the bottom of the screen and 255 is the top). Note also that foreground objects should never be placed higher than the region's horizon line (indicated by the region's `depth` field, described below).

If the object is contained by an avatar or another object, rather than by a region, the `y` field encodes the object's position within its container. Each container object has a limited number of containment "slots". The exact number varies with the class of the container object. The `y` field of the contained object must be a positive number that is less than the maximum number of slots in the container. Furthermore, you should be careful that no more than one object goes into a particular slot (i.e., no two objects in the same container have the same `y` value).

`style:` `bin15`

This is the graphic style of the object. The graphic style encodes which of a number of possible graphic images will be displayed for the object. Many objects only have a single possible style, so this value is usually 0. The number of available styles for any class of objects is usually quite small in any case. What the style numbers actually mean in terms of any particular object varies depending on the class. Consult the **Habitat Object Manual** for details.

`gr_state:` `bin15`

This is the graphic state of the object. The graphic state encodes which of a small number of variations of a particular image is to be displayed. Graphic state variations often correspond to functional state variations of the object itself. For example, the two states of a door image, 0 and 1, correspond to the door closed and the door open respectively. As with graphic style, consult the **Habitat Object Manual** for the specific graphic state information about each available class.

`orient:` `bin15`

This field contains the orientation of the object (i.e., whether it is facing right or left). The `orient` field also controls the color or pattern that the object will be displayed with. The value for `orient` is computed as

```
patcol*128 + color*8 + orientation
```

where `orientation` is 0 if the object is to have normal orientation (facing right) or 1 if the object is to have reversed orientation (facing left); `color` is the color or pattern number for the object (the colors are

just the normal 16 Commodore 64 colors; the patterns are the 16 we've all come to know and love); and `patcol` is 1 if `color` is to be interpreted as a color and 0 if `color` is to be interpreted as a pattern number.

`gr_width:` `bin15`

This field is obsolete. In the future it may be renamed and then reused for another purpose. Until then, ignore it.

`restricted:` `bit`

This is a bit-flag that, if set, indicates that the object is ''restricted''. The restriction mechanism is used, for example, to keep library books in the library and board game pieces in the arcade. If an object's `restricted` bit is set, the system enforces the following limitations on the behavior of an avatar holding it: (1) he cannot put the object in his pocket nor into any sort of portable container, (2) he cannot turn into a ghost, and (3) he cannot leave a region through any exit which has its corresponding exit restriction bit set in the region record.

`nitty_bits(31):` `bit`

These are 31 general purpose bit-flags that are not allocated to any particular purpose. They will be used up as features are added to the system. In addition, they may be used for various short-term purposes by games that we might invent which would run for perhaps a few days and then be removed from the system. Ordinarily, however, you need not be concerned with these bits.

`# fillers(5):` `bin15`

This field is simply a place holder. Don't touch it.

`# prop_length:` `bin15`

This field is used internally. ABSOLUTELY NEVER ALTER THIS FIELD!

`# property_data:` `bin15`

This is a dummy field that is used internally as a place-holder for the class-specific fields, if there are any. Don't mess with it.

The Avatar

These are the fields of the avatar record.

`# ident:` `avaid`

This is the global ID number of the avatar. DO NOT CHANGE THIS VALUE UNDER ANY CIRCUMSTANCES

`region:` `regid`

This is the region ID number of the region where the avatar is currently located. It should, of course, be the ID of an actual region.

`container:` `objid`

This is the global ID number of an object containing the avatar. This is used to handle avatars sitting in chairs and the like. If the avatar is not sitting in a chair (i.e., usually), then this field should be set to 0.

`x:` `bin15`

This is the horizontal position of the avatar on the screen. It is interpreted in the same way as the `x` field of an object (see above), with one exception. The exception is that an `x` value of -1 indicates that the avatar is currently a ghost.

`y:` `bin15`

This is the vertical position of the avatar on the screen. It is interpreted in exactly the same way as the `y` field of an object (see above).

`# not_really_turf:` `regid`

This is an historical artifact. Ignore it.

`gr_state:` `bin15`

This is the graphic state of the avatar. Its value is computed as

```
invisible*128 + onhold*64 + moonwalk
```

where `invisible` is 1 or 0 corresponding to whether or not the avatar is to be invisible; `moonwalk` is similarly 1 or 0 corresponding to whether or not the avatar is supposed to moonwalk (i.e., face backwards when walking); and `onhold` indicates whether the avatar is waiting to be displayed in a region. The only avatars who should ever be made invisible are oracle avatars in oracle regions; anything else would be inviting people to invade each others' privacy. Also, you should NEVER set the `onhold` bit.

`gr_width:` `bin15`

This field is obsolete. In the future it may be renamed and then reused for another purpose. Until then, ignore it.

`genl_flags(32):` `bit`

These are 32 general purpose bit-flags that are not allocated to any particular purpose. However, for the time being it is preferable to use the `nitty_bits` (see below), so don't touch these.

`orient:` `bin15`

This field contains the orientation of the avatar. It is similar to the `orient` field of an object (see above). However the interpretation of the number is somewhat different. The value is computed as

```
sex*128 + height*8 + orientation
```

where `orientation` is 0 if the avatar is to face right or 1 if the avatar is to face left (facing front or back is controlled by the `activity` field, described below); `height` is the height of the avatar in the range 0 to 15; and `sex` is 1 if the avatar is female and 0 if the avatar is male.

`# prof_length:` `bin15`

This field is used internally. ABSOLUTELY NEVER ALTER THIS FIELD!

`style:` `byte`

This field tells what sort of avatar this is (e.g., regular avatar, penguin, dragon, etc.). Ordinarily it is 0 (meaning a regular avatar body). See the **Habitat Object Manual** for details about the available avatar styles.

`# filler(27):` `character`

This field is simply a place holder. Don't touch it.

`name(10):` `character = ""`

This is the name of the avatar in ''internal'' form, i.e., all characters converted to lower case and all spaces squeezed out. You can look but DON'T EVER CHANGE THIS VALUE.

`screen_name(10):` `character = ""`

This is the name of the avatar as the user entered it. As with the `name` field, you can look at this value but DON'T EVER CHANGE IT.

`activity:` `bin15`

This field encodes the posture or gesture that the avatar is currently assuming. See the **Habitat Object Manual** for details on the possible values. In general, however, you should not alter the value of this field yourself except in specialized circumstances.

`action:` `bin15`

This is an internal value used for animation on the Commodore 64. Don't touch it.

`health:` `bin15`

This field indicates the relative physical health of the avatar. Allowed values are in the range 0 to 255, with a value of 255 indicating perfect health and a value of 0 indicating that the avatar is dead.

`restrainer:` `bin15`

This is an historical artifact. Don't mess with it.

`custom(3):` `bin15`

These three values encode the customization parameters of the avatar. `custom(1)` is

```
torsoPattern*16 + armPattern
```

where `torsoPattern` is the pattern number to be applied to the avatar's upper body and `armPattern` is the pattern for the arms. Both of these pattern numbers can take values in the range from 0 to 15. Similarly, `custom(2)` is just

```
legPattern
```

i.e., the pattern to apply to the avatar's legs (again, in the range 0 to 15). `custom(3)` is currently reserved for future use.

`bank_balance:` `bin31`

This is the avatar's bank account balance. Be careful how much money you give people.

`turf:` `regid`

This is the region ID of the region that is the avatar's turf. Changing this value changes where the avatar lives.

`stun_count:` `bin15`

This is a counter that is used to implement stun guns and magic which temporarily disables the avatar. If this field has a value greater than 0, then the avatar cannot move. Each time the avatar tries to move, the value of `stun_count` is decremented by one, until it reaches 0 and then avatar can move again. As a rule of thumb, it should never be set to more than 3.

`nitty_bits(31):` `bit`

These are 31 general purpose bit-flags that are not allocated to any particular purpose. They will be used up as features are added to the system. In addition, they may be used for various short-term purposes by games that we might invent which would run for perhaps a few days and then be removed from the system. Ordinarily, however, you need not be concerned with these bits.

`curse_immune:` `bit`

This bit indicates that the avatar is currently immune to curses. Ordinarily it is set by being cured of a curse, and then cleared daily by a batch process. By this means you can only get cursed once per day.

`true_orient:` `bin15`

This field holds the "true" value of the `orient` field, i.e., the value that it was set to when the avatar was hatched. Ordinarily you should not change it. This field (and the other `true_`*xxx* fields below) allow us to restore an avatar to his normal state after having been changed by magic and such.

`true_head_style:` `bin15`

This field holds the "true" `style` value of the avatar's head. This is set when the avatar is hatched and changed whenever the avatar changes his head voluntarily. However, if the avatar's head is changed by magic or other involuntary means, this field is *not* updated. This way we can figure out what the avatar's head is really supposed to be.

`true_custom(3):` `bin15`

These fields hold the "true" values of the `custom` fields (described above).

`curse_type:` `bin15`

This is the curse that the avatar is currently inflicted with, if any. If the avatar is not cursed (that is, almost always), this field should be set to 0. If the avatar *is* cursed, then the number here corresponds to what curse he has. You shouldn't set this field to a non-zero value unless you know what curse you are giving.

`curse_counter:` `bin15`

This is a counter that is used for the maintainance of curses. It is the number of "tags" that the avatar must issue before being freed of the curse. Ordinarily it has values such as 1 or 2.

```
last_on:          bin31
```

This field holds the time that the avatar last signed on. The `display` command will decode it into a readable time and date. You should never change this value yourself.

The Region

```
ident:            regid
```

This is the global ID number of the region. DO NOT CHANGE THIS VALUE UNDER ANY CIRCUMSTANCES

```
owner:            avaid = -1
```

This is the avatar ID of the avatar who ‘‘owns’’ this region, i.e., whose turf this region is. Most regions (‘‘public regions’’) are not owned by anyone. This is indicated by setting this field to -1. Setting this field to 0 indicates that this region is reserved to be a turf region, but has not yet been assigned to anyone.

```
light_level:      bin15 = 0
```

This is the illumination level that is ‘‘natural’’ for the region. A value of 0 or more indicates that the region is lit, while -1 or less indicates that the region is dark. Turning on a light in the region increments this value by one; turning the light off again decrements it. Normally, this value will be set to 0 or -1. Other values result in confusion for players.

```
depth:            bin15 = 32
```

This field indicates how ‘‘deep’’ the region is. That is, how far back from the front of the screen the region extends. In practical terms, this is the y-position of the region’s horizon line. The normal value for this field is 32, although specialized regions may deviate from this.

```
east_neighbor:    regid
west_neighbor:    regid
north_neighbor:   regid
south_neighbor:   regid
```

These fields are the region IDs of the adjoining regions in each of the given directions. A value of -1 indicates that there is no region in that direction.

```
class_group:      bin15
```

This is for future use. Ignore it for the time being.

```
orient:           bin15
```

This is the region’s orientation, or facing, with respect to the Habitat compass. The region’s orientation is the direction that is up, or to the back, when the region is displayed on the screen. The possibles values are 0 through 3, with 0=west, 1=north, 2=east and 3=south.

```
entry_proc:       bin15
exit_proc:        bin15
```

These are code numbers for special procedures to be executed whenever an avatar enters or leaves the region. For the time being, these fields should always be set to 0.

```
east_exit:        bin15
west_exit:        bin15
north_exit:       bin15
south_exit:       bin15
```

These are ‘‘exit types’’ for each of the specified directions. Special actions can thus be associated with each exit. For the time being, however, these fields should always be set to 0.

```
east_restriction:  bit
west_restriction:  bit
north_restriction: bit
south_restriction: bit
```

These are bit-flags that indicate restrictions on each of the respective exits from the region. If an exit's restriction bit is set, the system will not allow an avatar to leave in that direction if he is holding an object whose `restriction` bit is set.

```
weapons_free:       bit
```

This is a bit-flag that, if set, indicates that this is a weapons-free zone region. In such regions the system will not allow weapons to do any damage.

```
theft_free:         bit
```

This is a bit-flag that, if set, indicates that this is a theft-free zone region. In such regions the system will not allow avatars to grab things out of other avatars' hands.

```
nitty_bits(26):     bit
```

These are 26 general purpose bit-flags that are not allocated to any particular purpose. They will be used up as features are added to the system. In addition, they may be used for various short-term purposes by games that we might invent which would run for perhaps a few days and then be removed from the system. Ordinarily, however, you need not be concerned with these bits.

```
name(20):           character = ""
```

This is the "name" of the region. It is a string that is given when the player asks for HELP for the region. It should identify the region's location, if you want the player to be able to find this out. If the player is supposed to be lost when in this region, then set this to a blank string.

```
avatars:            byte
```

This is the maximum number of avatars allowed in the region. It should almost always be 6. It should NEVER be greater than 6.

```
# filler(35):       character
```

This field is simply a place holder. Don't touch it.

Other Objects

Each class of object may have fields which are unique to the class. However, there are several groups of important classes that are similar to one another and so are treated uniformly by much of the Habitat software. For complete details, see the **Habitat Object Manual**. However, the important cases are described below:

magic

The amulet (class 2), gemstone (33), knick-knack (43), magic staff (46), magic wand (47), ring (60), and switch (97) are magical. All magical objects have these fields:

```
magic_type:     bin15
magic_data:     bin31
```

`magic_type` indicates what sort of magic this item has. A value of 0 means that it is not magical after all. `magic_data` is a general-purpose data word whose meaning depends on the type of magic. The allowed magic types will not be described here. Consult your local guru (eventually this will be documented, but it isn't yet).

containers

The bag (class 5), box (13), chest (135), garbage can (32), hole (88), safe (150) are all containers that may be open or closed, locked or unlocked. Furthermore, the bed (class 130), bureaucrat (158), chair (134), couch (137), countertop* (18), display case* (22), glue* (98), pawn machine(96), table (155), vendo front* (85), and vendo inside* (86) are also containers, though they are always open. In addition, the door (class 23), while not a container, obeys the same open/close, locked/unlocked rules as containers do, and so can be treated as a container for many purposes. Containers all have these fields (the classes marked "*" also

have additional fields which are not discussed here; please consult the **Habitat Object Manual**).

```
open_flags:      bin15 = 0
key:             fatword = 0
```

`open_flags` encodes whether the container is open or closed, locked or unlocked. Its value is computed as

```
unlocked*2 + open
```

Where `unlocked` is 1 or 0 depending on whether the container is unlocked or locked, and `open` is 1 or 0 depending on whether the container is open or closed. Note that if the container is open, the value of `unlocked` is really irrelevant. `key` is the key number of the key that locks and unlocks this container. If there is no key for this container, this should be 0. The containers which are listed above as being permanently open should always have their `open_flags` field set to 3 and their `key` field set to 0.

books and paper

The book (class 10), plaque (55) and paper (54) objects all have a field

```
text_id:         bin31
```

that is the text database ID of the document to be displayed for the text of the object. Text IDs greater than 100,000 are reserved for mail. (The book and plaque objects have other fields in addition to this one; ignore them.)

buildings and doors

The building (class 132) and the door (class 23) have a field

```
connection:      regid = -1
```

that indicates which region the door or building is ‘‘connected to’’. Walking to a building or through an open door will take an avatar to this region. A value of -1 in this field indicates the default, which is whatever region is adjoining in the ‘‘up’’ direction from the region containing the building or door.

teleports

The elevator (class 28) and the teleport (class 74) have these fields

```
state:           bin15 = 0
take:            bin31 = 0
address(20):     character = ""
```

The only one of these which is important is `address`. This is the teleport address as it should be displayed by the player HELP (F7) function. `state` and `take` are irrelevant for the elevator. In the teleport they encode whether the teleport is active or not (don’t touch this yourself) and how much money the booth has taken in, respectively.

plants and rocks

The plant (class 58) and the rock (class 61) have a field

```
mass:            bin15 = 1
```

This indicates whether or not the plant or rock can be picked up by an avatar. A `mass` of 0 means that it *can* be picked up, a value of 1 means that it cannot.

Appendix A: Character Escape Codes

These are the character escape codes that you may use in strings to represent non-printing characters and Habitat sign formatting symbols:

`\n`	newline
`\t`	horizontal tab
`\b`	backspace
`\r`	carriage return
`\f`	form feed
`\e`	escape
`\\`	backslash
`\'`	apostrophe
`\"`	quote
`\ `	(backslash-space) half space
`\#`	double space
`\+`	increment text width
`\-`	decrement text width
`\(`	increment text height
`\)`	decrement text height
`\h`	half size characters
`\R`	sign carriage return
`\d`	move half character space down
`\i`	shift to inverse video
`\>`	cursor right
`\<`	cursor left
`\^`	cursor up
`\v`	cursor down
`\^`*c*	CONTROL-*c* (where *c* is any character).
`\`*ddd*	arbitrary byte (where *ddd* is one, two or three octal digits).
`\x`*hh*	arbitrary byte (where *hh* is one or two hexadecimal digits).

Appendix B: Operator Precedences

These are the precedences of the various expression operators. From highest to lowest:

```
( )
?
a o r
.
- ~ ^                (unary - and ^)
* /
+ -                  (binary -)
&
^                    (binary ^)
|
< lt > gt == eq <= leq >= geq != neq
! not
&& and
|| or
```