MUCK Manual Version

Written for TinyMuck version 2.2 fb5.64 & fb6.0
by Jessy @ FurryMUCK

Introduction

A `MUCK` is a computer-driven, text-based 'world'. Users, or 'players', log on to the `MUCK` computer via the Internet, and may then interact with other players. The program running the `MUCK` and the computer it's on are both called the 'server'. Each player controls a 'character', a virtual person or creature inhabiting the world of the `MUCK`. Your character might be very much like you, the player, or she might be very different. `MUCK` is one member of a group related servers, including `MUSH, MUX, MOO, MUSE,` and `MUD;` collectively, these servers are often referred to as `M*'s`, `MU*'s,` or `M***'s.`

Everything is in text; there are no graphics. You will 'see' things from the perspective of your character: whatever place you're in will be described, in words; people and things around you will also be described, and you can look at them. Succinct commands let you say and do things, interacting with other people on the `MUCK` (on a small `MUCK,` this might be six other people; on a large one, it might be 300). Some people view the text-only aspect of `MUCK`s as a limitation, others as a very positive feature. While a screen full of text lacks the instant impact of well done graphics, the world that comes alive in your imagination can be more vivid and colorful than anything computer graphics can produce. Just as the words on the page 'disappear' when one is immersed in a novel — and one instead imaginatively experiences the world described in the book — the words on your computer screen can 'disappear', and you imaginatively experience the world of the `MUCK`. Here, though, you are an active participant rather than a passive observer.

`MUCK`s can be constructed to model any sort of world imaginable, on any scale: a droplet populated by microbes, a one-room bar, an undersea kingdom, or a far-flung stellar empire. By (recent) historical accident, many `MUCK`s are worlds populated by furries. A furry is an anthropomorphic animal, an animal with human characteristics. You might meet an accountant or a college student on a `MUCK`, but you're more likely to meet a suave wolf who likes to quote poetry, or a tiger with a weakness for chocolate truffles. So, yes, furries are inherently silly. But they are also inherently noble: protean and indestructible, they embody who we might be in a world where anything is possible.

1.0 Basic Commands and Setting up a Character

1.1 Some Terms and Abbreviations

Frequently used terms:

Player:
: The person reading this manual; the entity who types things on behalf of a character.

Character:
: The virtual person or creature inhabiting the `MUCK,` controlled by a player. Players have characters; characters have players. The terms 'player' and 'character' are often used interchangebly.

Puppet:
: A player-like object controlled by a character. A 'virtual character', as it were. A puppet can move and act independently of the character; everything the puppet sees and hears is relayed to the character and player.

Zombie:
: Same thing as a puppet.

Robot (or 'bot):
: An object programmed to act like a character, or a character under the control of a program that causes her to perform certain actions automatically. 'Bots are sometimes called `AI's` (from 'artificial intelligence').

Flag:
: A setting on an object that influences or restricts its behavior. Flags are also called 'bits'. Flags are discussed in Section 2.1.

Object Type:
: An object may be a `PLAYER, THING, ROOM, EXIT,` or `PROGRAM.`

  A `PLAYER` is something that can be connected to via a name and password, can own things, and can act independently. Because this object type is called 'player' rather than 'character', many help documents (including this manual) will often use this term, though what's meant is the character object, not the living person who controls it. A `THING` is a mobile object that may be picked up, dropped, handed, and otherwise manipulated. A `ROOM` is a place in the world of the `MUCK,` though it need not be described as an interior room: rooms can also be described to simulate an outdoor area, or described in an abstract way so that moving through the room shows characters useful text (for example, a tutorial on building might be set up as a series of rooms, with the description of each room serving as a 'page' in a maunal). An `EXIT` is a link between two objects on the `MUCK.` Exits may be created such that they serve as virtual doors between rooms, or they may serve as user-created commands. (`EXITs` are also called `ACTIONS`.) A `PROGRAM` is a special instance of class `THING` which contains the code of an executable program. The type of an object is determined when it is created, and indicated by a flag. An object with a `P` flag is a `PLAYER`; an object with an `R` flag is a `ROOM`; an object with an `E` flag is an `EXIT`; an object with an `F` flag is a program. If an object has none of these flags, it is a `THING` . The collective term for all of these is is 'object': players, rooms, etc. are all objects.

Dbref:
: Every object in the database has a unique identifying number: its 'database reference number', or 'dbref'

for short. (Many objects may have the same name.) A dbref is usually preceeded by an # octothorpe, e.g. `#123.`

Client:

A program that replaces `Telnet` as a means to connect to the `MUCK` server. Clients have numerous special features specifically applicable to `M*'s,` such as automated logon, the capacity to switch between several worlds, hiliting certain kinds of text, line-wrapping, separating text being typed from text being displayed by the `MUCK,` paging, and scrollback. Different platforms require different clients. *TinyFugue* is a popular `UNIX` client. *MUDDweller* is a widely used Macintosh client. *SimpleMU*, *Pueblo*, *Phoca*, and *ZMud* are common Windows clients. A single client can connect to different servers... that is, you don't need a separate client for `MUSH,` `MUCK,` and `MUD.` Virtually all client programs are distributed as freeware or shareware, and are widely available over the Internet (see Appendix A).

Penny:

The unit of currency on a `MUCK.` Money is usually a non-issue on `MUCKs,` though some commands require an expenditure of pennies. The administrators of a `MUCK` may set the name of the currency: it might be pennies, or it might be pop-tarts, fleas, or lurid fantasies. You will often find money simply by moving through the `MUCK,` and will see a message such as 'You find a penny!' or 'You find a reason for living!' Many `MUCKs` also have banks or other places where you can get more pennies.

Save:

A period in which the server backs up the database to disk. At scheduled intervals, the server automatically executes relatively brief saves in which only those objects changed or created since the last save are backed up ('delta saves'), as well as longer saves in which the entire database is backed up ('full saves' or 'database saves'). In addition to automated saves, wizards may initiate a save via the `@dump` command. ('Saves' are also called 'dumps'.) During a save, activity on the `MUCK` is frozen: commands you enter during a save will be queued and executed after the save completes. On a very large `MUCK,` a full save can take ten or more minutes.

Lag:

(noun) A perceptible delay between the time you enter a command and the time it is executed. Lag may be caused by an overloaded server (too many people logged on or too many programs running), by an overly large database (the database is larger than available RAM, necessitating frequent swaps to disk), or by problems on the Internet. (verb) To experience lag or a 'locked-up' screen.

Spam:

(noun) Text scrolling by too fast to be read comfortably. (verb) To act or use programs in such a way that those around you are subjected to an excessive amount of frivolous or repetitive text.

Idle:

To be logged onto the `MUCK` but not doing anything.

Mav:

To say aloud something meant to be whispered. By extention, to say aloud something meant to be paged, or to whisper or page to the wrong person. Maving is a potential source of embarrassment or awkwardness. The name derives from a character in the early days of `M*s` who was chronically prone to this particular faux pas.

God:

The `MUCK's` overall controller or administrator, and the player object with dbref `#1.` God has access to a few commands unavailable to all other players, and may not be `@forced.` The term 'God' is used less frequently on `MUCKs` than on `MUSHes` and `MUDs.` In fact, leadership by a single wizard player is relatively rare. More often, a core group of wizards share responsibility for administering the `MUCK,` with each having access to the God character's password.

Wizard:
: An administrator of the `MUCK,` and a player with the `W` flag. Wizards have control over all objects in the database, and may use restricted commands available only to them. ('Wizard' is often shortened to 'wiz'.)

Realm Wizard or Realms Wizard:
: An administrator of a certain area within a `MUCK.` Realms wizards have partial command over objects and players within their area, but cannot use wiz commands. Use of the realms wizard system is somewhat uncommon.

Mortal:
: A non-wizard character. This term too is used less frequently on `MUCKs` than on `MUSHes` and `MUDs.` While wizards are technically players, people usually refer simply to 'wizzes' on the one hand and 'players' on the other. (Where the distinction is important, this manual will use the term 'mortal'.)

Staff:
: A `MUCK` administrator, who may or may not be a wizard, and may or may not have access to restricted commands. A common staff position is 'helpstaff': someone who agrees to help new players and answer questions, but seldom has access to restricted commands. Most wizards are staff, but occassionally a player will be given a wizbit without staff responsibilities and privileges. In other words, 'staff' is an administrative rather than technical term.

`MPI:`
: An online programming language with a `LISP`-like syntax. `MPI` is available to all players. Because it is universally available, `MPI` includes a number of security features that restrict its power. `MPI` is covered in Section 3.

`MUF:`
: An online programming language with a `FORTH`-like syntax. `MUF` security is handled through a system of privileges. In order to program in `MUF,` one must have a 'Mucker bit' (an `M` flag). Mucker bits range from `M1` (highly restricted) to `M3` (almost as powerful as a `Wizard` flag). On well established `MUCKS,` only highly trusted players with demonstrated programming skill are given `M3` bits. The power and efficiency of `MUF` make `MUCK` readily user-extensible. `MUF` is covered in Section 3.

User-Created Command or User-Created Program:
: These terms are not commonly used on `MUCKs,` but are often used in this manual, and so are mentioned here. Many of the commands people are accustomed to using on `MUCKs` are not part of the `MUCK` server, but rather separate programs created by players and wizards. Soft-coded commands, in other words. Some (such as the `say, page,` and `whisper` commands used on most `MUCKs`) are enhancements of server commands. Others (such as `ride, lsedit,` and `lookat`) are basic utilities that do something the server itself cannot. A large `MUCK` will also have a great many other user-created commands and programs: some invaluable, some highly specialized, and some frivolous.

Control:
: This term too is used quite frequently in the manual. In almost all cases, your permission to change an object is determined by whether or not you control it. For mortals, control is essentially synonymous

with ownership: you control everything you own; you don't control things you don't own, with one exception: anyone can control an unlinked exit. Wizards and realms wizards have extended control: Realms wizards control anything in their realm, including players; wizards control everything.

Frequently used abbreviations:

`VR:`

Virtual Reality. The `MUCK` world or worlds. Characters live there.

`RL:`

Real Life. The world outside the `MUCK.` Most players live there.

`M*` or `M***:`

A generic abbreviation for all flavors of text-based, Internet-accessible, interactive programs, including `MUD`, `MUSH`, `MUX`, `MOO`, `MUSE`, and `MUCK`.

`IC:`

In Character. Acting or speaking as your character, rather than as you, the player.

`OOC:`

Out of Character. Acting or speaking as you, the player, rather than the character. Medieval warriors arguing about Mac vs. Windows are `OOC.` In some situations, on some `MUCKs,` being `OOC` without signalling that you are doing so (by putting something like `'(OOC)'` or `'notes OOC'` before your poses and comments) is considered a breach of etiquette.

`IMHO:`

In My Humble (or Holy) Opinion, & `IMO,` In My Opinion

`LOL:`

Laughs Out Loud

`AFK:`

Away From Keyboard

`BRB:`

Be Right Back

`BBL:`

Be Back Later

`TS:`

TinySex. To make love or have sex on the `MUCK,` via the gestures and comments of your character. `TS` is both a verb and noun.

`TP:`

TinyPlot. A role-played, jointly-authored, predominantly improvised storyline acted out by a group of players. `TPs` are usually consensual: players agree ahead of time to participate in the `TP,` though other players may be drawn into the `TP` by the storyline's development. Usually players will plan the main conflict, premise, or events ahead of time — at least provisionally — and improvise their characters' individual contributions and reactions. (The terms 'TinySex' and 'TinyPlot' derive from 'TinyMUD', an early server from which `MUCK` grew out of.)

`RP:`
Role Playing. Acting `IC` in a way that is consistent with either the overall theme of the `MUCK,` a TinyPlot in which one is participating, or both. Some `MUCKs` are predominantly places to `RP;` some are mostly places to socialize, where `RP` is sporadic.

Sidebar: `MUCK` itelf is not an abbreviation or acronym. The names of other `M*` servers are: `MUD` stands for Multi-User Dungeon or Multi-User Domain; `MUSH` for Multi-User Shared Hallucination; `MOO` for `MUD,` Object Oriented. `MUCK` is simply a name with a sound and connotations rather like those of `MUD` and `MUSH.` Purportedly, the name derives from the fact that `MUF` gives players the ability to 'muck around with' the database.

About this Manual

The MUCK Manual comprises five sections. *Section 1* is an introduction for new players, covering basic commands and setting up your character. *Section 2* deals with server commands and the commands included in the standard start-up database. *Section 3* covers programming. Tutorials are presented in *Section 4*. *Section 5* discusses technical and nontechnical issues of administering a `MUCK.` New players should read *Section 1*; others may safely begin with later sections.

The *Overview* sections provide relatively thorough discussion of relevant concepts, but are not comprehensive. The *Reference* sections are comprehensive (or at least attempt to be), but are written in a terse style that assumes the reader has some familiarity with the topic or has read the *Overview*. The remaining sections discuss aspects of the section topic meriting special attention.

```
Examples in the manual are in fixed-width font like this.
```

Text enclosed in [square brackets] is optional. Text enclosed in <angle brackets> should be edited as appropriate for you and your character. For example, where the manual says 'type <your name>', type your name, rather than typing this literally. Within examples, lines beginning with a > greater than sign indicate material you should enter at your keyboard; the remaining lines are sample output from the `MUCK.` (In some longer programming entries, the > sign has been omitted so that you may cut and paste text from the web page to a `MUCK`.)

The MUCK Manual attempts to provide a single, relatively comprehensive reference for `MUCK,` as Lydia Leong's *MUSH Manual* does for `MUSH.` The organization of *The MUCK Manual* is in part based on her manual, and any indebtedness is gratefully acknowledged.

The MUCK Manual was written in collaberation with Winged of SPR and FurryMUCK, author of the online documentation for `MUCK,` version 6.0. Authors and editors of reference materials quoted include Foxen (foxen@belfry.com), Fre'ta (stevenl@netcom.netcom.com), Ioldanach (mortoj@rpi.edu).

This version of the manual was written for `MUCK` version `2.2 fb5.64.` Version `fb6.0` is given partial coverage. Note: this is this is the 'Fuzzball' branch of the `MUCK` server family, which is itself a branch of the `TinyMUD` branch of `M*` server families. In other words, while *The MUCK Manual* discusses `MUCK` as though there were a single standard version, this is not really the case. You should type `@version` on your `MUCK` to see what version you are on. If the version info includes the letters 'fb', you can be pretty sure that material presented in *The MUCK Manual* relates to your sever. (If you have expertise in alternate flavors of `MUCK` and would like to help me get up to speed so that I can expand the manual, then great, let's talk!)

The Manual may be freely copied, distributed, quoted, and archived.

1.2 Connecting to the MUCK

To play on a `MUCK,` you need to be logged onto a character. The most basic way to do this is with Telnet. A client program (see Appendix A) is vastly preferable to `Telnet,` but not essential. For now, examples will assume you are using `Telnet` from a `UNIX` account. If you don't have a character on the `MUCK,` you will need to log on as a Guest. You will also need to find the addresses and port numbers of some open `MUCKs.` Lists of such address are frequently posted on the newsgroup rec.games.mud.announce, and are easily found via web searches for keywords such as 'muck' or 'mud'. The example below use the current address for FurryMUCK (as of Summer, 1999).

Logging on involves two steps: connecting to the `MUCK,` and logging onto a specific character.

To connect to the `MUCK` via Telnet on a `UNIX` account, type `telnet`, followed by a numeric or domain name IP address, followed by a port number. Press enter.

```
> telnet furry.org 8888
```

If the connection is successful (i.e., if you typed everything correctly, the Net is not having a Bad Day, and the `MUCK` itself is running) a login screen will scroll onto your terminal. If you experience a delay, and then a messages such as 'connection incomplete: timed out', wait a few minutes and try again: it's possible that the `MUCK` was undergoing a save while you were trying to connect, or that there were transient connection problems on the Internet.

The login screen should show informational text, illustrations composed of `ASCII` characters, or a combination of these. It is possible to successfully connect to the `MUCK,` yet see a rather garbled looking login screen. Usually this is due to improper terminal settings. If this happens, continue with the login step. The problem may disappear once you've logged on, and if doesn't, people online may be able to help with terminal settings. (This problem is relatively rare.)

Once you've connected to the `MUCK,` you need to log onto a specific character. If you don't have a character on the `MUCK,` you can (usually) connect to a Guest character.

```
> connect guest guest
```

If you have a character, type `connect` followed by the character name and password.

```
> connect cashmere flipFlap
```

On large `MUCKs` (including our example, FurryMUCK) it is common to require Guests to page a wizard and ask to be let out of the Guest Room. This is a security measure: it is not unheard of for players who have been banned from the `MUCK` for causing problems to attempt to log on as a Guest and continue their improper behavior; a wizard can check the site from which a Guest is connecting, and refuse to free someone from a site known to be the home of problem-causers. If this step is required (the text you see after logging on as a Guest will say so), type `wizzes` to get a list of any wizards online, and page one asking him to let you out (paging is discussed below, in Section 1.4.3). If no wizards are online, you'll have to try again later. On most `MUCKs,` this step is unnecessary. You can get out of the Guest Room simply by typing `out`.

1.3 Getting a Character

The `MUCK` server has a provision for automatic character generation, but usually this is disabled. Instead, you'll need to send email requesting a character. The contents of the email you send will vary from `MUCK` to `MUCK:` some simply ask for the character name you want, others require your real name, your age, and your state or country of residence. Help files on the `MUCK` or people online should give you more specific information. Typing `news registration` or `news join for` a helpfile or `staff` for a list of people responsible for answering such questions are good first steps for finding this information.

Once your character is approved and created, you will be sent email containing the character's name (usually the name you asked for) and your password. Sometimes you request a password; sometimes it is randomly assigned. Random passwords may be changed to something memorable once you connect. Passwords are case-sensitive: 'flipFlap' and 'flipflap' are two different passwords.

Character creation by this method usually takes a day or two.

On `MUCKs` that allow automatic character creation, use the `create` command at the log on screen:

```
> create Cashmere flipFlap
```

1.4 Basic Commands

1.4.1 Looking

The `look` command (abbreviated 'l') shows you your surroundings. Typing `look` by itself shows you the room you are in. You automatically see the description of a room upon entering it; typing `look` shows the same description again (it may have scrolled off your screen). Typing

```
    look <object>
```

or

```
    l <object>
```

...shows you the description of `<object>`. The `lookat` command does a possessive look, showing something that is held by someone else. The syntax is `lookat <object>'s <thing>`.

```
    lookat jessy's ring
```

1.4.2 Moving

The `go` or `goto` command causes your character to move in a certain direction or use a specific exit. For example, typing `goto north` would cause your character to move one room to the north (assuming there is a room to the north — or, more accurately, there is an exit named 'north' in your current location, leading to another room). The `goto` command is not really necessary, and is almost never used: typing the exit name by itself will also cause you to 'go' in that direction, and directions are usually abbreviated to their first letter. Thus, typing `north` or `n` should produce the same result as `goto north`. If you can't go in a particular direction (because there is no exit in that direction, or the exit is locked against you), you will see a fail message such as 'You can't go that way.'. A commonly used exit name is 'out' or 'o': typing either of these in an interior room will often move you outside the room or building.

Frequently, the builder of a room makes use of a program that appends the names of 'Obvious Exits' to the description of a room (An 'obvious' exit is one that is not hidden by being set `Dark`). The names of these exits will then appear in a list below the room's description. The absence of such a list does not mean that there is no way out of the room. Read the room's description carefully to see if it indicates directions you can go. If you are still unable to determine valid directions, try common-sense directions such as 'west', 'n', or 'out'. If all else fails, you can get out of room by typing 'home' or 'gohome', which will return you to your home location.

1.4.3 Talking

The `say` command (abbreveated as a " quotation mark) causes your character to say something hearable by all other characters in the same room. Thus, if Cara's player types...

```
say Hi there!
```

or

```
"Hi there!
```

people in the same room will see

```
Cara says, "Hi there!"
```

Characters can communicate via gestures or 'poses' as well as speech. The `pose` command (abbreated as a : colon) causes your character to do something seeable by all other characters in the same room. So, if Cara's player types...

```
pose waves.
```

or

```
:waves.
```

people in the same room will see

```
Cara waves.
```

The pose command handles punctuation intelligently, omitting the space between the character's name and the pose when the text begins with punctuation. Thus, if Cara's player typed...

```
:'s going to go see a friend
```

characters around her would see...

```
Cara's going to go see a friend.
```

rather than

```
    Cara 's going to go see a friend.
```

You can communicate with a specific player, or a group of players, rather than the whole room, via the `whisper` command (abbreviated as 'w' or 'wh'). The syntax is `w <name or names> = <message>` For example, if Clearsong's player typed

```
    w cashmere = What say we blow this joint?
```

...Cashmere would see...

```
    Clearsong whispers, "What say we blow this joint?" (to
    you)
```

If Clearsong's player had typed `w cashmere cara = I thought that was illegal.`, both Cashmere and Cara would receive the whisper. To format a whisper as a pose, put a colon before the whispered message. For example, Cashmere could answer Clearsong by typing...

```
    w clearsong = :nods.
```

Whisper will search the room for partial name matches. Thus, if there is no one else named Clear<something> present, typing `w clear = :nods` would send the whisper to Clearsong. The whisper command 'remembers' the last person you whispered to, so to continue a whispered conversation, the name may be omitted: `w = <message>`.

You can communicate with people in other rooms via the `page` command (abbreviated as 'p'). The syntax for pages, page poses, and pages to multiple players is the same as that of whisper: `p <name or names> = <message>` (for a page) and `p <name or names> = : <pose>` (for a page pose). Like whisper, the page 'remembers' the last person you paged to and the name may be omitted.

(Note: Spaces have been placed around the = equal sign in the above examples for clarity. The spaces are not necessary, but can be included.)

1.4.4 Seeing Who's Around

Space on the `MUCK` is divided into 'rooms', which may or not be described to resemble actual interior rooms. There may be numereous people online, but not in the same room as you.

The names of characters in the same room as you will be appended to the room's description in a list of the room's 'Contents'. Some things in the list may not be connected characters, however: some may be things. some may be puppets, and some may be characters who are not online (they're 'asleep'). To get a list of online characters in the same room, type `who` in lower case. In rooms set `Dark,` the 'Contents' list will not appear: there may or may not be other players present. Guest Rooms and New Player Start rooms are often set `Dark.`

To see who's online on the `MUCK` as a whole, type `WHO`, all upper case, or `3who` for a shorter, three-column list.

1.4.5 Carrying Things

The commands for handling things are quite straightforward: `get <object>` causes you to pick up something; `drop <object>` causes you to drop it. When you go somewhere, objects you are carrying will go with you.

In order to get something, you must be in the same room, and it must not be locked against you.

You can hand things to other players, syntax `hand <object> to <player>`. Being able to hand or be handed to isn't automatic: type `hand #ok` to enable handing.

The `inventory` command, abbreviated `i` or `inv`, shows a list of everything you are carrying, and a line showing how many pennies you have.

1.5 Setting Up Your Character

A new character is essentially a blank slate, having only a name, a dbref, a password, a flag or two, and some pennies. In order to become a fully functioning part of the `MUCK` world, you will need to do a few things to get your character set up. The following section describes a very basic set up. Most of the things you'll do in this section involve setting 'properties'. A property is a named location on a `MUCK` object where information is stored. Properties are discussed fully in Section 2; you don't need to understand exactly how `MUCK` handles properties in order to set up your character. Any commands you issue at this stage are reversible, so don't worry about making mistakes. Simply re-enter the command correctly.

1.5.1 Describing Yourself

Until you describe yourself, people who look at you will see 'You see nothing special'. You can (and should) set what people see when looking you by entering a description, or 'desc' as it's usually called. The syntax is `@desc me = <description>`.

A 'straight desc' such as this is rather limited: the text remains the same at all times, you won't know when someone looks at you, you'll be limited to about twelve lines of text, and you can't divide the text into paragraphs.

Later, you can use `MPI` and the list editor (a program that lets you work with a 'list', a series of props which together act like a document) to create highly flexible descs that change in response to the time of day, who's looking at you, what you're wearing or what mood you're in, and so forth.

Many `MUCKs` have a user-created command that allows you to 'morph', or change descriptions with a succinct command. You may wish to type `morph #help` to see if such a program is available on your `MUCK.`

Most `MUCKs` also have an `MPI` macro that notifies you whenever someone looks at you, called 'look-notify'. To use it, put `{look-notify}` — curly braces included — anywhere in your description. For example:

```
    @desc me = A nymph with green hair.{look-notify}
```

1.5.2 Setting your Sex and Species

Your sex does not have to be male or female, but it's recommended. From time to time you'll encounter programs or commands that format their output based on the user's sex (pronoun substitution is the most common instance). No harm will be done if your sex is something other than male or female, but the program output may look a bit strange.

To set your sex, type ...

```
    @set me = sex : <value>
```

Your species can be anything appropriate to the `MUCK.` On `MUCKs` where everyone is human, the species prop is often used to indicate a profession or social position.

To set your species, type ...

```
    @set me = species : <value>
```

Many `MUCKs` have a user-created command that shows the sex and species of everyone in the room. The most common is `WhoSpecies` or `ws`. Type `ws` to see if it's available on your `MUCK.`

1.5.3 Locking Yourself

On some `M*` servers, locking yourself is a very worthwhile precaution, preventing other players from taking things from you or picking you up. On `MUCKs,` locking yourself really isn't crucial. The server won't let you pick up players, and it won't let you take things being held by other players. If you are not locked, people *can* 'rob' you, which means that they take one of your pennies. Most players have several thousand more pennies than they need.

Nonetheless, you can lock yourself by typing `@lock me = me`. (Locks are discussed in Section 2.)

A more practical issue is that of 'handing' and 'throwing'. You may or may not wish to allow people to hand and throw things to you. Both are convenient, but it is possible to abuse the `hand` and `throw` commands. Abuses range from throwing objects to someone who does not wish to be disturbed (a minor annoyance) to handing someone an innocuous looking object that can eavesdrop on their conversations (an offense that merits being banned from the `MUCK`). Most players do allow handing and throwing.

To allow handing, type `hand #ok`. To allow throwing, type `throw` #ok and `@set me = J`. To disallow them, type `hand #!ok` and/or `throw #!ok`. Both are disallowed by default. (`hand` and `throw` are user-created commands, but most `MUCKs` have them.)

1.5.4 Getting a Home

(Note: It is not necessary to get a home immediately, and sometimes the simplest approach is the best one: page a staff member and ask her to set you up with a home or tell you how to do so.)

'Home' has both a technical and non-technical meaning on `MUCKs.` In the technical sense, all players and things have a 'home', a place to which they are 'linked', and to which they will return if the `MUCK` doesn't know where to put them or if they are sent home by a command or program. A player's home must be a room. A thing's home may be a room or player.

In the non-technical sense, your 'home' is, naturally enough, where you live. Your 'non-technical home' (the place where you hang out, keep your belongings, sleep, etc.) does not necessarily have to be your 'technical home', but it is more convenient for both to be the same.

Getting a home on the `MUCK` involves both technical and non-technical issues. Read this section completely before typing any of the commands.

On the non-technical side, you will need to find a place where you can put a home. There may be places where the builder has installed a command that lets you put in a home: read 'news' and any materials in the place where you start off, or ask people online for information about such places.

If you don't find such a place, or if you find a public place where you would like a home but such a command is not available, you will need to contact the owner of the room and work with her to set up a home. Type `ex here to` find out the name of the room's owner, then contact her by `page` or `page #mail.` On most `MUCKs,` wizards or staff members work with new players to help set up a home. It's worthwhile asking people online how this matter is usually handled on your `MUCK.`

On the technical side, getting home involves three steps:

1. Creating a home.
2. Linking yourself to the home.
3. Linking the home to the rest of the MUCK.

If you are using a program that automates the process, creating a home is simply a matter of following instructions provided on a sign or directory. Usually you just type something like `claim <# or direction>`.

To create your own home, make a room with the `@dig` command, syntax `@dig <room name>`. (You must have a 'builder bit' — the `B` flag — in order to use `@dig.` On most `MUCKs,` all players have a builder bit.)

```
> @dig Cashmere's Den
  Cashmere's Den created with Room #1234.
  Parent set to Outdoor Environment Room(#101RJA).
```

Note the dbref of the newly created room (`#1234` in this example). You will need it in order to go there or link the room to the rest of the `MUCK.` If you forget the room's dbref, you can determine it with the `@find` command.

```
> @find cashmere's den
  Cashmere's Den(#1234RJ)
  End of list
  (1 object found)
```

(If you see the room's name, but not its dbref, you are set `Silent.` Type `@set me=!S to` clear the `Silent` flag set on your character.)

A newly created room is said to be 'floating': it exists, but is not linked to anything else on the `MUCK.` In other words, there are no 'doors' in and out of the room. If your `MUCK` has convenient 'global exits' (commands that take you to a centralized location from anywhere on the `MUCK`) it may be feasible to leave the room floating indefinitely.

If your `MUCK` has a convenient global exit, and you would like to set up a home without having to wait for help from someone else, `@dig` a room as described above, then use the `@link` command (syntax `@link`

<object> = <home>) to set your home and the `home` command to go there.

```
> @link me = #1234
  Home set.
> home
  You wake up at home, without your possessions...
  Cashmere's Den(#1234RJ)
```

As the example suggests, you can loose things when using the `home` command: when you type 'home', both you and anything you are carrying go home. If you are carrying things that do not belong to you, or that are linked to somewhere else on the `MUCK,` they will return to their homes when you use the `home` command (in other words, the `home` command is recursive). There are two ways to avoid this problem. Teleporting may or may not be allowed on your `MUCK` (usually players can teleport to and from rooms they own). If you are able to, teleport to the new room first, then link yourself there.

```
> @tel me = #1234
  You feel a wrenching sensation....
  Cashmere's Den(#1234RJ)
> @link me = here
  Home set.
```

The second alternative is to use `gohome` rather than `home.` The `gohome` command takes you to your home without causing the things you're carrying to also go home. However, `gohome` is a user-created command: though it's widely available, there is a possibility that your `MUCK` does not provide it.

```
> @link me = #1234
  Home set.
> gohome
  You head home, arriving safely.
  Cashmere's Den(#1234RJ)
```

If you plan to leave your room set floating for a while, and your `MUCK` has a `gohome` command, you can just use it whenever you want to go home. If there is no gohome command, it will be worthwhile to create an action that takes you there without causing you to loose things you are carrying. Use the `@action` command (abbreviated `@act,` syntax `@act <action name> = <origin or attachment point of action>`), and the `@link` command (syntax `@link <action> = <destination>`) to create the action

```
> @act myplace = me
  Action created with number #1236 and attached.
> @link myplace = #1234
  Linked to Cashmere's Den(#1234).
```

1.6 Getting Help

There are a number of online resources for getting help with questions or problems.

The server command `help` (syntax `help <topic>`) provides online documentation of many (but not all) server commands and features.

Many (but not all) user-created commands follow the convention of including a `#help` function (syntax `<command> #help`). For example, you can get information on using the page program by typing `page` #help. `MUF` and `MPI` both have online documentation: for `MUF,` type `man <topic>`; for `MPI,` type `mpi <topic>`.

Additional information can be found in the `news` and `info` files. Typing either of these commands should show a list of topics. The news and info files are written and updated by a wizard; their content and quality will vary widely.

The staff should also be able to provide help and information: type `staff` or `helpstaff` to get a list of available staff members. Smaller `MUCKS` may not have a separate helpstaff; in this case type `wizzes.`

And, there's always *The MUCK Manual*.

2.0 Commands

Section 2 documents server commands and the user-created commands provided in the standard start-up database.

2.1 Overview: Dbrefs, Names, Flags, Properties and Lists

All objects on a `MUCK,` of whatever type — Player, Thing, Room, Exit, or Program — have a unique identifying number, a 'database reference number', or 'dbref' for short. Multiple items may have the same name. The type and behavior of all database objects are determined by flags and properties. Both are ways of storing information about the object. Of the two, flags control more basic or fundamental aspects of the object. It might be helpful to think of flags as something that determine *what an object is* and properties as something that determine what *features* or *attributes* and object has (a property is in many ways comparable to an 'attribute' on `MUSH`). Multiple properties can be combined to form a 'list'... a collection of props that together act much like a file or document.

Dbrefs and Names:

A dbref is assigned to each object when it is created. In most cases, when specifying an object by its dbref, the number should be preceded by an # octothorpe. (Some user-created programs require that the octothorpe be omitted when the dbref is stored in a property.) The server assigns either the next available number or that of the most recently recycled object. For example, if the database holds 1,000 objects and all are valid (have not been recycled), the next object created will be given dbref `#1001.` If someone then recycles an object, say `#123,` the next object created would be given dbref `#123.` So, dbrefs are not a reliable guide to an object's age. Dbrefs cannot be changed. Unused dbrefs of recycled objects are 'garbage'.

Multiple objects can have the same name, except for players: there can be several hundred exits named 'out' on a `MUCK,` but only one player named 'Ruffin'... and there could (conceivably) be many things, exits, and programs named 'Ruffin'. All names can be changed with the `@name` command, syntax `@name <object> = <new name>`. To rename a player, the password must be supplied as well: `@name <'me' or old name> = <new name> <password>`. A player must have control of an object to rename it; programs can rename objects if the object and the program have the same controller, or if the program is set `W`. (Programs can only name players indirectly: if the player's password is available, a wizbitted program can force the player or a wizard to rename a player.).

Player names cannot include spaces. `'Madame_Bovary'` is an acceptable player name, but `'Madame Bovary`' is not. The names of other types of objects can include spaces.

When handling or modifying an object in the same vacinity as you, you can specify it by its name or part of its name. Partial names will work, provided that you specify enough characters to distinguish the object from others in the vacinity: `@desc super = <description>` will describe 'Superball', provided that 'Superman' or something else with a name beginning with 'Super-' is not in the same vacinity.

When handling or modifying an object that has the same name as something else in the vacinity, or an object not in the same vacinity as your character, the object will need to be specified by dbref. You can determine the dbref of an object controlled by you and in the same vacinity by examining it (ex <object>). You can determine the dbref of objects that you control but are not in the same vacinity with the `@find` command, syntax `@find <object's name>`.

Two substitution strings can be used in place of either a name or dbref: `'me'` and `'here'`. `'Me'` matches your character's dbref; `'here'` matches the dbref of the room you are in.

```
> @name here = Waterloo Station
  Name set.

> @name pen = Bic Four-Color Ballpoint
  I don't see that here.    (* You left the pen at home. *)
> @find pen
  Nyest Penal Colony, Massage Room(#855RJ) (* Includes 'pen'. Ignore. *)
  pen(#1237)
  End of list
  2 objects found.
> @name #1237 = Bic Four-Color Ballpoint
  Name set.

> @name me = Ruffin flipFlap
  You can't give a player that name. (* Already a player named Ruffin. *)
> @name me = Nebuchudnezer flipFlap
  Name set. (* But Nebuchudnezer works. *)
```

Flags:

Flags — also called 'bits' — provide an economical way to store important information about all objects in the database.

Usually you can see both the dbref and flags of any objects you control when they appear in a Contents or Inventory list, or in the case of rooms, simply by doing a look. Whether or not you can see this information is determined by the `S` flag set on your character: type `@set me =!S to` see dbrefs and flags, or `@set me = S` to hide them (in this context, the `S` flag means 'Silent'). Whether you are set `Silent` or not, you can see the flags set on an object you control by examining it.

The first flag listed after an object's dbref is its 'type flag'. The type flag functions differently than any remaining flags. It determines the type of the object; it is set at the time of the object's creation; it cannot be changed; and it determines the meaning or function of remaining flags. If an object is created with the `@dig` command, it will be a room and will have an `R` flag in the first position. If it is created with the `@action` or `@open` command, it will be an exit and will have an `E` flag. If it's created with the `@program` command, it will be a program and will have an `F` flag. If it's created with the `@pcreate` command, it will be a player and have a `P` flag. If it's created with the `@create` command, it will be a thing, and will have none of these flags. All flags are either 'set' or 'not set' at all times.

The meaning or function of the remaining flags depends on their context... that is, on what type flag the object has. For example, if a program (something with an `F` flag in the first position) is set `D,` the `D` flag means 'Debug', and debugging information is shown whenever the program runs. If a room (something with an `R` flag in the first position) is set `D,` the `D` flag means 'Dark': the 'Contents' list won't appear for a player who doesn't control the room, and no notices are emitted when players connect and disconnect in the room. In short, the same flag won't always mean the same thing. While the context-dependent meanings of flags can be confusing for new users, it provides an elegantly economical way to store important information. The meanings of each flag in relation to the type flags are listed in Section 2.5.

The type flag is set at the time of an object's creation. The remaining flags can be toggled with the `@set` command and the 'not operator' (an `!` exclamation point). The syntax for setting a flag is `@set <object> = <flag>`. For removing a flag, it's `@set <object> = !<flag>`. Flags are not case sensitive: `@set here = D` and `@set here = d` produce the same result.

```
> @set here = D
  Flag set.
> @set here = !D
  Flag reset.
```

Mortals' use of flags is restricted in a few ways. Most importantly, they can only set flags on things they control. Players cannot change the state of the `Zombie` (`Z`) or `Dark` (`D`) flags on themselves. They cannot set themselves or anything they own `Builder` (`B`) or `Wizard` (`W`). They must have a Mucker bit (flags `M1,` `M2,` or `M3`) in order to change the Mucker bit of a program. Players can set the Mucker bit of a program they own to a level lower than or equal to their own, but not higher. Wizard's control all objects and may change the state of any flag on any object, with two exceptions: (`1`) type flags can never be changed; (`2`) if the `MUCK` is compiled with the `god_priv` option (which it usually is), wizards cannot set players `W` or `!W`.

Properties:

A property is a named location on a database object that holds information. Both the server and user-created programs reference properties to obtain the data they need in order to carry out specific operations. For example, when someone looks at you, the server references your description property (`_/de`), retrieves the information stored there (your desc), and displays it to the person who's looking at you. Properties are often called just 'props'.

Props are organized into property directories (often called 'propdirs'). The structure and syntax of property directories are very much like those of file directories in the `UNIX` or `DOS` operating systems, with props being analogous to files and propdirs being analogous to directories. A directory can contain props or additional directories; names of props and directories are separated by a slash; props and directories are organized in a hiearchical 'tree' structure such that each prop has a unique path name. So, the desc prop mentioned above, `_/de`, is actually the `de` property in the _ underscore directory.

You can view the props on any object you control by examining it, syntax `ex <object> = <path>`. Typing `ex me = /` would show the 'first level' or 'root directory' of props and directories stored on your character. Typing `ex me = sex` would show your sex property. Typing `ex me = _page/` would show properties stored in the propdir created and modified when you use the `page` command. Directories will be prefaced by `'dir'` and will end with a slash. Properties will be prefaced by something different (usually `str` for 'string'), and will end with the value stored in the prop.

Like flags, properties are set and removed with the `@set` command, though the syntax is slightly different. The syntax for setting a prop is `@set <object> = <[path/]property>: <value>`. For removing a prop, it's `@set <object> = <[path/]property>:` (that is, a property name followed by just a colon). To clear a directory, it's `@set <object> = <propdir>/:` You can remove all the properties you have set on an object by typing `@set <object> = :clear` .

```
>@set me = obstreperous:yes
  Property set.
>@set me = obstreperous:
  Property removed.
>@set me = personality_traits/obstreperous:yes
  Property set.
>@set me = personality_traits/lascivious:yes
  Property set
>@set me = personality_traits/:
  Property removed.
>@set me = :clear
  All user owned properties removed.   (* oops *)
```

It is common practice to separate words in a property name with underscores (`@set me = my_favorite_color:blue`), but spaces can be used in property names (`@set me = my favorite color:blue`). However, spaces at the beginning and end of property names are removed when the prop is set. (Spaces at the beginning or end of a property *value* are not stripped: you can store a string beginning or ending with spaces, or even a string consisting of only spaces.)

The number, names, and content of properties are not pre-defined as they are in the case of flags. You can't 'make up' a new kind of flag and set it on your character (`@set me = G` or `@set me = 9`, say), but you can create and set any property you like and store any information there, as long as the syntax is correct and the amount of information stored doesn't exceed certain limits . If you wanted to do `@set me = number_of_pickles_in_my_jar:32`, you could, though the information might not be especially useful. (There are some restrictions on what properties you can set, discussed in Section 2.1.1)

While you can set virtually any property, the server and user-created commands will expect specific information to be stored in specific, predefined properties. The server will always reference the `_/de` prop when obtaining a desc; the `hand` command will always check your `_hand_ok` prop. So, using a program or configuring an object is often a matter of determining what props are referenced (by reading #help or program documentation, or by asking players or staff) and setting them appropriately. Important and frequently used properties are stored in the correct location by server commands: `@desc, @success, @osuccess, @drop, @odrop, @fail, @ofail`, and various `@lock` commands all store information in the `_/` directory. (Properties in the `_/` directory and their values are often called 'messages' and 'locks'. See Sections 2.1.2 and 2.3.)

```
>@create Feep
  Feep created with number #1237
>@desc feep = A cute little feep.
  Description set.
>@succ feep = You pick up the feep. It warbles contentedly.
  Message set.
>@osucc feep = picks up the feep. It warbles contentedly.
  Message set.
>@fail feep = You try to pick up the feep, but it scuttles away
whimpering!
  Message set.
>@ofail feep = tries to pick up the feep, but it scuttles away
whimpering!
  Message set.
>@drop feep = You set the feep down gently. It nuzzles your ankle.
  Message set.
>@odrop feep = sets the feep down gently. It nuzzles %p ankle.
  Message set.
>@lock feep = me
  Locked.
>@chlock feep = me
  Chown lock set.

  > ex feep = _/
  lok /_/chlk:Mistral(#100PWX)
  str /_/de:A cute little feep.
  str /_/dr:You set the feep down gently. It nuzzles your ankle.
  str /_/fl:You try to pick up the feep, but it scuttles away whimpering!
  lok /_/lok:Mistral(#100PWX)
  str /_/odr:sets the feep down gently. It nuzzles %p ankle.
  str /_/ofl:tries to pick up the feep, but it scuttles away whimpering!
  str /_/osc:picks up the feep. It warbles contentedly.
  str /_/sc:You pick up the feep. It warbles contentedly.
```

The properties in the `_/` directory trigger events or messages when the object is used in a certain way. For example, the `@success` message is displayed to a player who Successfully uses the object. If the object is a thing, 'successful use' means picking it up. For a room, 'success' means looking at the room. For an exit, it means passing through the exit or using the exit as a command. The `@osuccess` message uses the same definitions of 'success', but in this case the message is shown to Other players present, rather than the triggering player.

`@Fail` works similarly to `@success,` but in this case the message is shown when a player *fails* to use the object successfully, usually because it is locked against him (locks are discussed in Section 2.3), and `@ofail` has a similar relationship to `@osuccess.`

On a thing, a `@drop` message is shown to a player who drops the object; the `@odrop` message is shown to other players present when the object is dropped. When a `@drop` message is set on an exit, the message is shown to the player when he arrives in the destination room. The `@odrop` message is shown to other players in the destination room.

Lists:

In addition to directories, props can also be organized in 'lists'. A list is a group of properties which are handled together in such a way that they emulate a document or computer file. Lists can be created and edited with the list editor, command `lsedit,` syntax `lsedit <object> = <list name>`. This is useful for descriptions that need formatting such as paragraph breaks, indentations, and so forth. Complex `MPI` strings can be stored in a list rather than a property as well, in which case indentation and other whitespace can be used to make the code more readable than it would be as one long uninterrupted string. There are other uses.

```
>lsedit here = maindesc
   <    Welcome to the list editor. You can get help by entering '.h'     >
  < '.end' will exit and save the list. '.abort' will abort any changes. >
  <    To save changes to the list, and continue editing, use '.save'    >
  < Insert at line 1 >

            <you type type type some text text text>

> .end
  < Editor exited. >
  < list saved. >
```

Lists are stored as a set of properties sharing the same path name, and ending with `#/<line number>`.

```
> ex here = maindesc#/
  str /maindesc#/1: (a line of text you entered... )
  str /maindesc#/2: (another line... )
  str /maindesc#/3: (another line... )
  (etc)
```

The list name can be a path name that includes propdirs. For example, you could store multiple descs in propdir `_descs`.

```
> lsedit me = _descs/snazzy
  <enter your 'snazzy' desc>
> .end

> lsedit me = _descs/grungy
  <enter your 'grungy' desc>
> .end
```

(The server, recall, always references the `_/de` property for a description. If you write a description with lsedit, you will need to put an `MPI` string in this property that tells the server where to find the description and how to display it.

```
> lsedit me = _descs/snazzy
  <enter 'snazzy' desc... >
> look me
  You see nothing special.
> @desc me = {list:_descs/snazzy}
  Description set.
> look me
  Ooo la la! Today Mistral is modeling the latest from the Spiegal
  Catalogue: a gownless evening strap with... (etc etc)
```

The `MPI` string stored in `_/de` can be made considerably more elaborate and flexible than the one shown here. (See Section 3.1 for a more complete discussion of `MPI.`)

The syntax for removing a list is `@set <object> = <[path/]list name>#/:`

```
> @set me = descs/snazzy#/:
  Property removed.
```

2.1.1 Protected, Restricted, and Wizard Props

Property handling is governed by a system of privileges, though in most cases this will be transparent to the user: you will usually be able to set the properties you want without even being aware that the server is checking to see if you're allowed to do so. Besides 'normal' props, there are three classes of priviledged properties: protected, restricted, and wizard. The class of a property or propdir is determined by the first character in its name. A prop or propdir beginning with an _ underscore, `%` percent sign, or `.` period is 'protected'. A prop or propdir beginning with a ~ tilde is 'restricted'. A prop or propdir beginning with an `@` at-mark is 'wizard'. If the prop or propdir begins with any other character, it is 'normal' or 'unprotected'. (Note: the 'first character' restriction applies to any property or propdir. So, `@email, @/email`, and `data/personal/@email` are all wizard props: each includes a prop or propdir that begins with an `@` at-mark.)

Properties beginning with a an _ underscore can only be written to by the owner of the object on which the property is stored, or by programs owned by the owner, or by programs running at `M3` or `W`.

Properties beginning with a `.` period can only be written to *or read* by the owner of the object on which the property is stored, or by programs owned by the owner, or by programs running at `M3` or `W`.

Properties beginning with a `%` percent sign, like _ underscore properties, can only be written to by the owner or by programs owned by the owner. The `%` percent sign properties have the additional function of over-riding pronoun and name substitutions. For example, if Jessy does `@set me = %n:the Scamper Gal`, the prop will be protected, and will serve the additional function of causing `'the Scamper Gal'` instead of `'Jessy'` to be substituted in messages that use the `'%n'` substitution string. (See Section 2.1.7 for additional information on substitution strings.)

A restricted property (~) may be read like an unprotected prop, however it may only be modified by a wizard or a wizbitted program. A common use of restricted props is appointing non-wiz staff members. Staff commands, exits to administrative areas, and so forth, can be locked so that they may only be used by characters with specific restricted prop, such as `'~staff'`.

A wizard property (@) may only be read or modified by a wizard or a wizbitted program. The server records connection data in players' propdir @/. On some MUCKs, wizard props and propdirs are used to record administrative information such as players' email addresses. Some wizbitted programs record potentially sensitive data such as mail in a wizard propdir stored on players.

Locks can read any property—including restricted and wizard properties—but may only check for matches with a specific value. (See Section 2.3.)

2.1.2 Messages and Message Properties

As indicated earlier, a set of properties in the _/ directory is given special handling by the server. They can be set with a group of server commands such as @succ, @ofail, etc., and they cause strings to be parsed and displayed automatically whenever certain events happen. Collectively, this group of commands, properties, and their values are called 'messages'.

Command/Message Type	*Property*
@desc	_/de
@succ	_/sc
@osucc	_/osc
@fail	_/fl
@ofail	_/ofl
@drop	_/dr
@odrop	_/odr
@pecho	_/pecho
@oecho	_/oecho

(There does not seem to be a definitive pronunciation for messages: when speaking in RL, some people would say 'at-fail' for @fail, and others would say simply 'fail'.)

The @desc message is evaluated and displayed to any user who looks at an object of any type.

```
> @desc me = A nymph with green hair.{look-notify}
  Message set.
> l me
  [ Kiri looked at you. ]
  A nymph with green hair.
```

The @succ message is evaluated and displayed to any user who successfully uses an object. The meaning of 'success' varies depending on the object type. To successfully use a thing or a program means to pick it up. To successfully use a room means to look at it. To successfully use an exit means to pass through it or use it as a command that triggers a program. To successfully use a player means to steal one of her pennies with the rob command.

```
> @succ out = You step out back.
  Message set.
> out
  You step out back.
```

The @osucc message is evaluated and displayed to all players in a room where a player successfully uses the object, *other than* the player in question. The @osucc message (that is, the string stored in the object's _/osc property) is prefaced with the user's name when it is displayed.

```
> @osucc out = steps out back.
  Message set.

  (Kiri types 'out'... others see...)
  Kiri steps out back.
```

The `@fail` is evaluated and displayed to a user who fails to successfully use an object. The normal reason for failure is that the object is locked against the player. The terms for success or failure are those indicated above, for `@succ`.

```
> @fail vault = Ahem. Only authorized bank employees may open the
  vault. Various alarms begin to sound.
  Message set.
> @lock vault = ~banker:yes
  Locked.
> vault
  Ahem. Only authorized bank employees may open the vault.
```

The `@ofail` message is evaluated and displayed to all players in a room where a player fails to successfully use the object, *except* the player in question. The `@ofail` message is prefaced with the user's name when it is displayed.

```
> @ofail vault = tried to open the vault! Shrill alarms begin ringing!
  Message set.

  (Kiri types 'vault'... others see...)
  Kiri tried to open the vault! Shrill alarms begin ringing!
```

The `@drop` message is evaluated and displayed to a player who triggers a drop. For a thing or program, a drop is triggered when the object is dropped. For a room, a drop is triggered whenever an object is dropped in the room. For an exit, a drop is triggered when a player (or other object type) passes through th exit (using an exit/action linked to a program does not trigger a drop). For a player, a drop is triggered when he or she is killed.

```
> @drop grenade = BANG!
  Message set.
> drop grenade
  BANG!
```

The `@odrop` message is is evaluated and displayed to all players in a room where a player drops an object, except for the triggering player, or all other players in the room a player arrives in when she passes through an exit. The `@odrop` message is prefaced with the user's name when it is displayed.

```
> @odrop out = comes out of the house.
  Message set.

  (Kiri types 'drop out'... the players in the outside room see...)
  Kiri comes out of the house.
  Kiri has arrived.
```

```
> @odrop grenade = drops a grenade! Run! {null:{delay:3,{lit:
  {null:{tell:BANG!}}}}}
  Message set.

  (Kiri types 'drop grenade'... others see...)
  Kiri drops a grenade! Run!
  BANG!
```

`@pecho` and `@oecho` are somewhat different than the preceding messages, having to do with the format of messages transmitted (or 'broadcast') by puppets and vehicles.

By default, output broadcast from a puppet is prefaced by the puppet's name and a > greater than sign. Typing `@pecho <puppet name> = <new preface string>` sets the puppet object's `_/pecho` property, which will become the new preface string. (See Section 4.3 for more information on creating puppets.)

```
> z look
  Squiggy> Amberside Inn, Tavern
  Squiggy> At first glance the tavern seems little changed from
  Squiggy> days when pirate sloops sought haven in the protected
  Squiggy> coves of Amberside's cliffs: the beams are still low
  Squiggy> and smoke-stained...

> @pecho squiggly = *
  Message set.

> ex squig = _/
  str /_/pecho:*
  1 property listed.

> z look
  * Amberside Inn, Tavern
  * At first glance the tavern seems little changed from days when
  * pirate sloops sought haven in the protected coves of Amberside's
  * cliffs: the beams are still low and smoke-stained...
```

By default, messages broadcast from the exterior of a vehicle object to its interior will be prefaced by the string `Outside>` . Typing `@oecho <vehicle object> = <new preface string>` will set the vehicle's `_/oecho` property, which will become the new preface string. (See Section 4.5 for more information on creating vehicles.)

```
> @create 1967 Corvette Sting Ray
  1967 Corvette Sting Ray created with number #558.
> @set 1967 = V
  Flag set.
> @act getin = 1967
  Action created with number #559 and attached.
> @link getin = 1967
  Linked to 1967 Corvette Sting Ray(#558V)
```

```
> drop 1967
  Dropped.
> getin
  1967 Corvette Sting Ray(#558V)

> z :raps sharply on the window... "Can you hear me in there,
  mistress?"
  Outside> Squiqqy raps sharply on the window... "Can you hear
  me in there, mistress?"

> @oecho here = >>>
  Message set.

> z :raps again. "What about now?"
  >>> Squiggy raps again. "What about now?"
```

2.1.3 Data Types and Setting Properties

In the vast majority of cases, props can be correctly set by players and wizards with the `@set` command, as described above. However, in rare cases the 'data type' of a property needs special handling, and a different command is needed: `@propset`.

Data handled by the `MUCK` server is 'typed', as it is in many computer languages, including C (the language the server program is written in) and `MUF` (one of the two programming languages available on `MUCKs`). The server handles different types of data, and most operations require a specific type of data. In this context, the relevant types are 'string', 'dbref', 'integer', and 'float'. (There is also a fifth data type: 'lock', which is discussed in Section 2.3. Type 'float' is only available on `MUCK` versions `6.0` or higher.)

A string is a series of zero or more characters (a zero-length string is called a 'null string'). A dbref is a database reference number. An integer is a whole number. A float is a floating point decimal number. These values can *look* the same but have different *types*. The type of a datum is determined when it is stored. The string of numeral characters "123", the #dbref 123, and the integer 123, and the float 123.0 are *four different values*.

The `@set` command always stores property data as a string. The following three commands store information in three different properties, but it's the same value in each case: a string composed of the characters '1', '2', and '3'.

```
> @set me = my_favorite_dbref:123
  Propery set.
> @set me = my_favorite_integer:123
  Property set.
> @set me = my_favorite_string:123
  Property set.
```

If you typed these commands, and then did `ex me = /`, you would see the three properties, each prefaced by `str,` meaning 'this data is stored as a string'.

```
> ex me = /
  str /my_favorite_dbref:123
  str /my_favorite_integer:123
  str /my_favorite_string:123
```

Programs and commands often store data as dbrefs or integers rather than strings; occasionally, players will want or need to do so as well. The command for doing this is `@propset`, syntax `@propset <object> = <data type> : <[path/]property> : <value>`. The following three commands store information in three different properties, and although they look similar, it's a different value in each case: a dbref, an integer, and a string respectively.

```
> @propset me = dbref:my_favorite_dbref:#123
  Property set.
> @propset me = int:my_favorite_integer:123
  Property set.
> @propset me = str:my_favorite_string:123
  Property set.
```

(Type 'float' has been omitted from this example: at the time of this writing, `@propset` does not handle floating point numbers.)

If you typed these commands, and then did `ex me = /`, you would see the three properties, prefaced by `ref`, `int`, and `str` respectively, with each preface showing the type of the data stored in the property.

```
> ex me = /
  ref /my_favorite_dbref:Hortense(#123PBJ)
  int /my_favorite_integer:123
  str /my_favorite_string:123
```

Tip: there is also a server shortcut for setting properties with values of type integer: put a `^` carot before the number.

```
> @set me = lucky_number:^5
  Property set.
> ex me = lucky_number
  int /lucky_number:5
```

2.1.4 Triggering From Properties

In most cases, programs and server operations are triggered (i.e. caused to run or execute) by typing a command. However, they can also be triggered from a number of protected properties: all message props (`_/de`, `_/sc`, etc.), plus `_connect`, `_oconnect`, `_disconnect`, `_odisconnect`, `_arrive`, `_oarrive`, `_depart`, `_odepart`, and `_listen`.

Performing any of the actions implied by these property names sends information to the server and (if the property is set correctly) causes messages to be displayed, an `MPI` string to be parsed, or a program to run. For example, if you set a `_connect` prop on your character to trigger a certain program, the program will run each time you connect; if you put an `MPI` string in your `_/de` prop, it will be parsed each time someone looks at you. The server searches up the environment tree (see Section 2.2) for triggering props. For example, if `_connect` prop that triggers a program is set on your character, the program will run each time you connect. If it's set on a room, the program will run each time someone connects in that room. If it's set on the global parent room (`#0`), the program will run each time someone connects anywhere on the `MUCK.`

The manner in which the props are set differs depending on which prop it is (the `_/` directory is handled differently than the others, such a `_connect` or `_listen`) and on what the intended result is (messages are handled differently than MPI, which is handled differently than program calls).

To cause a message to be displayed by a `_/` prop (`_/de, _/sc`, etc), simply set the message as a string, with `@set` or the specific server command. `@o-` messages such as `@osucc, @ofail`, and `@odrop` are prepended with the name of the triggering player (or other object type).

```
> @desc out=A simple wooden door.
  Message set.
> look out
  A simple wooden door.

> @succ out=You decide to go outside for a bit...
  Message set.
> out
  You decide to go outsides for a bit...

> @set out=_/sc:You head outside.
  Property set.
> out
  You head outside.
```

`MPI` strings in `_/` props will be parsed automatically.

```
> @desc watch=You glance at your watch. The time is {time}.
  look watch
  You glance at your watch. The time is 01:44:31.
```

To trigger a `MUF` program from a `_/` prop, preceed the dbref of the program with an `@` at-mark. For example, let's assume the dbref of the 'Obvious Exits' program is `#123`:

```
> @succ here=@#123
  Message set.
> look here
  Messy Room(#545RJ)
  Boy, this place is a mess!
  Obvious Exits:
      Out <O>
```

The other triggering props (`_arrive, _oconnect`, etc) are handled slightly differently. Simple strings cause no result, `MPI` must be preceeded with an & ampersand in order to be parsed, and `MUF` programs can be called with just a string indicating the dbref.

```
> @set me=_connect:555
  Property set.
> @set me=_arrive:&{null:{otell:waltzes in.}}
```

```
  Property set.
```

These triggers can be set up as a propdir rather than a single prop, in order to trigger multiple results from the same action. For example, the following settings would trigger both programs `#581` and `#555` each time someone connects in the room. The propdirs are evaluated in alphabetical order, so `#581` would execute first. (Other than determining alphabetical order, the prop names following the first `/` slash mark have no effect: they can be whatever you like.)

```
> @set here=_connect/desc-check:#581
  Property set.
> @set here=_connect/my-wwf:#555
  Property set.
```

The `_listen` is triggered by *any* activity. As such, it is both very useful and very easily abused. Permissions safeguards are coded into the server for `_listen:` the only result that can be triggered is execution of a program; the program must be set `Link_OK` and have a Mucker level equal to or higher than a level set by the `MUCK` administrators. Usually this parameter is set to `3` or `4` (`M4` is 'wizard'). 'Bot programs and automatic 'noise' or 'event' programs are common examples. The prop is set simply by putting the dbref of the program to run in the property value.

```
> @find noises
  noises.muf(#812FLM3)
  ***End of List***
  1 objects found.
> @set here=_listen/noise:812
  Property set.
```

2.1.5 Registered Names

Objects can be specified by 'registered names' as well as by names and dbrefs. A registered name is an alias that can (like a dbref) be used regardless of the object's location, but (like a name) consisting of a memorable string. The primary use is to provide a convenient, memorable way of specifying an item, regardless of its location and ownership. For example, a player on a large `MUCK` might have a puppet with a long, difficult to remember dbref, such as `#128629.` If the player frequently wanted to teleport the puppet to her from somewhere else, she would need to either memorize the dbref or repeatedly retrieve it with the `@find` command. As an alternative, she could give the puppet an easy-to-remember registered name such as 'pup'. From that point on, the puppet could be specified with the name 'pup' preceeded by a `$` dollar sign, rather than by dbref.

Players can create registered names — usuable only by the player — with the `@register` command, syntax `@reg #me <object> = <registered name>`. (The information is stored in the player's `_reg/` directory.)

```
> @find Squiggy
  Squiggy(#128629XZ)
  ***End of List***
  1 objects found.
> @reg #me #128629 = pup
```

```
  Now registered as _reg/pup: Squiggy(#128629XZ) on Jessy(#2PWQX)
> @tel $pup = me
  Teleported.
> i
  You are carrying:
  Squiggy(#128629XZ)
  You have 5086 pennies.
```

Individual registered names may also be set when an object is created. The standard creation commands — `@create`, `@dig`, `@action`, and `@open` — each take two optional argument fields, separated by = equals signs. The first of these fields is specific to each command; the second field for all four may be used to specify a registered name.

```
    @create <name> = <cost in pennies> = <reg name>
    @dig <name>    = <parent room>     = <reg name>
    @action <name> = <source>          = <reg name>
    @open <name>   = <link>            = <reg name>
```

```
> @create Mary Poppins Umbrella == umbi
  Mary Poppins Umbrella created with number 226.
  Registered as $umbi
> ex $umbi
  Mary Poppins Umbrella(#226) Owner: Mistral
  Type: THING
  Created: Fri May 09 14:33:47 1997
  Modified: Fri May 09 14:33:47 1997
  Last used: Fri May 09 14:33:47 1997
  Usecount: 0
> ex me=_reg/
  ref /_reg/umbi:Mary Poppins Umbrella(#226)

> @dig OOCafe = #143 = cafe
  OOCafe created with room number 225.
  Trying to set parent...
  Parent set to OOC Environment(#143RL).
  Room registered as $cafe

> @open Enter Garden = $garden = gogard
  Exit opened with number 224.
  Trying to link...
  Linked to Secret Garden(#455R).
  Registered as $gogard
```

Wizards can set global registered names — usuable by all players — syntax `@reg <object> = <registered name>`. A frequent and convenient use of global registered names is to provide an alias for publicly available programs such as 'do-nothing' and 'obvious-exits'. Without global registered names, players would need to find the dbrefs of these programs each time they needed them for building purposes. Since the players do not control these programs, finding the dbrefs can be difficult. (To create a personal registered name as a wizard, use the `#me` option: `@reg #me <object> = <registered name>`)

```
> @find obv
  gen-Obvexits(#2002FLM3)
  ***End of List***
  1 objects found.
> @reg #2002 = exits
  Now registered as _reg/exits: gen-Obvexits(#2002) on The Void(#0R)
> @succ here = @$exits
  Message set.
```

2.1.6 Pattern Matching and Output Types

As indicated, the `@find` command can be used to locate items with a given name. For more flexible searches, you can use pattern matching (a limited version of regular expressions) to find objects whose name matches a certain pattern, rather than matching the name string literally. A pattern consists of logical and grouping operators, wildcard characters, and/or literal characters. `@Find` —and the related search commands `@owned, @contents`, and `@entrances` — can be used with a subset of standard regular expression conventions, and additional parameters that specify 'output types' (parameters for values such as memory used, time since used or modified, etc.)

Suppose that you have created a Coral Reef area, with interesting scenery and games: players try to avoid sharks, wrestle octopi, and find sunken treasure before they run out of breath and are forced to return to the surface. To use the rooms and games, a player must have a snorkle. You have set up a `'make snorkle'` action linked to an `M2` program that changes an object into a snorkle (it sets all the properties used by your games). And, you have a Snorkle Rental Booth: a room where can people rent snorkles or read 'The Complete Book of Snorkles' and 'Field Guide to the Lesser Coral Reef' (help on how to use the area). The reef rooms have names like 'Coral Reef, Sandy Wash' and 'Coral Reef, Moray's Lair'. Occassionally, you need to find these objects... to recall and update your snorkles or track down your wandering shark-bot, for example.

You can use literal matches to find objects whose name includes the string you are trying to match. Matching is not case-sensitive.

```
> @find complete book
  The Complete Book of Snorkles(#811SJ)
  1 objects found.

> @find coral reef
  Field Guide to the Lesser Coral Reef(#810SJ)
  Coral Reef, Sandy Wash(#802RJ)
  Coral Reef, Moray's Lair(#805RJ)
  Coral Reef, Near the Surface(#809RJ)
  Coral Reef, Weed-Shrouded Cave(#812RJ)
  Coral Reef, Sunken Hull(#815RJ)
  Coral Reef, Dark Cave(#817RJ)
  <etc.>
  <etc.>
  ***End of List***
  16 objects found.
```

The `*` and `?` wildcard characters allow you to search for names that match a pattern, rather than a literal string. The `*` asterix wildcard (sometimes called a 'glob') matches any zero or more characters; the `?` question mark matches any single character.

You could find your two cave rooms in the coral reef area (and leave out other cave rooms you might have) by beginning putting a glob between 'Coral' and 'Cave'.

```
> @find coral*cave
  Coral Reef, Weed-Shrouded Cave(#812RJ)
  Coral Reef, Dark Cave(#817RJ)
  ***End of List***
  2 objects found.
```

You could find all the reef rooms, leaving out the book 'Field Guide to the Lesser Coral Reef', by searching for 'reef' followed by a `?` question mark, which would require that there be at least one character after the string 'reef'.

```
> @find reef?
  Coral Reef, Sandy Wash(#802RJ)
  Coral Reef, Moray's Lair(#805RJ)
  Coral Reef, Near the Surface(#809RJ)
  <etc.>
  <etc.>
  ***End of List***
  15 objects found.
```

The `{curly braces}` grouping operators delimit word patterns. To find all your snorkle objects and the '`make snorkle`' action, but omit 'The Complete Book of Snorkles', you could delimit 'snorkle' as a word, and not simply a string.

```
> @find {snorkle}
  Snorkle Rental Booth(#856RJ)
  make snorkle(#857ED)
  Snorkle 1(#854)
  Snorkle 2(#859)
  Snorkle 3(#881)
  Snorkle 4(#882)
  Snorkle 5(#883)
  Snorkle 6(#884)
  Snorkle 7(#885)
  Snorkle 8(#886)
  Snorkle 9(#887)
  Snorkle 10(#888)
  Snorkle 11(#889)
  Snorkle 12(#890)
  ***End of List***
  13 objects found.
```

One can search for objects whose names include words from a group of valid words by separating the words with the 'or' operator, a `|` vertical bar. For example, you could find all objects that include the words 'sunken' or 'surface'.

```
> @find {sunken|surface}
  Coral Reef, Near the Surface(#809RJ)
  Coral Reef, Sunken Hull(#815RJ)
  ***End of List***
  2 objects found.
```

The `[square brackets]` grouping operators delimit character groups or ranges or characters.

A group of characters in square brackets are treated as valid single characters. A find for `'coral reef, [wdn]'` would find the coral reef rooms with either 'w', 'd', or 'n' following the string 'coral reef, '.

```
> @find coral reef, [wdn]
  Coral Reef, Near the Surface(#809RJ)
  Coral Reef, Weed-Shrouded Cave(#812RJ)
  Coral Reef, Dark Cave(#817RJ)
  ***End of List***
  3 objects found.
```

Instead of typing each valid character, you can also designate a range of valid characters, such as `[0-9]`, `[a-z]`, or `[A-Z]`. You could find all your snorkle objects, which all have a numeric character following the string 'Snorkle ', by using `[0-9]` as the range of characters.

```
> @find snorkle [0-9]
  Snorkle 1(#854)
  Snorkle 2(#859)
  Snorkle 3(#881)
  Snorkle 4(#882)
  Snorkle 5(#883)
  Snorkle 6(#884)
  Snorkle 7(#885)
  Snorkle 8(#886)
  Snorkle 9(#887)
  Snorkle 10(#888)
  Snorkle 11(#889)
  Snorkle 12(#890)
  ***End of List***
  12 objects found.
```

Note that the `[square brackets]` delimit *character ranges*, not *numeric ranges*. A find for `'snorkles [1-12]'` won't work... or won't work as one might intend. It finds all objects with either characters in the range of `'1 to 1'` or the character `'2'` following the string 'snorkles '.

```
> @find snorkle [1-12]
  Snorkle 1(#8454)
  Snorkle 2(#8459)
  snorkle 10(#8488)
  snorkle 11(#8489)
  snorkle 12(#8490)
  ***End of List***
  5 objects found.
```

Output Types

Searches with `@find` and the related commands (`@owned, @contents`, and `@entrances`) may also be narrowed by object type and several other values, and may return additional information such as the objects' owners and locations. The extended syntax is:

```
@find <name | pattern> = <parameter> = <output type>
```

One or several search parameters may be used. Valid search parameters may be a type flag, a Mucker level, or the following special values:

`U`
: Show only unlinked objects

`@`
: Show only old and unused objects

`~<size>`
: Show only objects with current memory used greater than <size>

`^<size>`
: Show only objects with total memory used greater than <size>

An output type parameter must be typed in full; only one output type may be used per search. Valid types include:

`owners`
: List owners along with objects

`links`
: List links along with objects

`size`
: Show memory size

`count`
: Don't list objects: just show total found

`location`
: Show objects' locations

Some examples:

This search would list any unlinked exits you control...

```
> @find = EU
  South;sout;sou;so;s(#528E)
  North;nort;nor;no;n(#533E)
  ***End of List***
  2 objects found.
```

This search would list any old and unused objects you control, along with their locations...

```
> @find = @ = location
  Faded rose(#761)         Mistral(#100)
  ***End of List***
  1 objects found.
```

Objects become 'old and unused' if none of their `'created'`, `'modified'`, or `'last used'` timestamps is more recent than the `aging_time` system parameter.

This search will find all your `M1` programs...

```
> @find = 1
  HelloWorld.muf(#976FM1)
  TrainingWheels.muf(#978FDM1)
  WhatsMyName.muf(#979FM1)
  CountMyBellyButton.muf(#980FM1)
  ***End of List***
  4 objects found.
```

This search will find any objects you control which use more than 2000 bytes of memory, along with the current memory size...

```
> @find = ~2000 = size
  Manhattan Phonebook(#1301)         1932373346 bytes.
  ***End of List***
  1 objects found.
```

For additional information on output types and pattern matching, see the entry for `@find` in the Server Command Reference (Section 2.6) and the entry for `SMATCH` in the `MUF` Reference (Section 3.2.5).

2.1.7 Pronoun and Name Substitution

Messages returned by fields such as `@success, @drop`, etc., and also the formatting of a number of commands and programs such as page, may be dynamically formatted for a player's name and gender: substitution strings (a `%` percent mark followed by a key character) are replaced by the appropriate name or pronoun. Standard substitution strings are:

```
    %a (absolute)       = Name's, his, hers, its.

    %s (subjective)     = Name, he, she, it.

    %o (objective)      = Name, him, her, it.

    %p (possessive)     = Name's, his, her, its.

    %r (reflexive)      = Name, himself, herself, itself.

    %n (player's name)  = Name.
```

Capitalizing the substitution string — such as `%S` or %R – causes the substitute value to be capitalized as well.

The server examines the `/sex` property of the triggering player (or other object type) and substitutes as needed. Supported values for the sex property are 'male', 'female', and 'neuter'. If the property is not set, the player's name is used instead.

```
> @set $pup = sex:male
  Property set.
> @osucc bonk = bonks %r on the head. %S exclaims, "I could
      have had a V8!"
  Message set.
> pp bonk
  Squiggy bonks himself on the head. He exclaims, "I could
      have had a V8!"
> @set $pup = sex:neuter
  Property set.
> pp bonk
  Squiggy bonks itself on the head. It exclaims, "I could
  have had a V8!"
```

The values for these substitutions may be over-ridden by setting a property with the same name as the substitution string. These settings give Squiggy a nickname and some way `PC` pronouns...

```
> @set $pup = %n:The Squigmeister
  Property set.
> @set $pup = %a:hes
  Property set.
> @set $pup = %s:s/he
  Property set.
> @set $pup = %o:hem
  Property set.
> @set $pup = %p:hes
  Property set.
> @set $pup = %r:hemself
  Property set.

> pp bonk
  Squiggy bonks hemself on the head. S/he exclaims, "I could
  have had a V8!"
```

2.2 Overview: Rooms and the Environment Tree.

Rooms are objects created with the `@dig` command and having the type flag `R.`

The syntax for the `@dig` command is

```
    @dig <room> [=<parent> [=<regname>]]
```

The position of rooms —and the resulting 'geography' of the `MUCK` — is determined in two ways. In addition to named exits creating the illusion of spatial relationships (e.g. having a room called 'The Village Green' that can be reached by travelling West from 'The Cove'), rooms exist in a hierarchical tree structure known as the 'environment tree'. One can lead a rich `VR` life without ever needing to know about the `MUCK's` environment tree, but builders and administrators will profit from an understanding of how it works.

As an analogy, one might think of the rooms on a `MUCK` as numerous nested boxes. Room `#0,` the 'global parent', would in our analogy be a large box containing all the other boxes... all the other rooms. Rooms inside `#0` can also contain rooms: the boxes can contain other boxes, in an unending series. The boxes (rooms) can contain items (players, things, etc) as well as other boxes. A room that contains another room is said to be a 'parent room'; a room contained in another room is said to be a 'daughter room'. A given room can be contained in another and at the same time contain other rooms: in this case, the room is both a parent and daughter room. Intermediate rooms of this type are often called 'environment rooms'. (Or, another analogy: rooms are like directories in a computer file system: the root directory is analogous to Room `#0;` environment rooms and rooms are analogous to directories and subdirectories within the root directory; players and objects in rooms are analogous to files in these directories.) Environment rooms are used to define areas of a `MUCK` and to provide commands or features that should only be available in certain areas (more on this below).

You can view the series of rooms containing the room you are located in by typing `@trace here`.

```
> @trace here
  Sinshe Village, by the Pier(#687RJ)
  Sinshe Parent Room(#635RA)
  Environment: Lowlands(#285RA)
  Rainforest Parent Room(#121RWA)
  Rainforest: Main Prarent(#118RA)
  Master Environment(#101RA)
  **Missing**
```

In this example, the administrators of the `MUCK` have carefully laid out a consistent, hierarchical environment tree. In addition to the 'geographical position' of Sinshe Village, each room on the `MUCK` has a specific and meaningful place in the environment tree. The village pier is nested inside — or 'parented to' — the `Sinshe Parent Room (#635),` which presumably contains all the rooms that make up the village of Sinshe. This room is in turn parented to `Environment: Lowlands (#285),` which would contain the parent rooms for all areas in the lowlands. The series continues up through rooms `#121, #118, #101,` and finally, room `#0,` which appears on this list as `**Missing**` (to mortals, rooms not set `Abode` and not controlled by them appear as `**Missing**` on a `@trace`; for security reasons, the global parent of a `MUCK` is usually not set `Abode,` and as a result the last item on the list will be `**Missing**`.)

In fact, not only rooms but all objects on a `MUCK` have a position in the environment tree. Exits are considered to be located in or on the object to which they are attached. Players and things are always located in a specific room or thing... but, unlike rooms and exits, they move around. The `@trace` command works on any object. If a player were holding an object called 'paper sack', `@trace paper sack` would show the sack's current position in the environment tree.

```
> @trace paper sack
```

```
  paper sack(#5474)
  Jessy(#2WQJ)
  Sinshe Village, by the Pier(#687RJ)
  Sinshe Parent Room(#635RA)
  Environment: Lowlands(#285RA)
  Rainforest Parent Room(#121RWA)
  Rainforest: Main Prarent(#118RA)
  Master Environment(#101RA)
  **Missing**
```

A newly created room is parented to the first room above it that is set `Abode,` or the first room controlled by the player issuing the `@dig` command. If no rooms in the environment path meet one of these criteria, it is parented to Room `#0.` Keeping parent rooms set Abode will insure that new rooms are correctly parented: if Jessy stood on the pier and typed `@dig Under the Pier`, the new room would be correctly parented to `Sinshe Parent Room(#635RA)`, the same parent as that of the pier proper.

Overview: Things

Things are objects created with the `@create` command and having no type flag. Both rooms and things can contain other objects.

The syntax for `@create` is:

```
    @create <object> [=<cost> [=<regname>]]
```

The standard start-up database includes the user-created commands `put` and `fetch. Put` allows you to put an item you are carrying inside another item (syntax `put <object> in <object>`). `Fetch` allows you to retrieve an item from inside another (syntax `fetch <object> from <object>`). Partial names may be used for `<object>`.

```
> put kitty snacks in backpack
  Putting Kitty Snacks in Backpack.
> fetch kitty from back
  Fetching Kitty Snacks from Backpack.
```

Things can also be set up as vehicles, by setting their `Vehicle` flag and creating an exit that is both attached and linked to the thing. For vehicles, and occassionally for other objects, you would want to describe the interior of the thing. The interior of a thing can be given a description with the `@idescribe` command, syntax `@idesc <object> = <interior description>`.

```
> @create tRanSMogriFIER
  tRanSMogriFIER created with number 5489.

> @desc trans = A large cardboard box with tRanSMogriFIER
  written on the side in marker, and an arrow that says ----> IN
  Description set.
> @idesc tRanSMogriFIER = An enthusiastic artist has made lots of
  buttons and monsters with a marker on the sides.
  Description set.

> l trans
  A large cardboard box with tRanSMogriFIER written on the side in
  marker, and an arrow that says ----> IN

> @act get in;in;enter = trans
  Action created with number #2543 and attached to tRanSMogriFIER(#5489)
> @link get in = trans
  Linked to gen-nothing.muf(#363FLM2)
> get in
  tRanSMogriFIER(#5489)
  An enthusiastic artist has made lots of buttons and monsters with
  a marker on the sides.
```

See Section 4.4 for more information on making vehicles.

2.2.1 Droptos

If a room is linked to another room or to a thing, the object linked to will serve as the room's 'dropto'. A dropto is a location to which dropped objects will be moved.

```
> @dig Lost and Found = #102 = lnf
  Lost and Found created with room number 198.
  Trying to set parent...
  Parent set to OOC Environment(#102RA).
  Room registered as $lnf

> @link here = #102
  Dropto set.
> Drop bic
  Dropped.
> @contents #198
  Bic Four-Color Ballpoint(#1237)
  ***End of List***
  1 objects found.
```

If a room's `Sticky` flag is set, the drop-to is delayed until all players have left the room. To remove a dropto, `@unlink` the room. To remove a `Sticky` bit, type `@set here = !S`.

2.2.2 Looktraps

'Looktraps' are details or 'fake objects' in a room. For example, rather than creating a 'Sign' object with instructions on how to use some local commands and placing it in a room, you could mention the sign in the room's desc and create it as a looktrap. There will be no separated dbref'd object named 'Sign', but players will still be able to do `look sign` and see it. The primary value of looktraps is efficiency: because no separate object is created, overall database size is somewhat smaller, and commands that must search the database have one fewer objects to examine. And, if your `MUCK` uses a quota system, there will be one fewer object counting against your quota.

Looktraps are stored as properties in the `_details/` directory. Names of looktraps follow the aliasing conventions of exits: strings delimited by ; semicolons serve as alias names.

```
> @set here = _details/sign;plaque;notice:To see who lives here,
  type 'look mailboxes'. To get a home here, type 'claim #',
  using an unclaimed home number for '#'.
  Property set.

> l plaque
  To see who lives here, type 'look mailboxes'. To get a home here,
  type 'claim #', using an unclaimed home number for '#'.
```

Looktraps are automatically parsed for `MPI.`

This looktrap puts the sign text in a list, allowing better control over formatting...

```
> @set here = _details/sign;plaque;notice:{list:signtext}
  Property set.
> lsedit here = signtext
  <    Welcome to the list editor. You can get help by entering '.h'
>
  < '.end' will exit and save the list. '.abort' will abort any changes.
>
  <    To save changes to the list, and continue editing, use '.save'
>
  < Insert at line 1 >
> ... text text text ...
> ... blah blah blah ...
> ... etc etc etc ...
  < Editor exited. >
  < list saved. >
```

This looktrap creates a 'delayed effect'. Players get a little message seven seconds after they look at the portrait.

```
> @set here=_details/painting;picture;portrait:A somber portrait in
  oils, depicting Baron Von Hoofenstaffen, former owner of this
  mansion:{null:{delay:7,{lit:{null:{tell:You're not quite sure: it
  seems, perhaps, that the eyes of the portrait are following your
  moves.}}}}}
```

Note: Many `MUCKs` have soft-coded 'look' commands that handle setting and removing looktraps, with syntaxes such as `look #add <detail> = <desc>`. Type `look #help` to determine if such a system is available on your `MUCK.`

2.3 Overview: Exits and Locks

An 'exit' is a link between two objects on a `MUCK.` When the two objects are rooms, the exit creates a virtual 'door' between them. When the destination is a program, the exit serves as a user-created command. Other combinations are possible. Exits are also called 'actions'. Use of exits (as well as objects of other types) is controlled by 'locks': an expression that evaluates as either 'true' (in which case the exit/object can be used) or 'false' (in which case it cannot). An exit is characterized by having a starting point (an object to which it is 'attached') and a destination point (an object to which it is 'linked').

Exits

Exits are created with either the `@open` or `@action` command. Both create an exit (an object with type flag E), but the syntax and defaults are slightly different.

The basic syntax of the `@open` command is `@open <exit name>`. An exit created in this way will be attached to (i.e., start from) the room in which one issues the command, and it will not be linked to anything (it won't lead anywhere). The exit can be linked to another object on the `MUCK` with the `@link` command, syntax `@link <exit name> = <destination>`. Since the destination will usually be somewhere else on the `MUCK,` it will need to be specified by dbref rather than name.

Hooking up an exit in two steps...
```
> @open out
  Exit opened with number #1766.
> @find hallway
  Ansley Inn, Hallway(#198R)
  1 objects found.
  ***End of List***
> @link out = #98
  Linked to Ansley Inn, Hallway(#98R)
```

Hooking up an exit in one step...
```
> @open out = #98
  Exit opened with #1766.
  Linked to Ansley Inn, Hallway(#198R)
```

An exit does not have to be attached to a room, however: an exit can be attached to anything except a program or another exit. The `@action` command (abbreviated as `@act`) creates an action/exit attached to an object specified at the time of creation:

```
> @act myplace = me
  Action created with number #1236 and attached.
> @link myplace = #1234
  Linked to Cashmere's Den(#1234).
```

Many `MUCKs` have a soft-coded `@action` command, that allows you to specify both the source and destination at the time of the exit's creation:

```
> @act myplace = me,#1234
  Action created with number #1236 and attached.
  Trying to link...
  Linked ot Cashmere's Den(#1234)
```

The attachments and links of exits can be changed. To relink an exit, issue the `@unlink` command and then `@link` the exit to the new destination.

```
> @unlink myplace
  Unlinked.
> @link myplace = #1768
  Linked to Cashmere's Bachelor Pad(#5784R).
```

To change an exit's point of attachment, use the `@attach` command, syntax `@attach <exit> = <new attachment point>`

```
> @attach refrigerator = here
  Action re-attached.
```

(Obvious-exit programs generally list exits in the order of first-attached to last-attached, or the reverse. Therefore, the order in which exits appear on the list can be changed by using `@attach` to re-attach exits to the room: the exit will then become the last-attached exit, and move to either the first or last position in the list.)

To reiterate, exits have a source (the object to which they are attached) and a destination (the object to which they are linked). This means that they are *one way*. This point often causes confusion for new builders: in order to create a 'door' between two rooms, one needs to create two exits, one leading in each direction. The following example illustrates this: `Cashmere's Bachelor Pad` has dbref `#5784.` From the Bachelor Pad, he will create a Bedroom, and then create two exits that make a door between the Pad and the Bedroom.

```
> @dig Cashmere's Bedroom
  Cashmere's Bedroom created with #5792.
  Parent set to BD's Environment Room(#4989RA).
> @open Bedroom = #5792
  Exit opened with number #5793.
> bedroom
  Cashmere's Bedroom(#5792)
> @open Out = #5784
  Exit opened with number #5784.
  Linked to Cashmere's Bachelor Pad(#5784R).
```

Exits linked to things move the the thing to the point of attachment when used (rather than moving the user to the thing).

```
> @act getpup = me
  Action created with number #4684 and attached.
> @link getpup = $pup
  Linked to Squiggy(#128629XZ).
> getpup
  done
> i
  You are carrying:
  Squiggy(#128629XZ)
  You have 10664 wet cats.
```

Exits' names can include 'aliases', other names that can be used as the exit name. An existing exit can be renamed to include aliases with the `@name` command, or the aliases can be specified at the time of the exit's creation.

Renaming an existing exit...

```
> @name bedroom = Bedroom <B>;bedroom;bed;b
  Name set.
```

Creating an exit with aliases...

```
> @open Out <O>;out;ou;o
  Exit opened with number #5785.
```

Each string separated by a ; semi-colon is an alias for the exit. In the 'out' example above, typing either `out <o>, out, ou,` or `o` would cause the player to use the out exit. Only the first name or alias is shown a list of obvious exits. The above examples follow the common and useful convention of supplying a 'full' exit name along with a simple abbreviation in the first alias.

```
> look
  Cashmere's Bachelor Pad(#5784R)
  Obvious Exits:
        Bedroom <B>      Out <O>
```

Locks:

Locks are expressions used to control the use of objects. The most common applications are to lock exits so that only some people can use them, and to 'lock down' things that you need to leave lying about in rooms (a sign or bulletin board, for example). Just what constitutes 'use' depends on the object type. To 'use' a thing means to pick it up. To 'use' an exit means to pass through it to another room, or to use it as a command. To 'use' a room means to look at it.

If an object is locked it may be used if and only if the lock expression is 'true' for the player (or other object type) attempting to use the object. 'True' in this context and in rather nontechnical terms means 'if the triggering player/object *is* or *has* or *is owned by* a result of the lock expression'.

A simple example...

```
> @lock closet = me
  Locked.
> ex out = _/
  lok /_/lok:Kenya(#75PBJM1)
```

A lock expression is stored as data type 'lock' (see also Section 2.1.3), as indicated indicated by the prefix `lok,` and meaning that the server will evaluate the expression for 'truth' in reference to the triggering player (or other triggering object type). The lock in this example evaluates to 'database object `#75'.` If the player or other object trying to use the exit 'is' `#75` (that is, if it is Kenya), then the lock 'passes': Kenya will be able to use the exit. Further, if the the triggering player/object 'has' Kenya, the lock would pass: if Kenya were inside a vehicle object, the vehicle would be able to use the exit. And, if the triggering object 'is owned by' Kenya, the lock would pass... for example, a puppet owned by Kenya would be able to use the exit. And (this is the whole point of locks) *only* these objects would be able to use the exit.

Property values as well as dbrefs can serve as lock expressions. The syntax for locking an object to a property value is `@lock` <object> = <property> : <value>

Only females can use this exit:
```
> @lock Powder Room = sex:female
  Locked.
> ex out = _/
  lok /_/lok:sex:female
```

If the triggering player/object 'has' the property value 'female' in her 'sex' property, the lock passes: in other words, only females can use this exit. (When the system parameter `lock_envcheck` is tuned to 'yes', the server searches rooms in the environment tree path for property matches on locks, as well as the triggering player/object. In this case, if room `#0` were set `sex:female`, the lock in the above example would always pass.)

Wildcard characters (see Section 2.1.6) may be used in locks for property values.

This exit may be used by males and females, but not fluffs... lyve wyth yt.
```
> @lock Normal Folks Bar and Grill = sex:*ale
  Locked.
```

Lock expressions may contain or evaluate to multiple elements. The elements must be separated by an 'or' operator (a `|` vertical bar) or by an 'and' operator (an & ampersand), and may optionally be preceded by a 'not' operator (an `!` exclamation point). Player names may be used, provided that they are preceded by an `*` asterix pointer.

This exit may be used by either of two players, Passiflora or Kenya...
```
> @lock Den of Iniquity = *passiflora|*kenya
  Locked.
```

This exit may only be used by female staff members...
```
> @lock Wizards' GunPowder Room = ~staff:yes&sex:female
  Locked.
```

This exit may be used by anyone (or anything) who is *not* Stinker...
```
> @lock Jessy's House = !*stinker
  Locked.
```

This exit may only be used by someone who *is* Jessy and who *is not* Jessy. In other words, this lock alway fails...
```
> @lock chair = *jessy&!*jessy
  Locked.
```

The operators have the following order of precedence:

1. ! not
2. | or
3. & and

The order of precedence can be over-ridden by enclosing sub-expressions within (parentheses).

This exit may be used by all females and all others who are *not* Stinker...
```
> @lock bar = !*stinker|sex:female
  Locked.
```

This exit may be used by all who are *not* female and *not* Stinker...

```
> @lock bar = !(*stinker|sex:female)
  Locked.
```

An object can be locked to a `MUF` program, in which case the program runs when someone uses or attempts to use the object, and the lock passes if the program returns a true *`MUF`* value (that is, `MUF's` truth conditions over-ride the truth conditions for locks). In `MUF,` a `""` null string, the integer `0,` the floating point value `0.00000,` and the dbref for 'nothing', `#-1,` are false. If the `MUF` program returns any of these values, the lock fails; if it returns any other value, the lock passes.

A lock cannot include `MPI` (or rather, will invariably fail if set to an `MPI` string), but an object can be locked to a property value, and the property value on a relevant object can be set to an `MPI` string.

This exit may only be used on the stroke of midnight (or by someone who has figured out the lock and set his 'time' property to `00:00:00`)...

```
> @lock time machine = time:00:00:00
  Locked.
> @set time machine = time:{time}
  Property set.
```

Locks may be removed with the `@unlock` command, syntax `@unlock <object>'`

2.3.1 Bogus Exits

It is (or was) a relatively common practice to create lookable details and realism-enhancing actions in rooms by means of 'bogus exits'... exits that do not lead anywhere. The exits can be given a description, so that doing `look <exit>` shows some detail of the room, and realistic messages can be put in the exit's `@fail/@ofail` or `@succ/@osucc`.

```
> @open Grandma's Rocker;grandmas rocker;rocker;chair;sit
  Exit opened with number #5797.
> @link chair = $nothing   Linked to gen-nothing(#114).
> @desc chair = An old, old rocker that has been in the family
  for generations.
  Description set.
> @succ chair = You take a seat in the old rocker.
  Message set.
> @osucc chair = takes a seat in the old rocker.
  Message set.
```

The above example creates a 'virtual chair'. Though it will not appear in the room's Contents list, people can look at it, and can sit in it by typing `sit` (or any of its other aliases).

Bogus exits may have their place, but builders should be aware that there are other ways of accomplishing the same goals without creating a separate exit for each item (an approach that quickly leads to dbase bloat). Lookable details can created with 'looktraps' (see Section 2.2.2) and many events like the 'sit' message in the above example can be handled by a single action (See Section 3.1.2, `MPI` Examples).

2.3.2 Unsecured Exits

An exit that is not linked to anything and not thoroughly locked is unsecure, in that its ownership transfers to anyone who uses the exit and anyone can link it. This creates a minor security risk: someone could take control of an exit attached to one of your rooms and make it do something annoying or harmful. So, always secure exits. An exit can be secured by locking it to a condition that always fails — such as `me&!me` — or by linking it to something. All established `MUCKs` provide a 'do-nothing' program, a program that produces no result, and thus serves as a convenient linking point for exits. Usually such a program is given the registered name `$nothing` or `$do-nothing`.

```
> @link sit = $nothing
  Linked to gen-nothing.muf(#363FLM2).
```

2.3.3 Exit Priorities

In our discussion of the environment tree, it was noted that the server searches up the environment tree for commands matching users' input. If more than one command with the same name is found, the server must resolve which command to execute. This is determined by the 'priority' of the exits, and the order of the search path. Both are affected by the system parameter `compatible_priorities`.

Wizards can set Mucker bits on exits as well as on programs and players. An exit with a higher Mucker bit runs at higher priority than an exit with a lower Mucker bit, or one with no Mucker bit.

For example, suppose a `MUCK` has a global exit named 'bank' linked to a program that gives players 100 pennies, and a player has an exit in his room named 'bank' linked to a program that gives players 500 pennies. If neither exit has a Mucker bit set, both are considered 'Priority 0' (zero). The *first* exit found in the search path would be executed: a player standing in the room with the 'local' exit would receive 500 pennies; elsewhere, the global 'bank' command would run, and the player would receive 100 pennies.

However, if a wizard set the global 'bank' exit `M1,` the global exit would now have higher priority. Even in the room with the 'local' exit, typing 'bank' would execute the global exit, and players would receive 100 pennies.

As indicated, if there are two exits with the same name and the same priority, the server executes the *first* exit found. But the order of the search path changes depending on whether the system parameter `compatible_priorities` is set to 'no' or 'yes'. (Wizards may set system parameters with the `@tune` command.)

If `compatible_priorities` is set to 'no', all non-prioritied exits (i.e., exits with no Mucker bit set) are considered 'priority 0', and the server uses the following search order:

1. On the room the player is located in
2. On objects the player's inventory
3. On objects in the room's inventory
4. On the player
5. Environment rooms containing the present room, beginning with the 'closest' room... the room furthest from room `#0`
6. Room `#0`
7. The server

If `compatible_priorities` is set to 'yes', all non-prioritied exits are considered 'priority 1', and the server uses the following search order:

1. On the room the player is located in
2. On the player
3. Environment rooms containing the present room, beginning with the 'closest' room... the room furthest from room #`0`
4. Room #`0`
5. Objects in the player's inventory
6. Objects in the room's inventory
7. The server

In our example, the wizard had just set the global 'bank' exit `M1,` so it had a higher priority than the local `M0` exit. If the wizard then did `@tune compatible_priorities = yes`, both exits would now be considered 'priority 1': the global is priority 1 because it is set `M1,` and the local is considered priority 1 because the system parameter is set to run all unprioritied exits at priority 1. The search order for players and inventories has changed, but in both cases the local room is checked before the global parent #`0.` So, now the local exit would run when player's type 'bank' in the room with the local exit. If the wizard then set the global exit `M2,` it would again have higher priority than the local exit, and would run regardless of where a player is standing when typing 'bank'.

If you have difficulty getting a local or personal exit to run in preference to a global of the same name, contact a wizard and discuss modifying priorities, either by raising your exit's priority, or by changing the system parameter.

2.4 User-Created Commands

We have used the term 'user-created commands' throughout the manual to indicate 'add-on' commands created by the wizards or players of a `MUCK.` Most other issues affecting such commands — the command search path, exit priorities, locks, etc. — have been discussed at various points above.

Because efficient and flexible commands can easily be added to a `MUCK,` the platform is highly customizable. In general, this is a Really Good Thing: `MUCKs` can be continually improved and tailored to their population's needs, without hacking the server code, which often results in bugs and invariably results in innumerable versions and patch levels of the program.

On the downside, you can't assume that something you've learned on one `MUCK` will work exactly the same way on a different `MUCK.` And, `The MUCK Manual` cannot provide a complete reference for any given `MUCK.`

Most `MUCKs` do share a core set of standard programs, libraries, and commands, distrubuted as the 'standard database'. Discussion of the standard commands is provided in Section 2.7, User-Created Command Reference. The programming libaries are discussed in Section 3.2.6, `MUF` Library Reference.

2.5 Flag Reference

A (Abode, Abate, Autostart)

On a Room: Anyone can set their home or the home of objects to the room.

On an Exit: The exit is lower priority than an exit without the `Abate` flag. If `compatible_priorties` is tuned to 'no', an abated exit's priority is 'less than 0'; If `compatible_priorites` is tuned to 'yes', an abated exit's priority is 'less than 1'.

On a Program: The program will automatically be loaded into memory and executed when `the MUCK` starts or restarts.

B (Builder, Bound, Block)

On a Player: Player can create and modify objects with the `@create, @dig, @link`, and `@open.` On most `MUCKs,` players start off with a `B` flag. A `B` flag on players is also called a 'builder bit'.

On a Room: Personal exits (exits attached to a player) cannot be used in the room.

On a Program: Any functions within the program run in preempt mode. If the program set `B` is called by another program, multi-tasking status returns to that of the calling program when execution exits from the called program.

(Only wizards can set and remove B flags.)

C (Chown_OK, Color)

On any object except Players: Anyone can take control of the object with the @chown command ('change ownership').

On a Player: On `MUCK` versions `5.x` and lower, no effect. on 6.x, `MUCK` output will be formatted with `ASCII` color, provided that the player's client handles color and that the text has been formatted with color.

D (Dark, Debug)

On a Room: Wizards and the owner of the room see all objects normally, but other players see only objects they own. If no objects would be seen, a 'Contents' list is not appended to the description of the room.

On a Thing: The object does not appear in the room's 'Contents' list.

On a Player: The player does not appear in the 'Contents' list of rooms or in the `WHO` list. Only wizards may set players dark.

On a Program: A stack trace of internal program operations is printed out to anyone who uses the program.

`E (Exit)`

Type Flag: The object is an Exit/Action.

`F (MUCK Forth Program)`

Type Flag: The object is a program.

`H (Haven, HardUID)`

On a Room: The `kill` command may not be used in that room.

On a Player: The player cannot be paged.

On a Program: The program runs with the permissions of the owner of the trigger, rather than with the permissions of the user of the program. When this is set in conjunction with the `Sticky` (`Setuid`, below) flag on a program, and the program is owned by a wizard, then it will run with the effective mucker level and permissions of the calling program. If the caller was not a program, or the current program is *not* owned by a wizard, then it runs with `Setuid`.

`J (Jump_OK)`

On a Room: Players can teleport to and from the room (assuming other conditions for teleporting are met). If the `MUCK` is configured with `secure_teleporting, J` indicates that exits attached to players and objects can be used to leave to leave the room, and `!J` indicates that they cannot.

On a Thing: The object can be moved by a program running at any Mucker level.

On a Player: The player can teleport to and from rooms (assuming other conditions for teleporting are met).

`K (Kill_OK)`

On a Player: The player can be killed with the `kill` command. A player who is 'killed' is simply sent home.

`L (Link_OK)`

On a Room: Anyone can link exits to the room.

On a Program: The program can be called by any program, and can be triggered by actions and propqueues not owned by the owner of the program.

`M1 (Mucker Level 1)`

(See also Section 3.2.1)

On a Player: The player is an 'apprentice' Mucker. He can use the `MUF` editor and create `M1` programs.

On an Exit: The exit runs at priority 1.

On a Program: The program runs with Mucker level 1 permissions. The program cannot get information about or send information to any object that is not in the same room. Some `MUF` primitives cannot be used. Program output to anyone except the triggering player is prepended with the triggering player's name. Instruction count is limited to about 20,000 instructions. The program follows permissions for protected props (see Section 2.1.1).

M2 (Mucker Level 2)

(See also Section 3.2.1)

On a Player: The player is a 'journeyman' Mucker. She can use the `MUF` editor and create `M2` programs. She can set the Mucker level of any program she controls to `M1` or `M2.`

On an Exit: The exit runs at priority 2.

On a Program: The program runs with Mucker level 2 permissions. Some `MUF` primitives cannot be used. Instruction count is limited to about 80,000 instructions. The program follows permissions for protected props (see Section 2.1.1).

M3 (Mucker Level 3)

(See also Section 3.2.1)

On a Player: The player is a 'master' Mucker. He can use the `MUF` editor and create `M3` programs. He can set the Mucker level of any program he controls to `M1, M2,` or `M3.`

On an Exit: The exit runs at priority 3.

On a Program: The program runs with Mucker level 3 permissions. Almost all `MUF` primitives can be used. There is no absolute limit to instruction count, unless the program is running in `PREEMPT` mode. The program may over-ride the permissions for protected props.

P (Player)

Type Flag: The object is a Player.

Q (Quell)

On a Player or Room: The `Quell` flag cancels the effects of a wizard flag. A wizard player set `Q` is effectively a normal player. A `Q` flag on a wizbitted room will cancel the realms-wiz powers of the room's owner.

R (Room)

Type Flag: The object is a Room.

S (Silent, Sticky, SetUID)

On a Thing: The object will return to its home when dropped.

On a Room: The room's drop-to is delayed until all players have left the room.

On a Player: The player will not see dbrefs on things she owns, and will not see objects in a `Dark` room. Control is unchanged however.

On a Program: The program runs with the permissions of the owner of the program, and not those of the user.

W (Wizard)

On a Room: The room's owner has Realms Wiz powers in that room and any rooms parented to it, provided that the `MUCK's realms_control` parameter is set to 'yes'.

On an Exit: The exit runs at priority 4.

On a Player: The player is a wizard. Wizards have control over all objects in the database (although with some restrictions in their control over God and other wizards). Wizards can use restricted, wiz-only commands, and can set programs, rooms, and things `W` and `B.` Some wizard powers are enabled or disabled by the system parameter `god_priv`.

On a Program: The program is effectively Mucker level 4. All `MUF` primitives may be used, and do not have a maximum instruction count unless the program is running in preempt mode.

X (Xforcicble)

On a Player or Thing: The player or thing may be forced by a player (or other object type) to which it is `force_locked`.

V (Vehicle)

On a Thing: The object is a vehicle.

On a Room: Vehicles may not enter the room.

On an Exit: Vehicles may not use the exit.

Z (Zombie)

On a Thing: The object is a Zombie: all output the Zombie sees or hears will be related to the controlling player.

On a Room: Zombies may not enter the room or be forced in the room.

On an Exit: Zombies may not use the exit.

2.6 Server Command Reference

@ACTION

```
 |  @ACT
@action <name>=<source> [=<regname>]
```

Creates a new action and attaches it to the thing, room, or player specified. If a `<regname>` is specified, then the `_reg/<regname>` property on the player is set to the dbref of the new object. This lets players refer to the object as `$<regname>` (eg: `$mybutton`) in `@locks`, `@sets`, etc. You may only attach actions you control to things you control. Creating an action costs 1 penny. The action can then be linked with the command `@link`.

@ARMEGEDDON

`@armedeggon`

Shuts down the `MUCK` without first doing a save. The primary purpose is to avoid over-writing the saved database if it becomes apparent the current database is corrupt, or when someone (usually a wizard) has done something quite stupid, destroying a large amount of objects or data. (Wizard only)

@ATTACH

`|  @ATT`

@attach <action> = <new source>

Removes the action from where it was and attaches it to the new source. You must control the action in question.

@BOOT

`@boot <player>`

Disconnects a player from the game. If a player is connected more than once it affects the most recent connection. (Wizard only)

@CHOWN

`@chown <object> [=<player>]`

Changes the ownership of `<object>` to `<player>`, or if no player is given, to yourself. If the `MUCK` is compiled with `player_chown,` all players are allowed to take possession of objects, rooms, and actions, provided the `Chown_OK` flag is set, with the following exception: mortals may `@chown` an exit if they own the object it is attached to, or an object it is linked to. Mortals cannot take ownership of a room unless they are standing in it, and may not take ownership of an object unless they are holding it. Wizards have absolute power over all ownership.

@CHOWN_LOCK

`|  @CHLOCK`

`@chown_lock <object> = <lock expression>`

Locks `<object>` such that it may only be chowned by players for whom `<lock expression>` is true. The object's `Chown_OK` flag must be set as well. Wizards may chown any item, regardless of its `chown_lock.`

@CONLOCK

`@conlock <object> = <lock expression>`

Locks `<object>` such that only those for whom `<lock expression>` is true may place things in or remove things from `<object>`.

@CONTENTS

`@contents [<object>] [= <flags/types> = [<output type>]]`

Searches the given object for items & exits that match the given flag string.

For an explanation of the flags/types modifiers and the output types, see the help entry for @find.

Example: `@contents here = DE = owner` will list all Dark Exits attached to your current location, giving the owner of each one. See also @find, @owned, @entrances

@CREATE

`@create <object> [=<cost> [=<regname>]]`

Creates a new object and places it in your inventory. This costs at least ten pennies. If `<cost>` is specified, you are charged that many pennies, and in return, the object is endowed with a value according to the formula: ((cost / 5) - 1). Usually the maximum value of an object is 100 pennies, which would cost 505 pennies to create. If a `<regname>` is specified, then the `_reg/<regname>` property on the player is set to the dbref of the new object. This lets players refer to the object as `$<regname>` (eg: `$mybutton`) in `@locks, @sets`, etc. Only a builder may use this command.

@CREDITS

`@credits`

Displays a screen listing the names of people and places who have contributed to TinyMUCK's development.

@DESCRIBE

```
 |  @DESC
@describe <object> [=<text>]
```

Sets the description field of `<object>` to `<text>`. If `<text>` is not specified, the description field is cleared. This is the same as `@set <object> = _/de:<text>`. A description is what is seen when a player looks at something.

@DIG

`@dig <room> [=<parent> [=<regname>]]`

Creates a new room, sets its parent, and gives it a personal registered name. If no parent is given, it defaults to the first `Abode` room down the environment tree from the current room. If it fails to find one, it sets the parent to the global environment, which is typically room #0. If no regname is given, then it doesn't register the object. If one is given, then the object's dbref is recorded in the player's `_reg/<regname>` property, so that they can refer to the object later as `$<regname>`. Digging a room costs 10 pennies, and you must be able to link to the parent room if specified. Only a builder may use this command.

DROP

```
drop <object>
```

Drops the `<object>` if you are holding it. It moves the object to the room you are in, unless its Sticky flag is set (in which case the object will go to its home), or the room has a drop-to (in which case the object will go to the room's drop-to). Programs are much like objects but are not affected by room droptos or Sticky flags. A 'drop' message can be set, which will be shown to the player dropping the object, and an 'odrop', which will be shown to the other players in the room. See `@odrop`.

@DROP

```
@drop <object> [=<text>]
```

Sets the drop field of `<object>` to `<text>`. If `<text>` is not specified, the drop field is cleared. The drop message on an object is displayed when you drop it. On an exit, it is displayed upon entering the destination room. On a player it is displayed to whoever kills him or her. On a room, it is displayed when an object is dropped there. This is the same as `@set <object> = _/dr:<text>`.

@DUMP

```
@dump [filename]
```

Saves the database from memory to disk. Automatically occurs every three hours, and when `@shutdown` is used. On a large `MUCK`, this can take a significant amount of time (ten or more minutes). If a filename is given, it will save the database to that file, and save any subsequent dumps to it as well. (Wizard only)

@EDIT

```
@edit <program>
```

Searches for a program and if a match is found, puts the player into edit mode. Programs must be created with `@program`.

@ENTRANCES

```
 |  @ENT
@entrances [<object>] [= <flags/types> = [<output type>]]
```

Searches through the database for items that you control linked to <object>.

For an explanation of the flags/types modifiers and the output types, see the help entry for `@find`.

Example: `@entrances here = ED = location` will list all Dark Exits that are linked to your current location, giving the location of each one. See also `@find`, `@owned`, `@contents`.

EXAMINE

```
 |  EX
examine <object> [=propdir]
```

If you control `<object>`, `examine` will give you a complete breakdown of all fields, flags, etc., that are associated with the object. If the optional propdir field is supplied, then it instead lists out all the properties directly under that propdir. To list the base propdir of an object, use `ex <object>=/`. Program-executing fields are displayed as their true text, rather than executing the program in question. If you do not control `<object>`, however, it prints the owner of the object in question, and, again, displays the true text of the description.

@FAIL

```
@fail <object> [=<message>]
```

`<Object>` can be a thing, player, exit, or room, specified as `<name>` or #`<number>` or 'me' or 'here'. Sets the fail message for `<object>`. The message is displayed when a player fails to use `<object>`. Without a message argument, it clears the message. This is the same as: `@set <object>=_/fl:[text]`. See also `@ofail` and `@desc`.

@FIND

```
@find [<name>] [= <flags/types> = [<output type>]]
```

Searches through the database for items that you control matching `<name>`. Players control only objects they own; wizards control all objects, so `@find` searches the entire database when they use it.

Flags or types can be specified, to specify that you only want to list objects that have that flag set, or that are of that type. You can also specify to list objects that are *not* of that specific type, or that do *not* have that flag. (A ! not operator before the modifier indicates that it is to be inverted.)

The flags that you can specify are: `Abode, Builder/Block, Chown_OK, Dark/Debug, Haven, Interactive, Jump_OK, Kill_OK, Link_OK, Mucker, Quell, Sticky/Silent, Vehicle, Wizard, Xforcible`, and `Zombie` (use the initial capitalized letter only).

You can also specify Mucker Levels by the level number: 1, 2, 3, or 4.

The types that you can specify are: `Exit, muF program, Garbage, Player, Room,` and `Thing` (use the capitalized letter only).

There are a few other modifiers you can specify:

`Unlinked`
: Specifies that you want to list only unlinked objects.

`@`
: Specifies to list objects longer than about 90 days old.

`~size`
: Will match all objs whose current memory usage is greater than or equal to size bytes. This must be the last modifier in the list of modifiers.

`^size`
: Will match all objs whose total memory usage, when fully loaded, is greater than size bytes. To do this, it loads the entire object into memory from disk. This modifier is only available to wizards. For regular players, this acts like `~size.` This must be the last modifier in the list of modifiers.

Again, use only the initial capital letter for `Unlinked.`

The output types that can be given are owners, links, size, count, & location. For output types, use the whole name rather than just the initial letter. You can use only one at a time.

`owners`
: Lists who owns each object.

`links`
: `Shows what each object is linked to, or *UNLINKED*, or, for exits linked to multiple things, *METALINK*`

`size`
: Displays how much memory is currently being used by an object. If this option is used with the ^ modifier, (see above) then this will display the true full size of the object, and not just how much is currently being used.

`count`
: Causes nothing to be shown but how many objects the `@find`, etc would match. That is, it doesn't display any of the matched objects.

`location`
: shows where the object is located.

The rules for matching on names are as follows:

- Individual words can be matched as `{word1|word2|...}`
- Individual characters can be matched as `[abc...]`
- A ? question mark matches any single character.
- An * asterix matches any number of characters, including none.
- Any of these special charcters can be matched by putting a \ backslash before it.

Examples:

```
@find north = EU = location
```
Finds all of your unlinked exits named 'north' and prints them along with their locations.

```
@find {big|little} = R!L
```
Finds all your rooms whose names contain 'big' or 'little' and are not Link_OK.

```
@find w[ei]ll
```
Find everything you control whose name contains 'will' or 'well'.

```
@find = E =links
```
Will list all exits that you control, and display where they are linked to.

```
@find button == locations
```
Will list all objects you control with 'button' in the name, and it will display where thay are located.

```
@find = ~2000 = size
```

Will list all your objects whose current memory usage is 2000 bytes or more, and it will display their size.

`@find = ^2000 = size`

Will, for a wizard, find all objects in the database that are 2000 or more bytes in total size, when fully loaded, and it will show their sizes. Note that this will load all of each object into memory to make the size determination. On some systems this can take a while, and on all systems this is an abuse to the diskbasing cache. (Wizard only)

See also `@owned`, `@entrances`,and `@contents`

@FORCE

`@force <player>=<command>`

Causes the game to process `<command>` as if typed by `<player>`. With the compile option `god_priv,` God cannot be forced by his or her sub-wizards.

@FORCE_LOCK

` |  @FLOCK`

`@force_lock` <object> = <lock expression>

Locks `<object>` such that it may only be `@forced` by players or other object types for whom `<lock expression>` is true, provided that the `XForcible` flag is set. Wizards may force any object, except God.

GET

`get <object>`

Attempts to pick up `<object>.` The lock on `<object>` is checked for a success (true), and the normal path of success/fail is then taken. On success the object is placed in your inventory.

Another variation on this is `get <container>=<object>`, which attempts to get `<object>` from the given container. The `_/clk` lock property on `<container>` is tested, and if it is true, when it checks to see if the standard `_/lok` lock property on `<object>` tests true. If both locks pass, then `<object>` is moved into the player's inventory. If there is no `_/clk` property on `<container>` it defaults to failing. The `_/lok` property, on `<object>,` on the other hand, defaults to passing. `@succ/@fail` messages are not displayed, when fetching something from a container.

GOTO

` |  GO`

`go[to] <direction>; go[to] home`

Goes in the specified direction. `go home` returns you to your starting location. The word `goto` or `go` may be (and usually is) omitted. `Move` is the same as `go`.

GIVE

```
give <player> = <# pennies>
```

Gives `<# pennies> to <player>`. For mortals, `<# pennies>` must be a positive number less than or equal to the number of pennies they have. `<# pennies>` is subtracted from the giving player's inventory. Wizards may give any amount from positive `max_integer` to negative `max_integer.` Giving does not affect the wizard's own inventory of pennies.

GRIPE

```
gripe <message>
```

Sends `<message>` to the system maintainer. Gripes are logged for later reference; also, if the system maintainer is connected he will receive the gripe real-time when the gripe is made.

HELP

```
help <topic>
```

Shows an online help document for `<topic>. Help` with no argument shows a brief list of basic commands. `Help help` shows a list of index topics.

HOME

```
home
```

Sends you home, no matter where you are. You retain your pennies, but any objects you are carrying leave your inventory and return to their own homes.

@IDESCRIBE

```
 |  @IDESC
@idescribe <object> [=<text>]
```

Sets the `idescription` field of `<object>` to `<text>.` If `<text>` is not specified, the description field is cleared. This is the same as `@set <object>=_/ide:<text>`. An idescription is what is seen on the inside of a vehicle, when a player inside it looks around.

INFORMATION

```
 |  INFO
information <topic>
```

Shows an online document about `<topic>.` If no topic is supplied, a list of available topics will be shown.

INVENTORY

```
 |  INV  |  I
inventory
```

Lists what you are carrying. This can be abbreviated to `inv` or `i`.

KILL

```
kill <player> [=<cost>]
```

A successful kill sends the player home, sends all objects in the player's inventory to their respective homes. The probability of killing the player is `<cost>` percent. Spending 100 pennies always works except against Wizards who cannot be killed. Players cannot be killed in rooms which have the `Haven` flag set. On systems where the `Kill_OK` flag is used, you cannot kill someone unless both you and they are set `Kill_OK`.

@KILL

```
@kill <processid|playername|programdbref|"all">
```

If passed a processid (a number without an # octothorpe preceeding it), it will kill the given process, if the player controls it. If passed a player name, it will kill all the processes controlled by that player. If passed a program dbref, it will kill all processes that that program is running in. If the argument passed is `all,` and the player is a wizard, it will kill all processes on the timequeue.

LEAVE

```
leave
```

Leave the vehicle you are currently inside.

@LINK

```
@link <object1>=<object2> [; <object3>; ... <objectn> ].
```

Links `<object1>` to `<object2>,` provided you control `<object1>,` and `<object2>` is either controlled by you or linkable. Actions may be linked to more than one thing, specified in a list separated by semi-colons.

@LIST

```
@list <program> [=[line1] [-] [line2]].
```

Lists lines in a program, provided you control it or it is `Link_OK.` Zero, one, or two line numbers may be specified, denoting the range of lines to list. If no lines are given, the entire program is listed.

@LOCK

```
@lock <object> = <key>
```

Locks `<object>` to a specific key(s). `<Object>` can be specified as `<name>` or #`<number>,` or as 'me' or 'here'. Boolean expressions are allowed, using `&` (and), `|` (or), `!` (not), and parentheses ('(' and ')') for grouping. To lock to a player, prefix their name with '*' (ex. `*Igor`). A key may be a player, an object, or 'property:value'.

LOOK

```
look <object>
```

Looks around at the current room, or at `<object>` if specified. For players, displays their description and inventory, for things, their description, and for rooms, their name, description, succ/fail message, and contents. Also triggers osucc/ofail messages on rooms. Programs are triggered accordingly on desc/succ/fail fields.

MAN

```
man [<subject>]
```

Displays the programmer's manual or a quick reference for `MUF.`

MOVE

See `GOTO`.

MPI

```
mpi [<subject>]
```

Displays the programmer's manual or a quick reference for `MPI.`

@NAME

```
@name <object>=<name> [<password>]
```

Sets the name field of `<object>` to `<name>. <Name>` cannot be empty; a null name is illegal. `<Password>` must be supplied to rename a player. Wizards can rename any player but still must include the password.

@NEWPASSWORD

```
@newpassword <player> [=<password>]
```

Changes `<player>'s` password, informing `<player>` that you changed it. Must be typed in full. If `god_priv` was defined, nobody can change God's password. (Wizard only)

NEWS

```
news [<topic>]
```

Displays the current news file for the game. Must be typed in full. If a topic is given, then it displays the information on that specific topic.

@ODROP

`@odrop <object> [=<text>]`

Sets the odrop field of `<object>` to `<text>.` If `<text>` is not specified, the odrop field is cleared. Odrop on an object is displayed prefixed by the player's name when s/he drops that object. On an exit, it is displayed upon a player's arrival to the destination room (or the location of the destination player). On a player, it is displayed after the 'name killed victim!' message. On a room, it is displayed when an object is dropped there, prefixed by the object's name. This is the same as: `@set <object>=_/odr:<text>`. See also `@drop`.

@OECHO

`@oecho <vehicle object> = <echo prepend string>`

Sets a string to prepend to messages relayed to the interior of a vehicle in the vehicle's `_/oecho` property. By default, the name of the vehicle object, followed by a `>` greater-than character is used to prepend such messages.

@OFAIL

`@ofail <object> [=<message>]`

The `@ofail` message, prefixed by the player's name, is shown to others when the player fails to use `<object>.` Without a message argument, it clears the message. `<object>` can be specified as `<name>` or #`<number>,` or as 'me' or 'here'. This is the same as: `@set <object>=_/ofl:<text>`. See also `@fail`.

@OPEN

`@open <exit> [=<object> [; <object2>; ... <objectn> ] [=<regname>]]`

Opens an exit in the current room, optionally attempting to link it simultaneously. If a `<regname>` is specified, then the `_reg/<regname>` property on the player is set to the dbref of the new object. This lets players refer to the object as `$<regname>` (eg: `$mybutton`) in `@locks, @sets`, etc. Opening an exit costs a penny, and an extra penny to link it, and you must control the room where it is being opened.

OUTPUTPREFIX

`OUTPUTPREFIX [string]`

Must be in all capitals, and typed in full. Prints the given line before the output of every command, setting them apart from other messages.

OUTPUTSUFFIX

```
OUTPUTSUFFIX [string]
```

Must be in all capitals, and typed in full. Prints the given line after the output of every command, setting them apart from other messages.

@OSUCCESS

```
  |  @OSUCC
@osuccess <object> [=<message>]
```

The `@osuccess` message, prefixed by the player's name, is shown to others when the player successfully uses `<object>.` Without a message argument, it clears the `@osuccess` message. It can be abbreviated `@osucc.` `<Object>` can be specified as `<name>` or #`<number>,` or as 'me' or 'here'. This is the same as `@set <object>=_/osc:<text>`. See also `@success`.

@OWNED

```
@owned <name> [= <flags/types> = [<output type>]]
```

Searches through the database for items that `<name>` controls.

For an explanation of the flags/types modifiers and the output types, see the entry for `@find`.

Example: `@owned Revar=F!L3=location` will list all Mucker Level 3 (`3`) programs (`F`) owned by Revar, that are *not* set `Link_OK` (`!L`), and it will show the location of each one.

Note that only wizards can do an `@owned` on other people. See also `@entrances`, `@find`, and

PAGE

```
page <player> [= <message>]
```

This tells a player that you are looking for them. They will get a message in the form of 'You sense `<pager>` is looking for you in `<location>.'` A `<message>` is optional, and is delivered in the form of `<pager> pages: <message>`. Your location is not revealed in message pages. If a player is set `Haven,` you cannot page them, and they will not be notified that you tried. You will instead be told, 'That player does not wish to be disturbed.' (Note: Most systems use a user-created program with a global `page` action, which takes the place of the built-in `page` command, and has more features.)

@PASSWORD

```
@password <old password> = <new password>
```

This changes your password.

@PCREATE

```
@pcreate <player> = <password>
```

This command creates a new player. It may only be used if `registration` is enabled. (Wizard only)

@PECHO

`@pecho <puppet object> = <echo prepend string>`

Sets a string to prepend to messages relayed from a puppet in the puppet's `_/pecho` property. By default, the name of the puppet object, followed by a > greater than character is used to prepend such messages.

POSE

`pose <message> or :<message>`

Causes `<message>` to be displayed to the room, prepended by your name.

@PROGRAM

`@program <program>`

Create a new program, or enter edit mode on an existing one. See also `@edit`.

@PROPSET

`@propset <object> = <data type> : <property> : <value>`

Sets `<object's>` `<property>` to `<value>,` with `<value>` stored as data type `<data type>`. You must control `<object>`.

@PS

`@ps`

Lists the status of the currently running `MUF` program processes. This lists all processes for a Wizard. Non-Wizards only see the muf processes that they can `@kill.` See `@kill`.

PUT

server: `put <object>`
user-created: `put <object> in <container>`

The server version of `put` is an alias of `drop.` The standard start-up database supplies a user created command that moves `<object>` into `<container>,` provided that you are holding both `<object>` and `<container>` and that `<container>` is not `@conlocked` against you.

QUIT

`QUIT`

Must be in all capitals, and typed in full. Logs out of your character and leaves the game. Your character remains at the location you are in when you log out, although it might be moved elsewhere while you are 'asleep.'

READ

```
read [<object>]
```

An alias for `look.` The standard start-up dbase supplies a bulletin board program for which `read` is a command to read bulletins.

@RECYCLE

```
 |  @REC
@recycle <object>
```

Destroy an object and remove all references to it within the database. The object is then added to a free list, and newly created objects are assigned from the pool of recycled objects first. You *must* own the object being recycled: even wizards must use the `@chown` command to recycle someone else's belongings.

@RESTRICT

`@restrict` <on|off>

Turning restriction on allows only wizards to log onto the `MUCK.` Turning it off returns to unrestricted access.

ROB

```
rob <player>
```

Attempts to steal one penny from `<player>.` The only thing you can steal are pennies.

SAY

```
say <message> or "<message>
```

Says `<message>` out loud. See also `pose`, `page`, and `whisper`.

SCORE

```
score
```

Displays how many pennies you are carrying.

@SET

```
@set <object> = [!] <flag> -or-
@set <object> = <property> : [ <string> ] -or-
```

`@set <object> = :`

`@set` does one of three things on TinyMUCK: it can modify flags, add properties to an object, or remove properties from an object.

Using the first format, you may set flags, which are: `Wizard, Link_OK, Dark [Debug], Filter, Sticky [SetUID], Jump_OK, Builder [Bound], Quell, Chown_OK, Havan [HardUID], Abode [Autostart], Vehicle, Zombie`, or `Mucker.` You can also set the Mucker (or Priority) Level of an object by using `0, 1, 2,` or `3` as the flag name. An optional flag which may or may not be on a given site is `Kill_OK`.

The second format sets `<property>` on `<object>` to `<string>,` or if `<string>` is not given, removes `<property>.`

The third format removes all properties from an object.

@SHUTDOWN

`@shutdown`

Shuts down the game. Must be typed in full. (Wizard only)

@STATS

`@stats [<player>]`

For mortal players, returns the highest number in the database, which includes garbage that has been generated with `@recycle.` For Wizards, gives this number as well as a breakdown of each type of object: rooms, exits, things, programs, players, and garbage. Wizards may also specify <player> which returns a similar display limited to the possessions of `<player>`.

@SUCCESS

`|  @SUCC`
`@success <object> [=<message>]`

Sets the success message for `<object>.` The message is displayed when a player successfully uses `<object>.` Without a message argument, it clears the message. It can be abbreviated `@succ.` `<object>` can be specified as `<name>` or `#<number>,` or as 'me' or 'here'. This is the same as `@set <object>=_/dr:<text>`. See also `@osuccess`.

@SWEEP

`@sweep [<object>]`

Returns a list of objects that are listening to `<object>.` `<Object>` defaults to 'here'.

TAKE

See `GET`.

@TELEPORT

` | @TEL`

`@teleport <arg1> [=<destination>]`

Moves `<arg1>` to `<destination>,` if `<destination>` is not given, moves you to `<arg1>.` Wizards may teleport anything to anywhere, provided it makes sense, and mortals are allowed to do two things: teleport rooms to change their parent fields, and the may teleport things to a room they can link to, provided they control either the thing or its location.

THROW

server: `put <object>`
user-created: `throw <object> to <player>`

The server version of `throw` is an alias of `drop.` Most `MUCKs` have a user-created program that moves `<object>` to `<player>,` announcing the move, provided that you are holding `<object>` and that you, `<object>,` and `<player>` are each configured in such a way as to allow the throw.

@TOAD

`@toad <player1> [= <player2>]`

Turns `<player1>` into a slimy toad, destroying their character. All possessions of `<player1>` are `@chowned` to `<player2>.` If no owning player is specified, objects will be `@chowned` to the wizard who is executing `@toad.` Must be typed in full. (Wizard only)

@TRACE

`@trace <object> [=<depth>]`

Starts with `<object>` and traces all location fields, until the global-environment room is reached or the optional `<depth>` is specified. This is generally useful for finding which rooms are parents in your heirarchy. If you cannot link to a particular location its name is replaced by `**MISSING**`.

@UNCOMPILE

`@uncompile <program>`

Uncompiles all programs in the database. (Wizard only)

@UNLINK

`@unlink <exit>; @unlink here`

Removes the link on the exit in the specified direction, or removes the drop-to on the room. Unlinked exits may be picked up and dropped elsewhere. Be careful, anyone can relink an unlinked exit, becoming its new owner (but you will be reimbursed your 1 penny). See `@link`.

@UNLOCK

`@unlock <object>`

Removes the lock on `<object>.` See `@lock`.

@USAGE

`@usage`

Displays system resource usage stats for the muck server process. (Wizard only)

@WALL

`@wall <message>`

Only wizards may use this command. Shouts something to every player connected. Must be typed in full.

WHISPER

`whisper <player>=<message>`

Whispers the message to the named person, if they are in the same room as you. No one else can see the message. Wizards can `whisper *<player>=<message>` to whisper to players in other rooms. (Note: Most systems use a user-created program in place of the built in whisper command. These programs generally provide many more useful features.)

WHO

`WHO [<player>]`

Must be in all capitals, and typed in full. Lists the name of every player currently logged in, and how long they have been inactive. If given a player name, it displays only the matching names and idle times. By typing `WHO*` wizards also get a display of the host the player is connected from.

2.7 User-Created Command Reference

This section provides a quick reference for user-created commands supplied in the standard start-up database (std-db.db). On many `MUCKS,` several or many of these programs will have been modified or replaced with versions that are either more recent or more specifically attuned to the needs of the `MUCK.` So syntax may very slightly, but you can reasonably expect to find a command that offers similar functionality. An established `MUCK` will have many other user-created commands besides these.

Most user-created commands follow the convention of a `#help` argument: typing the command name followed by `#help` will display a help screen.

3WHO

```
3who
```

Displays the names, online times, and idle times of all players online, formatted to three columns. A less spammy alternative to `WHO`.

@ARCHIVE

```
 |  @ARC
@archive <object>
```

The archive command essentially 'decompiles' an object, outputting all commands and settings that would be necessary to recreate the object. The output is simply printed to your screen: it is not saved in some archive on the `MUCK.` In order to use an `@archive` dump, you will need to capture the output, by cutting and pasting, logging with a client, or some other means, and save it to a file on your own computer.

The archived object can then be recreated by quoting or pasting the file into a `MUCK` window. The most common uses of archive dumps are porting objects between `MUCKs` or keeping a back-up copy offline to safeguard against loss or corruption of the `MUCK's` dbase.

The `@archive` command is recursive: `<object>` and anything contained by `<object>` will be archived. Thus, for example, an entire area can be dumped to an archive file by archiving the parent room: all rooms, exits, things, and programs in the area that are owned by the archiving player will become part of the dump.

For non-wizzes, @ wiz props will not be part of the dump. Restricted props (props that begin with a ~ tilde) will be included in the dump, but will not be recreated when the archive is quoted, unless the player is a wizard at that time. Also, exits leading to and from archived rooms or areas cannot be linked when the archive is quoted (though exits within an archived area can). It is normal to see various `'Huh?'` and `'Permission denied'` messages when recreating an object from an archive dump.

`MPI` and `MUF` that contains hard-coded dbrefs create another portability concern. A dbref, naturally, refers to a specific object on a specific `MUCK.` The corresponding object on a different `MUCK` will have a different dbref. To some extent, this problem can be reduced by using pointers and registered names when building, describing, coding, etc. For example, if you (player `#123`) used a personal version of `{look-notify}` ...

```
    {null:{tell:[ {name,me} looked at you. ],#123}}
```

... the code would result in a `'Permission denied'` error if you recreated yourself on another `MUCK` via an archive dump. This could be avoided with either of the following versions:

```
    {null:{tell:[ {name,me} looked at you. ],this}}
    {null:{tell:[ {name,me} looked at you. ],*Jessy}}
```

Recreating objects from archive dumps usually sets temporary registered name values on your character, in propdir `_reg/tmp.` It is not necessary to do anything with this directory, but to conserve dbase space, you might want to remove it after doing a large dump: `@set me = _reg/tmp:`

@BANSITE

```
@bansite sitename (Prevents logins from the given site)
@bansite !sitename (Re-allows logins from the given site)
@bansite #list (Lists all the sites that are banned)
```

This command prevents anyone from logging onto the `MUCK` from a banned site. Along with `@toad,` it is used to prevent problematic players from connecting to the `MUCK.` For example, a player might be toaded for violations of the `MUCK's` acceptable use policy, but continue to log on via guest and alternate characters. `@Bansite` prevents this by locking out connections from a given site regardless of the character.

Sites may be specified by host name or numeric address. Wildcards may be used in the site name and address. Wildcards are frequently necessary, since many `ISP's` use dynamic host names for customer's connected via `SLIP` or `PPP` connections. For example, an examination of a problematic player's `@/` directory might show that the last host used was:

```
    user12.vnn.luser.com
```

Doing `@bansite user12.vnn.luser.com would` be ineffective: in all likelihood, the first one or two sections of the hostname would different the next time the player logged on. In lieu of...

```
    @bansite user12.vnn.luser.com
```

... one should do:

```
    @bansite *vnn.luser.com       - or -
    @bansite *luser.com
```

`@Bansite` has two disadvantages: it locks out other players who connect from the same site, and it offers no protection against players who you want to ban but can log on from alternate sites. (Wizard only)

BOOTALL

```
bootall
```

Boots all but your lowest numbered connection. This program does not work well. `Cmd-@bootme,` which is widely available, is a better choice.

CHANGE

```
change <object>=<propname>:/<old>/<new>     - or -
change <object>=<mesgtype>;/<old>/<new>
```

Edits a property or message, replacing string `<old>` with string `<new>.` The syntax varies slightly — that is, it uses either a : colon or ; semi-colon delimiter — for messages and properties. 'Messages' are 'name', 'desc', 'succ', 'osucc', 'fail', 'ofail', 'drop', and 'odrop'. Examples:

```
    change here=_arrive/check:/444/445
    change me=desc;/top hat/beret
```

@CHECK

```
@check
```

Checks all objects in your location for completeness, with the definition of 'complete' being adjusted as appropriate for the object type. For example, exits are checked for links, descs, succs, osuccs, and odrops. Locked exits are checked for fails and ofails. The command prints a list of objects that need further work.

@DOING

```
@doing <msg>
```

Sets your 'sticky do' string... that is, the message shown with your name and times with commands `WHO` and `whodo.` The original purpose of doing strings was to provide a way for people to let others know what they are doing at the moment. Rather than frequently updating their doing string, most players set it to some (presumably) witty or interesting 'bumpersticker' type remark, and leave it.

EDIT

```
edit <object> = <prop>     or,
edit <object> = @<mesgtype>
```

This unusual and useful command is works only with the TinyFugue `UNIX` client. Entering the command and parameters pulls the value of `<prop>` or `<mesgtype>` into your client window, with the cursor positioned for editing. You can then use arrow keys, backspace, etc., to edit the property value. Pressing 'enter' stores the new property value. `<Mesgtype>` can be 'name', 'desc', 'succ', 'osucc', 'fail', 'ofail', 'drop', or 'odrop'.

Before using `edit,` you must define the following TinyFugue macro:

```
    /def -fg -p100 -t"##edit> *" = /grab %-1
```

Like most TinyFugue settings, this setting is 'volatile'... It is not retained between TinyFugue sessions. To avoid having to redefine it each time, put the definition line in TinyFugue's resource file, .tfrc, so that it will be defined each time you start the client.

FETCH

```
 |  RETRIEVE  |  GRAB
fetch <object 1> from <object 2>
```

Removes `<object` 1> from `<object` 2>. You must be carrying `<object` 2> and it must not be locked or `container_locked` against you. The object names do not have to be typed completely: partial strings sufficient to distinguish the object from others you are carrying will work. See also `put`, below.

LOOKAT

```
lookat <player's> <object>
```

Does a 'possessive look', showing you the description of an object being carried by another player. Example:

```
lookat tarka's linux club pin.
```

LSEDIT

```
 |  LSE
lsedit <object> = <list name>
```

Puts you in an interactive list editor. See Sections 2.1 and 4.1 for more information.

LSPROP

```
 |  LSP
lsprop <object> [= <path>]
```

Lists all properties on object and their values. If a path is specified, it will list only properties on that path. Examples:

```
    lsp me ........... Lists all properties on your character
    lsp me = _descs .. Lists all properties in your _descs/ directory
```

PROPCP

```
 |  CP
cp <object 1>=<prop 1>,<object 2>=<prop 2>
```

Copies `<prop 1>` and its value from `<object 1>` to `<prop 2>` on `<object 2>`. This example copies the @succ of exit 'out' to another exit:

```
    cp out=_/sc,#1344=_/sc
```

`Propcp` handles partial input well. If `<object 1>` is omitted, it assumes `<prop 1>` is on the user. If `<object 2>` is omitted, it assumes the destination object is the same as `<object 1>`... i.e., that you are copying from one prop to another on the same object. If `<prop 2>` is omitted, is assumes that it is the same as `<prop 1>`... i.e., that you are copying a property on one object to the same property on a different object. If more information is omitted, the program will ask for it. Rather than providing all — or even any — of the information on the command line, one can simply type cp, and respond to prompts. You must control both objects. See also `propmv`, below.

PROPMV

```
 |  MV
mv <object 1>=<prop 1>,<object 2>=<prop 2>
```

Moves `<prop 1>` and its value from `<object 1>` to `<prop 2>` on `<object 2>`, erasing `<prop 1>`. This example moves the fail message on 'out' to the ofail:

```
    mv out=_/fl,out=_/ofl
```

Propmv handles partial input well. If `<object 1>` is omitted, it assumes `<prop 1>` is on the user. If `<object 2>` is omitted, it assumes the destination object is the same as `<object 1>`... i.e., that you are moving a value from one prop to another on the same object. If `<prop 2>` is omitted, is assumes that it is the same as `<prop 1>`... i.e., that you are moving a property on one object to the same property on a different object. If more information is omitted, the program will ask for it. Rather than providing all — or even any — of the information on the command line, one can simply to `mv,` and respond to prompts. You must control both objects. See also `propcp`, above.

@PURGE

```
@purge <player> = yes
```

Recycles all of `<player's>` belongings. Most often, this will be used before `@toad.` It is a good idea to do `@owned <player>` first, and make suitable arrangements. Exits leading to rooms owned by the player will become unlinked, and should either be relinked or recycled. Public building — or rooms that are jointly used by the player and others — should be chowned to someone else. Mortals my purge only themselves; wizards may purge any player.

PUT

```
 |  REPLACE  |  STUFF
put <object 1> in <object 2>
```

Moves `<object 1>` from your inventory into the contents of `<object 2>`. You must be holding both objects, and `<object 2>` must not be locked or `container_locked` against you. Partial names for both objects will work. See also `fetch`, above.

READ

```
 |  WRITE  |  EDIT  |  ERASE  |  PROTECT
```

These commands operate bulletin boards that run from the standard bulletin board program, gen-mesgboard. `Read` is also an in-server alias for `look.` So, in a room that contains a bulletin board, `read` is a bulletin board command; elsewhere, it does a `look`.

`read`	Show the headers of all posted messages.
`read new`	Show headers of all messages less than 2 days old.
`read recent`	Show headers of all messages after last read messages.
`read <keyword>`	Show headers of all messages with matching `<keyword>`.
`read -<days>`	Show headers of all messages fewer than `<days>` old.
`read <mesgnum>`	Read the message referred to by the given message number.
`read -`	Read the next message, after the last one you read.
`read -recent`	Read all messages after last read message, in one continuous stream.
`write`	Post a message. Prompts for subject and keywords.
`write <subject>`	Post a message with given `<subject>`. Prompts for keywords.
`write <subject>=<keywords>`	Post a message with given `<subject>` and `<keywords>`.
`erase <message num>`	Lets message owner erase a previously written message.
`editmesg <message num>`	Lets message owner edit a previously written message.
`protect <message num>`	Lets a wizard protect a message from auto-expiration.

@REGISTER

```
 |  @REG
```

```
@reg <object> = <registered name>
@reg #me <object> = <registered name>
@reg #prop <target object>:<propdir> <object> = <registered name>
```

Sets a registered name for `<object>.` (See Section 2.1.5 for more information on registered names).

The default format, `@reg <object> = <registered name>`, creates a global registered name by setting property `_reg/<reg name>` on room #0 with the value of `<object's>` dbref. Because a property is being set on room #0, this will result in a 'Permission denied' error for a mortal player (unless she happens to own room #0). The following example, which would be typed by a wizard, gives the gen-nothing program the registered name of `nothing`; players would then be able to link actions to it by typing `@link <action> = $nothing`

```
    @reg gen-nothing = nothing
```

Registered names can be set on your character rather than on room #0, in which case they will be be meaningful only for you, by using the `#me` argument. The following example, which could be typed by a mortal, or by a wizard who wants to create a personal registered name, gives a puppet object the registered name 'pup'. It could then be specified by `$pup,` rather than its dbref, as in `@tel $pup = me`.

```
    @reg #me squiggy = pup
```

You can over-ride the target propdir, from `_reg/` to anything else, by using the `#prop` argument. If the target propdir does not begin with `_reg/` , the setting will not be usable as a registered name; this would simply be a way to set a property with data type dbref rather than string (which could also be accomplished with `@propset`). The following example sets property `_disconnect/cleanup` in the current room with the dbref of program gen-sweeproom.

```
    @register #prop here:_disconnect gen-sweeproom=cleanup
```

The same thing could be accomplished by typing:

```
    @propset here=dbref:_disconnect/cleanup:<#dbref of gen-sweeproom>
```

This example makes a setting in sub-propdir `_reg/lib,` giving the lib-strings library program the registered name `$lib/strings.`

```
    @register #prop #0:_reg/lib lib-strings = strings
```

The same thing could be accomplished by typing:

```
    @register lib-strings = lib/strings     - or -
    @propset #0 = dbref:_reg/lib/strings:<#dbref of lib-strings>
```

@RESTART

```
@restart
```

This command kills all active programs and restarts any set `Autostart.` It works by forcing the user to do `@kill all`, which means that God may not use it if the muck is running with the compile-time option `god_priv`. (Wizard only)

@SIZE

```
@size <object>
```

Calculates the memory used by `<object>`. Note that if `cmd-@sizer` is not set `Wizard`, the memory used by Wizard props will not be included in the total.

SWEEP

```
sweep (sweeps all sleepers in the room)
sweep <player> (sweeps <player>)
```

Sends home all players in a room who are sleeping, and who are not owners of the room. It also sends their contents that they do not own home when it homes them. If you control the room, you may also sweep online players who do not control the room.

The following prop settings, on either yourself or a room, as appropriate, are used to configure the command:

`_sweep/sweep`	message shown when you sweep the room.
`_sweep/swept`	message shown if you are swept.
`_sweep/fmt/`	propdir where sweep-player formats are stored.
`_sweep/fmt/std`	message shown if you sweep an individual player.
`_sweep/to`	where players swept in this room go to. ie: room #1234
`_sweep/authorized`	players authorized to sweep this room. Format: #12 #432 #190
`_sweep/immune`	players immune to general sweeps in this room.
`_sweep/public?`	if 'yes' means this room in sweepable by anyone.
`_sweep/immune?`	If 'yes', means this THING doesn't home from inventory when the player is swept.

UPTIME

```
uptime
```

Displays the time since the server was last restarted.

@VIEW

```
@view <program>
```

Displays documentation for `<program>`, if the program is appropriately configured.

To configure a program for use with `@view`, set it `Link_OK` and set the program's `_docs` property with a string that emulates a command that would show the documents. The most common way to do this is to set the `_docs` property to a command that `@lists` the header comment of the program. For example, if the first 54 lines of program `claimhome.muf(#47)` are a comment with documentation, one would configure the program for use with `@view` by typing:

```
    @set #47 = L
    @set #47 = _docs:@list #47 = 1 - 54
```

Other command formats are possible. Another example:

```
    @set #47 = _docs:info claimhome
```

@WHEN

`@when <object>`

Displays timestamp information for `<object>`: time created, time last modified, time last used, and usecount. `<object>` defaults to `'here'.` You must control `<object>`.

3.0 PROGRAMMING

`MUCK` supports two online programming languages, `MPI` and Muck `FORTH,` or `MUF.` `MPI` is an interpretted language (you don't need to compile it; the server reads the code directly) available to all players. `MUF` is a compiled, stack-based language, available to players who have a Mucker bit (an `M` flag). In general, `MPI` is better for one-off programming tasks and customizing messages (descriptions, exit `@osucc's,` etc.). `MUF` is better for public utilities and tasks that involve modifying the database.

The following sections describe both `MPI` and `MUF` in greater detail.

3.1 Overview: MPI

`MPI` is an interpretted language with a `LISP`-like syntax available to all players. Because it is universally available, `MPI` includes some security features that restrict its power. In general, `MPI` can only read props on the triggering object and on objects controlled by the controller of the object on which the `MPI` string is stored, and can only set props on the latter... on objects controlled by the owner of the `MPI` object. Other than setting props as mentioned, it is difficult (but not impossible) to significantly modify the database with `MPI,` but is ideally suited for message-handling. And because it is interpretted, it is well-suited for one-off programming tasks: no separate compiling and linking operations are needed, nor is a separate program object for holding the code.

`MPI's` syntax consists of nested functions enlcosed in curly braces that are evaluated left-to-right and 'from the inside out', much like mathematical expressions are evaluated outward from the innermost set of parentheses. For example...

```
    {if:{eq:{ref:me},#1},Hey it's God!,Hello world!}
```

The `MPI` parser will first evaluate the innermost function, `{ref:me}.` The `{ref}` function returns the dbref of its single argument — which in this case is `'me'` — so, `{ref:me}` returns the user's dbref. The returned expression 'replaces', as it were, the function. So, if the user's dbref were `#123,` the `MPI` string would in effect and at this moment be...

```
    {if:{eq:#123,#1},Hey it's God!,Hello world!.}
```

Then the next-innermost expression, effectively `{eq:#123,#1},` would be evaluated. The `{equals}` function returns true if the two arguments supplied are the same; otherwise it returns false. In this case, the two arguments are not the same, so `{equals}` will return false. At this point, the `MPI` value for false — the string "0" — will replace the function. (A "" null string is also false in `MPI.` Any value other than "" or "0" is considered true.) So, at this point the string would in effect be...

```
    {if:"0",Hey it's God!,Hello world!}
```

Finally, this, the outermost function will be evaluated. The `{if}` function takes three arguments. If the first argument is true, it returns the second argument. If the first argument is false, it returns the third argument. In this example, the first argument is false, so the the third argument will be returned: `MPI` will return the string `"Hello world!"` to player `#123.` If God had triggered the string, the `{if}` test would have been true, and the string `"Hey it's God!"` would have been returned instead.

The `{debug}` function displays the progress of the parser. Enclosing the whole of our example in a `{debug}` function will show the process described above.

```
> @succ testmpi = {debug:{if:{eq:{ref:me},#1},Hey it's God!,Hello
world!}}
  Message set.
> testmpi
  (@Succ) {IF:"{eq:{ref:me},#1}", "Hey it's God!", "Hello world!"}
    (@Succ) {EQ:"{ref:me}", "#1"}
      (@Succ) {REF:"me"}
        (@Succ) "me"
      (@Succ) {REF:"me"} = "#123"
      (@Succ) "#1"
    (@Succ) {EQ:"#123", "#1"} = "0"
    (@Succ) "Hello world!"
  (@Succ) {IF:"{eq:{ref:me},#1}", "Hey it's God!", "Hello world!"} =
"Hello world!"
  Hello world!
```

In the lines from the first half of the debugging output — where indentation is moving to the right — the parser is essentially finding the innermost, left-most function to be evaluated. The remaining portion, with lines ending in '`= <value>`' and indentation moving back to the left, depicts the series of returned expressions described above.

The keywords `'me'` and `'here'` can be used as normal. In addition, `MPI` supports the keyword 'this', which will be replaced by the dbref of the object on which the `MPI` is stored.

The variable functions `{&cmd},` `{&arg},` and `{&how}` may be used to retrive information about how `MPI` was triggered. `{&cmd}` stores the command name typed to trigger the `MPI.` `{&arg}` stores any arguments typed along with the command. `{&how}` stores the method by which `MPI` was triggered, and will have values such as `(@desc),` `(@succ),` `(@osucc),` `(@lock),` etc.

Functions can be nested up to 26 levels deep. Loops may iterate a maximum of 256 times, at which point the automatically exit. Lists may have a maximum of 256 lines, or 4096 characters, whichever is less.

An `MPI` string that appears in a `_/` prop (a `@succ` message, a `@desc,` and so forth) will be parsed automatically. For other triggering props, the parser must be called by an & ambersand at the beginning of the prop string.

```
> @set me=_oarrive/aaa:&{null:{otell:pads in quietly.}}
  Property set.
```

The arguments of functions are separated by commas. Commas appearing in argument strings will confuse the parser: functions will seem — to it — to have too many arguments. So, commas in argument strings must be 'escaped'... i.e., preceded with a `\` backslash escape character, which tells the parser to treat the next character literally, rather than as an `MPI` instruction. For example, if we wanted our first example to say `"Hey, it's God!"` or `"Hello, world!",` the commas would need to be escaped with a backslash character.

```
    {if:{eq:{ref:me},#1},Hey\, it's God!,Hello\, world!}
```

Complex or very long `MPI` instructions are often better stored in a list, where whitespace can be used to make the code more readable, rather than in a single prop where all will run together in an opaque mass of characters. A simple pointing string using the `{lexec}` ('execute list') function can then be placed in the triggering prop.

```
> lsedit harley = bikecode
  <    Welcome to the list editor. You can get help by entering '.h'
>
  < '.end' will exit and save the list. '.abort' will abort any changes.
>
  <    To save changes to the list, and continue editing, use '.save'
>

> {null:
>    {if:
>        {
>        (lots of complicated really cool
           motorcycle code goes here)
>         }
>     }
> }
> .end
  < Editor exited. >
  < list saved. >

> @succ ride harley;ride motorcycle;ride cycle = {lexec:bikecode}
  Message set.

> ride harley
  (The Harley does something amazing.)
```

The `{lexec}` function executes `MPI` in a list. The `{exec}` function executes `MPI` in a property, and thus provides another way to break code up into managable pieces. `MUSH` coders especially might find this method more intuitive.

3.1.1 MPI Macros

Frequently used portions of `MPI` code can be defined as macros, by setting properties in the `_msgmacs/` directory of an object located in the command search path (see Section 2.3.3. Macros defined in the `_msgmacs/` directory of room `#0` can be used by all players and objects.

An example: To define a global 'look-notify' — a macro that tells users when someone looks at them அ we could set the following property:

```
 @set #0=_msgmacs/look-notify:{null:{tell:*{name:me} looked at
you*,this}}
```

Some standard `MPI` macros are listed here. Wizards can copy these macros to the `MUCK's` global `MPI` macro directory. Alternatively, players can copy individual entries to `_msgmacs/` directory on themselves or objects they own.

3.1.2 MPI Examples

Here are a few examples of `MPI` code. Contributions are welcome.

This example describes an exit such that users can see who is on the other side by looking at the exit:

```
@desc cafe=Glancing in the Cafe door, you see {if:{commas: {contents:
#179}}, {commas:{contents:#179}, and ,item,{name:{&item}}},that it is
empty}.
```

(more to come)

3.1.3 MPI Reference

Connection Functions

awake	idle	online	ontime

DB Functions

contains	contents	controls	dbequals	exits
flags	force	fullname	holds	istype
links	location	locked	money	name
nearby	owner	ref	testlock	type

List-Handling Functions

commas	count	lcommon	lmember	lrand
sublist	lremove	lsort	lunion	luniqu
mklist				

Loop Functions

filter	fold	for	foreach	parse
while				

Logical Functions

and	equals	ge	gt	if
le	lt	not	notequals	or
xor				

Math Functions

absolute	addition	decrement	dice	distan
divide	increment	maximum	minimum	modulo
multiply	sign	subtraction		

Miscellaneous Functions

attr	debug	debugif	delay	fox
func	isnum	isdbref	kill	macros
muckname	muf	version		

Property Handling

concat	delprop	exec	exec!	index
index!	lexec	list	listprops	prop
prop!	propdir	rand	select	store
timesub				

String Functions

center	eval	instr	left	litera
midstr	nl	null	otell	pronou
right	smatch	strip	strlen	subst
tell	tolower	toupper		

Time Functions

convsecs	convtime	created	date	delay
ftime	lastused	ltimestr	modified	secs
stimestr	time	timestr	tzoffset	timesu
usecount				

Variable-Handling Functions

&arg	&cmd	&how	set	variab
with				

Definitions

Trigger
: The object that the `MPI` script is evaluated from.

List
: A string containing several individual substring items, seperated by carriage return characters.

Property-Based List

A set of consecutively numbered properties that each contain one string in a list of strings. Property based lists are often numbered like: `listname1, listname2, listname3, listname4,` etc. Another popular format is `listname#/1, listname#/2, listname#/3,` etc. `MPI` can read in either of those formats, and several more, for that matter.

False

A string value of `"0",` or a null string (`""`).

True

Anything that is not False.

ABS

```
 |  ABSOLUTE
{abs:expr}
```

Returns the absolute value of `expr.`

ADD

```
 |  ADDITION
{add:expr1,expr2}
{add:expr1,expr2,expr3...}
```

Returns the sum of the values of `expr1` and `expr2.` If more than two args are given, then this will add all the args together and return the result.

AND

```
 |  &&
{and:expr1,expr2...}
```

Returns true if `expr1` and `expr2` evaluate as true. Otherwise, this returns false. If `expr1` was false, this doesn't bother to evaluate `expr2,` as it does C-style shortcutting. If there are more than two arguments, then this will evaluate all of them until either one returns false, in which case this function returns false, or until it has evaluated all of the arguments. This function returns true only if all the arguments evaluate as true.

ARG

```
 |  &ARG
{&arg}
```

The `{&arg}` variable contains a string with the value of the command line arguments the user entered. This is so that you can have stuff like `MPI` in the fail of an exit, and when the user triggers the exit, and has some extra text on the line they entered, other than the exitname, the `MPI` can take that extra stuff as arguments for use. Note that you need to set an action `Haven` to get it to accept command line arguments.

ATTR

```
| ATTRIBUTE
```

{attr:attr1,attr2...,text}

This formats `text` with color and formatting attributes specified as `attr1, attr2,` etc. You may specify up to 8 attributes. Example, to format the string '`WARNING!`' in bold red letters, use `{attr:bold,red,WARNING!}`. Nesting attributes is not recommended. (fb6.0+)

AWAKE

```
{awake:player}
```

Returns how many times player is connected. This means that it will returns 0 if the player is not connected. If the given object is *not* a player, it will return 0. In all other cases, it will return a positive number, being how many times the given player is connected to the server.

CENTER

```
{center:string}
{center:string,fieldwidth}
{center:string,fieldwidth,padstring}
```

Takes a string and pads it to fit the given fieldwidth, with the string center justified. If no padstring is given, it assumes that it will pad the string with spaces. If no fieldwidth is given, it assumes that the field width is 78 characters. Example: `{center:Hello,10,1234567890}` would return the string "123Hello12"

COMMAS

```
{commas:list}
{commas:list,lastsep}
{commas:list,lastsep,var,expr}
```

Takes a list and returns a plain english comma delimited string with the items in it. For example, `{commas:{mklist:Tom,Dick,Harry}}` will return "Tom, Dick and Harry". If you specify the lastsep argument, you can replace the "and" with something else, such as "or" to get a result like "a, b or c". Note: You need to be careful to include spaces around the "or" or else you might get a result like "a, borc".

Example:

```
  {commas:{mklist:a,b,c}, or }
```

If the var and expr arguments are passed, then for every item in the list, it will set the value of the given variable name (which it will declare) to the item, then evaluate expr, and use the result in the string it outputs. Example: `{commas:{contents:here},\, or ,v,{name:{&v}}}` will return the name of every object in the room in a comma separated list, using ", or " as the final conjunction. ie: "Tom, Can of SPAM, Dick, or Harry."

CMD

```
 |  &CMD
{&cmd}
```

The `{&cmd}` variable contains the command name the user used, that caused the `MPI` to run. This is generally the exit name that the player triggered. For example, if the player typed 'east', and triggered the exit named 'east;e;out', which ran some `MPI` commands, the `{&cmd}` variable would have a value of "east".

CONCAT

```
{concat:listname}
{concat:listname,obj}
```

Returns a string, containing the concatenated lines of a property based list. It concatenates the list semi-intelligently, putting a single space between lines normally, and two spaces between lines when the previous one ended with a period, exclamation mark, or question mark. A property based list is a series of properties that are consecutively numbered. The server understands several different formats, and can also read in property lists in either the propnameX format, or the `propname#/X` format. It does *not* evaluate the contents of the list for embedded `MPI` commands. If no `obj` argument is supplied, then it looks for the list somewhere down the environment from the trigger object. Otherwise, it looks for the list down the environment from the given object.

CONTAINS

```
{contains:obj1}
{contains:obj1,obj2}
```

Returns true if `obj1` is within `obj2,` or within anything it contains, or within anything they contain. If `obj2` is not given, then it checks to see is `obj1` is held by the player, or by anything they hold, etc. Basically, this just sees if `obj1` is within the locational environment of `obj2.`

CONTENTS

```
{contents:obj}
{contents:obj,type}
```

Returns a list of the contents of the given object. If a second argument is passed to it, it restricts the listing to only those objects that are of the given type. Either the object must be nearby the trigger object, or else the owner of the trigger object must control the object. Otherwise this will error out with a `Permission Denied` error. The valid object type values are `Room,` `Thing,` `Exit,` `Player,` `Program,` and `Bad.` *Hint*: If you need to get a list of two types of objects from the room, just concatenate the lists from two calls to this function, with each object type you want. For example:

```
  {mklist:{contents:here,player},{contents:here,thing}}       - or -
  {contents:here,player}{nl}{contents:here,thing}
```

CONTROLS

```
{controls:obj}
{controls:obj,player}
```

If one argument is given, then this returns true ("1") if the trigger object's owner controls the given object. If two arguments are given, then it returns true if the given player controls the given object. Otherwise, this returns false. ("0") Wizards control everything

CONVSECS

```
{convsecs}
```

Converts systime seconds into a readable time string.

CONVTIME

```
{convtime:string}
```

Converts `"HH:MM:SS` Mo/Dy/Yr" format time string to systime seconds.

COUNT

```
{count:list}
{count:list,sep}
```

This counts the number of `\r` delimited items that are in the given list. This is effectively a list item count. If the sep argument if given, then it counts the number of sep delimited substrings in list. ie: The default sep `is` \r. (A carriage return.)

CREATED

```
{created:obj}
```

Returns the systime when `obj` was created.

DATE

```
{date}
{date:timezone}
```

Returns a date string in the form `mm/dd/yy.` If the timezone argument is given, then it offsets the date returned by that number of hours.

DBEQ

```
 |  DBEQUALS
{dbeq:obj1,obj2}
```

Returns true if `obj1` and `obj2` refer to the same `object.` This does name matching, so `{dbeq:*Wizard,#1}` will return true if `#1` is named Wizard.

DEBUG

```
{debug:expr}
```

This will show `MPI` debugging information for all the commands within the given expression. This is useful for seeing why something isn't working. This returns the result of the evaluation of `expr.`

DEBUGIF

```
{debugif:condition,statement}
```

If `condition` evals true, use debug mode is used when evaluating statement. Otherwise the statement is evaluated in regular mode.

DEC

```
 |  DECREMENT
{dec:var}
{dec:var,val}
```

Decrements the value of the given variable by one, returning the result. If a value argument is given, then it will subtract that from the variable, instead of the value `1.`

DELAY

```
{delay:secs,expr}
```

Evaluates the given expression, then puts the result of that on the timequeue, to execute after the given number of seconds. At that time, the string is evaluated again, and displayed to the user, or to the room, depending on whether it was run from a regular message such as `@succ,` or from an omessage such as `@osucc.` Since the expression is evaluated both before and after being delayed, you need to put `MPI` code that is to run after the delay within a `{lit:expr}` command. If a `{delay}` evaluation is a null string, then the notify or notify_except will not be done. `{Delay}` will return the process ID of the event it puts on the timequeue.

DELPROP

```
{delprop:propname}
{delprop:propname,object}
```

This function will remove a property and all of its subsidiary properties in the case that it is a propdir. This will delete the property on the trigger object, unless an object argument is specified. If one is, then it will delete the property on that given object. This function returns a null string. If you specify a propname that is protected, you will get an error of `Permission Denied`. You are only allowed to delete properties from objects that are owned by the owner of the trigger object.

DICE

```
{dice:X}
{dice:X,Y}
{dice:X,Y,Z}
```

Given one parameter, picks a random number between `1` and `X.` `(1dX)` Given two parameters, it randomly generates `Y` numbers between `1` and `X,` and adds them together. `(YdX)` A third parameter, if given, is just added to this sum as a modifier. `(YdX+Z)`

DIST

```
 |  DISTANCE
{dist:x,y} Returns distance from 0,0 that x,y is.
{dist:x,y,z} Returns distance from 0,0,0 that x,y,z is.
{dist:x,y,x2,y2} Returns distance between x,y and x2,y2.
{dist:x,y,z,x2,y2,z2} Returns distance between x,y,z and x2,y2,z2.
```

Given two arguments, this calculates the distance of a 2D point from the origin. Given three arguments, this calculates the distance of a 3D point from the origin. Given four arguments, this calculates the distance between a pair of 2D points. Given six arguments, this calculates the distance between a pair of 3D points.

DIV

```
 |  DIVIDE
{div:expr1,expr}
{div:expr1,expr2,expr3...}
```

Returns the value of `expr1` divided by `expr2.` Division by zero will return zero. If more than two arguments are given, then the first argument is divided by the second, and the result is divided by the third, etc, for all of the arguments. For example: `{div:180,6,3,5}` would be read like `180 / 6 / 3 / 5`, and a result of 2 would be returned.

EQ

```
 |  EQUALS  |  ==
{eq:expr1,expr2}
```

If `expr1` and `expr2` evaluate out to the same value, then this returns true. Otherwise, this returns false. If both expressions evaluate out to numbers, then this compares them numerically.

EVAL

```
{eval:string}
```

Sort of the exact opposite of `{lit:}`. This takes a string, and evaluates it for `MPI` commands embedded within it. This can be used on the output of `{list:}`, for example.

EXEC

```
{exec:propname}
{exec:propname,obj}
```

Returns the string value of the given property, after having evaluated any embedded `MPI` commands that it contained. If no object parameter is passed to it, it looks for the property somewhere down the environment from the trigger object. Otherwise, it looks down the environment from the object specified. If the property is not found, this returns an empty string. If the property that it tries to access is read restricted and the owner of the trigger object does not own the object that the property is found on, then the `MPI` script stops with a `Permission denied` error.

EXEC!

```
{exec!:propname}
{exec!:propname,obj}
```

Returns the string value of the given property, after having evaluated any embedded `MPI` commands that it contained. If no object parameter is passed to it, it looks for the property on the trigger. Otherwise, it looks for the property on the object specified. If the property is not found, this returns an empty string. If the property that it tries to access is read restricted and the owner of the trigger object does not own the object that the property is found on, then the `MPI` script stops with a `Permission denied` error.

EXITS

```
{exits:obj}
```

Returns a list of all the exits on the given object. The owner of the trigger object has to control `obj,` or else this errors out with `Permission denied`. Programs and exits never have exits attached to them.

FILTER

```
{filter:var,list,expr}
{filter:var,list,exp,sep}
{filter:var,lst,exp,sep,s2}
```

This evaluates `expr` for each and every item in the given list. On each evaluation, the temporary variable `var` will contain the value of the item under scrutiny. This function returns a list containing all of the items from the input list, for which `expr` evaluated true. `Var` will only be defined for the duration of `expr,` and will be undefined after the `{filter}` construct finishes. If sep is given, then it uses that string as the item seperator in the input list, instead of the usual carriage return character. If `s2` is defined, then it will use that string to seperate the items in the list it returns, instead of the normal carriage return. `Sep` and `s2` can be multiple characters long.

FLAGS

```
{flags:obj}
```

Returns a flaglist string from `obj.`, such as `PM2J.` The object must either be in the vicinity, or it must be controlled by the owner of the trigger object.

FOLD

```
{fold:var1,var2,list,expr}
```

```
{fold:var1,var2,lst,expr,sep}
```

This takes a list and stores the first two items in `var1` and `var2,` then evaluates `expr.` The value returned by `expr` is then put in `var1,` and the next list item is put in `var2. Expr` keeps being evaluated in this way until there are no more list items left. This returns the last value returned by `expr.` If a sep argument is given, the input list is assumed to have its individual items delimited by that string, otherwise it assumes a carriage return.

FOR

{for:varname,start,end,increment,command}

Acts as a counting loop, like BASIC's `for` loops. The `varname` is the name of the variable that it will create and use to store the count value. The start value will be the initial value of the variable, and the end value will be the value that the variable is working towards. The increment is how much the variable will be incremented by in each loop. The command will be evaluated for each value of the variable between the beginning and ending values. For example: `{null:{for:i,10,1,-1,{tell:{&i}}}}` will echo a countdown from ten to one, inclusive, to the user.

FORCE

```
{force:object,command}
```

Forces the given player or thing to do the given command. The thing forced must be `@flock`'ed to the trigger object, or the trigger object's owner, and it must be set *Xforcible*, or else this function will get a `Permission denied` error. This function returns a null string. `{Force}` cannot force a thing-object to do something, if it is set `Dark,` is in a room set `Zombie,` if it has the same name as a player, or is owned by a player set `Zombie.`

FOREACH

```
{foreach:var,list,expr}
{foreach:var,list,expr,sep}
```

This evaluates `expr` for each and every item in the given list. On each evaluation, the temporary variable `var` will contain the value of the item under scrutiny. `Var` will only be defined for the duration of `expr,` and will be undefined after the `{foreach}` construct finishes. If `sep` is given, then it uses that string as the item seperator in list, instead of the usual carriage return character. `Sep` can be multiple characters long. This structure returns the result of the last evaluation of `expr.` Example: `{foreach:thing, {contents:here},{store:1,_seen}}`

FOX

```
{fox}
```

Returns the string `YARF!`

FTIME

```
{ftime:format}
{ftime:format,tz}
{ftime:format,tz,secs}
```

Returns a time string in the format you specify. See the MUF Reference entry on timefmt for the `%subs` that you can use in the format string. If specified, `tz` is the number of hours offset from GMT. If specified, `secs` is the systime to use, instead of the current time. `{ftime:%x %X %Y,8,0}` will return the date and time for systime 0, for the Pacific time zone.

FULLNAME

```
{fullname:obj}
```

Returns the name of the given object. In the case where the object is an exit, then the full name of the exit is given, including all the ; aliases. The object must be in the immediate vicinity, or be controlled by the owner of the trigger object.

GE

```
 |  >=
{ge:expr1,expr2}
```

Evals `expr1` and `expr2,` then returns true if `expr1` was larger or equal.

GT

```
 |  GREATERTHAN  |  >
{gt:expr1,expr2}
```

Evaluates `expr1` and `expr2,` then returns true if `expr1` was larger.

HOLDS

```
{holds:obj1}
{holds:obj1,obj2}
```

Returns true if the location of `obj1` is `obj2.` If no `obj2` argument is given, then this will return true if the location of `obj1` is the player.

HOW

```
 |  &HOW
{&how}
```

The `{&how}` variable is a short string telling what ran the `MPI` command. It can have the values `'(@desc)', '(@succ)', '(@osucc)',` etc. for when it is run from an `@desc,` an `@succ,` an `@osucc,` or whatever. It can also have the value `'(@lock)'` for when it is run from a lock test.

IDLE

```
{idle:player}
```

Returns player idle time in seconds. If the given player is not connected, or is not a player object at all, then this will return -1. This returns the idle time for the most recently connected connection, if there are multiple connects.

IF

```
{if:check,true}
{if:check,true,false}
```

This is a simple conditional command. It evaluates the `check` argument and if it is true, then it evaluates the `true` argument and returns its result. If `check` does not evaluate as true, then it will evaluate the `false` argument, if there is one, and returns its result. If there is no false argument, and `check` evaluated false, then it returns a null string. Example:

```
    Your computer is {if:{eq:2,3},broken!,All right.}
```

INC

```
 |  INCREMENT
{inc:var}
{inc:var,val}
```

Increments the value of the given variable by one, returning the result. If a value argument is given, then it will add that value to the variable, instead of the value `1.`

INDEX

```
{index:propname}
{index:propname,obj}
```

Returns the string value of the property whose name is stored in the given property. This sounds confusing, but it's basically just the same as `{prop:{prop:propname}}`. If no object parameter is passed to it, it looks for both the index property and the referenced property somewhere down the environment from the trigger object. Otherwise, it looks down the environment from the object specified for both of them. If either property is not found, this returns an empty string. If the property that it tries to access is read restricted, and the owner of the trigger object does not own the object that the properties are found on, then the `MPI` script stops with a `Permission denied` error.

INDEX!

```
{index!:propname}
{index!:propname,obj}
```

Returns the string value of the property whose name is stored in the given property. This sounds confusing, but it's basically just the same as `{prop!:{prop!:propname}}`. If no object parameter is passed to it, it looks for both the index property and the referenced property on the trigger object. Otherwise, it looks on the specified object for both of them. If either property is not found, this returns an empty string. If the property that it tries to access is read restricted, and the owner of the trigger object does not own the object that the properties are found on, then the `MPI` script stops with a `Permission denied` error.

INSTR

```
{instr:str1,str2}
```

Lists the position of the first substring within `str1` that matches `str2.` If no such substring exists, then this returns a 0.

ISDBREF

```
{isdbref:dbref}
```

Returns true if the string passed to it is a valid dbref.

ISNUM

```
{isnum:number}
```

Returns true if the string passed to it is a valid number.

```
ISTYPE
{istype:obj,typ}
```

Returns true if the given object if of the given type. Valid types are: `Bad,` `Room,` `Exit,` `Thing,` `Player,` and `Program.`

KILL

```
{kill:0}
{kill:processID}
```

Kills a process on the timequeue, that was possibly created by `{DELAY}.` If the process ID it is given is 0, then it will kill all processes done by that trigger object. If the process to be killed was not set off by that trigger, and was not set off by any object that the owner of the trigger owns, then this will error out with `Permission denied`. If no process is found, this returns 0. If a process was found, and the permissions were okay, then the process is killed, and `{kill}` returns the number of processes killed. Usually one.

LASTUSED

```
{lastused:obj}
```

Returns the systime when obj was last used.

LCOMMON

```
{lcommon:list1,list2}
```

Creates a list containing every item that appears in *both* `list1` and `list2.` Any duplicate items in the resulting list are removed.

LE

```
 |  <=
{le:expr1,expr2}
```

Evals `expr1` and `expr2,` then returns true if `expr1` was smaller or equal.

LEFT

```
{left:string}
{left:string,fieldwidth}
{left:string,fieldwidth,padstring}
```

Takes a string and pads it to fit the given fieldwidth, with the string left justified. If no padstring is given, it assumes that it will pad the string with spaces. If no fieldwidth is given, it assumes that the field width is 78 characters. Example: `{left:Hello,10,_.}` would return the string `"Hello_._._"`

LEXEC

```
{lexec:listname}
{lexec:listname,obj}
```

This takes a property based list, and concatenates all its lines together, stripping spaces from the beginning and end of each one. It then evaluates the result for `MPI` commands, and returns the resulting string. A property based list is a series of properties that are consecutively numbered. The server understands several different formats, and can also read in property lists in either the `propnameX` format, or the `propname#/X` format. If no `obj` argument is supplied, then it looks for the list somewhere down the environment from the trigger object. Otherwise, it looks for the list down the environment from the given object.

LINKS

```
{links:obj}
```

Returns the object reference of what the given object if linked to. Since exits can be meta-links, linked to multiple exits, if there is more than one link, then this function returns a list of all the destinations, seperated by carriage return characters. (\r)

LIST

```
{list:listname}
```

```
{list:listname,obj}
```

Returns a string, containing a carriage-return delimited list of individual lines from a property based list. A property based list is a series of properties that are consecutively numbered. The server understands several different formats, and can also read in property lists in either the `propnameX` format, or the `propname#/X` format. It does `not` evaluate the contents of the list for embedded `MPI` commands. If no `obj` argument is supplied, then it looks for the list somewhere down the environment from the trigger object. Otherwise, it looks for the list down the environment from the given object.

LISTPROPS

```
{listprops:propdir}
{listprops:propdir,object}
{listprops:propdir,object,pattern}
```

This function will return a list that contains the full names of all the sub-properties contained by the given propdir. If not given, `object` defaults to the trigger object. If a pattern is given, the sub-properties in the propdir are each compared against the smatch wildcard pattern, and only those that match are returned in the list. This comparison is only done on the last part of the property name after the last `/.` See also propdir and smatch.

LIT

```
 |  LITERAL
{lit:string}
```

Returns the literal string given as its parameter. This means you can have things that look like `MPI` commands within it, and it will not evaluate them, but will rather just treat them as a string.

LMEMBER

```
{lmember:list,item}
{lmember:list,item,delimiter}
```

Returns `0` if the given item is *not* in the given list, otherwise, it returns the item's position in the list. The first list item in the list would return `1,` and the third would return `3,` for example. If the delimiter argument is given, then it treats the list as if it were delimited by that string, instead of by carriage returns. (`\r`'s) Example: `{lmember:{mklist:a,b,c,d,e,f},d}` would return `4.`

LOC

```
 |  LOCATION
{loc:obj}
```

Returns the location of the given object. The object must either be in the vicinity, or it must be controlled by the owner of the trigger object.

LOCKED

{locked:player,obj}

Tests the `_/lok (@lock)` standard lock property on obj against the given player. Returns true if the lock is locked against the player.

LRAND

```
{lrand:list}
{lrand:list,seperator}
```

Returns a randomly picked stringlist item from the given list. If the seperator argument is given, then it will assume that the stringlist has its items delimited by the given seperator string, instead of by carriage returns.

LUNION

```
{lunion:list1,list2}
```

Combines the contents of `list1` and `list2`, removing any duplicates.

LUNIQUE

```
{lunique:list}
```

Returns `list` with all duplicate items removed.

```
LREMOVE
{lremove:list1,list2}
```

Returns the contents of `list1`, with any items that match an item in `list2` removed. The resulting list has all duplicate items removed.

LSORT

```
{lsort:list}
{lsort:list,var1,var2,expr}
```

Returns the sorted contents of `list.` If 4 arguments are given, then it evaluates expr with a pair of values, in `var1` and `var2.` If `expr` returns true, then it will swap the positions of the two values in the list. It runs this comparison on every pair of items in the list, so it will be evaluated N*N times, where N is the number of items in the list. This method can also be used to randomize a list. Example:

```
    {lsort:{&list},v1,v2,{gt:{dice:100},50}}
```

LT

```
 |  LESSTHAN  |  <
{lt:expr1,expr2}
```

Evaluates `expr1` and `expr2`, then returns true if `expr1` was smaller.

LTIMESTR

`{ltimestr:secs}`

Given a time period, in seconds, this will return a string, including a breakdown of all the time units of that period. For example, given a number of seconds, it might return `"1 week, 2 days, 10 mins, 52 secs".`

MAX

`| MAXIMUM`
`{max:expr1,expr2}`

Returns the greater of the values of `expr1` and `expr2.`

MKLIST

`{mklist:value...}`

Returns a list with all the given values as list items, seperated by carriage returns. (`\r`'s) Example: `{mklist:Tom,Dick,Harry}` returns `"Tom\rDick\rHarry"`. Note: A maximum of nine items can be passed to the `{mklist}` function. If you need more, you can chain `{mklist}s` together. Example:

```
    {mklist:{mklist:a,b,c,d,e,f,g,h,i},j,k,l,m,n,o,p}
```

MIDSTR

`{midstr:str,pos}`
`{midstr:str,pos1,pos2}`

Returns the substring that starts at `pos1` within `str.` If no `pos2` is given, then the returned string is only the character at the given `pos1` position. If a `pos2` position is given, then it returns the substring beginning at `pos1` and ending at `pos2,` inclusive. If `pos1` or `pos2` are negative, then they represent the position that is that absolute number of characters from the end of the string. The first character in `str` is 1, and the last one can always be referenced by -1. If a position would be before the beginning of the string, it is assumed to be at the beginning of the string. If it would be beyond the end of the string, it is assumed to be at the last character. If the starting position is later in the string than the ending position, then the returned string has the characters in reverse order. If either `pos1` or `pos2` are 0, then this returns a null string. (`""`)

MIN

`| MINIMUM`
`{min:expr1,expr2}`

Returns the lesser of the values of `expr1 and` expr2.

MODIFIED

`{modified:obj}`

Returns the systime when `obj` was last modified.

MOD

```
 |  MODULO
{mod:expr1,expr2}
```

Returns the leftover remainder of `expr1` divided by `expr2.` If more than two arguments are given, then the first arguments is modded by the second, then the result of that would be modded by the third, and so on and so forth. For example: `{mod:91,20,3}` would be read as `91 % 20 % 3`, and a result of `2` would be returned.

MONEY

```
{money:obj}
```

This returns the value of an object of type `Thing`, or returns how many pennies a player has.

MUCKNAME

```
{muckname}
```

Returns the muck name string. Example: FurryMUCK

MUF

{muf:prog,arg}

Runs the given `MUF` prog with the string arg on the stack. This returns the top stack value when the prog exits. If the `MPI` code was run from a propqueue like `_listen,` or `_connect,` then `{muf}` cannot run a `MUF` program with a Mucker level of less than 3

MULT

```
 |  MULTIPLY
{mult:expr1,expr2}
{mult:expr1,expr2,expr3...}
```

Returns the product of the values `expr1` and `expr2.` If more than two args are given, then they are all multiplied together to get the result.

NAME

{name:obj}

Returns the name of the given object. If the object is an exit, the name returned is the first exit name it has before the first '`;`'. The object must be in the vicinity, or controlled by the owner of the trigger object.

NE

```
 |  NOTEQUALS  |  !=  |  <>
{ne:expr1,expr2}
```

If `expr1` and `expr2` evaluate out to the same value, then this returns false. Otherwise, this returns true. If both expressions evaluate out to numbers, then this compares them numerically.

NEARBY

```
{nearby:obj}
{nearby:obj,obj2}
```

If one argument is given, then this returns true ("1") if the given object is nearby to the trigger object. If two arguments are given, then it returns true if the two objects are nearby one another. Otherwise, this returns false. ("0") Nearby is defined as: 1) The two objects are in the same location, or 2) One object contains the other, or 3) the two objects are in fact the same object.

NL

```
 |  \r
{nl} or \r
```

Returns a carriage return character. This can be used to seperate items in a list, or can split the string at that point, starting a new line. Example: the string:

```
    This is\ran example{nl}of using newlines.
```

would print out like:

```
This is
an example
of using newlines.
```

NOT

```
 |  !
{not:expr}
```

Returns the logical `NOT` of `expr. If` expr was true, this returns false. If `expr` was false, this returns true.

```
NULL
{null:expr...}
```

Returns a null string, no matter what the expressions within it return. This can take up to nine arguments, though you could pass the output of several commands as one argument.

ONLINE

```
{online}
```

Returns a list of players who are online. This function can only be executed by wizbitted objects.

ONTIME

```
{ontime:player}
```

Returns `player`'s online time in seconds. If the given player is not connected, or is not a player object at all, then this will return -1. This returns the online time for the most recently connected connection, if there are multiple connects.

OR

```
{or:expr1,expr2...}
```

Returns true if `expr1` or `expr2` evaluate as true. Otherwise, this returns false. If `expr1` was true, this doesn't bother to exaluate `expr2,` as it does C-style shortcutting. If there are more than two arguments, then this will evaluate them until either one returns true, or until it has evaluated all the expressions. This returns false only if all of the expressions return false.

OTELL

```
{otell:string}
{otell:string,room}
{otell:string,room,player}
```

This will tell the given string to all the players in the room, except for the given player. If no room argument is given, it is assumed to be the room that the triggering player is in. If no player is given, then it assumes that you want to skip sending the message to the triggering player. If you pass it a player of #-1, it will send the message to all the players in the room. This returns the message that was sent. If the trigger isn't a room, or an exit on a room, and if the message doesn't already begin with the user's name, then the user's name will be prepended to the message.

OWNER

```
{owner:obj}
```

Returns the owner of the given object. The object must be in the vicinity, or be controlled by the trigger object's owner.

PARSE

```
{parse:var,list,expr}
{parse:var,list,expr,sep}
{parse:var,list,expr,sep,s2}
```

This evaluates `expr` for each item in the given list. On each evaluation, the temporary variable `var` will contain the value of the item under scrutiny. This function returns a list containing the output of `expr` for each item within the list. This lets you do direct translation of a list of dbrefs, for example, into a list of names. `var` will only be defined for the duration of `expr,` and will be undefined after the `{filter}` construct finishes. If `sep` is given, then it uses that string as the item seperator in the input list, instead of the usual carriage return character. If `s2` is defined, then it will use that string to seperate the items in the list it returns, instead of the normal carriage return. `Sep` and `s2` can be multiple characters long.

PRONOUNS

```
{pronouns:string}
{pronouns:string,object}
```

If passed one argument, evaluates the string and does pronoun substitution with regards to the using player. If given two args, it does the pronoun substitution with regards to the given object.

PROP

```
{prop:propname}
{prop:propname,obj}
```

Returns the literal string value of the given property. If no object parameter is passed to it, it looks for the property somewhere down the environment from the trigger object. Otherwise, it looks down the environment from the object specified. If the property is not found, this returns an empty string. If the property that it tries to access is read restricted and the owner of the trigger object does not own the object that the property is found on, then the `MPI` script stops with a `Permission denied` error.

PROP!

```
{prop!:propname}
{prop!:propname,obj}
```

Returns the literal string value of the given property. If no object parameter is passed to it, it looks for the property on the trigger. Otherwise, it looks for the property on the object specified. If the property is not found, this returns an empty string. If the property that it tries to access is read restricted and the owner of the trigger object does not own the object that the property is found on, then the `MPI` script stops with a `Permission denied` error.

PROPDIR

```
{PROPDIR:propname,obj}
```

If the given property on the given object is a propdir, containing sub-properties, then this returns true. Otherwise it returns false. Object will default to the trigger object, if not given.

RAND

```
{rand:listname}
{rand:listname,obj}
```

Returns the value of a randomly picked list item from a property based list. If no `obj` parameter is given, then it looks down the environment from the trigger object for the list. Otherwise, it looks down the environment from the given object.

REF

```
{ref:obj}
```

Returns the dbref of the given object in the form `#xxxx.` The `object` must be in the vicinity, or controlled by the owner of the trigger object.

RIGHT

```
{right:string}
{right:string,fieldwidth}
{right:string,fieldwidth,padstring}
```

Takes a string and pads it to fit the given fieldwidth, with the string right justified. If no padstring is given, it assumes that it will pad the string with spaces. If no fieldwidth is given, it assumes that the field width is 78 characters. Example:

```
    {right:Hello,10,_.}
```

would return the string

```
    "_._._Hello"
```

SECS

```
{secs}
```

Returns system time: the number of second since midnight 1/1/70 GMT

SELECT

```
{select:value,listname}
{select:value,listname,object}
```

Returns the value of a single list item from a sparse property list. The item chosen is the one who's line number is the largest one that is less than or equal to the given value. If the list is missing any items, then `{select}` will return the item in the list with the highest line number that is less than or equal to the given value. For example, if the list has the following entries:

```
_junk#/1:one
_junk#/5:two
_junk#/16:three
_junk#/20:four
```

Then `{select:9,_junk}` will return `"two",` `{select:16,_junk}` will return `"three",` and `{select:25,_junk}` will return `"four".`

SET

```
{set:var,value}
```

This sets the value of the given named variable to the given value. If no variable with that given name is currently defined, then this gives an error message complaining about that.

SIGN

```
{sign:expr}
```

Returns -1 `if` expr is negative. Returns `1` if `expr` is positive. If `expr` is `0,` then it returns `0.`

SMATCH

```
{smatch:str,pattern}
```

Matches `str` against the wildcard pattern. If there is a match, this returns true, or `"1".` If it doesn't match, this returns a value of `"0",` or false. In wildcard patterns, the following characters have the following meanings:

```
  *               matches any number of any character
  ?               matches one character, of any type
  [abcde]         matches one char, if it is a, b, c, d, or e
  [a-z]           matches on char, if it is between a and z, inclusive
  [^abd-z]        matches one char if it is NOT a, b, or beteen d and z
  {word1|word2}   matches one word, if it is word1, or word2
  {^word1|word2}  matches one word, if it is NOT word1 or word2
  \               escapes any of the prev chars, making it not special
```

STIMESTR

```
{stimestr:secs}
```

Given a time period, in seconds, this will return the most significant time unit of that time period. For example, a number of seconds, that is equivalent to 9 days, 23 hours, 10 minutes, and 52 seconds, will be have the value `9d` returned, as the abbreviated most significant time unit.

STORE

```
{store:val,prop}
{store:val,prop,obj}
```

Stores a string value in a given property. If no obj parameter is given, then it stores the property on the trigger object. Otherwise, it will store it on the given object. If you specify a propname that is protected, you will get a `Permission denied` error. You are only allowed to store properties on objects controlled by the owner of the trigger object. The trigger object is the object that triggered the evaluation of the `MPI` commands. This function returns the string that is stored as the prop value. If you store a null value in the property, then it will remove the property if it is not a propdir. It will clear the value of the prop if it *is* a propdir.

STRIP

```
{strip:string}
```

Returns a copy of string with all the spaces stripped from the beginning and the end.

SUBLIST

```
{sublist:list,pos1}
{sublist:list,pos1,pos2}
{sublist:list,pos1,pos2,sep}
```

Takes a list, and returns a subset of the list items within it. The subset is all the list items between list item `pos1,` and list item `pos2,` inclusive. If the `pos2` argument is omitted, it assumes that `pos2` is the same as `pos1.` If `pos2` is less than `pos1,` then all the list items between `pos2` and `pos1` are returned, in reversed order. If `pos1` or `pos2` are negative, it counts that many list items back from the end of the list, so `-1` is the last list item, and `-5` would be the fifth from last list item. The input list is assumed to be delimited by carriage returns (`\r`) unless the sep argument is given.

SUBST

```
{subst:str,old,new}
```

Returns a copy of `str` with all substring instances of `old` replaced by the text specified by `new.` Basically just substitutes the new text for the old text in `str.` Example: `{subst:Hello World!,l,r}` would return `"Herro Worrd!"`

SUBT

```
 |  SUBTRACTION
{subt:expr1,expr2}
{subt:expr1,expr2,expr3...}
```

Returns the difference of the values `expr1` and `expr2.` If more than two args are given, all values are subtracted from the first value in sequence. For example: `{subt:10,3,2,4}` would be read as `10 - 3 - 2 - 4,` and it would return a result of `1.`

TELL

```
{tell:string}
{tell:string,player}
```

If passed only a string, tells the user that string. If passed both a string, and a player dbref, it will tell the given player the message. This returns the message that was sent. If the trigger isn't a room, or an exit on a room, and if the message doesn't already begin with the user's name, then the user's name will be prepended to the message. The two exceptions to this are that if the messages is being sent to the owner of the trigger, or to the user, then the user's name will not be prepended.

TESTLOCK

```
{testlock:obj,prop}
{testlock:obj,prop,who}
{testlock:obj,prop,who,def}
```

Tests the lock property `prop,` on `obj` against the given player `who.` If no `who` argument is given, then it checks the lock against the using player. If a `def` argument is given, then the lock will default to that value, if there is no lock property of the given name on the given object. Returns true if the lock is locked against the player.

TIME

```
{time}
{time:timezone}
```

Returns a time string in the 24hr form `hh:mm:ss.` If the timezone argument is given, then it offsets the time returned by that number of hours.

TIMESTR

```
{timestr:secs}
```

Given a time period in seconds, this will return a concise abbreviated string representation of how long that time was. This might return a value like `"9d 12:56"` for `9 days, 12 hours, and 56 minutes`.

TIMESUB

```
{timesub:period,offset,listname}
{timesub:period,offset,listname,object}
```

This is sort of like `{list},` except that it will only return one line of the given named property list. The line it chooses depends on the time. The period is the length of time, in seconds, that it takes for `{timesub}` to cycle through the entire list. The offset is the number of seconds to offset into the time period, if you actually need to synchronize the `{timesub}` with something. The offset usually is just left at zero. If the object argument is not passed, it looks for the list on the trigger. What this all means, is that if you have, for example, a period of 3600 (one hour), an offset of zero, and a property list that has six items in it, then `{timesub}` will return the first line of the property list during the first ten minutes of the hour, the second line during the next ten minutes, and so on, until it returns the last line during the last ten minutes of the hour. Then it returns the first line for the beginning of the next hour. Here's an example:

```
    {timesub:86400,0,_sunmoon}
```

This example will show different property list lines, depending on the time of day. The period is 86400 seconds, which is one day. If the property list has 24 items in it, then a different line will be returned for each hour of the day.

TOLOWER

```
{tolower:string}
```

Returns a copy of string, with all uppercase characters converted to lowercase.

TOUPPER

```
{toupper:string}
```

Returns a copy of string, with all lowercase characters converted to uppercase.

TYPE

```
{type:obj}
```

Returns the type of an object. The possible values are: `Bad`, `Room`, `Exit`, `Thing`, `Player`, and `Program`.

TZOFFSET

```
{tzoffset}
```

Returns local time zone offset from GMT in seconds.

USECOUNT

```
{usecount:obj}
```

Returns the usecount of `obj`.

V

```
 |  VARIABLE  |  &
{v:var}
{&var}
```

These are two ways of trying to do the same thing. They return the value of the named variable `var`. If there is no variable with the given name currently defined, then this gives an error stating as much. Variables can be defined either with the `{with:}` function or within a looping command.

VERSION

```
{version}
```

Returns the version string for the server.

WHILE

`{while:check,expr}`

This is a looping structure. It evaluates `check,` and if it evaluates true, then it evaluates `expr`, and repeats the process. If `check` evaluates false, then the loop is exited. This returns the result of the last evaluation of `expr.`

WITH

`{with:var,val,expr..}`

This defines a new variable with the given name, and sets its value to the given `val.` Up to 7 `expr's` are allowed, but the only value returned to `{with}'s` caller, is the value returned by the evaluation of the last `expr.` If there is already a variable of the same name, then this command will override that variable, for the duration of the `{with:}` command. The new variable is only valid to use within the confines of the `{with:}` command, and it will go away after the command completes. This provides scoped variables quite effectively. *Note*: There can be no more than 32 variables defined at any one time, total. This includes variables that are defined within macros, or properties or lists that are executed with `{exec:}` or `{lexec:}.` Here's an example to illustrate the scope of variables inside of `{with:}` commands:

```
  {prop:_mydesc}                          <- {&people} not defined
  {with:people,{contents:here,players},  <- Defining. Not available yet
    {if:{count:{&people}},                <- It's usable now
      The players awake here are
      {lit: }                             <- just puts in a space
      {commas:{&people},{lit: and },
        who,{name:{&who}}                 <- uses {&who} as temp var
      }                                   <- {&who} no longer defined
    }
  }                                       <- {&people} no longer defined
```

XOR

` | EXCLUSIVEOR`
`{xor:expr1,expr2}`

Returns true if `expr1` or `expr2` evaluate as true, but false if both do. Otherwise, this returns false.

3.2 Overview: MUF

`MUF` — a dialect of `FORTH` — is one of two programming languages implemented on all `MUCKs,` the other being `MPI.` The speed and efficiency of `MUF` make `MUCKs` readily user-extensible: powerful new commands and programs can be soft-coded into the database. Although the only place you'll be able to use it is on `MUCKs, MUF` is a real programming language: once you've learned it, you can truthfully say you know how to program computers, and concepts and habits of thought you pick up as a Mucker will be useful in learning languages with widespread `RL` applications.

`MUF` is an extensible, structured, stack-based, compiled language.

Extensible:
If there isn't a command to do what you want, you can make one.

Structured:
A `MUF` program consists of 'functions' or 'words' ... blocks of code that are executed as a unit and 'call' each other as needed. It doesn't matter (to the computer) whether you put all your code on one long line or every word on a new line (it matters a lot to people who are trying to read your code). White space (any combination of spaces, tabs, or new lines) separates words, and the order of the words and their positioning between the symbols that start and end a function are what matter. The program is composed of 'functions' or 'words' that each perform a specific task.

Stack-based:
You do everything by manipulating a stack, a set of 'last-in-first-out' values like HP calculators use.

Compiled:
You write a program that people can read, with semi-normal words like `pop` and `rot` and `remove_prop` (this is your source code) and then a part of the server, the `MUF` compiler, turns it into arcane stuff that computers can read (this is your object code). You won't ever see the object code.

Before You Begin:

In order to program in `MUF,` you'll need a Mucker bit, a flag that lets you use the `MUF` editor. A wiz will need to set this, so page one and ask. There are three levels of mucker bits, `M1` (apprentice), `M2` (journeyman), and `M3` (master) (wizards are considered `M4`).

As a new Mucker, you'll get an `M1` bit. `M1`'s have access to most of the functions, but not all; output from an `M1` program to anyone other than the owner of the program is prefaced by the user's name; and the program won't be able to send messages to or retrieve information from remote locations. An `M1` bit is essentially `MUF` with training wheels. Once you've written a couple `M1` programs that work, you can show them to a wiz and ask for an `M2` bit. `M2`'s can't use all the functions, but you can make truly useful programs at the `M2` level: bulletin board or book programs, specialized exits and locks, morphing programs, etc.

`M3` bits are dandy to have, but they are hard to come by on large mucks, and for good reason. An `M3` bit gives its owner considerable power over the data base, approaching that of a wizard: an inept or malintentioned `M3` coder could cause serious problems. In fact, `M3`'s are in some ways more of a security risk than Wizard bits: wiz bits are more powerful, but all commands issued by a wizard are logged; this is not necessarily true of `M3` programs and players. To get an `M3` bit on a well established `MUCK,` you will need to write some very good programs and have shown yourself to be a trustworthy player over time. In general, it's easier to get an `M3` on newer, smaller `MUCKs.`

There are two commands that are good to know Before you Begin. One is `man,` the online manual command. Typing `man pop` would tell you about the `MUF` primitive `pop.` The other is `@q.` This aborts a foreground program you have running. If you find yourself in a run-away or locked-up program, type `@q` to get out of it.

3.2.1 Mucker Levels

There are — in effect, if not literally — five MUF permission levels: 0 to 4.

Mucker Level 0 refers to players who do not have a Mucker or Wizard bit. They cannot create new programs or use the `MUF` editor. `MUF` programs owned by them run as if they were Level 1 Muckers.

Mucker Level 1 players are 'apprentices'. They can use the editor and create `M1` programs. Their programs can only retrieve information about objects in the same room. For example, `#123 name` will fail if object `#123` is not in the same room as the user running the program. `M1` programs always run as if they are set Setuid. Output (from `NOTIFY, NOTIFY_EXCEPT,` and `NOTIFY_EXCLUDE`) can only go to players in the same room, and will usually (depending on server compiler directives) be prefaced with the user's name. They cannot use `ADDPENNIES. M1` programs have an instruction-count limit of about 20,000 instructions.

Mucker Level 2 players are 'journeymen'. They can use most but not all `MUF` primitives. They can create `M2` programs, and set programs they own `M2` or `M1. M2` programs are limited to about 80,000 instructions.

Mucker Level 3 playes are 'masters'. They can set programs they own `M1, M2,` or `M3.` They can use the connection information primitives (`CONDBREF, ONLINE,` etc.), can read the `EXITS` list of any room, can use `NEXTOBJ` on objects, can use `NEWROOM, NEWOBJECT, NEWEXIT,` and `COPYOBJ` without limitations, can use `QUEUE` and `KILL,` and can override the permissions restrictions of `MOVETO.` There is no absolute limit on `M3` processes' intruction count.

Wizards and wizbitted programs are effectively Mucker Level 4. They can set programs they own `M1, M2,` or `M3,` and `W` or `!W.` They can use primitives not available to other Mucker Levels, including `RECYCLE, CONHOST, FORCE,` and `SETOWN.` All properties are readable and settable by `Wizard` programs.

The effective Mucker Level of a program is its own Mucker Level or its owner's Mucker Level, whichever is lower. There is one exception: Programs owned by wizards who do not have a Mucker bit set run at Mucker Level 2, unless the program is set `Wizard.`

To set Mucker bits, use just the number as a flag: `@set cashmere=2.`

MUF Libraries

Libraries allow `MUF` programs to share code: frequently used, generic routines can be written and compiled once, in a library; other programs can use these routines by using an `$include` statement to include the library. Once a library has been included, any routines defined in the library can be used in the local program. A number of standard libraries are available, and will be needed to compile a number of the standard `MUF` programs. Section 3.2.6, the `MUF` Library Reference, gives comprehensive documentation of routines available from the standard libraries. In addition, you can create your own libraries.

Using Library Functions:

To use library functions in your programs, include the library or libraries needed by placing an `$include` statement in your code, before any library functions are called, and outside of any functions. (Most programmers put all `$include` statements at the top of their programs). Libraries are specified by their registered names. For example, to use the reflist-handling functions from lib-reflist, you would put the following line of code in your program:

```
    $include $lib/reflist
```

At any point after this line, functions defined in lib-reflist can be used in your program.

To list the the libraries available on your `MUCK,` type `@register lib`. To view a library's documentation, type `@view $lib/<library name>`.

Creating Libraries

Creating libraries is relatively straightforward. You simply write code that could be useful to include in multiple programs, make individual functions available to other programs with the `public` declaration, and set some properties that give the server the information it will need to coordinate between programs and programmmers the information they will need in order to make good use of your library. The remainder of this page illustrates these by example: we'll create, configure, and use lib-sort, a library of sorting functions. Take glance at the file now, and refer to it as we discuss its implementation.

Step 1: Create a program with useful, generic routines. Programs frequently need to sort data, but neither `MUF` nor the standard libraries include functions for doing this. Lib-sort addresses this need. It uses a simple bubble sort to sort a stack range of items. (A 'stack range' is a group of items on the stack and an integer indicating how many items are to be considered part of the range. A more complete discussion of stack ranges is available in Section 3.2.6.) For example, if we had five strings on the stack that we wanted to sort alphabetically...

```
"mink" "timber wolf" "otter" "woolly pterydactyl" "linsang"
```

... we could do so by putting an integer that gives the 'count' of the range on top of the stack, and then calling lib-sort's `sort` function...

```
"mink" "timber wolf" "otter" "woolly pterydactyl" "linsang" 5
sort
```

Lib-sort would return the five items in alphabetical order, with the count integer removed:

```
"linsang" "mink" "otter" "timber wolf" "woolly pterydactyl"
```

Lib-sort provides two public functions: `sort`, which returns the range sorted in ascending order, and `sort-d`, which returns the range in descending order. All items in the range must be of the same data type (`string`, `integer`, or `dbref`). Lib-sort looks at the data type of the top item in the range to determine how to sort the range.

(A bubble sort is not the most efficient algorithm: it is relatively slow when sorting large sets (though it is probably the most efficient algorithm for very small sets), and it does not finish any faster if the data is already sorted or mostly sorted. Other algorithms, such as a qsort, would be better in many situations. Implementing a sorting library with more sophisticated algorithms would be a good advanced `MUF` programming exercise.)

Step 2: Declare the appropriate public functions. Lib-sort includes a total of eight functions, but only two of them are meant to be called by other programs: `sort` and `sort-d`. These two functions look at the stack range and call one of the remaining six functions, which sort, ascending or descending, according to datatype. So, `sort` and `sort-d` need to be declared public, and the others should not be. We do this by putting the lines...

```
public sort
public sort-d
```

...in the program, after the functions are defined, outside of any function definitions. Here, we put them after each function. Another common convention is to put all the public declarations at the end of the program.

Step 3: Document the program. Lib-sort uses the most common documentation method, a header comment. Lines 1 to 49 explain how to install and use the program.

Step 4: Set the program's `_docs` property to allow users to `@view` the documentation: `@set lib-sort=_docs:@list $lib/sort=1-49.`

Step 5: Set the program `Link_OK`, so that other programs can access its public functions and users can view its documentation with `@view`: `@set lib-sort=L`.

Step 6: Configure the interface for each public function by setting a `_defs/` property. These properties give the server the information it needs to match statements in local programs to public functions in the library.

```
    @set lib-sort=_defs/sort:"$lib/sort" match "sort" call
    @set lib-sort=_defs/sort-d:"$lib/sort-d" match "sort-d" call
```

That's it!

3.2.3 MUF Macros

As with `MPI,` you can define macros அ snippets of reusable code called by a single statement அ with `MUF.` In fact, viewing defined macros and defining new ones is easier with `MUF` than with `MPI.`

To view, define, or delete `MUF` macros, you need to be in the `MUF` editor. You can use any program when invoking the editor; the defined macros are available to all programs.

Viewing Macros: In the editor, type either `s` (to see the long version of macro definitions) or `a` (to see the abridged version).

(The example output below is slightly reformatted.)

```
> @edit tinker.muf
> s

  abs          Jessy ( i -- i' ) dup 0 < if -1 * then

  after_match  Jessy (d -- ) dup #-1 dbcmp swap #-2 dbcmp or if exit then


  atodbref     Jessy ( s -- d ) "" "#" subst atoi dbref

  .
  . ( .... entries omitted .... )
  .

  wizard?      Jessy ( -- i ) "W" flag?

  yes?         Jessy ( s -- i ) ( checks that string starts with y ) 1
  strcut pop "y" stringcmp not
  End of list.

> a
 abs              addpropstr     after_match    atodbref       confirm
 debug-line       debug-off      debug-on       me_wiz?        no?
 notify_nospam    otell          pmatch         puppet?        quote
 rstrfmt          spc80          split          tell           wizard?
 yes?

  End of list.
```

Defining Macros: In the editor (but not while you are inserting code), type `def <macro name> <macro definition>`.

```
> @edit tinker.muf
> def nukestack ( x..x' -- ) begin depth while pop repeat
  Entry created.
```

Deleting Macros: Only wizards can delete macros, so, if you are a non-wizard Mucker entering macros, be sure you have it right when you enter it. If you are a wizard and want to delete a macro, type `<macro name> kill` while in the editor.

```
> @edit tinker.muf
> nukestack kill
  Macro entry deleted.
```

3.2.4 MUF Compiler Directives

Compiler directives — additional instructions that the server executes before or while compiling your code — were introduced in the MUF Overview. Below is a more complete reference of the available compiler directives.

`$define <definition name> <definition> $enddef`

This redefines `<definition name>` such that all instances of it will be replaced with `<definition>` when the program is compiled.

`$undef <defname>`

This undefines `<defname>`, making it an undefined value for the compiler. Its primary use is to allow compiler directives to be included in the source code without their being invoked by the compiler. For example, you might have a program that should be compiled differently if running at Mucker Level 3 rather than Mucker Level 2. You could include the following lines in your code...

```
$define MLev2     (* Undefine this if the program is set M3 *)
$undef  MLev3       (* Define this if the program is set M2 *)
```

This way, the person compiling the program would have an example of each valid form, even though only one is being invoked.

```
$echo <string>
```

This outputs <string> to the person compiling the program, at compile time. Example:

```
$echo Compiling multi-guest.muf...
$echo See header comment for configuration instructions.
```

```
__version
```

This is a pre-defined macro that is replaced at compile time with the current server version. It would be useful for tests to ensure that the server version can support all the primitives used in your program. The replacement string will be the same as that returned by the `VERSION` primitive.

```
$ifdef <condition>
  <compile-if-true>
$else
  <compile-if-false>
$endif

$ifndef <condition>
  <compile-if-true>
$else
  <compile-if-false>
$endif
```

These cause conditional compilation of blocks of code, based on `<condition>`. `<Condition>` can be a `$defined name`, or a test that consists of a comparator such as =, <, or > and a test value, all in one word, without whitespace. The else clause is optional. Comiler directives are nestable. Example:

```
$ifndef __version>Muck2.2fb3.5
  $define envprop .envprop $enddef
$endif

$define ploc

$ifdef proplocs .proploc $else $endif $enddef
```

```
$include <program>
```

This creates `$defines` based on the `_defs/` propdir of `<program>.` For example, if object #`345` had the following properties...

```
/_defs/desc: "_/de" getpropstr
/_defs/setpropstr: dup if 0 addprop else pop remove_prop then
/_defs/setpropval: dup if "" swap addprop else pop remove_prop then
/_defs/setprop: dup int? if setpropval else setpropstr then
```

... then a program that contains the line `$include #345` would be compiled such that all instances of `desc`, `setpropstr`, `setpropval`, and `setprop` would be preplaced with their respective definitions on `#345`.

Escaped statements will be read literally, rather than interpreted and expanded by the compiler, which allows primitives and macros to be included in `$defines`. Example:

```
$define addprop over over or if \addprop else pop pop remove_prop
$enddef
```

In this case, `ADDPROP` will be expanded in the program itself, but the compiler will not recursively expand it... it will call the actual, in-server `ADDPROP`.

3.2.5 MUF Reference

Note: Some listed functions are only available in MUCK server version fb6.0 and following. Such cases are marked with the notation '(fb6.0+)' in individual entries.

ray-Handling Primitives:

rray_count	array_delitem	array_delrange	array_explode	array_firs
rray_getitem	array_getrange	array_insertitem	array_insertrange	array_keys
rray_last	array_make	array_make_dict	array_ndiff	array_next
rray_nintersect	array_notify	array_nunion	array_prev	array_seti
rray_setrange	array_vals			

twise Operators:

itand	bitor	bitshift	bitxor

onnection-Handling Primitives:

vake?	concount	condbref	conidle	contime
onhost	conboot	connotify	condescr	
escrflush	descriptors	online	nextdescr	

ontrol Structures:

oort	begin	break	call	continue
or	else	execute	exit	if
mp	kill	repeat	then	until
hile				

ata Conversion Primitives:

toi	ctoi	dbref	float	ftostr
nt	intostr	itoc	stod	strtof
ariable				

ror-Handling Primitives:

lear	clear_error	error?	error_bit	error_name
rror_num	error_str	is_set?	set_error	

put/Output Primitives:

otify	notify_except	notify_exclude	read	tread

ock-Handling Primitives:

etlockstr	locked?	parselock	prettylock	setlockstr
estlock	unparselock			

ath and Comparison Primitives:

- / %	< >	=	<=	>=
cos	and	asine	atan	atan2
eil	cos	dbcmp	dist3d	exp
abs	floor	frand	getseed	inf
og	log10	modf	not	number?
r	pi	polar_to_xyz	pow	random
ound	setseed	sine	sqrt	srand
trcmp	stringcmp	strncmp	tan	xyz_to_pol

essage-Management Primitives:

esc	drop	fail	name	odrop
fail	osucc	setdesc	setdrop	setfail
etname	setodrop	setofail	setosucc	succ

iscellaneous Primitives:

ddpennies	checkargs	checkpassword	contents	copyobj
btop	exits	flag?	force	getlink
nterp	location	match	mlevel	movepennie
oveto	newexit	newobject	newroom	next
extowned	owner	part_pmatch	pennies	pmatch
rog	recycle	rmatch	set	setlink
etown	stats	sysparm	textattr	trig
ersion				

ultitasking:

ackground	bg_mode	fg_mode	fork	foreground
ill	mode	pr_mode	preempt	queue
etmode	sleep			

roperty-Handling Primitives:

ddprop	envpropstr	getlockstr	getpropfval	getpropstr
etpropval	nextprop	propdir?	remove_prop	

ack-Handling Primitives:

epth	dup	dupn	ldup	lreverse
ver	pick	pop	popn	put
everse	rot	rotate	swap	

ring-Handling Primitives:

xplode	fmtstring	instr	instring	midstr
ronoun_sub	tolower	toupper	rinstr	rinstring
match	split	strcat	strcut	strdecrypt
trencrypt	stringpfx	strip	striplead	striptail
rlen	subst	unparseobj		

me Primitives:

ystime	timesplit	timefmt	date	gmtoffset
ime	timestamps			

pe-Checking Primitives:

rray?	dbref?	dictionary?	exit?	flag?
loat?	int?	ispid?	lock?	ok?
layer?	program?	propdir?	number?	room?
ring?	thing?			

riable-Handling Primitives:

	!	localvar	lvar	public
ar	variable			

`+ - * / %    ( x1 x2 -- i )`

These words perform arithmetic operations on integer and floating point numbers.

`+ is addition:`	`x1 + x2`
`- is subtraction:`	`x1 - x2`
`* is multiplication:`	`x1 times x2, or x1 * x2`
`/ is division:`	`x1 divided by x2, or x1 / x2`
`% is modulo:`	`remainter of x1 / x2, or x1 % x2`

The result of an integer division is truncated of fractions.

Modulo cannot be used on floating point values: see `MODF.`

All math operations may also be performed where `x1` is a variable type. This is mainly useful in calculating an offset for a variable array.

`< > = <= >=    ( x1 x2 -- i )`

Perform relational operations on integers or dbrefs `x1` and `x2.` These return `i` as `1` if the expression is true, and `i` as `0` otherwise.

`@    ( v -- x )`

Retrieves variable `v's` value `x.` (Pronounced 'fetch'.)

`!    ( x v -- )`

Sets variable `v's` value to `x.` See also `VARIABLE,` `VAR,` `LVAR,` `LOCALVAR,` and `miscellaneous.` (Pronounced 'store'.)

`ABORT    ( s -- )`

Aborts the MUF program with an error. For example, `'"Bad vibes." abort'` would stop the MUF program and tell the user a message like:

```
   Programmer error. Please tell Revar the following message:
   #1234 (line 23) ABORT: Bad vibes.
```

`ACOS   ( f -- f' )`

Returns the inverse cosine of `f`. (fb6.0+)

`ADDPENNIES    ( d i -- )`

`d` must be a player or thing object. Adds `i` pennies to object `d.` Without Wizard permissions, `addpennies` may only give players pennies, limited to between zero and `MAX_PENNIES.`

`ADDPROP    ( d s1 s2 i -- )`

Sets property associated with `s1` in object `d.` Note that if `s2` is null `"",` then `i` will be used. Otherwise, `s2` is always used. All four parameters must be on the stack; none may be omitted. If the effective user of the program does not control the object in question and the property begins with an underscore `'_',` the property cannot be changed. The same goes for properties beginning with a dot `'.'` which cannot be read without permission. If you store values, you must ensure that it they are never zero. Otherwise, when the user stores a non-zero number into the string field, (users may only access string fields) the next time TinyMUCK is dumped and loaded up again, the value field will be replaced with `atoi` (string field). If it is necessary to store zero, it is safer to just add 1 to everything.

`ADDRESS?    ( ? -- i )`

Returns true if the top stack item is a function address.

`AND    ( x y -- i )`

Performs the boolean 'and' operation on `x` and `y,` returning `i` as `1` if both `i1` and `i2` are `TRUE,` and returning i as `0` otherwise.

`ARRAY?   ( ? -- i )`

Returns true if the top item on the stack is an array. (fb6.0+)

`ARRAY_COUNT   ( a -- i )`

Returns a count of the items in an array. (fb6.0+)

`ARRAY_DELITEM   ( a @ -- a' )`

Removes a given item from an array. (fb6.0+)

`ARRAY_DELRANGE   ( a @ @ -- a' )`

Deletes a range of items from an array, between two indexes, inclusive. Returns the resulting array. (fb6.0+)

`ARRAY_FIRST   ( a -- @ i )`

Returns first the index in array, and a boolean. Bool is false if no items are in array. (fb6.0+)

`ARRAY_EXPLODE   ( a -- {@ ?} )`

Explodes array into stackrange of index/value pairs, such as `"idx0" "val0" "idx1" "val1" 2`. (fb6.0+)

`ARRAY_GETITEM   ( a @ -- ? )`

Gets a given item from an array. (fb6.0+)

`ARRAY_GETRANGE   ( a @ @ -- a' )`

Gets range between two indexes (inclusive) from an array, returning it as an array. (fb6.0+)

`ARRAY_INSERTITEM   ( ? a @ -- a')`

Inserts a given value into an array. (fb6.0+)

`ARRAY_INSERTRANGE   ( a1 @ a2 -- a' )`

Inserts items from array `a2` into `a1`, starting at the given index. Returns the resulting array. (fb6.0+)

`ARRAY_KEYS   ( a -- {@} )`

Returns the keys of an array in a stackrange. (fb6.0+)

`ARRAY_LAST   ( a -- @ i )`

Returns last index in array, and a boolean. Bool is false if no items are in array. (fb6.0+)

`ARRAY_MAKE   ( {?} -- a )`

Makes a list array from a stackrange. (fb6.0+)

`ARRAY_MAKE_DICT   ( {@ ?} -- a )`

Makes a dictionary associative array from a stackrange of index/value pairs. (fb6.0+)

`ARRAY_NDIFF   ( {a} -- a )`

Returns a list array, containing the difference of all the given arrays. Multiple arrays are consecutively processed against the results of the previous difference, from the top of the stack down. (fb6.0+)

`ARRAY_NINTERSECT   ( {a} -- a )`

Return a list array, containing the intersection of all the given arrays. Multiple arrays are consecutively processed against the results of the previous intersection, from the top of the stack down. (fb6.0+)

`ARRAY_NEXT   ( a @ -- @ i )`

Returns the next index in array, following index `a,` and a boolean. Bool is false if no items left. (fb6.0+)

`ARRAY_NOTIFY   (a1 a2 -- )`

Notifies all dbref items in array `a2` with all string items in array `a1.` (fb6.0+)

`ARRAY_NUNION   ({a} -- a)`

Returns a list array, containing the union of values of all the given arrays. (fb6.0+)

`ARRAY_PREV   ( a @ -- @ i )`

Returns previous index in array, following index `a,` and a boolean. Bool is false if no items left. (fb6.0+)

`ARRAY_SETITEM   ( ? a @ -- a' )`

Overwrites item a from array @ with value ?. (fb6.0+)

`ARRAY_SETRANGE   (a1 @ a2 -- a')`

Sets items in list `a1` to values from list `a2,` starting at the given index. Returns the resulting array. (fb6.0+)

`ARRAY_VALS   ( a -- {?} )`

Returns the values of an array in a stackrange. (fb6.0+)

`ASINE    ( f -- f' )`

Returns the inverse cosine of `f.` Operates in the range of `-pi/2` to `pi/2.` (fb6.0+)

`ATAN    ( f -- f' )`

Returns the inverse tangent of `f.` Operates in the range of `-pi/2` to `pi/2.` (fb6.0+)

`ATAN2  ( fy fx -- f )`

Equivalent to `ATAN,` but takes the signs of the arguments into account, and avoids division-by-zero errors. (fb6.0+)

`ATOI    ( s -- i )`

Turns string `s` into integer `i.` If `s` is not a string, then `0` is pushed onto the stack.

`AWAKE?    ( d -- i )`

Passed a player's dbref, returns the number of connections they have to the game. This will be `0` if they are not connected.

`BACKGROUND    ( -- )`

A way to turn on multitasking. Programs in the background let the program user go on and be able to do other things while waiting for the program to finish. You cannot use the `READ` command in a background program. Once a program is put into background mode, you cannot set it into foreground or preempt mode. A program will remain in the background until it finishes execution.

`BEGIN ( -- )`

Marks the beginning of begin-until or begin-repeat loops.

```
BG_MODE    ( -- i )
pr_mode
fg_mode
```

These are all standard built in defines. They are used with `MODE` and `SETMODE` to show what mode the program is running in, or to set what mode it will run in. For example, `MODE` returns an integer on the stack, that you can compare against `pr_mode,` `fg_mode,` or `bg_mode,` to determine what mode the program is in. `pr_mode` is defined as `0,` `fg_mode` is defined as `1,` and `bg_mode` is defined as `2.`

`BITOR    ( x x -- i )`

Does a mathematical bitwise or.

`BITAND    ( i i -- i )`

Does a mathematical bitwise and.

`BITSHIFT    ( i i -- i )`

Shifts the first integer by the second integer's number of bit positions. This works like the C `<<` operator. If the second integer is negative, it is equivalent to `>>.`

`BITXOR    ( i i -- i )`

Does a mathematical bitwise exclusive or.

`BREAK    ( -- )`

Breaks out of the innermost loop. Jumps execution to the instruction after the `UNTIL` or `REPEAT` for the current loop.

`CALL    ( d -- ?? )`

Calls another program `d. d` must have been compiled already. `d` will inherit the values of `ME, LOC, TRIGGER,` and all other variables.

`CALLER    ( -- d )`

Returns the dbref of the program that called this one, or the dbref of the trigger, if this wasn't called by a program.

`CEIL    ( f -- f )`

Returns the next highest integer number as a float. (fb6.0+)

`CHECKARGS    ( ??? s -- )`

Takes a string argument that contains an expression that is used to test the arguments on the stack below the given string. If they do not match what the expression says should be there, then it aborts the running program with an appropriate `Program Error` message. The expression is formed from single character argument tests that refer to different argument types. The tests are:

```
a - function address
d - dbref                         (#-1, #-2, #-3 are okay)
D - valid, non-garbage dbref      (#-1, #-2 NOT allowed #-3 is okay)
e - exit dbref                    (#-1, #-2 allowed)
E - exit dbref                    (#-1, #-2 NOT allowed)
f - program dbref                 (#-1, #-2 allowed)
F - program dbref                 (#-1, #-2 NOT allowed)
i - integer
p - player dbref                  (#-1, #-2 allowed)
P - player dbref                  (#-1, #-2 NOT allowed)
r - room dbref                    (#-1, #-2 allowed) (#-3 is a room)
R - room dbref                    (#-1, #-2 NOT allowed) (#-3 is a room)
s - string
S - non-null string
t - thing dbref                   (#-1, #-2 allowed)
T - thing dbref                   (#-1, #-2 NOT allowed)
v - local or global variable
? - any stack item type
```

Tests can be repeated multiple times by following the test with a number. For example, `"i12" checkargs` would test the stack for 12 integers.

The last test in the string expression will be done on the top stack item. Tests are done from the top of the stack down, in order, so the last test that fails in a string expression will be the one that the Program Error will be given for. ie: `"sdSi" checkargs` will test that the top stack item is an integer, then it tests that the next item down is a non-null string, then it tests the third item from the top to see if it is a dbref, and lastly it tests to make sure that the 4th item from the top is a string.

Spaces are ignored, so `s d i` is the same as `sdi.` However, multipliers are ignored if they follow a space, so `s 4d i` is also the same as `sdi.` This is because you are basically telling it to repeat the space 4 times, and since spaces are ignored, it has no effect.

If you have a function that takes a stack item of any type, you can use the `?` test. `?` will match a string, integer, dbref, or any other type.

Since sometimes arguments are passed in ranges, such as the way that the explode primitive returns multiple strings with an integer count on top, there is a way to group arguments, to show that you expect to recieve a range of that type. For example, `"{s}" checkargs` would test the stack for a set of strings like `"first" "second" "third" "fourth" 4` where the top stack item tells how many strings to expect within the range.

Sometimes a function takes a range of paired arguments, such as: `"one" 1 "two" 2 "three" 3 "four" 4 4` where the count on the top of the range refers to the number of pairs. To test for the range given above, you would use `"{si}" checkargs` to tell it that you want to check for a range of paired strings and integers. You can group as many argument tests together in a range as you would like. ie: you could use `{sida}` as an expression to test for a range of related strings, integers, dbrefs, and function addresses.

Since the argument multipliers refer to the previous test `OR` range, you can test for two string ranges with the test `"{s}2" checkargs.` ie: It would succeed on a stack of: `"one" "two" "three" 3 "four" "five" 2. "{s2}" checkargs,` however, would test for one range of paired strings. ie: It would succeed with a stack of: `"one" "1" "two" "2" "three" "3" 3.`

If, for some reason, you need to pass a range of ranges to a function, you can test for it by nesting the braces. ie: `"{{s}}" checkargs`.

Now, as one last example, the primitive `notify_exclude,` if we were to test the arguments passed to it manually, would use the test `"R{p}s" checkargs` to test for a valid room dbref, a range of player dbrefs or `#-1s,` and a string.

`CHECKPASSWORD    ( d s -- )`

Returns true if the password of player `d.` (wizbit only)

`CLEAR    ( -- )`

Clears all error flags. See also `CLEAR_ERROR`, `ERROR?`, `ERROR_STR`, `ERROR_NAME`, `ERROR_BIT`, `ERROR_NUM`, `IS_SET?`, and `SET_ERROR`   (fb6.0+)

`CLEAR_ERROR    ( i -- )`

Clears error flag `i`. See also `CLEAR`, `ERROR?`, `ERROR_STR`, `ERROR_NAME`, `ERROR_BIT`, `ERROR_NUM`, `IS_SET?`, and `SET_ERROR` (fb6.0+)

`CONBOOT    ( i -- )`

Takes a connection number and disconnects that connection from the server. (wizbit only)

`CONCOUNT    ( -- i )`

Returns how many connections to the server there are. (Requires Mucker Level 3)

`CONDBREF    ( i -- d )`

Returns the dbref of the player connected to this connection. (Requires Mucker Level 3)

`CONDESCR   ( i -- i )`

Takes a connection number and returns the descriptor number associated with it. (Requires Mucker Level 3) See `DESCRIPTORS,` `DESCRCON`.

`CONHOST    ( i -- s )`

Returns the hostname of the connection. (wizbit only)

`CONIDLE    ( i -- i )`

Returns how many seconds the connection has been idle. (Requires Mucker Level 3)

`CONNOTIFY   ( i s -- )`

Sends a string to a specific connection to the server. (Requires Mucker Level 3)

`CONTENTS   ( d -- d' )`

Pushes the dbref of the first thing contained by `d.` This dbref can then be referenced by `next` to cycle through all of the contents of `d. d` may be a room or a player.

`CONTIME    ( i -- i )`

Returns how many seconds the given connection has been connected to the server. (Requires Mucker Level 3)

`CONTINUE    ( -- )`

Jumps execution to the beginning of the current loop.

`COPYOBJ    ( d -- d' )`

Creates a new object (returning `d'` on top of the stack), that is a copy of object `d.` Each program is allowed to create only one new object per run.

`COS    ( f -- f' )`

Returns the cosine of `f.` Operates in the range of -pi/4 and pi/4. (fb6.0+)

`CTOI ( s -- i )`

Converts the first character in a string to its `ASCII` equivalent integer. (fb6.0+)

`DATE    ( -- i i i )`

Returns the monthday, month, and year. For example, if it were February 6, 1992, `date` would return `6 2 1992` as three integers on the stack.

`DBCMP    ( d1 d2 -- i )`

Performs comparison of database objects `d1` and `d2.` If they are the same object, then `i` is `1,` otherwise i is `0.`

`DBREF   ( i -- d )`

Converts integer `i` to object reference `d.`

`DBREF?    ( x -- i )`

Returns true if `x` is a dbref.

`DBTOP    ( -- d)`

Returns the dbref of the first object beyond the top object of the database. `dbtop ok?` would return a false value.

`DEPTH    ( -- i )`

Returns the number of items on the stack.

`DESC    ( d -- s )`

Takes object `d` and returns its description (`@desc`) string field.

`DESCRCON    ( i -- i )`

Takes a descriptor and returns the associated connection number, or `0` if no match was found. See `DESCRIPTORS, CONDESCR.`

`DESCRFLUSH    ( i -- )`

Flushes output text on the given descriptor. If `-1` is passed as the descriptor, it flushes output on all connections.

`DESCRIPTORS   ( d -- ix...i1 i )`

Takes a player dbref, or `#-1,` and returns the range of descriptor numbers associated with that dbref (or all for `#-1`) with their count on top. Descriptors are numbers that always stay the same for a connection, while a connection number is the relative position in the `WHO` list of a connection. See `DESCRCON, CONDESCR.`

DICTIONARY? (? -- i) Returns true if the top item on the stack is a dictionary array. (fb6.0+)

`DIST3D   ( fx fy fz -- f )`

Returns the distance of `XYZ` coords `(fx, fy, fz)` from the origin. (fb6.0+)

`DROP    ( d -- s )`

Takes object `d` and returns its drop (`@drop`) string field.

`DUP    ( x -- x x )`

Duplicates the item at the top of the stack.

`DUPN    ( ?n...?1 i -- ?n...?1 ?n...?1 )`

Duplicates the top `N` stack items. See also `LDUP.`

`ELSE` (See `IF`)

`ENVPROPSTR ( s d -- s d )`

Takes a starting object dbref and a property name and searches down the environment tree from that object for a property with the given name. If the property isn't found, it returns `#-1` and a null string. If the property is found, it will return the dbref of the object it was found on, and the string value it contained.

`ERROR? ( -- i )`

Returns true if any error flags have been set. See also `CLEAR`, `CLEAR_ERROR`, `ERROR_STR`, `ERROR_NAME`, `ERROR_BIT`, `ERROR_NUM`, `IS_SET?`, and `SET_ERROR` (fb6.0+)

`ERROR_BIT ( s -- i )`

Given the error flag string name, returns the bit identifier. See also `CLEAR`, `CLEAR_ERROR`, `ERROR?`, `ERROR_STR`, `ERROR_NAME`, `ERROR_NUM`, `IS_SET?`, and `SET_ERROR` (fb6.0+)

`ERROR_NAME ( i -- s )`

Given the error flag bit identifier, returns the string name for the error flag. See also `CLEAR`, `CLEAR_ERROR`, `ERROR?`, `ERROR_STR`, `ERROR_BIT`, `ERROR_NUM`, `IS_SET?`, and `SET_ERROR` (fb6.0+)

`ERROR_NUM ( -- i )`

Returns the total number of error flag types.

```
DIV_ZERO   (0)  Division by zero attempted
NAN        (1)  Result was not a number
IMAGINARY  (2)  Result would be imaginary
FBOUNDS    (3)  Floating-point inputs were out of range
IBOUNDS    (4)  Calculation resulted in an integer overflow or underflow
```

See also `CLEAR`, `CLEAR_ERROR`, `ERROR?`, `ERROR_STR`, `ERROR_NAME`, `ERROR_BIT`, `IS_SET?`, and `SET_ERROR` (fb6.0+)

`ERROR_STR ( i -- s )`

Given an error identifier, returns a user-readable error string. See also `CLEAR`, `CLEAR_ERROR`, `ERROR?`, `ERROR_NAME`, `ERROR_BIT`, `ERROR_NUM`, `IS_SET?`, and `SET_ERROR` (fb6.0+)

`EXECUTE ( a -- ?? )`

Executes the function pointed to by the address a on the stack.

`EXIT ( -- )`

Exits from the word currently being executed, returning control to the calling word, at the statement immediately after the invokation of the call (exiting the program if applicable).

`EXIT? ( d -- i )`

Returns `1` if object `d` is an exit object, `0` if otherwise. See also `player?, program?, room?, thing?, ok?.`

`EXITS    ( d -- d' )`

Returns the first exit in the linked exit list of room/player/object `d.` This list can be transversed with `next.`

`EXP    ( f -- f' )`

Returns `e` raised to the `fth` power. (fb6.0+)

`EXPLODE    ( s1 s2 -- ... i )`

`s2` is the delimiter string, and `s1` is the target string, which will be fragmented, with `i` pushed on top of the stack as the number of strings `s1` was broken into. For instance:

```
   "Hello world" " " explode
```

will result in

```
   "world" "Hello" 2
```

on the stack. (Note that if you read these items off in order, they will come out `"Hello"` first, then `"world"`.) For TinyMUCK 2.2, `s2` may be any length. But `""` (null string) is not an acceptable string for parameter `s2.`

`FABS    ( f -- f' )`

Returns the absolute value of `f.` (fb6.0+)

`FAIL    ( d -- s )`

Takes object `d` and returns its fail (`@fail`) string field.

```
FG_MODE    ( -- i )
pr_mode
bg_mode
```

These are all standard built in defines. They are used with `MODE` and `SETMODE` to show what mode the program is running in, or to set what mode it will run in. For example, `MODE` returns an integer on the stack, that you can compare against `pr_mode,` `fg_mode,` or `bg_mode,` to determine what mode the program is in. `pr_mode` is defined as `0,` `fg_mode` is defined as `1,` and `bg_mode` is defined as `2.`

`FLAG?    ( d s -- i )`

Reads the flag of object `d,` specified by `s,` and returns its state: `1 = on`; `0 = off.` Different flags may be supported in different installations. `flag?` returns `0` for unsupported or unrecognized flags. You can check the `"interactive"` flag to see if a player is currently in a program's `READ,` or if they are in the MUF editor.

`FLOAT    ( i -- f )`

Converts `i` to a floating point. (fb6.0+)

`FLOAT?    ( x -- i )`

Returns true if `x` is a floating point. (fb6.0+)

`FLOOR    ( f -- f )`

Returns the next lowest integer number as a float. (fb6.0+)

`FOREACH   ( a -- @ ? )`

A loop construct for arrays. (fb6.0+)

`FMTSTRING`

`FMTSTRING` can be used to format complicated and long strings, as well as multi-lined (with embedded new-lines) strings. These strings can consist entirely of user-specified text, formatted variable entries (as values taken from the stack) or a combination of both.

The syntax for the format string is as follows:

```
Format -+---+-%[-,|][+, ][0][number][.number]type->+----->
        ^ | |
        | +-text-------------------------------->|
        | |
        +----------------------------------------+
```

Here, text can be any string that does not contain a `%` percent mark. This can be circumvented by replacing any occurance of a `%` with `%%. Number` is any standard integer number, and type should be one of the following single character identifiers (case is important):

- `i - integer argument`
- `s - string argument`
- `? - unknown type argument, will print a string stating what the variable type is`
- `d - dbref number, in the form of #123`
- `D - dbref name reference; given a dbref, will print the associated name for that object - terminates on bad reference`
- `l - pretty-lock, given a lock, will print the description`
- `f - float in xxx.yyy form`
- `e - float in x.yyEzz form`
- `g - shorter of forms e or f`

A sample format string might look something like:

```
"There are %i apples in the box."
```

If there was an integer value of '5' on the stack, this would return:

```
"There are 5 apples in the box."
```

Variables can be formatted in very specific manners. For numeric entries, zero padding can be specified, as well as control over the sign field. The total output field size for the variable can be specified, as well as the minimum and maximum output lengths. Within a field, output can be left, right or center justified. The `[-,|]` denotes the justification style to use. With no argument, a field will be right justified. A `-` will change this to left and a `|` will make the field centered. If justification is to be used, the first numeric entry must be specified. This denotes the total size of the variable field. If the output of the variable is not limited by the maximum output number (see below), and the total size of the output is greater than the given field width, justification will have no meaning. The second number field (after the period) specifies the maximum number of characters to be printed for string variables, or the precision for numerics. When dealing with numerics, the `+, ' '` and `'0'` characters also can be used before the numeric fields. Specifing `+` will make sure that a sign character is always printed before the number, wether it be a plus or a minus. Default is to only print a minus sign as needed. If a `' '` is used instead of the `+,` a positive number will be prefaced with a space instead of a plus-sign. When a `0` is given, zeros will be prefaced to the printed number in order to fill the field width.

Tabbing and new-lines are allowed within the format strings as well. Tabs are defaulted to the equivilent of eight spaces. Starting a new line with `\r` will reset the tab count. Unless explicitly reset with a new-line character, tab will continue to count in mod-eight across the given output line. To insert a tab character, use `\t.`

Variables should be on the stack 'under' the format string, and should be in order of reference. The first variable referenced by the format string should be the next item in the stack, and so forth. (fb6.0+)

`FOR    ( i1 i2 i3 -- i )`

A loop structure, iterating from `i1` to `i2,` with a step of `i3.` The current count is pushed onto the stack at each iteration. `FOR` is used just like `BEGIN,` marking the beginning of a loop, with `REPEAT` or `UNTIL` marking the end. However unlike other loops, `FOR` loops are limited to a ridiculously high nesting of 512, enforced at runtime. The `FOR` primitive is actually `( i1 i2 i3 -- )` and the `FORITER` primitive, used to control the iteration count is `( -- [i] i1)` where `i1` is a boolean signaling weather or not to continue, and if continuing, `i` is the currnet count. The primitive `FORPOP ( -- )` cleans up the `FOR` nesting stack by popping off the data for the nested loop.

`FORCE    ( d s -- )`

Forces player `d` to do action s as if they were `@forced.` (wizbit only)

`FOREGROUND    ( -- )`

To turn on multitasking, you can issue a foreground command. While a program is in foreground mode, the server will be multitasking and handling multiple programs at once, and input from other users, but it will be blocking any input from the user of the program until the program finishes. You cannot foreground a program once it is running in the background. A program will stay in foreground mode until it finishes running or until you change the mode.

`FORK    ( -- i )`

This primitive forks off a `BACKGROUND` (muf) process from the currently running program. It returns the pid of the child process to the parent process, and returns a `0` to the child. If the timequeue was full, then it returns a `-1` to the parent process, and there is no child process. (Requires Mucker Level 3)

`FRAND   ( -- f )`

Returns a random number between `0` and `1.` (fb6.0+)

`FTOSTR    ( f -- s )`

Converts a floating point value to a string. (fb6.0+)

`GETLINK    ( d -- d' )`

Returns what object `d` is linked to, or `#-1` if `d` is unlinked. The interpretation of link depends on the type of `d:` for an exit, returns the room, player, program, action, or thing that the exit is linked to. For a player, program, or thing, it returns its 'home', and for rooms returns the drop-to.

`GETLINKS    ( d -- dn..d1 n )`

Returns information on metalinks. Returns `0` when the `obj` is an unlinked exit, or if the `obj` is a program. Returns `0` if the `obj` is a room with no `dropto.` Returns `#-3` and a count of `1` if the dropto is linked to `HOME.` (fb6.0+)

`GETLOCKSTR    ( d -- s )`

Returns the lock expression for the given object in the form of a string. Returns `*UNLOCKED*` if the object doesn't have a lock set.

`GETPROPSTR    ( d s -- s' )`

Retrieves string associated with property `s` on object `d.` The value stored in s must be a string. If the property is cleared, `""` (null string) is returned.

`GETPROPFVAL    ( d s -- f )`

Retrieves the floating point associated with property `s` on object `d.` The value stored in s must be a float. If the property is cleared, `<<<?>>>` is returned.

`GETPROPVAL    ( d s -- i )`

`s` must be a string. Retrieves the integer value `i` associated with property `s` in object `d.` If the property is cleared, `0` is returned.

`GETSEED   ( -- s )`

Returns the the current `SRAND` seed string. See also `SETSEED` and `SRAND` (fb6.0+)

`GMTOFFSET    ( -- i )`

Returns the machine's offset from GMT in seconds.

`IF ... [ else ... ] then    ( x -- )`

Examines boolean value `x.` If `x` is `TRUE,` the sequence of statements after the `if` up until the `then` (or until the `else` if it is present) performed. If it is *fails*, then these statements are skipped, and if an `else` is present, the statements between the `else` and the `then` are performed. Control continues as usual at the statement after the `then.` Note that checking the top of the stack actually pops it, so if you want to re-use it, you should dup (see `DUP`) it before the if. For every `IF` in a word, there *must* be a *then*, and vice-versa. `ELSE` is optional.

`INF    ( -- f )`

Returns an infinite result. (fb6.0+)

`INSTR    ( s s1 -- i )`

Returns the first occurrence of string `s1` in string `s,` or `0` if `s1` is not found. See also `RINSTR.`

`INSTRING    ( s s1 -- i )`

Returns the first occurrence of string `s1` in string `s,` or `0` if `s1` is not found. Non-case sensitive. See also `RINSTRING,` `INSTR,` and `RINSTR`. This is an inserver define to `tolower swap tolower swap instr`.

`INT    ( x -- i )`

Converts variable or object `x` to integer `i.`

`INT?    ( x -- i )`

Returns true if `x` is an integer.

`INTERP    ( d d2 s -- ?)`

Takes a program dbref to run, the trigger to use, and the top stack item string. It runs the `MUF` program with the given trigger (with the given string on top of the stack) and returns the top stack item that the `MUF` program exits with.

`INTOSTR    ( x -- s )`

`x` must be an integer or a dbref. Converts `x` into string `s.`

`IS_SET?    ( i -- i )`

Returns true if a specific error flag is set. See also `CLEAR`, `CLEAR_ERROR`, `ERROR?`, `ERROR_STR`, `ERROR_NAME`, `ERROR_BIT`, `ERROR_NUM`, and `SET_ERROR` (fb6.0+)

`ISPID?    ( i -- i)`

Takes a process id and checks to see if an event with that pid is in the timequeue. It returns `1` if it is, and `0` if it is not. *NOTE*: since the program that is running is not on the timequeue *while* it is executing, but only when it is swapped out letting other programs run, `pid ispid?` will always return `0.`

`ITOC    ( i -- s )`

Converts an integer to its ASCII equivalent character. If it is not a valid display character, a null string is returned. (fb6.0+)

`JMP    ( a -- )`

The JMP primitive takes an address like those supplied by 'funcname and moves execution to that point. It is one early way that was used to do tail-recursion loops without as much overhead, and without failing due to system stack overflows. It is mostly obsolete now, except that it's one of the three or four internal primitives used to implement if-else-then and begin-while-repeat loops and such.

Example of JMP as a tail-recursion optimization:

```
: countforever ( i -- )
  1 +
  dup intostr .tell
  'countforever jmp
;
```

A better ways to do the same thing with looping primitives would be:

```
: countforever ( i -- )
  begin
    1 +
    dup intostr .tell
  repeat
```

```
   ;
```

`KILL    ( i -- i )`

Attempts to kill the given process number. Returns `1` if the process existed, and `0` if it didn't. Any process can kill itself; killing other processes equires Mucker Level 3.

`LDUP    ( {?} -- {?} {?} )`

Duplicates a stackrange on top of the stack. See also `DUPN.`

`LOCALVAR    ( i -- l )`

Takes an integer and returns the respective local variable. Similar to the `VARIABLE` primitive.

`LOCATION    ( d -- d' )`

Returns location of object `d` as object `d'.`

`LOCK?    ( ? -- i )`

Returns true if the top stack item is a lock. See also `GETPROP,` `SETPROP,` `PARSELOCK,` `UNPARSELOCK,` `PRETTYLOCK,` and `TESTLOCK.`

`LOCKED?    ( d d -- i )`

Takes, in order, the dbref of the object to test the lock on, and the dbref of the player to test the lock against. It tests the lock, running programs as necessary, and returns a integer of `0` if it is not locked against her, or `1` if it is.

`LOG    ( f -- f' )`

Returns the natural log of `f.` `f` must be greater than zero. Very small values will return INF.

`LOG10    ( f -- f' )`

Returns the log base 10 of `f.` `f` must be greater than zero. Very small values will return INF.

`LREVERSE    ( ?n...?1 i -- ?1...?n i )`

Reverses top `N` items, leaving count. See also `REVERSE.`

`LVAR <varname>`

This declares a variable as a local variable, that is local to a specific program. If another program calls this program, the values of the local variables will not be changed in the calling program, even if the called program changes them.

`MATCH    ( s -- d )`

Takes string `s,` first checks all objects in the user's inventory, then checks all objects in the current room, as well as all exits that the player may use, and returns object `d` which contains string `s.` If nothing is found, `d = #-1.` If ambiguous, `d = #-2.` If HOME, `d = #-3.`

`MIDSTR    ( s1 i1 i2 - s2 )`

Returns the sub-string specified by positions `i1` to `i2` (inclusive) from within string `s1.`

`MLEVEL    ( d -- i )`

Returns the Mucker (or Priority) Level of the given object.

`MODE    ( -- i )`

Returns an integer denoting the current multitasking mode. This ignores `BOUND` bits on programs. The integer this returns will be the same as one of those defined by the standard `$defines` `bg_mode,` `fg_mode,` and `pr_mode,` being `background,` `foreground,` and `preempt` mode, respectively. Also see `PR_MODE.`

`MODF    ( f -- f f )`

Returns the integral and fractional parts of `f,` both as floating point numbers. Ex:

```
   1.5 MODF
```

would put

```
   1.0 0.5
```

on the stack.

`MOVEPENNIES   ( d1 d2 i -- )`

Moves `i` pennies from player/thing `d1` to player or thing `d2.` (fb6.0+)

`MOVETO    ( d1 d2 -- )`

Moves object `d1` to object `d2.` `MOVETO` is affected by the following rules:

1. If the object being moved is `!Jump_OK` and is it being moved by someone other than the object's owner, then the moveto fails.
2. If the object being moved is a person and either the source or destination rooms (if not owned by the person being moved) are `!Jump_OK,` the moveto fails.
3. If the object being moved is not a player, is owned by the owner of either the source or destination rooms, and either room where the ownership matches is `!Jump_OK,` the moveto fails.

The `MOVETO` succeeds under any other circumstances. `MOVETO` rules follow the permissions of the current effective userid. `MOVETO` will run programs in the `@desc` and `@succ/@fail` of a room when moving a player. If the object to be moved is an exit, the program must be `M3.`

`NAME    ( d -- s )`

Takes object `d` and returns its name (`@name`) string field.

`NEWEXIT    ( d s -- d )`

Takes a location and name and returns the new exit's dbref. Owner is the person running the program. (wizbit only)

`NEWOBJECT    ( d s -- d )`

Takes a location and name and returns the new thing's dbref. Owner is the person running the program. (wizbit only)

`NEWROOM    ( d s -- d )`

Takes the dbref of the parent and the name of the room. It returns the dbref of the created room. Owner is the person running the program. (wizbit only)

`NEXT    ( d -- d' )`

Takes object `d` and returns the next thing in the linked contents/exits list of `d's` location.

`NEXTDESCR    ( i -- i )`

Takes a descriptor number, and returns the next connected descriptor number. To get the first descriptor number, use `1 condescr.` Between these, you can step through the descriptors list. If you try to use nextdescr on an invalid descriptor, it will return `0.` If you have reached the end of the descriptor list, it returns `0.` (requires Mucker Level 3.)

`NEXTOWNED    ( d -- d )`

Returns the dbref of the first object owned by player `d.` When called this object's dbref, `NEXTOWNED` returns the next object owned by the same player. When there are no more objects left owned by that player, then `#-1` is returned. The order of the objects is not guarenteed, but when used correctly, each object owned by that player will be returned exactly once. The player object itself will *not* be returned. This is used similarly to the `NEXT` primitive. Ex:

```
   me @ begin dup while dup unparseobj .tell nextowned repeat
```

(fb6.0+)

`NEXTPROP    ( d s -- s )`

This takes a dbref and a string that is the name of a property and returns the next property name on that dbref, or returns a null string if that was the last. To *start* the search, give it a propdir, or a blank string. For example,

```
   #10 "/" NEXTPROP
```

or

```
   #28 "/letters/" NEXTPROP
```

A blank string is the same as `"/".` (Requires Mucker Level 3) `NEXTPROP` will skip properties if they would not be readable by the program with the given permissions and effective user id.

`NOT    ( x -- i )`

Performs the boolean 'not' operation on `x,` returning `i` as `1` if `x` is `FALSE,` and returning `i` as `0` otherwise.

`NOTIFY    ( d s -- )`

`d` must be a player object. `s` must be a string. Tells player `d` message `s.` If s is null it will print nothing. This primitive will trigger the `_listen`'er property on the object the message is sent to, unless the program that would be run is the same as one one currently running.

`NOTIFY_EXCEPT    ( d1 d2 s -- )`

`d1` must be a room object, s must be a string. Tells everyone at location `d1` except object `d2` message `s.` If object `d2` is not a player or `NOTHING` (`#-1`) all players are notified. If `s` is null it prints nothing. *Note*: `notify_except` is now only an inserver `$define.` It is translated to `1 swap notify_exclude.` See also NOTIFY_EXCLUDE

`NOTIFY_EXCLUDE    ( d dn ... d1 n s -- )`

Displays the message `s` to all the players (or `_listening` objects), excluding the `n` given players, in the given room. For example:

```
#0 #1 #23 #7 3 "Hi!" notify_exclude
```

would send `"Hi!"` to everyone in room `#0` except for players (or objects) `#1, #7,` and `#23.` `_listener's` will not be triggered by a `notify_exclude` if the program they would run is the same as the current program running.

`NUMBER?    ( s -- i )`

Returns `1` if string on top of the stack contains a number. Otherwise returns `0.`

`ODROP    ( d -- s )`

Takes object `d` and returns its odrop (`@odrop`) string field.

`OFAIL    ( d -- s )`

Takes object `d` and returns its ofail (`@ofail`) string field.

`OK?    ( x -- i )`

Takes `x` and returns `1` if `x` is a type dbref, as well as `0` or above, below the top of the database, and is not an object of type garbage. See also `EXIT?, PLAYER?, PROGRAM?,` and `THING?.`

`ONLINE    ( -- d ... i )`

Returns a dbref for every connection to the game, and lastly the number of connections.

`OR    ( x y -- i )`

Performs the boolean 'or' operation on `x` and `y.` Returns `i` as `1` if either `x` or `y` is `TRUE,` returns `i` as `0` otherwise.

`OSUCC    ( d -- s )`

Takes object `d` and returns its osuccess (`@osucc`) string field.

`OVER    ( x y -- x y x )`

Duplicates the second-to-top thing on the stack. This is the same as `2` pick.

`OWNER    ( d -- d' )`

`d` is any database object. Returns `d',` the player object that owns `d.` If `d` is a player, `d'` will be the same as `d.`

`PARSELOCK    ( s -- l )`

Parses a lock string into a lock. If the parsing failed, then the lock returned will be a true_boolexp, Which is logically false to an 'if' test. See also `UNPARSELOCK, LOCK?, PRETTYLOCK, TESTLOCK, GETLOCKSTR, SETLOCKSTR,` and `LOCKED?.`

`PARSEPROP    ( d s s i -- s )`

Returns the string output of the MPI parser, given an object, a property name to parse, an input string for the `{&cmd}` variable, and an integer that should either be `1,` for when you want `{delay}` messages to be sent to the player only, and `0,` when you want the rest of the players in the room to get the omessages. Note: for security reasons, you cannot use parseprop on an object you don't control, if the property is not a `_prop` or a `~prop.` The exception to this is if the `MUF` program is at least Mucker Level 3. Then parsing of normal props is allowed. If the `MUF` program is wizbit, it can also parse `@props` and `.props.`

`PART_PMATCH    ( s -- d )`

Takes a player name, or the first part of the name, and matches it against the names of the players who are currently online. If the given string is a prefix of the name of a player who is online, then their dbref is returned. If two players could be matched by the given string, it returns a `#-2.` If none of the players online match, then it returns a `#-1.`

`PENNIES    ( d -- i )`

Gets the amount of pennies player object `d` has, or the penny value of thing `d.`

`pi    ( -- f )`

Returns pi.

`PICK    ( ni ... n1 i -- ni ... n1 ni )`

Takes the `i`'th thing from the top of the stack and pushes it on the top. `1 pick` is equivalent to `dup,` and `2 pick` is equivalent to `over.`

`PLAYER?    ( d -- i )`

Returns `1` if object `d` is a player object, otherwise returns `0.` If the dbref is that of an invalid object, it will return `0.` see also `PROGRAM?, ROOM?, THING?, EXIT?, OK?.`

`POLAR_TO_XYZ    ( fr ft fp -- fx fy fz )`

This converts the spherical polar coordinate `(fr, ft, fp)` to the `XYZ` coord `(fx, fy, fz).` (fb6.0+)

`PMATCH    ( s -- d )`

Returns the dbref of the player with name `s.` (fb6.0+)

`POP    ( x -- )`

Pops the top of the stack into oblivion.

`POPN    ( ?n...?1 i -- )`

Pops the top `n` stack items.

`POW    ( f f' - f'' )`

Returns `f` raised to the `f`'th power. If `f` is zero, `f'` must be greater than zero. If `f` is less than zero, `f'` must be an integer value.

`PR_MODE    ( -- i )`
fg_mode
bg_mode

These are all standard built in defines. They are used with mode and setmode to show what mode the program is running in, or to set what mode it will run in. For example, mode returns an integer on the stack, that you can compare against `PR_MODE`, `FG_MODE`, or `BG_MODE`, to determine what mode the program is in. `PR_MODE` is defined as `0`, `FG_MODE` is defined as `1`, and `BG_MODE` is defined as `2`.

`PREEMPT    ( -- )`

Prevents a program from being swapped out to do multitasking. Needed in some cases to protect crutial data from being changed while it is being worked on. A program will remain in preempt mode until its execution is completed. Basically what this command does is to turn off multitasking, but then you have a limit on how many instructions you can run without needing either to pause with a sleep, or have a wizbit on the program.

`PRETTYLOCK    ( l -- s )`

Unparses a lock into a string fit for players to see. Also see `LOCK?`, `PARSELOCK`, `UNPARSELOCK`, `TESTLOCK`, `GETLOCKSTR`, `SETLOCKSTR`, and `LOCKED?`.

`PROG    ( -- d )`

Returns the dbref of the currently running program.

`PROGRAM?    ( d -- i )`

Returns `1` if object `d` is a program, otherwise returns `0`. If the dbref is that of an invalid object, it will return `0`. See also `PLAYER?`, `ROOM?`, `THING?`, `EXIT?`, `OK?`.

`PRONOUN_SUB    ( d s -- s' )`

Takes database object `d` and substitutes string `s` according to `o-message` rules. For example:

```
   me @ "%n has lost %p marbles." pronoun_sub
```

would return:

```
   "igor has lost his marbles."
```

If the player's name was Igor and his sex were male. `d` does not have to be a player for the substitutions to work. The substitutions are:

```
   %a/%a for absolute possessive (his/hers/its, his/hers/its)
   %s/%s for subjective pronouns (he/she/it, he/she/it)
   %o/%o for objective pronouns (him/her/it, him/her/it)
   %p/%p for possessive pronouns (his/her/its, his/her/its)
   %r/%r for reflexive pronouns (himself/herself/itself,
   %n/%n for the player's name. (himself/herself/itself)
```

`PROPDIR?    ( d s -- i )`

Takes a dbref and a property name, and returns whether that property is a propdir that contains other props. (Requires mucker level 3)

`PUBLIC <functionname>`

Declares a previously defined function to be public for execution by other programs. This is a compile-time directive, not a run-time primitive. To call a public function, put the dbref of the program on the stack, then put a string, containing the function name, on the stack, then use `CALL`.

For example:

```
#888 "functionname" call
```

`PUT    ( nx...n1 ni i -- nx...ni...n1 )`

Replaces the `i`'th thing from the top of the stack with the value of `ni.` `1 put` is equivalent to `swap pop`

Example:

```
"a" "b" "c" "d" "e" 3 put
```

would return on the stack:

```
"a", "e", "c", "d"
```

`QUEUE    ( i d s -- i )`

Takes a time in seconds, a program's dbref, and a parameter string. It will execute the given program with the given string as the only string on the stack, after a delay of the given number of second. Returns the pid of the queued process, or `0` if the timequeue was full. (Requires Mucker Level 3)

`RANDOM    ( -- i )`

Returns a random integer from `0` to the `MAXINT` of the system running the MUCK. In general this number is `(2^31)-1` or `2,147,483,647` (2.1 billion).

`READ    ( -- s )`

Reads a string `s` from the user. This command should not be used in a program that is locked (as opposed to linked) to an object, as the lock will always fail and print the fail messages at read time. It cannot be used in a program associated with a room object.

`RECYCLE    ( d -- )`

Recycles the given object `d.` Will not recycle players, the global environment, the player starting room, or any currently running program. (Can recycle objets owned by uid if running with Mucker Level 3 permissions. Can recycle other people's items with wizbit)

`REMOVE_PROP    ( d s -- )`

Removes property `s` from object `d.` If the property begins with an underscore, `'_'` or a dot `'.',` and the effective user does not have permission on that object, the call fails.

(*Note*: There is a slight bug with `REMOVE_PROP` in versions `fb5.46` and earlier. If you use the primitive to remove a prop, and then later in the same process try to remove a prop whose name contains that of the earlier prop, the second property will not be removed. For example:

```
loc @ "banned_lock" remove_prop
loc @ "banned_lock/program" remove_prop
```

If these two lines were executed in this order, no error would be issued, but property `banned_lock/program` would not be removed. If the order of the two lines is reversed, both properties will be removed.)

`REPEAT    ( -- )`

Jumps execution to the instruction after the `BEGIN` in a `BEGIN-REPEAT` loop. Marks the end of the current loop.

`REVERSE    ( ?n...?1 i -- ?1...?n i )`

Reverses the order of the top `i` items on the stack, returning the number of items reversed. See also `LREVERSE.`

`RINSTR    ( s s1 -- i )`

Returns the last occurrence of string `s1` in string `s,` or `0` if `s1` is not found. `"abcbcba" "bc" rinstr` returns `4.` See also `INSTR.`

`RINSTRING    ( s s1 -- i )`

Returns the last occurrence of string `s1` in string `s,` or `-1` if `s1` is not found. Non-case sensitive. See also `INSTRING,` `INSTR,` and `RINSTR.` This is an inserver define to `tolower swap tolower swap rinstr`.

`RMATCH    ( d s -- d' )`

Takes string `s,` checks all objects and actions associated with object `d,` and returns object `d'` which matches that string. For example, matches actions and inventory objects for a player object, actions on a thing object, etc. If nothing is found, `d'= #-1.` If ambiguous, `d' = #-2.` If HOME, `d' = #-3.`

`ROOM?    ( d -- i )`

Returns `1` if object `d` is a room, otherwise returns `0.` If the dbref is that of an invalid object, it will return `0`.

`ROT    ( x y z -- y z x )`

Rotates the top three things on the stack. This is equivalent to `3 rotate`.

`ROTATE    ( ni ... n1 i -- n(i-1) ... n1 ni )`

Rotates the top `i` things on the stack.

`ROUND    ( f -- f' )`

Returns `f,` rounded to the nearest whole number, as a floating point.

`SET    ( d s -- )`

Sets flag `s` to object `d.` Currently settable things are: `Abode, Chown, Dark, Haven, Jump, Link,` and `Sticky.` Boolean operations (e.g. '!abode') work as expected. See also `SETNAME,` `SETDESC,` and `FLAG?`.

`SET_ERROR    ( i -- )`

Sets a specific error flag to `ON.` (fb6.0+)

`SETDESC SETSUCC SETFAIL SETDROP SETOSUCC SETOFAIL SETODROP    (d s -- )`

Takes object `d,` and sets the string field specified to `s.` A program may only set string fields of objects that are owned by the effective user of the program, or any object if the program is `Wizard.` These are all actually `$defines` to `addprop` with the apprpriate property name. They are effectively defined as:

```
  $define setdesc    "_/de"    swap 0 addprop $enddef
  $define setsucc    "_/sc"    swap 0 addprop $enddef
  $define setfail    "_/fl"    swap 0 addprop $enddef
  $define setdrop    "_/dr"    swap 0 addprop $enddef
  $define setosucc   "_/osc"   swap 0 addprop $enddef
  $define setofail   "_/ofl"   swap 0 addprop $enddef
  $define setodrop   "_/odr"   swap 0 addprop $enddef
```
See also SET, SETNAME, ADDPROP, GETPROPSTR, REMOVE_PROP, DESC, SUCC, FAIL, DROP, OSUCC, OFAIL, and ODROP.

`SETLINK    ( d1 d2 -- )`

Takes an exit dbref `d1,` and sets its destination to `d2.` You must have control of the exit, and if the exit is already linked, it must be unlinked first by doing setlink with `#-1` as the destination. Wizbitted programs can `SETLINK` regardless of whether the exit is already linked or who controls it.

`SETLOCKSTR    (d s -- i)`

Tries to set the lock on the given object to the lock expression given in the string. If it was a success, then it will return a `1,` otherwise, if the lock expression was bad, it returns a `0.` To unlock an object, set its lock to a null string.

`SETMODE    ( i -- )`

Sets the current multitasking mode to the given mode. The integer this uses will be the same as one of those defined by the standard `$defines` `bg_mode,` `fg_mode,` and `pr_mode,` being `background,` `foreground,` and `preempt` mode, respectively. Programs set `BLOCK` will run `PREEMPT,` ignoring this mode.

`SETNAME    ( d s -- )`

Takes object `d,` and sets the name to `s.` A program may only set the names of objects that are owned by the effective user of the program, or any object if the program is `Wizard.` The name of a player can never be set, since that would normally require a password. See also SET, NAME, and SETDESC.

`SETOWN    ( d d -- )`

Sets the ownership of the first object to the player given in the second dbref. (wizbit only)

`SETSEED   ( s -- )`

Sets the seed for `SRAND.` Only the first thirty-two characters are significant. If `SRAND` is called before `SETSEED` is called, then `SRAND` is seeded with a semi-random value. See also GETSEED and SRAND. (fb6.0+)

`SINE    ( f -- f' )`

Returns the sine of `f.` Operates in the range of `-pi/4` and `pi/4.`

`SLEEP    ( i -- )`

Makes the program pause here for `i` seconds. The value of `i` cannot be negative. If the sleep is for more than `0` seconds, then the program may not thereafter use the `READ` primitive.

SMATCH (s s -- i)

Takes a string `s`, and a string pattern, `s2`, to check against. Returns `TRUE` if the string fits the pattern. This is case insensitive. In the pattern string, the following special characters will do as follows:

- A `?` matches any single character.
- A `*` matches any number of any characters.
- `{word1|word2|etc}` will match a single word, if it is one of those given, separated by `|` characters, between the `{}`'s. A word ends with a space or at the end of the string. The given example would match either the words `"word1"`, `"word2"`, or `"etc"`. `{}` word patterns will only match complete words: `"{foo}*"` and `"{foo}p"` do not match `"foop"` and `"*{foo}"` and `"p{foo}"` do not match `"pfoo"`. `{}` word patterns can be easily meaningless; they will match nothing if they:

  (a) contain spaces,
  (b) do not follow a wildcard, space or beginning of string,
  (c) are not followed by a wildcard, space or end of string.

- If the first char of a `{}` word set is a `'^'`, then it will match a single word if it is `NOT` one of those contained within the `{}`s. Example: `'{^Foxen|Fiera}'` will match any single word *except* for Foxen or Fiera.
- `'[aeiou]'` will match a single character as long as it is one of those contained between the `[]`s. In this case, it matches any vowel.
- If the first char of a `[]` char set is a `'^'`, then it will match a single character if it is *not* one of those contained within the `[]`s. Example: `'[^aeiou]'` will match any single character *except* for a vowel.
- If a `[]` char set contains two characters separated by a `'-'`, then it will match any single character that is between those two given characters. Example: `'[a-z0-9_]'` would match any single character between `'a'` and `'z'`, inclusive, any character between `'0'` and `'9'`, inclusive, or a `'_'`.
- The `'\'` character will disable the special meaning of the character that follows it, matching it literally.

Example patterns:

- `"d*g" matches "dg", "dog", "doog", "dorfg", etc.`
- `"d?g" matches "dog", "dig" and "dug" but not "dg" or "drug".`
- `"M[rs]." matches "Mr." and "Ms."`
- `"M[a-z]" matches "Ma", "Mb", etc.`
- `"[^a-z]" matches anything but an alphabetical character.`
- `"{Moira|Chupchup}*" matches "Moira snores" and "Chupchup arghs."`
- `"{Moira|Chupchup}*" does NOT match "Moira' snores".`
- `"{Foxen|Lynx|Fier[ao]} *t[iy]ckle*\?" Will match any string starting with 'Foxen', 'Lynx', 'Fiera', or 'Fiero', that contains either 'tickle' or 'tyckle' and ends with a '?'.`

SRAND (-- i)

Generates a seeded random number. See also `GETSEED` and `SETSEED` (fb6.0+).

`SQRT    ( f -- f' )`

Returns the square root of `f`. `f` must be greater than or equal to zero.

`STATS    ( d -- total rooms exits things programs players garbage )`

Returns the number of objects owned by `d`, or the total objects in the system if `d == #-1`. This is broken up into a total, rooms, exits, things, programs, players, and garbage. This functions much as the `@STAT` command. (Requires Mucker Level 3)

`STOD    ( s -- d )`

Takes a string and attempts to extract a dbref number from it. Recognizes both plain numbers as well as numbers prepended with the # sign.

`STRCAT    ( s1 s2 -- s )`

Concatenates two strings `s1` and `s2` and pushes the result `s = s1s2` onto the stack.

`STRCMP    ( s1 s2 -- i )`

Compares strings `s1` and `s2`, and returns `i` as `0` if they are equal. Otherwise, `STRCMP` returns `i` as the difference between the first non-matching character in the strings. For example, `"a" "z"` strcmp returns `25`. While returning `0` (`false`) for a match may seem counter-intuitive, this arrangement allows the primitive to be used for things such as string sorting functions. Unlike STRINGCMP, STRCMP is case sensitive. See also `STRNCMP`.

`STRCUT    ( s i -- s1 s2 )`

Cuts string `s` after its `i`'th character. For example,

```
"Foobar" 3 strcut
```

returns

```
"Foo" "bar"
```

If `i` is zero or greater than the length of `s`, returns a null string in the first or second position, respectively.

`STRING?    ( x -- i )`

Returns true if `x` is a string.

`STRINGCMP    ( s1 s2 -- i )`

Compares strings `s1` and `s2`. Returns `i` as `0` if they are equal, otherwise returns `i` as the difference between the first non-matching character in the strings. For example, `"a" "z"` stringcmp returns `25`. This function is not case sensitive, unlike `STRCMP`. See also `STRNCMP`.

`STRINGPFX    ( s s2 -- i )`

Returns `1` if `s2` is a prefix of `s`. Case insensitive. Returns `0` if `s2` is *not* a prefix of `s`.

`STRIP    ( s -- s )`

This is a built in `$define`. It is interpreted as `striplead striptail` It strips the spaces from both ends of a string.

`STRIPLEAD    ( s -- s )`

Strips leading spaces from the given string.

`STRIPTAIL    ( s -- s )`

Strips trailing spaces from the given string.

`STRLEN    ( s -- i )`

Returns the length of string `s.`

`STRNCMP    ( s1 s2 i -- i' )`

Compares the first `i` characters in strings `s1` and `s2.` Return value is like `STRCMP.` See also `STRINGCMP.`

`STRTOF    ( s -- f )`

Converts string `s` to a floating point number. `s` may be in the format `xxx.yyy` or `xxx.yyyEzz.` See also `FTOSTR.`

`SUBST    ( s1 s2 s3 -- s )`

`s1` is the string to operate on, `s2` is the string to change all occurences of `s3` into, and `s` is resultant string. For example:

```
   "HEY_YOU_THIS_IS" " " "_" subst
```

results in

```
   "HEY YOU THIS IS"
```

`s2` and `s3` may be of any length.

`SUCC    ( d -- s )`

Takes object `d` and returns its success (`@succ`) string field `s.`

`SWAP    ( x y -- y x )`

Takes objects `x` and `y` on the stack and reverses their order.

`SYSPARM    ( s -- s )`

Takes a tuneable system parameter and returns its value as a string. For an integer it returns it as a string, a time is returned as a string containing the number of seconds, a dbref is returned in standard dbref format, and boolean is returned as `'yes'` or `'no'` Checking an invalid parameter or a parameter with higher permissions then the program has will return an empty string.

Parameters available:

```
(str) dumpwarn_mesg          - Message to warn of a coming DB dump
(str) deltawarn_mesg         - Message to warn of a coming delta dump
(str) dumpdeltas_mesg        - Message telling of a delta dump
(str) dumping_mesg           - Message telling of a DB dump
(str) dumpdone_mesg          - Message notifying a dump is done
```

(str) penny	- A single currency
(str) pennies	- Plural currency
(str) cpenny	- Capitolized currency
(str) cpennies	- Capitolized plural currency
(str) muckname	- The name of the MUCK
(str) rwho_passwd	- Password for RWHO servers (Wizbit only)
(str) rwho_server	- RWHO server to connect to (Wizbit only)
(str) huh_mesg	- Message for invalid commands
(str) leave_mesg	- Message given when QUIT is used
(str) register_mesg	- Message for a failed 'create' at login
(time) rwho_interval	- Interval between RWHO updates
(time) dump_interval	- Interval between dumps
(time) dump_warntime	- Warning prior to a dump
(time) monolithic_interval	- Max time between full DB dumps
(time) clean_interval	- Interval between unused object purges
(time) aging_time	- When an object is considered old and unused
(int) max_object_endowment	- Max value of an object
(int) object_cost	- Cost to create an object
(int) exit_cost	- Cost to create an exit
(int) link_cost	- Cost to link an exit
(int) room_cost	- Cost to dig a room
(int) lookup_cost	- Cost to lookup a player name
(int) max_pennies	- Max number of pennies a player can own
(int) penny_rate	- Rate for finding pennies
(int) start_pennies	- Starting wealth for new players
(int) kill_base_cost	- Number of pennies for a 100 percent chance
(int) kill_min_cost	- Minimum cost for doing a kill
(int) kill_bonus	- Bonus for a successful kill
(int) command_burst_size	- Maximum number of commands per burst
(int) commands_per_time	- Commands per time slice after burst
(int) command_time_msec	- Time slice length in milliseconds
(int) max_delta_objs	- Max percent of changed objects for a delta
(int) max_loaded_objs	- Max percent of the DB in memory at once
(int) max_process_limit	- Total processes allowed
(int) max_plyr_processes	- Processes allowed for each player
(int) max_instr_count	- Max preempt mode instructions
(int) instr_slice	- Max uninterrupted instructions per time slice
(int) mpi_max_commands	- Max number of uninterruptable MPI commands
(int) pause_min	- Pause between input and output servicing
(int) free_frames_pool	- Number of program frames pre-allocated

(int) listen_mlev	- Minimum Mucker level for _listen programs
(ref) player_start	- The home for players without a home
(bool) use_hostnames	- Do reverse domain name lookup
(bool) log_commands	- The server logs commands (Wizbit only)
(bool) log_failed_commands	- The server logs failed commands (Wizbit only)
(bool) log_programs	- The server logs programs (Wizbit only)
(bool) dbdump_warning	- Warn about coming DB dumps
(bool) deltadump_warning	- Warn about coming delta dumps
(bool) periodic_program_purge	- Purge unused programs from memory
(bool) support_rwho	- Use RWHO server
(bool) secure_who	- WHO works only in command mode
(bool) who_doing	- Server support for @doing
(bool) realms_control	- Support for realm wizzes
(bool) allow_listeners	- Allow listeners
(bool) allow_listeners_obj	- Objects can be listeners
(bool) allow_listeners_env	- Listeners can be up the environment
(bool) allow_zombies	- Zombie objects allowed
(bool) wiz_vehicles	- Only wizzes can make vehicles
(bool) force_mlev1_name_notify	- M1 programs forced to show name on notify
(bool) restrict_kill	- Can only kill KILL_OK players
(bool) registration	- Only wizzes can create players
(bool) teleport_to_player	- Allow use of exits linked to players
(bool) secure_teleport	- Check teleport permissions for personal exits
(bool) exit_darking	- Players can set exits dark
(bool) thing_darking	- Players can set objects dark
(bool) dark_sleepers	- Sleepers are effectively dark
(bool) who_hides_dark	- Dark players are hidden (Wizbit only)
(bool) compatible_priorities	- Backwards compatibility for exit priorities
(bool) do_mpi_parsing	- Parse MPI strings in messages
(bool) look_propqueues	- Look triggers _lookq propqueue
(bool) lock_envcheck	- Locks will check the environment
(bool) diskbase_propvals	- Allow diskbasing of property values

`SYSTIME    ( -- i )`

Returns the number of second from Jan 1, 1970. This is compatible with the system timestamps and may be broken down into useful values through 'timesplit'.

`TAN    ( f -- f' )`

Returns the tangent of `f.` Operates in the range of `-pi/4` and `pi/4.` (fb6.0+)

`TESTLOCK    ( d l -- i )`

Tests the player dbref against the given lock. If the test was successful, then this returns a `1.` If the test failed, then this returns a `0.` See also `LOCK?,` `PARSELOCK,` `UNPARSELOCK,` `PRETTYLOCK,` `GETLOCKSTR,` `SETLOCKSTR,` and `LOCKED?`

`TEXTATTR   ( s1 s2 -- s3 )`

Takes plain text string `s1` and adds color and formmatting codes specified in string `s2,` returning the enhanced string as `s3.` The attributes specified in `s2` should be a comma-separated list of attribute names. For example:

```
  "WARNING!" "bold,red" textattr me @ swap notify
```

(fb6.0+)

`THEN` See `IF`

`THING?    ( d -- i )`

Returns `i` as `1` if object `d` is a thing, otherwise returns `i` as `0.` See also `PLAYER?,` `PROGRAM?,` `ROOM?,` `EXIT?,` `OK?.`

`TIME    ( -- s m h )`

Returns the time of day as integers on the stack, seconds, then minutes, then hours.

`TIMEFMT    ( s i -- s )`

Takes a format string and a `SYSTIME` integer and returns a string formatted with the time. The format string is `ASCII` text with formatting commands:

```
    %% -- "%"
    %a -- abbreviated weekday name.
    %A -- full weekday name.
    %b -- abbreviated month name.
    %B -- full month name.
    %C -- "%A %B %e, %Y"
    %c -- "%x %X"
    %D -- "%m/%d/%y"
    %d -- month day, "01" - "31"
    %e -- month day, " 1" - "31"
    %h -- "%b"
    %H -- hour, "00" - "23"
    %I -- hour, "01" - "12"
    %j -- year day, "001" - "366"
    %k -- hour, " 0" - "23"
    %l -- hour, " 1" - "12"
    %M -- minute, "00" - "59"
    %m -- month, "01" - "12"
    %p -- "AM" or "PM"
    %R -- "%H:%M"
    %r -- "%I:%M:%S %p"
    %S -- seconds, "00" - "59"
    %T -- "%H:%M:%S"
    %U -- week number of the year. "00" - "52"
    %w -- week day number, "0" - "6"
    %W -- week# of year, starting on a monday, "00" - "52"
    %X -- "%H:%M:%S"
    %x -- "%m/%d/%y"
    %y -- year, "00" - "99"
    %Y -- year, "1900" - "2155"
```

%Z -- Time zone. "GMT", "EDT", "PST", etc.

`TIMESPLIT    ( i -- is im ih id im iy iw iyd )`

Splits a systime value into `8` values in the following order: seconds, minutes, hours, monthday, month, year, weekday, yearday. Weekday starts with sunday as `1,` and yearday is the day of the year (1-366).

`TIMESTAMPS    ( d -- i i2 i3 i4 )`

Returns the following for a program, the time created (`i`), the time last modified (`i2`), the time last used (`i3`), and the number of uses(`i4`) for any object.

`TOLOWER    ( s -- s )`

Takes a string and returns it with all the letters in lowercase.

`TOUPPER    ( s -- s )`

Takes a string and returns it with all the letters in uppercase.

`TREAD    ( i -- i | s i )`

Acts like a timed `READ` call. If the user does not provide input within the given number of seconds, the `READ` call will time-out and return a failure to the program, otherwise it returns a success and the string value entered.

`TRIG    ( -- d )`

Returns the dbref of the original trigger.

`UNPARSELOCK    ( l -- s )`

Unparses a lock into a string fit for program editing. Also see `LOCK?,` `PARSELOCK,` `PRETTYLOCK,` `TESTLOCK,` `GETLOCKSTR,` `SETLOCKSTR,` and `LOCKED?.`

`UNPARSEOBJ    ( d -- s )`

Returns the name-and-flag string for an object. It always has the dbref and flag string after the name, even if the player doesn't control the object. For example: `"One(#1PW)"`

`UNTIL    ( i -- )`

If the value on top of the stack is false, then it jumps execution back to the instruction afer the matching `BEGIN` statement. (`BEGIN-UNTIL,` `BEGIN-REPEAT,` and `IF-ELSE-THEN`'s can all be nested as much as you want.) If the value is true, it exits the loop, and executes the next instruction, following the `UNTIL.` Marks the end of the current loop.

`VAR <name>`

VAR is not a 'true' primitive in that it must always be used outside words and does not alter the stack in any way. When the compiler sees a `VAR` statement, it allows the use of `<name>` as a variable in all words sequentially defined after the var declaration. See also `@,` `VARIABLE,` and `LOCALVAR.`

`VARIABLE    ( i -- v )`

Converts integer `i` to variable reference `v.` Of the pre-defined variables, `me` corresponds to integer `0, loc` to `1,` and `trigger` to 2. Thus:

```
   me @
```

and

```
   0 variable @
```

will do the same thing (return the user's dbref). User-defined variables are numbered sequentially starting at `3` by the compiler. Note that these variable numbers can be used even if variables have not been formally declared, making implementation of such things as arrays conveniently easy. See `@,` `!,` and `VAR`

`VERSION    ( -- s )`

Returns the version of this code in a string. "Muck2.2fb6.0", currently.

`XYZ_TO_POLAR   ( fx fy fz -- fr ft fp )`

This converts the `XYZ` coordinate `(fx, fy, fz)` to the spherical polar coord `(fr, ft, fp).` (fb6.0+)

`WHILE    ( i -- )`

If the value on top of the stack is false, then this causes execution to jump to the instruction after the `UNTIL` or `REPEAT` for the current loop. If the value is true, however, execution falls through to the instruction after the `WHILE.`

3.2.4 MUF Library Reference

The standard database and package of programs includes the following programming libraries:

- lib-case
- lib-edit
- lib-editor
- lib-index
- lib-lmgr
- lib-look
- lib-match
- lib-mesg
- lib-mesgbox
- lib-props
- lib-reflist
- lib-strings
- lib-stackrng

Note: Since the libararies were written, many of the functions they provide have been defined as `MUF` primitives. The libraries are retained in their current form for compatibility with existing programs. If a primitive providing the functionality you want is available, it will be more efficient than a library call. For more information on using libraries, see Section 3.2.2.

Functions by Library:

Lib-Case

An ingenious library that uses a few simple `$defines` to give `MUF` 'case' or 'switch' handling like you would find in other programming languages. See the program's header comment for a good, terse explanation.

- case
- default
- end
- endcase

Lib-Edit

This library includes string-editing routines, with capabilities for handling ranges of strings such as would be present by reading a list or range from a list onto the stack.

- EDITcenter
- EDITcopy
- EDITdisplay
- EDITformat
- EDITfmt_rng
- EDITjoin
- EDITjoin_rng
- EDITindent
- EDITleft
- EDITlist
- EDITmove
- EDITreplace
- EDITright
- EDITsearch
- EDITshuffle
- EDITsort
- EDITsplit
- instring
- STRasc
- STRchr
- STRright
- rinstring

Lib-Editor

A set of functions providing an interface with the list editor. Note: this library is in the public domain, but its terms of use state that the author of the library, Foxen, must be credited in the header comment of any program that makes use of it.

- EDITOR
- EDITORloop
- EDITORparse
- EDITORheader

Lib-Index

This library provides a set of routines for handling 'indexes'. See Indexes. *Note*: There are two versions of this library in circulation. Currently, `basedb.db` and `fbmuf` provide the older version. The second version — which may or may not be available on your `MUCK` — provides the additional public routines `index-setmatchstr,`, `index-getmatchstr,` and `index-propname.` Use `@view $lib/index` to determine which version you have. The standard version is discussed here. Also, the standard version as implemented by `basedb.db` and the upload script in `fbmuf` only set `_def/` props in the `.period-prefix` form: routine names must be joined with a leading period in your code, unless the wizards on your `MUCK` have added the no-period form. Example: to use the routine `index-add,` put `.index-add` in your code. (The newer version may or may not require the `.format`). There are some problems with the implementation of several functions in this library, as discussed in individual entries below.

- index-add
- index-add-sort
- index-delete
- index-envmatch
- index-match
- index-matchrange
- index-remove
- index-set
- index-value
- index-write

Lib-Match

A library of matching functions.

- .noisy_match
- .noisy_pmatch
- .controls
- .match_controlled
- .multi_rmatch
- .table_match
- .std_table_match

Lib-LMGR

Lib-Lmgr is one of several libraries holding functions used by `lsedit.` These functions are also useful for other programs that need to manipulate lists without going through the list editor, or to create their own list editors. *Note*: The documentation provided in the program's header comment, and listed with `@view,` is inaccurate. The function `LMGR-CopyRange` (listed in the header) does not exist, but the following undocumented public functions do: `LMGR-PutBRange,` `LMGR-GetCount,` and `LMGR-SetCount.` The public function `LMGR-ExtractRange` (undocumented, but included as a `_def/`) does not work.

Despite the inaccuracies of documentation, the functions provided by `Lib-Lmgr` are useful and efficient, and provide a number of capabilities not duplicated by primitives.

- LMGR-ClearElem
- LMGR-ClearRange
- LMGR-DeleteList
- LMGR-DeleteRange
- LMGR-FullRange
- LMGR-GetBRange
- LMGR-GetCount
- LMGR-GetElem
- LMGR-Getlist
- LMGR-GetRange
- LMGR-InsertRange
- LMGR-MoveRange
- LMGR-PutBRange
- LMGR-PutElem
- LMGR-PutRange
- LMGR-SetCount

Lib-Look

A set of routines adapted to 'look' functions. *Note*: the standard version of `lib-look` as implemented by `basedb.db` and the upload script in `fbmuf` only set `_def/` props in the `.period-prefix` form: routine names must be joined with a leading period in your code, unless the wizards on your `MUCK` have added the no-period form. Example: to use the routine `'safecall',` put `'.safecall'` in your code.

- contents-filter
- cmd-look
- db-desc
- dbstr-desc
- get-contents
- list-contents
- long-display
- safecall
- short-display
- short-list
- str-desc
- unparse

Lib-Mesg

This library provides functions for handling 'messages'. See Messages.

- MSG-append
- MSG-count
- MSG-create
- MSG-destroy
- MSG-info

- MSG-insitem
- MSG-item
- MSG-message
- MSG-setinfo
- MSG-setitem
- MSG-delitem

Lib-Mesgbox

A library of message and message-box handling routines. See Messages.

- MBOX-append
- MBOX-badref?
- MBOX-count
- MBOX-create
- MBOX-delmesg
- MBOX-destroy
- MBOX-insmesg
- MBOX-message
- MBOX-msginfo
- MBOX-num2ref
- MBOX-ref2num
- MBOX-ref2prop
- MBOX-setinfo
- MBOX-setmesg

Lib-Props

This library contains several useful property-handling functions.

- setpropstr
- envprop
- envsearch
- locate-prop

Lib-Reflist

A library of reflist-handling routines. See Reflists.

- REF-add
- REF-delete
- REF-first
- REF-next
- REF-inlist?
- REF-list
- REF-allrefs
- REF-filter
- REF-editlist

Lib-Strings

This library provides a number of routines for formating strings.

- instring
- rinstring
- STRasc
- STRblank?
- STRchar
- STRcenter
- STRfillfield
- STRleft
- STRrsplit
- STRright
- STRsls
- STRsms
- STRsplit
- STRstrip
- STRsts

Lib-Stackrng

This library provides routines for handling items within a specified range of the stack. See Ranges

- sr-catrng
- sr-copyrng
- sr-deleterng
- sr-extractrng
- sr-filterrng
- sr-insertrng
- sr-poprng
- sr-swaprng

`case   ( -- )`

Begins a `case` statement. See header for lib-case. (Lib-Case)

`cmd-look   ( s -- )`

Does a `match` function on `s,` then calls `db-desc` with the results, simulating the server `look` command. (Lib-Look)

`contents-filter   ( a d -- d' ... d'' i )`

Takes the address of a 'filter' function and a dbref, and returns the objects from `d's` contents for which the filter function returns true (`1`) as a range on the stack. The first item found (the 'top' of a `Contents` or `Carrying` list) will be at the start of the range. Example:

```
: PlayerCheck   ( d -- i )
    player? if 1 else 0 then
;
: GetPlayers   ( -- d' ... d'' i )
    'PlayerCheck me @ location contents-filter
;
```

This code puts all the players in the room on the stack as a range of dbrefs, filtering out objects of other types. `'PlayerCheck` (a function name joined with an `'` apostrophe) puts the address of the the filter function `PlayerCheck` function on the stack. (Lib-Look)

`.controls   ( d1 d2 -- i )`

Returns true if player `d1` controls object `d2.` The same capability is provided by the `CONTROLS` primitive. (Lib-Match)

`db-desc   ( d -- )`

Prints a description of `d,` including the name and triggering succ and fail if `d` is a room. The desc is not parsed for `MPI.` Programs run with `d` stored in the variable `trigger.` (Lib-Look)

`dbstr-desc   ( d s -- )`

Prints `s` as a description, like `str-desc,` using `d` as the effective trigger value. The string is not parsed for `MPI.` (Lib-Look)

`default   ( x -- )`

Declares default action to take if no cases in a `case` statement test true. See header for lib-case. (Lib-Case)

`EDITcenter   ( range ... offset i1 i2 i3 -- range' )`

Center justifies all strings between `position1` and `position2` for a screen `i1` characters wide. That is, the strings are padded to a length of `i1` characters with leading and trailing spaces. (Lib-Edit)

`EDITcopy   ( {rng} ... offset dest start end -- {rng'} ... )`

Copies text within a string range from one line to another, inserting it in the new location. (Lib-Edit)

`EDITdisplay   ( range -- )`

Displays the range of strings on the stack to the user. See also `EDITlist.` (Lib-Edit)

`EDITformat   ( range ... c i1 i2 -- range' )`

Formats the strings in `range` to strings between right margin `i1` and wrap margin `i2` in length. Short strings are joined to fill this field width; long strings are split at the last occurance of split character `c` found between `i1` and `i2.` See aslo `EDITfmt_rng.` (Lib-Edit)

`EDITfmt_rng   ( range ... offset i position1 position2 -- range' )`

Formats the strings of the range between `position1` and `position2` to strings `i` characters wide, splitting long strings and joining short strings. A string that consists only of spaces is considered a paragraph delimiter and is not joined. `i` must be equal to or greater then `20.` The string at `position1` must `2` or more characters long. See also `EDITformat.` (Lib-Edit)

`EDITindent   ( range ... offset i position1 position2 -- range' )`

Indents all strings between `position1` and `position2` by `i` spaces. That is, the strings are padded with `i` space characters. If `i` is a negative number, it reduces indentation by `i` characters, but will not unindent past the left margin. (Lib-Edit)

`EDITjoin   ( range -- s )`

Joins a range of strings on the stack into one string. (Lib-Edit)

`EDITjoin_rng   ( range ... offset position1 position2 -- range' )`

Joins the lines of the subrange between `position1` and `position2` into a single string, returning the string range that results. Leading and trailing spaces are stripped from the strings to be joined, and a single space is inserted in the combined string at points where strings were joined. See also `EDITjoin.` (Lib-Edit)

`EDITleft   ( range ... offset position1 position2 -- range' )`

Left justifies all strings between `position1` and `position2.` That is, leading spaces are stripped from these strings. (Lib-Edit)

`EDITlist   ( range ... offset i1 i2 i3 -- range )`

Prints the strings `i2` to `i3` from range to the user's screen. If `i1` is true, the lines will be prepended with line numbers. See also `EDITdisplay.` (Lib-Edit)

`EDITOR   ( range -- range' s )`

Puts the user in the list editor, using range as the values for the list. The list can be interactively edited as with command lsedit. Exits with the modified range and the editor's exit string (`end` or `abort`) on the stack. See also `EDITORloop.` (Lib-Editor)

`EDITORheader   ( -- )`

Prints the standard header informing the user that he is entering the list editor and giving succinct help. Called automatically by `EDITOR.`

```
        < Entering editor. Type '.h' on a line by itself for help. >
         < '.end' will exit the editor. '.abort' aborts the edit. >
          < Poses and says will pose and say as usual. To start a >
          < line with : or " just preceed it with a period ('.') >
```

(Lib-Editor)

`EDITORloop   ( range s1 i1 -- range' s1 i2 s2 i3 i4 s3 )`

Puts the user in the list editor, using `range` as the values for the lines of the list. The list can be interactively edited as with command `lsedit.` The `EDITOR` function provides similar functionality; `EDITORloop` gives greater control over data passed to and from the editor.

`EDITORloop` takes a range, a space-separated string `s1` containing any commands that can be used to return from the editor in addition to `end` and `abort,` cursor position `i1,` and the first editor command to be executed.

`EDITORloop` returns the modified range, the string `s1,` the cursor postion at time-of-exit, exit arguments `s2,` `i3,` and `i4,` and the command string used to exit the editor on the stack. Exit arguments are parameters returned to the program when `EDITORloop` exits. They follow the syntax of other editor commands:

```
  command start_line end_line = argument_string
```

`start_line` is `i3.` `end_line` is `i4.` `argument_string` is `s2.`

Example: Suppose you create a program that allows players included in a list holding their dbrefs to update documents in a news reader program. The program includes and `#admin` function that puts the user into the editor to modify this list. You want to include, in your version of the editor, a command that shows the names of the players who are currently in the list, optionally followed by information such as the last time they logged on or their position/title in the `MUCK` news staff. The user could then display the names (and other information) for players whose dbrefs are in the list, or a range from the list, by entering `.names` or `.names <range>` or `.names [<range>] = <field>.` You could do this by reading the list onto the stack and supplying `names` as a command string that would cause `EDITORloop` to return, with code such as the following:

```
"_staff" trig LMGR-Getlist       (*read list onto stack as a range*)
"names" over 1 + ".i" EDITORloop (*put user in editor, ready to insert
                                   text, at the next free line, with
                                   'names' defined as a .command that
                                   will cause EDITORloop to return*)
```

If the user typed `.names` while at line 6 of a 12-line list of dbrefs, the following would be put on the stack:

```
... #123, 12, "names", 6, "", 0, 0, "names"
```

Using comparison primitives such as `SMATCH` or `STRINGCMP,` you could check the top value on the stack and, if it is the string `"names"` (as it is here), execute a `ShowNames` function in your program, then return to the editor with another instance of `EDITORloop.`

If the user had typed `.names 1 8,` the following would be put on the stack:

```
... #123, 12, "names", 6, "", 1, 8, "names"
```

Your `ShowNames` function would need to use the values `1` and `8` to show only names for the dbrefs in lines 1 - 8.

If the user had typed `.names 1 - 8 = title,` the following would be put on the stack:

```
... #123, 12, "names", 6, "title", 1, 8, "names"
```

`ShowNames` would use this information to show the names of players listed in lines 1 - 8 of the list, concattenated with their title... perhaps a value stored in their `_news/title` property. See also `EDITOR.` (Lib-Editor)

`EDITORparse   ( range s1 i1 s2 -- range' s1 i1 i2 )`

Parses range with the editor command `s2,` returning the modified range, the original paramaters `s1` and `i1,` and the last line handled as `i2.`

Example:

```
"mink" "otter" "linsang" 3   "quit" 1 ".indent 1 3 = 5" EDITORparse
```

This would enter and exit the editor, indenting each string in the range by 5 spaces, with the last line handled on top of the stack as `i2.` The standard editor output would be shown to the user:

```
< Indented 3 lines starting at line 1, 5 columns. >
```

(Lib-Editor)

`EDITmove   ( {rng} ... offset dest start end -- {rng'} ... )`

Moves text within a string range from one line to another location, deleting the original. (Lib-Edit)

`EDITreplace   ( range ... offset s1 s2 position1 position2 -- range' )`

Performs a case-sensitive seach of the range items from `position1` to `position1,` inclusive, for all occurances of string `s1,` replacing each with string `s2.` (Lib-Edit)

`EDITright   ( range ... offset i position1 position2 -- range' )`

Right justifies all strings between `position1` and `position2` for a screen width of `i` characters. That is, these strings are padded with trailing spaces to a string length of `i` characters. (Lib-Edit)

`EDITsearch   ( range ... offset s position -- range ... i2 )`

Perfroms a case-sensitive search of the strings in `range` for an occurance of string `s.` The search begins with the item at `position.` `i2` is the position of the first item that includes an occurance of string `s.` (Lib-Edit)

`EDITshuffle   ( range -- range' )`

Randomizes the order items in a range. (Lib-Edit)

`EDITsort   ( range i1 i2 -- range' )`

Alphabetically sorts `range.` If `i1` is true, the sort will be in descending order: `"aardvark"` would be near the top of the stack and `"zebra"` would be near the bottom. If `i2` is true, the sort will be case-sensitive: `"Leviathon"` would be come before `"lapidary",` since `"L"` and `"l"` are two different characters in a case-sensitive comparison, with `"L"` preceding `"l".` (Lib-Edit)

`EDITsplit   ( s c i1 i2 -- range )`

Splits string `s` on the last split character `c` found between right margin `i1` and wrap margin `i2.` If no split character is found in this range, the string is split at right margin `i1.` (Lib-Edit)

`end   ( -- )`

Declares the end of an action block in a `case` statement. See header for lib-case. (Lib-Case)

`endcase   ( -- )`

Ends a `case` statement. See header for lib-case. (Lib-Case)

`envprop   ( d s -- s' )`

Searches up the environment tree from object `d,` looking for a property with the name `s.` `Envprop` returns the value stored in the first occurance of prop `s` found, or a null string if it wasn't found. (Lib-Props)

`envsearch   ( d s -- d' )`

Searches up the environment tree from object `d` for an occurance of property `s.` `Envsearch` returns the dbref of the first object the property is found on, or `#-1` if the propety is not found. (Lib-Props)

`get-contents   ( d -- d' ... d'' i )`

Passes the contents of object `d` through a standard filter that emulates a server contents list: dark rooms do not have contents lists, unless you control the objects or the room; dark objects you don't control do not show; you do not show. Those objects which would show in a `Contents/Carrying` list by these criteria are returned as a range of dbrefs. The first contained object (the 'top' of a `Contents` or `Carrying` list) is at the start of the range. (Lib-Look)

`index-add   ( d s1 s2 s3 -- i )`

Adds name `s2,` associated with value `s3,` to index `s1` on object `d.` Returns error code `i.` If `s2` is already a member of the index (matching exactly), index-add returns error code `0,` and no changes are made to the index. Otherwise, the name and value are added to the index and error code `1` is returned (operation successful). See also Indexes and `std-index-add`. (Lib-Index)

`index-add-sort   ( d s1 s2 s3 -- i )`

Like `index-add,` `index-add-sort` adds name `s2` and value `s3` to index `s1,` stored on object `d.` index-add always adds a name to the end of the index; `Index-add-sort,` *in theory*, inserts the new name in alphabetical order, immediately before the element it would precede alphabetically. *In theory*, if names are always added to the index with this routine, the index will remain in alphabetical order. However, this function does not behave as advertised: like `index-add,` it always puts new names at the end of the index. Returns error code `i:` `1` is `'no error: item added',` `0` is `'error: name already a member of index; index unchanged'.` See also Indexes and `std-index-add`. (Lib-Index)

`index-delete   ( d s1 s2 -- )`

Removes name `s2,` and its associated value, from index `s1` on object `d,` with no error checking. See also Indexes and `std-index-delete`. (Lib-Index)

`index-first   ( d s1 -- s2 )`

Returns the first name in index `s1` on object `d` as `s2.` See also Indexes and `std-index-first`. (Lib-Index)

`index-envmatch   ( d s1 s2 -- d2 s3 i )`

Returns name `s2` from index `s1` on object `d` as `s3,` the object on which the name was found as `d2,` and error code `i` (`1` if no error; `0` if no match). If no match is found on `d,` continues searching up the environment tree, checking indexes called `s2` on these objects as well. If index `s1` is was created with `index-add,` or with `index-setmatchstr` with the `'w'` name parameter, `index-envmatch` will return the first match found. That is, there is no check for ambiguity. See also Indexes and `std-index-envmatch`. (Lib-Index)

index-last (d s1 -- s2)

In theory, returns the last name in index `s1` on object d as `s2.` In practice, returns a null string. See also Indexes and `std-index-last`. (Lib-Index)

`index-match   ( d s1 s2 -- s2' i )`

Returns name `s2` from index `s1` on object `d` and error code `1` (no error) if `s2` is a member of index `s1.` If `s2` is not a member of `s1,` `index-match` returns a null string and error code `0` (`error: no match`). If index `s1` was created with `index-add,` or with `index-setmatchstr` with the `'w'` name parameter, `index-match` will return the first match found. That is, there is no check for ambiguity. If the index was created with `index-setmatchstr` without the `'w'` parameter, only complete, explicit matches will be returned, in which case ambiguity is impossible. See also Indexes and `std-index-match`. (Lib-Index)

`index-matchrange   ( d s1 s2 -- string_range )`

Performs a partial-word match search on all members of index `s1` on object `d`, returning those which match true as a string range (see Ranges). Ignores match-type parameters: any full or partial matches will be returned. Not available in `'std-'` form. See also Indexes. (Lib-Index)

`index-next   ( d s1 s2 -- s3 )`

Returns the name immediately following name `s2` from index `s1` on object `d`. See also Indexes and `std-index-next`.(Lib-Index)

`index-prev   ( d s1 s2 -- s3 )`

Returns the name immediately preceding name `s2` from index `s1` on object `d`. See also Indexes and `std-index-prev`.(Lib-Index)

`index-remove   ( dbref index name -- error )`

Removes name s2 and its associated value from index s1 on object d, and returns an error code: 1 if the name existed in the index and was successfully removed, and 0 otherwise. See also Indexes and `std-index-remove`. (Lib-Index)

`index-set   ( d s1 s2 s3 -- i )`

Adds name `s2`, associated with value `s3`, to index `s1` on object `d`, and returns error code `i`. Unlike `index-add` and `index-write`, this routine does not check for and return errors: if the name is not presently a member of the index, it will be added; if the name is already a member, its value will be edited. See also Indexes and `std-index-set`. (Lib-Index)

`index-value   ( d s1 s2 -- s3 )`

Returns the value associated with name `s2` in index `s1` on object `d`. If the index does not include name `s1`, `s3` will be a null string.See also Indexes and `std-index-value`. (Lib-Index)

`index-write   ( d s1 s2 s3 -- i )`

In index `s1` on object `d`, sets the value of existing index member with name `s2` to value `s3`. Returns error code `i`: `1` is `'no error: value changed'`, `0` is `'error: name not a member of this index; index unchanged'`. See also Indexes and `std-index-write`. (Lib-Index)

Indexes

"In theory, there's no difference between theory and practice. But in practice, there is." Keep this firmly in mind when working with Lib-Index functions. The following is a brief discussion of how Lib-Index functions are supposed to work. However, several do not behave as advertised. Such cases are noted in the entries for specific Lib-Index functions.

An index is a set of name/value pairs. The index as a whole is stored as a set of associated properties, in a single propdir, consisting of a list of all elements (stored as a single string) and a property for each 'name', which holds a 'value' associated with the name.

Most routines are also available in `'std-'` form (the routine name is prefaced with `std-`, such as `.index-add` to `.std-index-add`). Routines which store information (add members to the index) have the following stack effect:

```
( d s1 s2 s3 -- d s1 s2 s3 i )
```

where `d` is the object the index is stored on, `s1` is the name of the index, `s2` is the 'name' value, and `s3` is the 'value' value. All four items are left on the stack unchanged, and an integer error code is returned: `1` for `'no error: item successfully added',` and `0` for `'error: s2 is already a member of the index; index unchanged'.`

Routines that retrieve information (perform matches) have the following stack effect:

```
( d s1 s2 s3 -- d s1 s2' s4 i )
```

where `d` is the object the index is stored on, `s1` is the name of the index, `s2` is the name to be matched, and `s3` is a string holding 1 - 3 characters specifying matching criteria:

```
x: exact match only, ignores the index itself
e: exit-style match: exact match of ;-separated value, and select the
   first match found if there are more than one
w: normal operation: match the beginning of a space separated word,
   and fail if more than one matched. So, 'mat' matches 'mattress'
   and 'lit match', but not 'rematch'.
```

Arguments `d` and `s1` are left on the stack unchanged. The name to be matched is, if found, returned as `s2'.` If the match was unsuccessful, `s2'` will be a null string. If `s2` were a partial match, `s2'` would be returned as the full name. The match criteria paramters, `s3,` are returned as null string `s4.` Finally, an error code is returned (`1` is no error, match successful; `0` is error, match unsuccessful).

The default matching criteria (employed with routines called *without* the `std-` prefix) is `xew`: all three criteria are used, and the first match made by any of the criteria will be returned.

`instring   ( s1 s2 -- i )`

Returns the position of the first occurance of `s2` in `s1,` case-insensitive. The same capability is provided by the `INSTRING` primitive. (Lib-Strings)

`list-contents   ( s d -- )`

Calls `get-contents` followed by `long-display` to print out all of the contents of the given dbref. If there are any contents listed, then the string on the stack is printed out, for `"Contents:"` or the like. If the contents list is empty, the string is ignored. The triggering player's name is omitted from the contents list of a room (that is, you don't see yourself listed). (Lib-Look)

`LMGR-ClearElem   ( i s d -- )`

Clears the content of line `i` from list `s` on object `d.` The size of the list (number of lines) remains unchanged.

`LMGR-ClearRange   ( i1 i2 s d -- )`

Clears `i1` lines from list `s` on object `d,` beginning with line `i2.` The number of lines in the list does not change. See also `LMGR-DeleteRange`

`LMGR-DeleteList   ( s d -- )`

Deletes list `s` from object `d.`

`LMGR-DeleteRange   ( i1 i2 s d -- )`

Deletes `i1` lines from list `s` on object `d`, beginning with line `i2`. Lines in the orginal list positioned after line `i2` are shifted to earlier positions. See also `LMGR-ClearRange.`

`LMGR-FullRange   ( s d -- i 1 s d )`

Returns the count of lines in list `s`, an index value of `1`, and string `s` and dbref `d` unchanged. These are the parameters that would be needed to put a complete list on the stack using `LMGR-GetRange.`

`LMGR-GetBRange   ( i1 i2 s d -- range )`

Returns puts `i1` lines from list `s` on object `d`, followed by their count, on the stack, beginning with line `i2`. The first line of the list will be closest to the top of the stack. See also `LMGR-GetRange.`

`LMGR-GetCount   ( s d -- i )`

Returns the count of lines in list `s` on object `d`.

`LMGR-GetElem   ( i s d -- )`

Returns the content of line `i` from list `s` on object `d`.

`LMGR-Getlist   ( s1 d -- s s' ... s'' i )`

Puts the contents of list `s1` on object `d` and the total number of lines on the stack.

`LMGR-GetRange   ( i1 i2 s d -- range )`

Puts `i1` lines from list `s` on object `d`, followed by their count, on the stack, beginning with line `i2`. The last line of the list will be closest to the top of the stack. See also `LMGR-GetBRange.`

`LMGR-InsertRange   ( sx sx' ... sx'' i1 i2 s d -- )`

Inserts `i1` strings (range `sx sx' ... sx''`) in list `s` on object `d`, beginning with line `i1`. Lines in the original list positioned after `i2` are shifted to later positions. See also `LMGR-PutRange`

`LMGR-MoveRange   ( i1 i2 i3 s d -- )`

Moves the contents of the `i2` lines of list `s` on object `d`, beginning at line `i3`, to the lines beginning at line `i1`. The contents of lines `i1` to `i1+i2` are over-written.

`LMGR-PutBRange   ( sx sx' ... sx'' i1 i2 s d -- )`

Stores `i1` strings (`range sx sx' ... sx''`) in list `s` on object `d`, beginning with line `i2`. The string at the top of the range is stored at line `i2`, and the remaining strings are stored on successive lines (in other words, strings are stored in descending order). on The previous contents of the lines are over-written. See also `LMGR-PutRange` and `LMGR-InsertRange.`

`LMGR-PutElem   ( s1 i s2 d -- )`

Stores string `s1` in line `i` of list `s2` on object `d`. The previous contents of the line are over-written.

`LMGR-PutRange   ( sx sx' ... sx'' i1 i2 s d -- )`

Stores `i1` strings (`range sx sx' ... sx''`) in list `s` on object `d`, beginning with line `i2`. The string at the bottom of the range is stored at line `i2`, and the remaining strings are stored on successive lines (in other words, strings are stored in ascending order). on The previous contents of the lines are over-written. See also `LMGR-PutBRange` and `LMGR-InsertRange.`

`LMGR-SetCount   ( i s d -- )`

Sets the count of lines for list `s` on object d to `i.`

`locate-prop   ( d s -- d' )`

Given a property name and dbref, locate-prop finds the property, whether on the dbref itself, an environment of the dbref, or a proploc of the dbref. If no matching property is found, returns `#-1.` (Lib-Props)

`long-display   ( d' ... d'' i -- )`

Prints the string form of a range of dbrefs to the user's screen, using unparse to determine the strings' format. The start of the range is printed first. Example:

```
    me @ location get-contents long-display
```

This code duplicates the `Contents` portion of a server look in a room (note: the same effect could be achieved with a single library call to 'list-contents'). (Lib-Look)

`.match_controlled   ( s -- d )`

Searches the vacinity for objects with names matching `s,` like .noisy_match and with similar messages for negative or ambiguous results, but returns `#-1` and prints `"Permission denied."` if the user does not control the found object. (Lib-Match)

`MBOX-append   ( range s1 s2 d -- i )`

Creates a new message with base `s2` on object `d,` with items `range,` on object `d,` appending it to existing messages in the box. The newly-created message's number is returned. (Lib-Mesgbox)

`MBOX-badref?   ( i s d -- i' )`

Returns true if message item `i` in the message with base `s` on object `d` does not exist. (Lib-Mesgbox)

`MBOX-count   ( s d -- i )`

Returns the number of messages contained in message box is on object `d.` (Lib-Mesgbox)

`MBOX-create   ( s d -- )`

Creates a new message box with base `s` on object `d` with no messages in it. This consists of a property `'<base>#/i'` with the string `0` stored in it. (Lib-Mesgbox)

`MBOX-delmesg   ( refnum base dbref -- )`

Delete the given message number in the message box. It moves the rest of the messages after it up in the message box.

`MBOX-destroy   ( s d -- )`

Destroys message box `s` on object `d` and all of its contents. (Lib-Mesgbox)

`MBOX-insmesg   ( range s1 i1 s2 d -- i2 )`

Creates a new message with the given message items and info string and inserts it before the given message number in the message box. Returns the message's number.

`MBOX-message   ( refnum base dbref -- {strrange} )`

Returns the contents of the given message number in the message box as a range of strings.

`MBOX-msginfo   ( refnum base dbref -- infostr )`

Returns the info string of the goven message number in the message box.

`MBOX-ref2num   ( i s d -- i' )`

Returns the absolute message number (the number assigned when the message was created) for the message with base `s` on object `d` with reference number `i` (the reference number of a message is its current position in the message list). Note: the first message created has absolute message number `0.` This is also the value that will be returned if no message at postion `i` exists. Code that uses `MBOX-ref2num` should include error checking to deal with this contingency. See also `MBOX-num2ref` and `MBOX-ref2prop.` (Lib-Mesgbox)

`MBOX-ref2prop   ( i s d -- s' d )`

Returns the propterty name of message number `i` in the message box with base `s` on object `d.` Note: The first message created for a box, number 1, is stored in property `<base>#/0` (example: `"1" "+news" trig MBOX-ref2prop` would return `"+news/0#" #123`). This is also the the data that will be returned if message `i` does not exist. Code that uses `MBOX-ref2prop` should include error checking to deal with this contingency. See also `MBOX-ref2num` and `MBOX-num2ref.` (Lib-Mesgbox)

`MBOX-num2ref   ( i s d -- i' )`

Returns the reference number (the current position in the message list) for the message with base `s` on object `d` that has the absolute number `i` (the number assigned when the message was created). If no such message exits, `0` is returned. See also `MBOX-ref2num` and `MBOX-ref2prop.` (Lib-Mesgbox)

`MBOX-setinfo   ( refnum base dbref -- )`

Sets the info string for the given message number in the message box.

Messages

A message is a set of associated list properties consisting of a 'base' (the list name), one or more 'items' (lines in the list), 'item strings' (the strings stored in the list, usually holding players' dbrefs or names), and an 'information' string (the content of the message). Some Lib-Mesg functions handle 'string ranges': sets of strings adjacent on the stack, followed by their count. Multiple, related messages are stored in a propdir that constitutes a 'message box'. Example:

```
  str /msgs/1#/1:2
  str /msgs/1#/2:3
  str /msgs/1#/3:13
  str /msgs/1#/i:This is the content of a message for players
                 with dbrefs #2, #3, and #13.
```

This message has three items (`1, 2,` and `3`) holding the item strings `"2", "3",` and `"13"` respecitively. Its base is `msgs/1.` Its information string is `"This is the content of a message for players with dbrefs #2, #3, and #13."` The message is stored in message box `msgs.`

`MBOX-setmesg   ( {strrange} infostr refnum base dbref -- )`

Sets the given message number in the given message box to contain the given message items and info string.

`MSG-append   ( s1 s2 d -- )`

Appends message item `s1` to the message with base `s2` on object `d.` (Lib-Mesg)

`MSG-count   ( s d -- i )`

Returns the number of items in message with base `s` on object `d.` (Lib-Mesg)

`MSG-create   ( range s1 s2 d -- )`

Creates a new message with the items in a stringe range and the information string `s1,` stored as list properties with base `s2` on object `d.`

Example: The following code reads list `_staff` from the trigger action onto the stack as a range, and creates a message with the contents of lvar ourString for each staff member.

```
"_staff" trig LMGR-Getlist        (*read list onto stack as range*)
ourString @                       (*put message on stack *)
"st-msgs/" dup trig LMGR-Getcount (*get next available line number*)
1 + intostr strcat                (*use count to create message base*)
trig MSG-create                   (*create message, stored on trig*)

                                  (*note: this same effect could be
                                    achieved more efficiently with
                                    MSG-append or MBOX-append*)
```

Assume that a staff memeber uses the `stnews` command (#555) with the message `"Please check '+read 17'";` that list `_staff` contains 5 lines, each holding a staff member's dbref in string form (`2, 3, 13, 99, and 244`); and that there are currently two other staff messages. In this case, the code would put the following values on the stack:

```
"2", "3", "13", "99", "244", 5, "Please check '+read 17'",
"st-msgs/3", #555
```

Then, `MSG-create` would be called, creating propdir `st-msgs/3#/,` which other functions could use to relay the message to staff members online, at log-in, or when they check staff news. `Ex #555 = st-news/3#/` would show the following values:

```
str /st-msgs/3#/1:2
str /st-msgs/3#/2:3
str /st-msgs/3#/3:13
str /st-msgs/3#/4:99
str /st-msgs/3#/5:244
str /st-msgs/3#/i:Please check 'news staff'
```

The data passed to `MSG-create` would be cleared from the stack. (Lib-Mesg)

`MSG-delitem   ( i s d -- )`

Deletes message item `i` from the message with base `s` on object `d.` (Lib-Mesg)

`MSG-destroy   ( s d -- )`

Clears and removes the message with base `s` from object `d.` (Lib-Mesg)

`MSG-info   ( s d -- s2 )`

Returns the information string for the message with base `s` on object `d.` (Lib-Mesg)

`MSG-insitem   ( s1 i s2 d -- )`

Inserts a new message item with string value `s1` at position `i` into the message with base `s2` on object `d.` (Lib-Mesg)

`MSG-item   ( i s d -- s2 )`

Returns item number `i` from message with base `s` on object `d.` (Lib-Mesg)

`MSG-message   ( s d -- range )`

Reads the items for the message with base `s` on object `d` onto the stack as a string range. (Lib-Mesg)

`MSG-setinfo   ( s1 s2 d -- )`

Sets the `s1` as the information string for message with base `s2` on object `d.` (Lib-Mesg)

`MSG-setitem   ( s1 i s2 d -- )`

Sets the value of item `i` in message `s2` on object do the the string value `1.` (Lib-Mesg)

`.multi_rmatch   ( d s -- d' ... d'' i )`

returns all objects contained by object `d` whose name matches string `s,` with the total number of successful matches. the search string can include wildcard and grouping operators such as those used by the `smatch` primitive. See also `SMATCH.` (lib-match)

`.noisy_match   ( s -- d )`

Checks the room, objects in the room, the player, objects carried by the player, and exits that the player may use for objects whose name matches `s,` returning the dbref. Partial name strings will work, provided that enough characters are specified to distinguish the object from other similarly named objects. The function is 'noisy' in that it supplies explanatory messages for negative or ambiguous results. If no matching object is found, `#-1` will be returned, and the string `"I don't see that here!"` will be printed to the user's screen. If the match is ambiguous, `#-2` will be returned, and the string `"I don't know which one you mean!"` will be printed. (Lib-Match)

`.noisy_pmatch   ( s -- d )`

returns the dbref of the player with name `s.` the player need not be in the vacinity of the user, but the name must be supplied completely (not case-sensitive). the function is 'noisy' in that it supplies explanatory messages for negative or ambiguous results. if no matching player is found, `#-1` will be returned, and the string `"i don't recognize anyone by that name."` will be printed to the user's screen. (lib-match)

Ranges

A number of library functions handle 'ranges': sets of related items adjacent to each other on the stack. The documentation for these functions uses the following terms:

range:
    A set of related items on the stack, and their count. Example: "mink" "otter" "linsang" 3

count:
    The number of items in a range. The count of the range in the above example is 3.

offset:

The number of stack items between the range and the parameters such as 'offset', 'position', & 'number'. That is, how 'deep' in the stack the specified range lies. Example: "mink" "otter" "linsang" 3 "wolf" The range of three items has an offset of 1.

position:
: The position of the first item in a subrange within a range. The first item in a range is at position 1. Example: "wolf" "mink" "otter" "linsang" 3 Within this range, "mink" is at position 1, "linsang" is at position 3, and "wolf" is at position 0.

number:
: The number of items within a to handle (e.g., to copy or delete).

start:
: The first item in a range, farthest from the top of the stack.

end:
: The last item in a range, closest to the top of the stack.

`REF-add   ( d1 s d2 -- )`

Adds dbref `d2` to reflist `s` on object `d1.` If `d1` is already a member of the list, it is moved to the final position in the list. `REF-add` does not check to ensure that `d1` is a valid dbref. (Lib-Reflist)

`REF-allrefs   ( d s -- d' ... d'' i )`

Returns the contents of reflist `s` on object `d` as a range of dbrefs (Lib-Reflist)

`REF-delete   ( d1 s d2 -- )`

Removes dbref `d2` from reflist `s` on object `d1.` (Lib-Reflist)

`REF-editlist   ( i d s -- )`

Puts the user in an interactive editor that allows dbrefs to be added and removed from reflist `s` on object `d.` If `i` is true, only player dbrefs may be added. If `i` is false, objects of any type may be entered. In either case, the editor rejects invalid dbrefs. The reflist must exists, and all dbrefs contained in it must be valid. `REF-editlist` displays the following header when the user enters the editor:

```
To add an object, enter its name or dbref.  To remove an object,
enter its name or dbref with a ! in front of it.  ie: '!button'.
To display the list, enter '*' on a line by itself.  To clear the
list, enter'#clear'.  To finish editing and exit, enter '.' on a
line by itself.Enter '#help' to see these instructions again.
```

(Lib-Reflist)

`REF-filter   ( a d s -- d' ... d'' i )`

Tests each dbref in reflist `s` on object `d` in the filter function at address `a,` returning the those for which the function returns true as a range of dbrefs. The filter function (supplied by your code) should return `1` for true or `0` for false.

Example: the following code tests the dbrefs in reflist `_members,` stored on the trigger action, filtering out garbage dbrefs and returning the others as a range.

```
: CheckRefs
```

```
        ok? if 1 else 0 then
    ;

    : GetMembers
        'CheckRefs
        trig "_members"
        REF-Filter
    ;
```

In function `GetMembers`, `'CheckRefs` (a function named joined with an apostrophe) puts a pointer to the `CheckRefs` function on the stack. The next line reads the reflist onto the stack. `REF-Filter` then tests each dbref in the list, using the test provided in `CheckRefs:` those dbrefs which are currently valid will be returned as a range.

`REF-first   ( d s -- d' )`

Returns the first dbref in reflist `s` on object `d.` (Lib-Reflist)

`REF-inlist?   ( d1 s d2 -- i )`

Returns true (`1`) if dbref `d2` is a member of reflist `s` on object `d1`, or false (`0`) if it is not. (Lib-Reflist)

`REF-list   ( d s -- s' )`

Returns a string containing a comma-separated list of the names of all objects with dbrefs in reflist `s` on object `d.` (Lib-Reflist)

`REF-next   ( d1 s d2 -- )`

Returns the next dbref after `d2` from reflist `s` on object `d1.` If there are no dbrefs after `d1`, `#-1` is returned. (Lib-Reflist)

Reflists

A 'reflist' is string, stored in a property, containing space- and # octothorpe-delimited dbrefs, such as `"#2 #3 #13 #3175 #244".` A reflist will contain only one instance of any one dbref. Like all strings stored in properties, a reflist is limited to 4096 characters (about 500 dbrefs).

`rinstring   ( s1 s2 -- i )`

Returns the position of the last occurance of `s2` in `s1`, case-insensitive. The same capability is provided by the `RINSTRING` primitive. (Lib-Strings)

`safecall   ( x d -- )`

Calls program `d`, passing `x` (usually a string) as an argument. This routine ensures no garbage is left on the stack by the program called, and that the variables `'me'`, `'loc'`, `'trigger'`, or `'command'` are unchanged: even if they are modified by program `d`, they will have their original values after the program call. (Lib-Look)

`setpropstr   ( d s1 s2 -- )`

Sets `d's` property `s1` to the string value `s2`, or removes prop `s1` if `s2` is a null string. Similar capability is provided by the `SETPROP` primitive. (Lib-Props)

`short-display   ( d' ... d'' i -- )`

Calls `short-list` for the range of dbrefs on the stack, then prints `"You see <contents string>"` to the user. *Note*: Contrary to documentation provided with `@view`, `short-display` will crash if not passed a range of valid dbrefs containing at least one member. (Lib-Look)

`short-list   ( d' ... d'' i -- s )`

Returns a range of dbrefs as a space- and comma-separated string, punctuated and formatted intellegently. For example, the contents of a room might be returned as

```
"Kenya, Passiflora, PowerMac 7100, and Sign"
```

(Lib-Look)

`sr-catrng   ( range1 range2 -- range )`

Concatenates two ranges into one range. (Lib-Stackrng)

`sr-copyrng   ( range ... offset number position -- range ... range2 )`

Copies a subrange of `number` items out of a range in the stack, beginning with the item at `position.` For example,

```
"a" "b" "c" "d" 4 0 3 2 sr-copyrng
```

would make take the 4-item stack `"a" "b" "c" "d",` and copy from it, making a new, 3-item range, beginning with the second item in the 4-item range.

```
"a" "b" "c" "d" 4 "b" "c" "d" 3
```

would be left on the stack. (Lib-Stackrng)

`sr-deleterng   ( range ... offset number position -- range' )`

Deletes a subrange of `number` items from a range on the stack, beginning with the item at `position.` For example,

```
"a" "b" "c" 3 "d" 1 2 1 sr-deleterng
```

would delete 2 items, beginning with the first item in the range, from the 3-item range. The range has an offset of 1: that is, there is 1 item (`"d"`) between the range and the parameters for `sr-deleterng.` This would leave

```
"c" 1
```

on the stack. (Lib-Stackrng)

`sr-extractrng   ( range ... offset number position -- range' ... subrange )`

Extracts a subrange of `number` items from a range in the stack, beginning with the item at `position.` The subrange is removed from the original range, and placed on top of the stack as a new range. For example,

```
"a" "b" "c" "d" 4 0 2 2 sr-extractrng
```

would remove 2 items from the 4-item range, beginning with the second item, leaving

```
"a" "d" 2 "b" "c" 2
```

on the stack. (Lib-Stackrng)

`sr-filterrng   ( range function_address -- range' filtered_range )`

Passes each item in a range to the function at `function_address.` If this function returns a non-zero value, `sr-filterrng` removes that item from range and puts it in `filtered_range.` The items in range may be of any type. To determine the address of a function, preceed the function name with an ' apostrophe. For example, the following code extracts the items of type string from all items on the stack.

```
: StringTest
string? if 1 else 0 then
;

: PullStrings
depth 'StringTest sr-filterrng
;
```

Calling PullStrings with

```
#123 "a" "b" 666 "c"
```

should leave

```
#123 666 2 "a" "b" "c" 3
```

on the stack. Unfortunately, this potentially useful function does not work as advertised. The code above will actually leave

```
#123 "a" 666 3 "c" "b"
```

on the stack. (Lib-Stackrng)

`sr-insertrng   ( range1 ... range2 offset position -- range )`

Inserts a subrange into a range on the stack, between the items at `position` and `position + 1.` (Lib-Stackrng)

`sr-poprng   ( range1 -- )`

Removes a range from the stack. Also defined as `popn` in `lib-stackrng.` The same capability is provided by the `POPN` primitive. (Lib-Stackrng)

`sr-swaprng   ( range1 range2 -- i range2 range1 )`

Swaps two ranges on the stack, inserting a 0-length range before the two ranges. There must be `x` items on the stack below `range1,` where `x` is equal to the larger of `range1's` count or `range2's` count. That is,

```
"" "a" "b" 2 "c" "d" "e" 3 sr-swaprng
```

would crash due to stack underflow, but

```
"" "null" "null" "null" "a" "b" 2 "c" "d" "e" 3 sr-swaprng
```

would put

```
"" "null" "null" "null" 0 "c" "d" "e" 3 "a "b" 2
```

on the stack. For this reason, it will often be necessary to copy a 'work space' range with `sr-copyrng` before using `sr-swaprng.` (Lib-Stackrng) `std-index-add   ( d s1 s2 s3 -- d s1 s2 s3 i )`

Like `index-add,` this routine adds name `s2` and value `s3` to index `s1,` stored on object `d.` All four items are left on the stack unchanged, and error code `i` is returned: `1` is `'no error: item added',` `0` is `'error: name already a member of index; index unchanged'.` See also Indexes and `index-add`. (Lib-Index)

`std-index-add-sort   ( d s1 s2 s3 -- d s1 s2 s3 i )`

Alphabetically adds a name/value pair to an index, like `index-add-sort,` but leaves all four items on the stack, returning error code `i:` `1` is `'no error: item added',` `0` is `'error: name already a member of index; index unchanged'.` *In theory*, it should add the name to the index in alphabetical order, but in practice and like `index-add,` it simply adds the name to the end of the index. See also Indexes and `index-add-sort`. (Lib-Index)

`std-index-delete   ( d s1 s2 -- d s1 s2 )`

Removes name `s2,` and its associated value, from index `s1` on object `d,` with no error checking. All three items are left on the stack unchanged. See also Indexes and `index-delete`. (Lib-Index)

`std-index-envmatch   ( d s1 s2 s3 -- d s1 s2' s4 i )`

Performs an environment match like `index-envmatch,` using the match criteria specified in paramater string `s3.` Arguments `d` and `s1` are left on the stack; parameter string `s3` is returned as null string `s4.` See also Indexes and `index-envmatch`. (Lib-Index)

`std-index-first   ( d s1 s2 s3 -- d s1 s2' s4 )`

Takes object to holding index (`d`), an index name (`s1`), and two null strings (`s2` and `s3`). Returns four items, with `s2` being replaced by `s2',` the first name in the index. See also Indexes and `index-first`. (Lib-Index)

`std-index-last   ( d s1 s2 s3 -- d s1 s2' s4 )`

In theory, takes object to holding index (`d`), an index name (`s1`), and two null strings (`s2` and `s3`), returning four items, with `s2` being replaced by `s2',` the last name in the index. In practice, `s2'` will still be a null string. See also Indexes and `index-last`. (Lib-Index)

`std-index-match   ( d s1 s2 s3 -- d s1 s2' s4 i )`

Performs a name match like `index-match,` using the match criteria specified in parameter string `s3.` Arguments `d` and `s1` are left on the stack; parameter string `s3` is returned as null string `s4.` See also Indexes and `index-match`.(Lib-Index)

`std-index-next   ( d s1 s2 s3 -- d s1 s2' s3 )`

Takes object holding index (`d`), an index name (`s1`), a name (`s2`), and a null string (`s3`), returning the four items with `s2` modified to `s2',` the name in the index immediately following `s2.` See also Indexes and `index-next`.(Lib-Index)

`std-index-prev   ( d s1 s2 s3 -- d s1 s2' s3 )`

Takes object holding index (`d`), an index name (`s1`), a name (`s2`), and a null string (`s3`), returning the four items with s2 modified to s2', the name in the index immediately preceding s2. See also Indexes and `index-prev`.(Lib-Index)

`std-index-remove   ( d s1 s2 s3 -- d s1 s2 s3 i )`

Takes object holding index (`d`), an index name (`s1`), a name (`s2`), and a null string (`s3`), and removes name s2 and its associated value from the list. The four initial arguments and an error code are left on the stack. See also Indexes and `index-remove`.(Lib-Index)

`std-index-set   ( d s1 s2 s3 -- d s1 s2 s3 )`

Adds a name/value pair to an index, with no error checking, like index-set. All four items are left on the stack unchanged. See also Indexes and `index-set`.(Lib-Index)

`std-index-value   ( d s1 s2 s3 -- d s1 s2' s4 )`

Performs a match for name `s2` in index `s1` on object `d,` using matching criteria specified in parameter string `s3.` The dbref and index name are left on the stack unchanged. Name `s2` is returned in complete form (partial matches are expanded). Returns the value associated with name `s3` as string `s4.` If the match was unsuccessful, `s4` will be a null string. See also Indexes and `index-value`.(Lib-Index)

`std-index-write   ( d s1 s2 s3 -- d s1 s2 s3 i )`

Sets a value for an existing index member, like `index-write,` but leaves all four items on the stack and returns an error code. See also Indexes and `index-write`.(Lib-Index)

`.std_table_match   ( x1 x2 sn pn ... s1 p2 i sm -- sx px | x1 | x2 )`

The standard table match takes `i` comparator/data pairs in the same manner as `.table_match,` and performs a `SMATCH` on each comparator, matching against string `sm.` Comparators must be strings; it is not necessary to supply a matching functin and its address. `.Std_table_match` returns the comparator/data pair that matchs `sm,` or `x1` if no match was found, or `x2` if more than one match was found. See also `.table_match` and `SMATCH.` (Lib-Match)

`str-desc   ( s -- )`

Prints `s` as a description, matching the `'@###'` and `'@$prog'` values properly, and uses them with the present trigger value. If neither of these exist, or if they are invalid, the string is simply printed. The string is not parsed for `MPI.` (Lib-Look)

`STRasc   ( s -- i )`

Converts character `s` to its `ASCII` number equivalent. The same capability is provided by the `CTOI` primitive. (Lib-Strings)

`STRblank?   ( s -- i )`

Returns true if `s` null string or only spaces. (Lib-Strings)

`STRcenter   ( s i -- s )`

Centers `s` in a field `i` characters wide.

```
    "Welcome to " "muckname" sysparm 78 STRcenter .tell
```

...would display a welcome line including the name of the `MUCK` centered for a standard screen. (Lib-Strings)

`STRchr   ( i -- s )`

Converts `ASCII` code number `i` to its character equivalent. The same capability is provided by the `ITOC` primitive. (*Note*: the header document for Lib-Strings lists the name of this funtions as `STRchar,` but this is incorrect: use `STRchr`.)

`STRfillfield   ( s1 s2 i -- s3 ]`

Returns a string consisting of as many characters `s2` as would be needed to concattenate with `s1` to create a string `i` characters long. Useful for aligning columns of output.

```
      me @ name " " 20 STRfillfield strcat
```

...would return...

```
    "Jessy               "
```

(Lib-Strings)

`STRleft   ( s i -- s )`

Returns `s,` padded with trailing spaces to a string length of `i` characters. (Lib-Strings)

`STRright   ( s i -- s )`

Returns `s,` padded with leading spaces to a string length of `i` characters. (Lib-Strings)

`STRrsplit   ( s1 s2 -- s3 s4 )`

Splits `s1` on last occurence of delimiter string `s2.`

```
"#add here=detail=chair" "=" STRrsplit
```

...would put...

```
"#add here=detail", "chair"
```

...on the stack. If `s1` does not include `s2, s3` will be identical to `s1` and `s4` will be a null string. The same capability is provided by the `RSPLIT` primitive. See also `STRsplit.` (Lib-Strings)

`STRsts   ( s -- s' )`

Strips leading spaces from `s.` The same capability is provided by the `STRIPLEAD` primitive. (Lib-Strings)

`STRsls   ( s -- s' )`

Strip trailing spaces from `s.` The same capability is provided by the `STRIPTAIL` primitive. (Lib-Strings)

`STRsms   ( s -- s' )`

Strips multiple internal spaces from `s.`

```
  "It's a long way to Tipperary" STRsms
```

...would return...

```
  "It's a long way to Tipperary"
```

(Lib-Strings)

`STRstrip   ( s -- s' )`

Strips leading and trailing spaces from `s.` The same capability is provided by the `STRIP` primitive. (Lib-Strings)

```
STRsplit   ( s1 s2 -- s3 s4 ]
```

Splits `s1` at the first occurence of delimiter string `s2.`

```
  "mink otter linsang" "otter" STRsplit
```

...would put...

```
  "mink ", " linsang"
```

...on the stack. If `s1` does not include `s2, s3` will be identical to `s1` and `s4` will be a null string. The same capability is provided by the `SPLIT` primitive. See also `STRrsplit.` (Lib-Strings)

```
.table_match   ( x1 x2 cn pn ... c1 p1 i x3 address -- cx px )
```

This function tests successive pairs of 'comparator' and 'data' elements with a separate function (that you create), returning the pair that (according to your function's criteria) 'matches'. Although `.table_match` is supplied as a matching function, it can have other applications: the comparator/data pairs in effect constitute a hash table, and the 'matching' function can perform whatever operations you require.

`.Table_match` takes parameters `x1` — the item to be returned if no match is found — and `x2` — the item to be returned if more than one pair matches. `x1` and `x2` can be of any type. Example: `#-1` and `#-2` are the conventional items used to indicate negative or ambiguous match results for objects of type dbref. These parameters are followed by comparator/data pairs: the comparator is the item to be tested for a match; the data item is a stack item associated with the comparator. An integer specifying the number of such pairs to be tested is then supplied, followed by the value `x3` — which will be used to test the match — and the address of the testing function. The testing function should return `1` for true results and `0` for false results. `.Table_match` returns the comparator/data pair that matches (`cx px`), or the paramters for negative or ambiguous results (`x1` or `x2`)

Example: Suppose you want a function supplementing `.pmatch` and `.noisy_pmatch` that can find players by 'nicknames', which are often set as a player's `%n` property. One way (among many possibilities) to create such a function would be to supply comparator/data pairs of players' `%n` properties and their dbrefs, and calling `.table_match.` The nickname to be matched is stored in `lvar ourString.` Function `GetPairs` puts the prop/dbref pairs and their count on the stack. Function `CheckNick` performs a `SMATCH,` returning `1` if a comparator matches ourString, and `0` if it does not. Code for calling `.table_match` might then look like this:

```
  #-1 #-2 GetPairs ourString @ 'CheckNick .table_match
```

If a player calls the program with a search for `"Dr. Cat",` and `GetPairs` returns three players who have `%n` nicknames (and thus are candidates for possible matches) the stack created by this code might have values such as the following immediately before `.table_match` is called:

```
  #-1, #-2, "the Scamper Gal", #2, "the terribly velvet-furred linsang",
  #13, "Dr. Cat" #244, "Dr. Cat", 'CheckNick
```

`#-1` and `#-2` are the codes `.table_match` is to return for negative and ambiguous matches. The next six items on the stack are three comparator/data pairs: The dbrefs of players `#2, #13,` and `#244,` paired with their nicknames. The next item (the second occurance of string `"Dr. Cat"`) is the string to be matched, which was fetched from `lvar ourString`. `'CheckNick` — a function name preceded by an `'` apostrophe — is a pointer to the address for the `CheckNick` function.

In this case, the results are not negative or ambiguous: one of the players does have the nickname to be matched: player `#244` is Dr. Cat. `.Table_match` would clear all the above data from the stack, and return...

```
"Dr. Cat", #244
```

See also `.std_table_match.` (Lib-Match)

```
unparse   ( d -- s )
```

Returns the name of object `d`, concattenated with its dbref and flags (such as `"Bulletin Board(#177S)"`) if the user controls object `d`. If the user does not control `d`, or if the user is set `Silent`, only its name is returned (`"Bulletin Board"`). (Lib-Look)

```
when   ( x -- )
```

Performs a `TRUE|FALSE` condition test within a `case` statement. See header for lib-case. (Lib-Case)

4.0 TUTORIALS

This section provides tutorials on how to use the commands and programs discussed throughout *The MUCK Manual* to perform common tasks and projects. Note: in many cases, there is more than one way to accomplish the goal.

4.1 Using the List Editor

Lists (introduced in Commands Overview: Lists) are the `MUCK` equivalent of files or documents: adjacent, related props hold lines of text, which together make up a list. Lists allow you to include formatting information such as paragraph breaks and indentation in descriptions and other text output. Lists are normally created and edited using the list editor, command `lsedit.`.

If you have used a line editor before, such as `UNIX's ex`, the `MUCK` list editor will seem familiar and easy to use. But if your only experience with text editors has been through screen editors such as Notepad or Simpletext, or `WYSIWYG` word processors, the list editor will definitely take some getting used to.

With a line editor, you work with individual lines of text or with groups of lines, rather than seeing all lines at the same time, with your position in the file marked with a cursor. In order to see the contents of the list, you will need to enter a command that causes text to be displayed. Similarly, you need to enter explicit commands to move around in the list and tell the editor which line you are working with.

The following is a walk-through of using the list editor to give a room — a lounge on a high-tech world — a two-paragraph description.

The syntax for lsedit is `lsedit <obj> = <list name>`. `<Obj>` here is the object holding the props that make up the list. We do not necessarily have to store the list on the room, but since the list holds the room's desc, it makes sense to do so. The list name can be anything we like. We'll call it 'desc', just to keep things simple. So, to get started we start the list editor:

```
> lsedit here = desc
<    Welcome to the list editor.  You can get help by entering '.h'     >
< '.end' will exit and save the list.  '.abort' will abort any changes. >
<    To save changes to the list, and continue editing, use '.save'     >
< Insert at line 1 >
```

At this point, we can either type in the text directly, or we can copy text from a text editor on our client computer, and paste it into our `MUCK` client's window (often a better choice).

Let's say that we enter the following description for the room, using either method:

```
  This lounge, one notices immediately, is oddly anacoustic: something
  -- perhaps the walls covered in a short-nub, grey carpet, perhaps the
  somewhat unusual layout of shoulder-high, paper-thin partitions
  between groups of tables, or perhaps the alcoves set into the walls --
  something seems to swallow sound here. Footfalls fall muted;
  conversations coalesce into a low, murmuring susurrus.

  A seemingly continuous tube of cobalt blue neon adorns the room,
  running along the walls eight inhces below the ceiling. Its
  eye-thwarting hue reflects in dabs and streaks from the furniture: the
  tables, chairs, barstools, and the bar itself... sleek amalgamations
  of stainless steel, glass, and buffed black leather. Behind the bar,
  between shelves holding ornate bottles, sits a single bonsai tree, its
  tortured form hilighted in the beam of a hidden drop light.
```

When we enter the text, nothing seems to happen. Did we successfully enter it? To see the contents of our list, we can type `.l` (to list the lines, without line numbers) or `.p` (to print the lines, with line numbers.)

```
> .l
  This lounge, one notices immediately, is oddly anacoustic: something
  -- perhaps the walls covered in a short-nub, grey carpet, perhaps the
  somewhat unusual layout of shoulder-high, paper-thin partitions
  between groups of tables, or perhaps the alcoves set into the walls --
  something seems to swallow sound here. Footfalls fall muted;
  conversations coalesce into a low, murmuring susurrus.
  A seemingly continuous tube of cobalt blue neon adorns the room,
  running along the walls eight inhces below the ceiling. Its
  eye-thwarting hue reflects in dabs and streaks from the furniture: the
  tables, chairs, barstools, and the bar itself... sleek amalgamations
  of stainless steel, glass, and buffed black leather. Behind the bar,
  between shelves holding ornate bottles, sits a single bonsai tree, its
  tortured form hilighted in the beam of a hidden drop light.
  < listed 2 lines starting at line 1 >

> .p
  1: This lounge, one notices immediately, is oddly anacoustic:
  something -- perhaps the walls covered in a short-nub, grey carpet,
  perhaps the somewhat unusual layout of shoulder-high, paper-thin
  partitions between groups of tables, or perhaps the alcoves set into
  the walls -- something seems to swallow sound here. Footfalls fall
  muted; conversations coalesce into a low, murmuring susurrus.
  2: A seemingly continuous tube of cobalt blue neon adorns the room,
  running along the walls eight inhces below the ceiling. Its
  eye-thwarting hue reflects in dabs and streaks from the furniture: the
  tables, chairs, barstools, and the bar itself... sleek amalgamations
  of stainless steel, glass, and buffed black leather. Behind the bar,
  between shelves holding ornate bottles, sits a single bonsai tree, its
  tortured form hilighted in the beam of a hidden drop light.
  < listed 2 lines starting at line 1 >
```

Hmmm... This looks more or less ok, but a couple things have happened to our text as it has been processed by the list editor. Our paragraphs are each currently in one long line, and the blank line that separated them has disappeared.

Truly blank lines get 'swallowed' by your client, and never make it to the `MUCK` or the list editor. So, if we want to put blank lines in a list, we need to put a space or two on the line to hold its place. We decide that we do want a blank line separating our paragraphs here, so we use `lsedit's .insert` command (syntax `.i <line to insert>`), and enter a couple spaces on line 2.

```
> .i 2
  < Insert at line 2>
> ( ...we type a couple spaces here... )
> .p
  1: This lounge, one notices immediately, is oddly anacoustic:
  something -- perhaps the walls covered in a short-nub, grey carpet,
  perhaps the somewhat unusual layout of shoulder-high, paper-thin
  partitions between groups of tables, or perhaps the alcoves set into
  the walls -- something seems to swallow sound here. Footfalls fall
  muted; conversations coalesce into a low, murmuring susurrus.
  2:
  3: A seemingly continuous tube of cobalt blue neon adorns the room,
  running along the walls eight inhces below the ceiling. Its
  eye-thwarting hue reflects in dabs and streaks from the furniture: the
  tables, chairs, barstools, and the bar itself... sleek amalgamations
  of stainless steel, glass, and buffed black leather. Behind the bar,
  between shelves holding ornate bottles, sits a single bonsai tree, its
  tortured form hilighted in the beam of a hidden drop light.
  < listed 3 lines starting at line 1 >
```

We could leave our text in long lines, one per paragraph, but sometimes clients have trouble with line wrapping. If we leave it like this, it's possible that a few players will not be able to see all the text. `Lsedit's .format` command (syntax `.format <start line> <end line> = <columns to format to>`) will break the lines up into reasonable lengths.

```
> .format 1 3 = 76
  < Formatted 3 lines starting at line 1 (Now curr line) to 76 columns. >
> .p
  1: This lounge, one notices immediately, is oddly anacoustic: something --
  2: perhaps the walls covered in a short-nub, grey carpet, perhaps the somewhat
  3: unusual layout of shoulder-high, paper-thin partitions between groups of
  4: tables, or perhaps the alcoves set into the walls -- something seems to
  5: swallow sound here. Footfalls fall muted; conversations coalesce into a
  6: low, murmuring susurrus.
  7:
  8: A seemingly continuous tube of cobalt blue neon adorns the room, running
  9: along the walls eight inches below the ceiling. Its eye-thwarting hue
 10: reflects in dabs and streaks from the furniture: the tables, chairs,
 11: barstools, and the bar itself... sleek amalgamations of stainless steel,
 12: glass, and buffed black leather. Behind the bar, between shelves holding
 13: ornate bottles, sits a single bonsai tree, its tortured form hilighted in
 14: the beam of a hidden drop light.
< listed 14 lines starting at line 1 >
```

So, we successfully inserted a blank line, separating our paragraphs. But on second thought, this isn't how we want to format it. The blank line will break things up too much: the first paragraph will seem to 'go with' the name of the room, and the second paragraph will seem to 'go with' the room's contents list. The desc itself won't look like a unified whole on the screen. So, we decide instead to delete the blank line (using `lsedit's .delete` command, syntax `.del <line to delete>`), and indent the paragraphs two spaces, using `lsedit's .indent` command.

The syntax for `.indent` is `.indent [<first line to indent> [<last line to indent>]] = <number of spaces to indent>.` That looks somewhat complicated when expressed formally, but it's not difficult in practice.

Some examples...

This indents the current line by five spaces:

```
> .indent 5
  < Indenting 1 lines starting at line 2, 2 columns. >
```

This indents line 14 by five spaces:

```
> .indent 14 = 5
  < Indenting 1 lines starting at line 14, 2 columns. >
```

This indents lines 18 to 20 by five spaces:

```
> .indent 18 20 = 5
  < Indenting 3 lines starting at line 18, 2 columns. >
```

So, back to our main example: in the list editor, working with our 'desc' list for a room, we want to delete line 7 (the blank line) and indent the two remaining lines by two spaces.

```
> .del 7
  < Deleting 1 lines, starting at line 7  (Now current line) >
> .indent 1 = 2
  < Indented 1 lines starting at line 1, 2 columns >
> .indent 7 = 2
  < Indented 1 lines starting at line 7, 2 columns >
> .l
    This lounge, one notices immediately, is oddly anacoustic:
  something -- perhaps the walls covered in a short-nub, grey carpet,
  perhaps the somewhat unusual layout of shoulder-high, paper-thin
  partitions between groups of tables, or perhaps the alcoves set into
  the walls -- something seems to swallow sound here. Footfalls fall
  muted; conversations coalesce into a low, murmuring susurrus.
    A seemingly continuous tube of cobalt blue neon adorns the room,
  running along the walls eight inhces below the ceiling. Its
  eye-thwarting hue reflects in dabs and streaks from the furniture: the
  tables, chairs, barstools, and the bar itself... sleek amalgamations
  of stainless steel, glass, and buffed black leather. Behind the bar,
  between shelves holding ornate bottles, sits a single bonsai tree, its
  tortured form hilighted in the beam of a hidden drop light.
```

OK, this is looking good. But we need to make one last change. In the process of working with the list, we notice a typo in line 8 of the second paragraph: `'inhces'` should be `'inches'.` We want to fix that, and would rather not have to retype everything. The right tool for the job here is `lsedit's .replace` command (syntax `.repl <line number> = /<old string>/<new string>`)

```
> .repl 8 = /inhces/inches
  < Replaced.  Going to line 8 >
  8: along the walls eight inches below the ceiling. Its eye-thwarting
  hue
```

Now, finally, an important point, we need to *get out of the list editor*. (Forgetting to get out of the list editor, and typing a bunch of commands into the list you were just working on is a fairly common mishap.) To do so, type `.end,` on a line by itself.

```
> .end
  < Editor exited. >
  < list saved. >
```

So we're done! Or are we? We do a `'look'` to see the final product, and get a disappointment. The desc doesn't show up. So far, we have created the list, but we haven't told the server to display this list as the room's desc. To do so, we need to set the room's desc with a bit of `MPI` that will cause the newly created list to be displayed when people look at the room.

```
> @desc here = {list:desc}
  Description set.
> l
    This lounge, one notices immediately, is oddly anacoustic:
  something -- perhaps the walls covered in a short-nub, grey carpet,
  perhaps the somewhat unusual layout of shoulder-high, paper-thin
  partitions between groups of tables, or perhaps the alcoves set into
  the walls -- something seems to swallow sound here. Footfalls fall
  muted; conversations coalesce into a low, murmuring susurrus.
    A seemingly continuous tube of cobalt blue neon adorns the room,
  running along the walls eight inches below the ceiling. Its
  eye-thwarting hue reflects in dabs and streaks from the furniture: the
  tables, chairs, barstools, and the bar itself... sleek amalgamations
  of stainless steel, glass, and buffed black leather. Behind the bar,
  between shelves holding ornate bottles, sits a single bonsai tree, its
  tortured form hilighted in the beam of a hidden drop light.
```

Voila! We're done.

4.2 Making a Multi-Action

Over time, many players end up creating several handy little macro commands that provide shortcuts for frequent tasks, or create rooms that have a number of specific, local commands. Rather than creating a separate action for each command (an approach that quickly leads to dbase bloat and eats up your quota), you can combine a number of commands in a single action. The basic technique is to give the action an alias name for each command, store `MPI` strings in properties with the same names as the aliases, and use the command name typed by the user to determine which property should be executed.

The following example creates a multi-action that lets you determine the dbref of something in the same room, enter or review a page of 'to do' notes, and remotely lock or unlock the door into your home. First, create the action attached to your character, and lock it to a condition that always fails. Since the action is attached to you, no one else should be able to use it... Nonetheless, it would be a good idea to secure the exit by linking it to something, such as a do-nothing program.

```
> @act ref;note;notes;lockhome;unlockhome = me
  Action created with number 9456 and attached.
> @lock ref = me&!me
  Locked.
> @link ref = $nothing
  Linked to gen-nothing.muf(#363FLM2).
```

The lock will fail unless the user *is* and *is not you*... it will always fail. Now, set the action's `@fail` with an `MPI` string that will cause other `MPI` strings stored in properties on the object to be executed.

```
> @fail ref = {exec:{&cmd}}
  Message set.
```

`{&cmd}` is a variable... it holds whatever command name the user typed. If you used the action by typing `'ref',` then the string `'ref'` would be substituted for `{&cmd}` when the `@fail` is parsed; if you used the the action by typing `'notes',` then `'notes'` would be substituted for `{&cmd}.`

`{exec}` evaluates the property following it, executing any `MPI` contained in it, and returning any resulting strings to the user. So, typing `'notes'` would cause the `'notes'` property on the object to be evaluated: `{&cmd}` would be replaced by `'notes',` and `{exec}` would use this value to determine which property it should evaluate.

Now, set a property for each alias, containing `MPI` that performs a function. One of the simplest ways to do this, is to force yourself to use a command. In order to do so, you need to be set `X(forcible)` and `force_locked` to yourself.

```
> @set me = X
  Flag set.
> @flock me = me
  Force lock set.

> @set ref =ref:{ref:{&arg}}
  Property set.
> @set note =note:{force:me,lsedit notes=notes}
  Property set.
> @desc note ={list:notes}
  Description set.
> @set notes =notes:{force:me,look notes}
  Property set.
> @set lockhome =lockhome:{force:me,@lock #17212=me&!me}
  Property set.
> @set unlockhome =unlockhome:{force:me,@unlock #17212}
  Property set.
```

The one action can now be used to do several different things. Typing `'ref here'` would show the dbref of the room you are in (`{&arg},` like `{&cmd},` a variable: it holds whatever was typed following the command name. If you typed `'ref here',` the `&arg` variable would hold the string `'here'`). Typing `'note'` would force you to use `lsedit` to edit your page of notes. Typing `'notes'` would force you to look at the notes: the list would be printed on your screen. Typing `'lockhome'` would force you to lock the exit into your home, specifying it by dbref so that it doesn't matter where on the `MUCK` you are located. Typing `'unlockhome'` would unlock the exit in the same way.

(Notice that several of the actions world by 'forcing' you to do something. This is an easy way to make little macro commands. However, be aware that in order for it to work, you must be set 'Xforcible', and you must be 'force_locked' to yourself. So, do the following as well:

```
> @set me=X
  Flag set.
> @flock me=me
  Locked.
```
)

A single multi-action like this can be extended indefinitely, by adding aliases and a corresponding prop. To add a 'look at my watch and check the time' function, you could rename the action and set a 'watch' property that uses simple `MPI.`

```
> @name ref = ref;note;notes;lockhome;unlockhome;watch
  Name set.
> @set watch = watch:You glance at your watch. The time is {time}.
  Property set.

> watch
  You glance at your watch. The time is 14:18:46.
```

4.3 Making Puppets

A puppet (or 'zombie') is a player-like object of type thing, set up so that it can move and act independently, relaying everything it sees and hears to the controlling character.

In the strictest sense, all that is required to turn an object into a zombie is to set its `Z(ombie)` flag. This will cause it to relay messages, and to be treated as a zombie by programs which distinguish between zombies and players. In practice, two other steps are required to create a working puppet: locking and setting the puppet so that it can be forced, and creating an action to control it. First, create the puppet object. Since the puppet will often be in a different location than your character, it's a good idea to give it a registered name as well. Then, set its `Z(ombie)` flag.

```
> @create Squiggy == pup
  Squiggy created with number 128629.
  Registered as $pup
> @set squig = Z
  Flag set.
```

Next, set the puppet's `X(forcible)` flag, and `force_lock` it to you.

```
> @set squiggy = X
  Flag set.
> @flock squiggy = me
  Force lock set.
```

It would be possible to stop at this point, and use the `@force` command to make the puppet do what you want.

```
> @force $pup =:jumps!
  Squiggy jumps!
```

In practice, it will be more convenient to create an action that simplifies frequent typing. Use an action name that's short and easy to type, and won't conflict with other common exit names. `'Z'` is a frequently used command name for controlling a zombie. Then, link the action to a do-nothing program, lock it to a condition that always fails, such as `'me&!me',` and set its fail with an `MPI` string that forces the puppet.

```
> @act z = me
  Action created with number 128630 and attached.
> @link z = $nothing
  Linked to do-nothing.muf(#197FLM2)
> @lock z = me&!me
  Locked.
> @fail z = {force:$pup,{&arg}}
  Message set.
```

`'&arg'` is a variable: it holds whatever was typed after the command. If you typed `'z` :jumps!', `&arg` would hold the string `':jumps!'.` The fail set on the control action would force the puppet with `':jumps!'.`

```
> z :jumps!
  Squiggy jumps!
> z out
  Squiggy heads out to the park.
  Squiggy has departed.
  Squiggy> You head out to the park...
  Squiggy> Alcot Park
  Squiggy> A sweep of close-cropped green grass extends...
```

As indicated in the example above, output from the puppet is preceded with the puppet's name, and a > greater than symbol to distinguish it from your own output. This prefix string can be changed with the `@pecho` command: `@pecho <puppet object> = <prefix string>`

```
> @pecho squiggy = *
  Message set.
> z look
  *Alcaot Park
  *A sweep of close-cropped grass extends...
```

4.4 Making Vehicles

`MUCK` has a rather limited set of server commands for using vehicles: objects configured such that players may get inside them, and see what happens outside the vehicle. Many `MUCKs` also have more elaborate `MUF` programs for vehicles. But, since these will vary widely from `MUCK` to `MUCK,` only the server vehcile commands are discussed here.

A vehicle is in many ways much like a puppet. Whereas a puppet is created by setting the `Z(ombie)` flag on an object of type `THING`, a vehicle is created by setting the `V(ehicle)` flag on an object of type `THING`. Puppets broadcast anything hearable in their location to their owner; vehicles broadcast anything hearable in their location to all players inside the vehicle. The preface string marking puppet output is set with the `@pecho` commmand; the preface string marking vehicle output is set with the `@oecho` command. As with puppets, it is usually necessary to `force_lock` the vehicle to users who are allowed to drive (or fly or steer) it, and to create an action that implements this. One significant difference between puppets and vehicles: you need to create an action that lets you get inside a vehicle. This done by creating an action that is both attached and linked to the vehicle:

```
> @create 1967 Corvette Sting Ray
  1967 Corvette Sting Ray created with number #558.
> @set 1967 = V
  Flag set.
> @act getin = 1967
  Action created with number #559 and attached.
> @link getin = 1967
  Linked to 1967 Corvette Sting Ray(#558V)
> drop 1967
  Dropped.
> getin
  1967 Corvette Sting Ray(#558V)
```

To describe the interior of the vehicle, use the `@idesc` command:

```
> @idesc here = The 'vette's interior is pristine: gleaming chrome,
  smooth blue vinyl, and rubbery floormats.
> look
  The 'vette's interior is pristine: gleaming chrome, smooth blue
  vinyl, and rubbery floormats.
```

You do not need to create an action that takes you out of the vehicle: the server command `leave` will do this for you.

You do, however, need to set up an action that will let you control the vehicle. The action will work by forcing the vehicle, so it needs to be set `X(forcible)` and `force_locked` to you.

```
> @set 1967 = X
  Flag set.
> @flock 1967 = me
  Force lock set.
```

If you wanted anyone to be able to drive the vehicle, you would need to set the force lock to something that would succeed for all players, such as `@flock 1967 = !#0`.

To make an action that controls the vehicle, create an action as normal, attached to the vehicle, lock it to a condition that always fails, and set its `@fail` message with `MPI` that forces the vehicle to act as specified by the action's argument:

```
> @act drive = 1967
Action created with number 560 and attached.
```

```
> @lock drive = me&!me
Locked.
> @fail drive = {force:#558,{&arg}}
Message set.
> drive :vroom vrooOOOOmms!
Outside> 1967 Corvette Sting Ray vroom vrooOOOmms!
> drive n
Outside> ( ... the vette goes north; we see the appropriate output )
```

By default, all emits outside the vehicle will be broadcast to the vehicle's interior, prepended with the string `outside>.` You can change this prepend string with the `@oecho` command:

```
  (Outside the vehicle, Holbrook waves.)
  Outside> Holbrook waves.

> @oecho 1967 = >>>
  Message set.

  (Holbrook waves again for us.)
  >>> Holbrook waves again.
```

Note: Soft-coded look, pose, and say commands may alter the behavior of vehicles from what is described here.

4.5 Creating Rooms and Areas

Quality `MUCK` building requires a blend of creativity and technical savvy. At a minimum, you should ensure that rooms in your areas are literately desc'd, and that exits have the standard message props (`@desc`, `@succ`, `@osucc`, and `@odrop`).

The building commands are discussed elsewhere in the manual. What follows is a narration of building a village inn, with cross-references to relevant commands. The process should illustrate most of the issues a builder will face. Only two rooms are discussed, but the same techniques could be used over a much larger area.

Creating an Environment Room:

If you are building an area, as opposed to one or a few rooms for personal use, it's a good idea to parent the area to an environment room. Doing so will allow you to create commands that can be used throughout the area, and will let you do a backup of your work with a single `@archive` command. As discussed in Section 2.2, rooms by default have the same parent room (or environment room) as the room from which they are are created. A room can be reparented by `@teleporting` it to a new environment room. A reasonable approach to building an area is to `@dig` an environment room, `@dig` the first room in the area with the just-created environment room as its parent, and move to the new room before creating additional rooms in the area. This way, standard `@dig` commands will cause the additional rooms to be parented to the area's environment room.

```
> @dig Amberside Environment Room  (create the environment room)
  Amberside Environment Room created with room number 4801.
> @register #4801 = aer  (give it a registered name for easy access)
  Now registered as _reg/aer:Amberside Environment Room(#4801) on
  Shadowkat(#1131PBJ)
> @set $aer = A  (set env room Abode, so it will show up in @trace)
  Flag set.
> @dig Amberside Inn: Tavern = $aer
  Amberside Inn: Tavern created with room number 4802.
  Trying to parent...
  Parent set to Amberside Environment Room(#4801RA).
> @register #4802 = ai   (register this too, since we'll be going there)
  Now registered as _reg/ai:Amberside Inn: Tavern(#4802) on
  Shadowkat(#1131PBJ)
> @tel me = $ai   (go there)
  You feel a wrenching sensation...
  Amberside Inn: Tavern(#4802R)
  Teleported.
```

If system parameters on your `MUCK` are set such that you cannot teleport to your new room, it will be worthwhile to create a personal exit that takes you there, for use while you are building the area. After the area is built and linked to the rest of the `MUCK,` you can recycle it.

```
> @act ai = me
  Action created with number 4803 and attached.
> @link ai = $ai
  Linked to Amberside Inn: Tavern(#4802)
> ai
  Amberside Inn: Tavern(#4802R)
> @trace here
  Amberside Inn: Tavern(#4802R)
  Amberside Environment Room(#4801RA)
  Environment: Lowlands(#285RA)
  Rainforest Parent Room(#121RWA)
  Rainforest: Main Parent(#118RA)
  Master Environment(#101RA)
  **Missing**
```

The Amberside Environment Room is parented to `Environment: Lowlands(#285RA)`, because we were under it when we issued the first `@dig` command. This may be where we want it when the area is finished, or there may be some other area of the `MUCK` that is more suitable. After the area is finished, you would contact the `MUCK's` builder wiz and discuss where to put the area. For now, the current location should be fine. Additional rooms that we `@dig` while under `The Amberside Environment Room(#4801RA)` will automatically be parented to there.

```
> @dig Amberside Inn: West Wing
  Amberside Inn: West Wing created with room number 4804.
> @trace #4804
```

```
  Amberside Inn: West Wing(#4804R)
  Amberside Environment Room(#4801RA)
  Environment: Lowlands(#285RA)
  Rainforest Parent Room(#121RWA)
  Rainforest: Main Parent(#118RA)
  Master Environment(#101RA)
  **Missing**
```

One way we could make use of the enivonment room is to provide a command available throughout the area we are building, and only in that area. If we make an action called `map;look map;loo map;lo map;l map`, attaching it to the environment room, and setting its `@succ` to display a list with and ASCII map of the area, users will be able to view a map of the area from anywhere within it.

```
> @act map;look map;loo map;lo map;l map = $ae
  Action created with number 4805 and attached.
> @link map = $nothing
  Linked to do-nothing.muf(#97)
> @succ map = {list:the_map,this}
  Message set.
> lsedit map = the_map

     ( insert map )
```

Creating and Propping Exits:

To avoid leaving floating rooms lying about, it's a good idea to create the exits linking rooms immediately after `@digging` the rooms. As discussed in Section 2.3, exit names can include aliases: alternate names, separated by semi-colons, that can be used in place of the full name. The first alias is the one that will be shown by obvious-exits programs. A common and worthwhile convention is to set the first alias as a combination of the full name and a notation indicating valid abbreviated aliases.

And, since putting message props on exits is the kind of niggling detail that's easy to forget if you leave it to later, it's also a good idea to set the message props immediately after `@opening` the exits.

```
> @open West Wing <W>;west wing;west;w = #4804
  Exit opened with number 4805.
  Trying to link...
  Linked to Amberside Inn: West Wing(#4804R)
> @desc w = A sturdy wooden door leading to rooms on the west wing.
  Description set.
> @succ w = You pull open the door to the West Wing hallway...
  Message set.
> @osucc w = goes into the West Wing.
  Message set.
> @odrop w = comes in from the Tavern.
  Message set.
```

Exits are one way: in order to create a 'door' between rooms, you need two exits, one leading in each direction. So, we complete the door between the Tavern and the West Wing by going to the West Wing and making another exit leading back to the Tavern. We gave the Tavern the registered name 'ai', so it will be easy to link back to. But, if we had not done this, and the room's dbref had scrolled off the screen, we could determine it with the `@find` command.

```
> w
  You pull open the door to the West Wing hallway...
  Amberside Inn: West Wing(#4804R)
> @find tavern
  Amberside Inn: Tavern(#4802R)
  ***End of List***
  1 objects found.
> @open Tavern ;tavern;t;east;e = #4802
  Exit opened with number 4806.
  Trying to link...
  Linked to Amberside Inn: Tavern(#4802R)
> @desc t = A strong wooden door.
  Description set.
> @succ t = You open the door into the inn's tavern.
  Message set.
> @osucc t = goes into the Tavern.
  Message set. > @odrop t = comes out of the west hallway.   Message set.
> t
  You open the door into the inn's tavern.
  Amberside Inn: Tavern(#4802R)
```

We gave the exits user-friendly aliases, but the exits are not showing up at all when we look at the room. The server `'look'` command, and most user-created `look` programs as well, do not show obvious exit lists by default. In order to have such a list appended to our rooms' descriptions, we will need to set the `@succ` message on the rooms to trigger an obvious-exit programs. You will probably need to page a wizard or helpstaff member to determine the dbref or registered name of such a program. As discussed in Section 2.1.4, the dbref or regname of `MUF` programs triggered by message props need to be prefaced by an @ at-mark.

```
> l
  Amberside Inn: Tavern(#4802R)
> p jessy = Hiya... Can you tell me the dbref or regname of the
  obvious exits program?
  You page, "Hiya... Can you tell me the dbref or regname of the
  obvious exits program?" to Jessy.
  Jessy pages, "It's dbref #183... reg name $obvex."
> p = OK, thanks.
  You page, "OK, thanks." to Jessy.
> @succ here = @$obvex
  Message set.
> l
  Amberside Inn: Tavern(#4802R)
  Obvious Exits:
    West Wing <W>
```

We will need to set this `@succ` message on all rooms in the area for which we want obvious-exit lists.

Describing Rooms:

Room descs are a somewhat demanding genre. You need to convey a sharp sense of place in a very brief space. You often need to describe a number of rooms with similar characteristics without becoming repetitive. The description often needs to incorporate cues that geographically orient the player. Since other events online are competing for the reader's attention, your writing needs to be exceptionally clear. The following discussion first covers technical aspects of setting room descs, then offers some editorial suggestions for writing them.

As with any other object, the description of a room can be set with the `@desc` command. And, as with any other object, you may want to do some hanky-panky with `lsedit` and `MPI` to provide formatting (indentation, paragraph breaks, etc.) or flexibility (dynamically generated text describing the time of day, players present, or random events). A desc string bracketted with an `MPI` `{eval}` function will cause any `MPI` in the desc itself to be parsed. The following description of the tavern uses this technique to include the names of present players in the description.

```
> @desc here = {eval:{list:maindesc,here}}
  Description set.
> lsedit here = maindesc
  <    Welcome to the list editor. You can get help by entering '.h'    >

  < '.end' will exit and save the list. '.abort' will abort any changes.
>
  <    To save changes to the list, and continue editing, use '.save'   >

  > Insert at line 1>
    The tavern is little changed from days when pirate sloops found haven
  in the coves nearby: the beams are still low and smoke-stained, the
tables
  still rough-hewn and knife-hacked, the floorboards still trecherously
  uneven. What light there is comes from three flickering oil-lamps and a
  sharp-scented wood fire burning steadily on the hearth. Glasses and
  bottles along the back wall catch and toss the dim, moving light. The
  innkeeper standing easily behind the bar is a grave of secrets.
  {eval:{prop:_folks,here}}
  .end
  < Editor exited. >
  < list saved. >
> @set here=_folks:{if:{commas:{lremove:{contents:here,player},
{owner:me}}},
  {commas:{lremove:{contents:here,player},{owner:me}}, and ,item,{name:
  {&item}}} are here\, at various places in the room,Currently\, you're
the
  only other one here}.
  Property set.
```

If you don't want formatting such as paragraphs or the two-space indentation in the example above, it would be more efficient *not* to embed the desc in a list

```
> @desc here = The tavern is little changed from days when pirate sloops
  found haven in the coves nearby: the beams are still low and smoke-,
  stained, the tables still rough-hewn and knife-hacked, the floorboards
  still trecherously uneven. What light there is comes from three
flickering
  oil-lamps and a sharp-scented wood fire burning steadily on the hearth.
  bottles along the back wall catch and toss the dim, moving light. The
  innkeeper standing easily behind the bar is a grave of secrets.
  {eval:{prop:_folks,here}}
  Description set.
```

Including queues that indicate looktraps and local commands is a common and helpful convention. A clueful `MUCKer` would deduce from the following room desc that it's worth taking a look at the innkeeper and sign:

```
> @desc here = The tavern is little changed from days when pirate sloops
  found haven in the coves nearby: the beams are still low and smoke-,
  stained, the tables still rough-hewn and knife-hacked, the floorboards
  still trecherously uneven. What light there is comes from three
flickering
  oil-lamps and a sharp-scented wood fire burning steadily on the hearth.
  bottles along the back wall catch and toss the dim, moving light. The
  'innkeeper' standing easily behind the bar is a grave of secrets.
  {eval:{prop:_folks,here}} There is a small 'sign' sitting
  on the end of the bar.
  Description set.

> @open ip;is = $nothing ( create a local command )
  Exit opened with number 4808.
  Trying to link....
  Linked to do-nothing.muf(#97)
> @succ ip = {exec:{&cmd}}
  Message set.
> @set ip = ip:{null:{tell:Ishmael {&arga}},{otell:Ishmael {&arg}}}
  Property set.
> @set is = is:{null:{tell:Ishmael growls, "{&arg}"},{otell:Ishmael
growls, "{&arg}"}}
  Property set.
> @set is = D   ( set the exit dark, so it won't show up as an obv-exit )

    ( set some looktraps to indicate the presense of local commands )
> @set here=_details/innkeeper:Call him Ishmael. This ancient wolf looks
  old enough to have been on a first name basis with the aforementioned
  pirates. ('look sign')
> @set here=_details/sign:{list:signtext}
  lsedit here = signtext
  <    Welcome to the list editor. You can get help by entering '.h'
>
  < '.end' will exit and save the list. '.abort' will abort any changes.
>
  <    To save changes to the list, and continue editing, use '.save'
>
  < Insert at line 1 >

  The innkeeper, Ishmael, is interactive:

    ip <pose> ....... Ishmael <pose>
    is <comment> .... Ishmael growls, "<comment>"

  .end
  < Editor exited. >
  > list saved. >

> l sign

  The innkeeper, Ishmael, is interactive:

    ip <pose> ....... Ishmael <pose>
    is <comment> .... Ishmael growls, "<comment>"

> is What will you have, stranger?
  Ishmael growls, "What will you have, stranger?"
```

(Note that we avoided creating a puppet object for Ishmael... the room desc says that he's here, and a named object in the room's contents list is not really necessary for our purposes. Also, the two commands — 'ip' and 'is' — are created using one exit object, and we could modify the action to provide additional local commands using the multi-action technique discussed in Section 4.2. It is especially worthwhile to follow such dbase-conserving practices when building large areas.)

Some builders feel that the obvious exit list after room descs is intrusive and artificial looking. If you decide not to use an obvious exit list, you should include text in the desc that will tell a careful reader which ways she can go.

```
> @desc here = The tavern is little changed from days when pirate sloops
  found haven in the coves nearby: the beams are still low and smoke-,
  stained, the tables still rough-hewn and knife-hacked, the floorboards
  still trecherously uneven. What light there is comes from three
flickering
  oil-lamps and a sharp-scented wood fire burning steadily on the hearth.
  bottles along the back wall catch and toss the dim, moving light. The
  innkeeper standing easily behind the bar is a grave of secrets.
  {eval:{prop:_folks,here}} There is a small 'sign' sitting
  on the end of the bar. A sturdy wooden door opens onto a hallway
  to the west; another, at the front of the tavern, opens onto the
  village green.
  Description set.
```

Modern graphic workstations can display a huge amount of text, but the standard computer terminal only displays a screen 80 characters wide and 24 lines tall. While standard character terminals may seem obsolete to some, it's worthwhile to respect this 'lowest common denominator': a number of people do still access `M*'s` through standard terminals... either `UNIX`/Linux computers on which they have chosen not to install or run graphics, or virtual terminals being run from a shell account. Further, the vast majority of these `UNIX` folks will be using TinyFugue as their client. With TinyFugue's 'visual' mode turned on, the bottom four lines are used as a typing window. Additionally, not all clients can display the full width of 80 characters. So, if you want people to be able to see the entire desc in one screen (a good thing to strive for), you should limit descriptions to a 78 x 20 field of text. You can choose to violate this guideline, of course, and often will be justified in doing so... but be aware of the tradeoff you make.

A further 'less-is-more' point: a common mistake made by new builders is to create rooms that serve only to provide a way to get from point A to point B. A forest path leading from the stream to the glade, a foyer leading from the street to office lobby, etc. Such rooms (and the extra exits required to link them) seldom earn their keep: they take up extra space in the database, eat up your quota, and make the area feel bland or 'loose'. As an alternative, consider setting detailed `@succ` and `@drop` messages that convey a sense of travel over a distance. Or, in cases where the effect is still too fast and a greater sense of distance is required, consider writing or using a program that generates a travel event of some sort (one such program is provided here).

Writing Room Descriptions:

Your rooms are your rooms: *The MUCK Manual* makes no claim to be an authoritative source on the 'right' way to write room descriptions. At the same time, a few basic techniques — some of which would be presented in any creative writing course, and some arising from the specific nature of this very specific genre — may prove helpful as you work to make your building memorable and enjoyable.

Favor the Specific over the General

As does a narrative essay or work of fiction, a room desc strives to evoke a sharp sense of place... to give the reader the information necessary to vividly imagine the scene and to vicariously experience it, as though she were there. Specific information serves this purpose much more effectively than general information. It is much easier to visualize 'a white picket fence clambered with Morning Glory' than it is to visualize 'a fence'... and this is true even if we don't know exactly what a Morning Glory looks like. When faced with a choice (and writing is in its essence a continuous stream of choices), always give the specific instance first consideration over the general case. Put the 'top-heavy dahlia, just beginning to lose its petals' into your desc, rather than the 'flower'. Furnish the conference room with 'wobbley chairs with stainless steel legs and orange naugahide seats, that look like they were stolen out of dorm rooms', rather than 'seats' or 'chairs'. Let your too-slick cardshark smoke a 'Romeo y Julietta Bellicoso' or 'Montecristo No. 2' rather than simply 'a cigar'.

Appeal to Multiple Senses

Many builders rely too heavily on the sense of sight. Use this to your advantage: make your own rooms stand out, favorably, in contrast, by appealing to multiple senses. Do include visual description — show your reader the tropical water's seductive shadings of blue and green and the fractured dance of sunlight on its surface — but complement these with the tawdry and insistent call of the seagulls, the weighty smell of seaweed, the congenial contest between warm sunlight and a chill seabreeze, and the wholesome tang of salt when an unexpected burst of spray gifts the lips.

Use Active Verbs

The much-maligned passive voice (using some form of 'to be' as the verb of a sentence) has valuable uses in all forms of writing, and one should regard prose purists who insist that is inherently inferior with grave suspicion. Nonetheless. In place descriptions, active verbs do help bring energy and verve to the writing, especially when the subject matter itself is passive or static. Consider the following:

> The gentle surf polishes the tiny bits of jade, amethyst, and sapphire that dot the expanse of dark obsidian sand, extending forever out under the clear bluegreen water of the Dragon Sea. Transparent, crystalline crabs dance over the sands, plucking a particularly scintillating grain now and then for the tiny hoards upon their backs. Sweeping back towards the northern woods, the warm black sand borders square patches of cool grass, following the checkered pattern of a harlequin's breast. The sand stretches to the west and east, past a low outcropping of obsidian to the southeast, where the entrance to a little cove is visible. Various homes can be seen interspersed with the trees to the north. Hanging low in the southern sky, you can make out the silhouette of the Floating Island. A directory of the local homes is posted on a tree.
>
> (Cymoryl's Shimmer Beach, on FurryMUCK)

The surf 'polishes' the stones. The crabs 'dance' and 'pluck'. The shoreline 'sweeps' this way, and 'stretches' that way. There is little doubt that this desc would have been much less effective had Cymoryl relied on the passive voice.

Use Contrast

When trying to convey a specific tone or feeling, guard against sounding the same note throughout the description. Instead, use contrast to focus the reader's awareness of the desired tone or feeling. Consider: if we want to convey a sense of snug warmth, we *could* write a description saturated with details such as warm blankets, coals in the fireplace, thick walls, and so forth. But the reader might be better able to experience the desired effect if we instead provide several *contrasting* details — cold drafts lurking about the baseboards, and a still, star-lit night outside, with ice-flowers of frost forming on the window — and then provide close narration of being bundled in ample quilts, with only our noses peeking out.

Consider Providing a 'Focus'

Most of the guidelines offered so far would apply to narrative writing at least as strongly as they do to desc writing. But in some ways the demands of the two forms are very different. One of these differences has to do with 'pace' or 'rhythm'. In narrative, you usually want to create a sense of flow and continuity. `MUCK` rooms, however, exist 'by themselves' to a greater extent than paragraphs or scenes from a narrative. Rooms are discrete 'pieces', that need to be judged on their own merits, with relatively little consideration to what went before or comes after (something you cannot control in the same way that a narrative writer can).

Sometimes, we take special care to convey a room's relationship to other nearby places (we describe the path leading down to the stream, the hill just above this field, the ridge leading on to the plateau) and we describe a number of aspects of the room itself (the flowers speckling through the field's grass, the surrounding trees, the sky), and the result is... unaccountably bland. Quite often, this can be traced to a failure to give the reader something to focus on. Everything present is there because of its relationship to something else, and there is no single visual or logical point that gives the room an identity of its own. So... if a room description just doesn't seem to work, try providing a single, specific, dominant feature, preferably near the beginning or end of the description, rather than buried in the middle. The field room in our parenthetical examples here might be saved by inclusion of a focal feature such as a fire pit in its center, or a lone, towering pine, or a monolith of some sort.

Consider Using 'Catelogues'

The writers of ancient epics had a nifty trick: they evoked very specific constellations of connotation and emotion simply by listing things. Lots of things. Homer did not have resources such as a wide screen or a symphony orchestra to convey the pagentry and thrill of the Achean forces setting out for Troy. He had words, and one way he used them was the 'epic catelogue': instead of saying that a thousand ships set forth, he listed them... and who was on them... and who their parents were... and what they took with them. The cummulative effect builds and builds, and by its close the audience has not only a great deal of information but also, in effect, 'a recipe for a feeling'. *These* ships, *these* men, *these* weapons... and in such numbers. When the audience has this in mind, it cannot help but experience a unique and vivid effect.

When writing a room desc, you must work on a much smaller canvas, but the technique can be adapted. Try evoking a place or a world by simply listing salient figures and objects within it. Consider the following:

> Olevin Causeway
> The Causeway connects Sidney-Down-Over and the Ambly Island Spaceport, but is a world unto itself. Street merchants working from ramshackle stalls extol light sabres and life insurance, armorlite and absynthe, chronometers and Chrysellian fire oil. Strolling vendors insistently offer vibrablades, nebutol, and ice cream. Prophets and prostitutes hawk their wares, the former with greater passion, the latter with greater conviction. A crowd of drunken spacers skip Stellars on the heaving waters. A slumming heiress keeps ennui at bay for one more night, and the eyes of a starving child ask a question with no answer.

Homer it is not, but the list does convey the jaded, carnevalesque atmosphere of the Causeway in a small amount of screen space.

Consider Directly Addressing the Reader

Most accomplished builders avoid directly addressing the reader, because doing so forces an intrusive editorial presence upon the reader. This is a good rule to follow. But, like all the guidelines here, you might want to consider breaking the rule occassionally. Just be aware when you are doing so. Some readers consider the following desc too intrusive; others consider it highly effective.

> Bedroom Loft
> Yes, the ceiling is a bit low, the angles a trifle confining. But was there ever a better place to lie in a loved one's arms? Can you see the sailing moon outside the dormer window? Can you smell the faint but bracing scent, of cedar and lavendar, of musk and the night and something from your childhood? Turn down this faded quilt... Slip between these cool sheets... Unfold into this repose, and in it create

yourself anew.

4.6 Archiving Your Belongings

The `@archive` command (syntax `@archive <object>`) 'decompiles' objects. That is, it examines them, and then outputs the commands that would be necessary to recreate them. Unless you simply want to create an epic amount of spam, you need to capture this output in some way in order to make use of it. The normal way is to use your client to start a log file, and then issue the `@archive` command, then close the log. The resulting file can be used to recreate the object(s) if you accidently mess them up, or if the `MUCK` experiences database loss or corruption, or if you want to move the object to a different `MUCK.` To do so, simply quote or paste the file into your client window on the destination `MUCK.`

The command operates in a recursive fashion: a single `@archive` command will decompile the specified object, and any objects contained within it. This is usually a good thing: you can, for example, `@archive` an entire area with a single command by `@archiving` the area's parent room. But a minor warning: if you issue the command while you are in the area, your character object will be archived as well. This isn't a calamity, but it's probably not the best approach. The resulting archive file will be much larger than it needs to be, and if you quote it back without deleting the portions that apply to your character, you will get a great many error messages. So, if you want to `@archive` a room or area, do so by moving to a remote location before issuing the `@archive` command. (You can, of course, `@archive` your character as well... it's a good idea in fact. You would need the help of a wizard to recreate the character object, but once this is done, you could use the file to return your character to a previous state).

In the following example, we `@archive` the area created in the previous tutorial, using the logging commands for the TinyFugue client. Other clients will have other logging commands.

```
> gohome   ( leave area, so character won't be archived )
  ShadowKat's Lair
  .
  .
  .

> /log amberside
> @archive $ae
  .
  .
  . ( lots of text scrolls by )
  .
  .
> /log off
```

That's all there is to it! Since the `@archive` command is recursive, and since the whole area is — directly or indirectly — contained within the environment room referenced by the registered name `$ae,` the whole area will be `@archived.`

To recreate the area, we would simply quote the log file:

```
> /quote -S 'amberside
```

It is normal to see a few error messages when you recreate something from a log file. Common causes are exits that cannot be properly relinked because they led to something not included in the log and restricted properties that the `@archive` command could read but that you cannot set when you quote the file. Other, less obvious problems are sometimes caused by dbrefs included in properties: if you are moving the object to a different `MUCK,` these dbrefs will now be inaccurate. The effects of this will vary, but regardless you need to examine and test objects recreated from archive files.

5.0 ADMINISTERING A MUCK

Putting a `MUCK` online, creating a world that will attract players, and dealing with the minor and major crises that will inevitably crop up over time is an inherently demanding and time-consuming task. For most, though, it is a labor of love, and the burdens can be shared among staff members. Administering a `MUCK` involves both technical and non-technical issues. Both can be quite insistent and pressing during a `MUCK's` early, formative stages: there are innumerable technical details to attend to, and the small population creates an artificially incestuous atmosphere in which non-technical issues — especially personality conflicts — easily get blown out of proportion. As the world develops and attracts players, things stabilize, but both issues will remain. On the technical side, problems of dbase management and backward compatibility will replace the tasks of getting everything working right. On the non-technical side, player relations and changing levels of commitment from staff members will come to the fore. Both sets of issues are discussed in this section of the Manual.

5.1 Technical Issues

5.1.1 Selecting a Site

Of the various matters you'll need to consider, selecting a site is in many ways the easiest. You need a 7 x 24 connection to the Internet and a stable `IP` address, for a machine running some version of `UNIX,` with permission to run a constant process. This may reduce the number of options to one or none. If it's one, the decision is made: go with the site you have. If it is none, and you're set on running a `MUCK,` your options are:

1. Buy 7 x 24 access, with a stable `IP` address. In most markets, at the current time, this costs about $100 per month. You will of course also need a computer at the receiving end of that connection.
2. Find someone who will let you run a `MUCK` on their site. This is not impossible. Both requests and offers for sites appear rather frequently on the rec.games.mud.announce newsgroup. Bulletin board posts and public shouts on large `M*'s` may well turn up altruists: the freeware and volunteerism ethos remains alive and well in the `M*` universe.

If you are in the fortunate position of being able to choose sites, consider the following factors:

1. Connection reliability, connection speed, and sizable `RAM` are all more important than processor speed. A `MUCK` can be run quite successfully on a '486 under Linux. A faster `CPU` is of course nicer, especially if you do other things on the machine, but in all likelihood, processor speed is the last bottleneck you'll hit.

2. It's hard to judge connection reliability ahead of time. If you know someone who works on a site you're considering, or at least through the same ISP, solicit their feedback.
3. Faster connection hardware is, obviously, better than slower hardware. A 28.8 modem is adequate, but an ISDN is better. The new cable modem and ASDN links would work quite nicely. A T1 or partial T1 connection is also acceptable.
4. Hardware is not the only consideration for connection speed. Some ISP's simply cannot provide reliably fast connections. Sometimes this is their own fault (they sold more bandwidth than they have), and sometimes it's due to factors beyond their control (the ISP's ISP sold more bandwidth than *they* have, or the whole set up is situated behind some chronic bottleneck). You can get a rough idea of comparative speeds by doing a traceroute to sites under consideration. The first hops shown will be those on your end of the connection, and are less relevant: other user's won't have to make those hops. The last several hops, however, are telling. If one site consistently posts better times than another on the last two or three hops, this is worth taking into consideration. Although the times you see are in scant milleseconds, final hops of more than 300 - 400 milleseconds may indicate a bottleneck at the server. Also, try pinging the sites (`ping <address>`) and comparing the indicated times. Let ping run for a couple minutes, then stop it with ctrl-c. You should see a percentage for 'packets lost'. Dropped packets have to be resent, which significantly slows transmissions between a client and server.
5. You need adequate RAM. MUCK is memory-based. That is, it keeps the database loaded in RAM while operating. If this results in a process larger than available RAM, portions of memory are frequently swapped to disk ('paged'). This causes a significant degradation in performance. So, have enough memory: for a small MUCK, it's not a lot. A dbase with 2500 objects should create a process of 4 - 4.5 megs.

The ideal is to run the MUCK on a machine that you own: compiling the server will be considerably easier, and you'll be able to restart the MUCK quickly in the case of a power outage or other failure. Console access is also a significant security consideration: see Section 5.1.4

5.1.2 Compiling a MUCK

UNIX gurus and C buffs should have little difficulty compiling a MUCK. For the rest of us, it is a potentially frustrating experience. This section of the manual is addressed to the rest of us: sysadmins and C developers, you can use this time to go toggle in a new kernal from the front panel of a PDP-11, just to keep your hand in.

MUCK is not a shrink-wrapped, plug-and-play product. It is, rather, a large freeware application developed over a number of years by skilled coders who are willing to devote innumerable hours to making something for other people to use and enjoy. It is assumed that you — the site administrator — have reasonable facility with the UNIX operating system and a basic understanding of how to configure the program by editing C source code configuration and header files. In other words, like UNIX itself, MUCK is quite user-friendly, but rather choosy about who its friends are. The following overview may help you get on speaking terms with your new server.

It is of course madness to try to set up a MUCK without knowing UNIX. Nonetheless, people often try, and often succeed. A good book on `UNIX` will be a worthwhile investment if you are going to be the `MUCK's` site administrator.

If the set up goes smoothly — that is, if your system has everything where `MUCK` expects it to be — this information should be all you need. If you encounter compilation errors, you'll need to enlist help. Those sysadmins and C developers will be through toggling in their microkernals by the time you've gotten that far, and will no doubt be `MUCK`'ing somewhere. Go to a large `MUCK,` and try a public shout, or paging helpstaffers and wizards, asking if someone can lend a hand compiling a `MUCK.`

Getting your server up and running involves the following steps:

1. Getting a compressed, archived file containing the source code.
2. Uncompressing the files.
3. Editing configuration files
4. Compiling the source code
5. Specifying database files
6. Starting and logging onto the server

Getting the Server

The `MUCK` platform has evolved over a number of years, from TinyMUD, written by James Aspnes, to its current incarnation: TinyMUCK, FuzzBall version 6.0, developed primarily by Garth Minette. The most current and authoritative version should be available at ftp.belfry.com/pub. You may want to get the archive of standard `MUF` programs and a start-up database as well.

1. `pub/fuzzball/fb6.0.tar.gz`
2. `pub/fuzzball/fbmuf.tar.gz`
3. `pub/fuzzball/basedb.tar.gz`

Put the `fb6.0` file where you want the top level of your `MUCK` directory to go, perhaps in your home directory.

Uncompressing the Files

The files you just got are compressed archives of a great many files and directories.

To uncompress them, type `gunzip <filename>.` For example

```
gunzip fb6.0.tar.gz
```

If you get something like `gunzip: command not found,` try `unzip:`

```
unzip fb6.0.tar.gz
```

This uncompresses the archive. You now need to extract individual files from the archive with the `tar` command (`'tar'` for `'tape archive'`).

```
tar -xvf fb6.0.tar
```

The switches `-xvf` tell the system that you want to eXtract files from an existing archive File, and that you'd like it to do so Verbosely, so you can see what it's doing.

Type `ls` to 'list files' in your current directory. You should see, among other things, an entry for `fb6.0/` (the `/` slash may or may not appear). This is the directory holding the server.

If you got the dbase and `MUF` files, move them to the correct spots in the server directories...

```
mv fbmuf.tar.gz fb6.0/game/muf
mv basedb.tar.gz fb6.0/game/data
```

... and change directories (`cd`) down to the directories holding the files and unpack them in the same fashion.

Editing the Configuration Files

In directory `fb6.0,` you should see a file called `INSTALLATION.` It gives succinct instructions for setting up the `MUCK.` Type `cat INST*` to list it on your screen.

You will be editing a few files (`include/config.h`, `game/restart,` and possibly `include/params.h` or `include/autoconf.h).` It is strongly recommended that you make a copy of these files before you start modifying them, so you can start over if something obscure goes wrong. In the appropriate directories...

```
cp config.h config.bak
cp params.h params.bak
cp autoconf.h autoconf.bak
cp restart restart.bak
```

You can either edit the files directly on the server, with text editors such as `vi` or `pico,` or you can download the files to your computer, make changes, and then upload them. Hereafter, the Manaul will simply instruct you to 'edit' the files; do so in whatever way works best for you.

Much of what you'll be doing when editing these files consists of 'defining' or 'undefining' terms. A term is defined by beginning a line with the `#define` preprocessor directive, followed by the term and (optionally) its definition.

If a term is defined without a definition, such as...

```
#define GOD_PRIV
```

... this simply means that the term is 'true': the compiler can in effect check 'Are we using God privileges?', and get a yes/no answer. In this case, the term is defined; it has a true value: so, yes, we're using God privileges.

If a definition is supplied, then the term is true, and has a specific value. For example,

```
#define TINYPORT 8000
```

This means that, yes, we do have a specified default port to connect to: port 8000.

You can 'undefine' a term in either of two ways: you can comment it out, or you can explicitly undefine it. To undefine a term, use the `#undef` preprocessor directive:

```
#undef ANONYMITY
```

To comment out a term (that is, to change it into a comment that people can read, but to be ignored by the compiler), enclose it in the strings `'/*'` and `'*/'`

```
/* #define DISKBASE */
```

In general, you should `#undefine` terms rather than commenting them out: doing so will undefine the term, even if it was defined somewhere else.

You won't need to change a very much.

In `include/config.h:`

Edit the file to the port you want to use.

```
  #define TINYPORT 8000
```

Port numbers below 1024 are reserved for system processes; use something higher than 1024, and lower than 65,534.

It's recommended that you leave all other settings the same, until you're familiar with each of them and have a specific reason for changing them.

You shouldn't need to change anything in `include/params.h.`

In `game/restart:`

This file is a shell script, a set of commands that execute conditionally, rather like a `DOS` batch file. In addition to starting and restarting the `MUCK,` it does some logging and error checking: it bails out if the `MUCK` is already running, so you won't have duplicate processes running, keeps a log of when the `MUCK` was restarted, and warns of conditions such as missing or damaged database files, or insufficient memory. You will need to make a couple changes to this file.

Near the top of the file, change the line which sets the variable holding the path name for the `MUCK.` If you left the name of the directory created when you un-tar'd the server as `fb6.0,` and didn't rename any sub-directories in the server directory, you would set the path variable as follows:

```
  set GAMEDIR = $HOME/fb6.0/game
```

The port number needs to be specified in restart as well:

```
  set PORT = 8000
```

The server process is called `'netmuck'` (the executable file that actually runs the server is `'netmuck'`). `'Restart'` includes necessary references to `'netmuck'.` On sites that have several `M*'s` running, it's polite to rename `'netmuck'` to something else, or to call it by an alias, so that the sysadmin can tell at a glance what's what. If you're running the `MUCK` on your own machine, and know you'll only have one `MUCK,` the following step can be omitted.

Find the line that says...

```
  You probably won't need to edit anything after this line.
```

... so you'll know where to start making changes. A few lines into forbidden territory, you'll see the following line:

```
  set muck = 'ps -aux | grep netmuck | wc -l'
```

Replace the word `'netmuck'` with something indentifiable as your `MUCK.` If your `MUCK` is called `'Vanity Fair',` you might change it to:

```
  set muck = 'ps -aux | grep VanFair | wc -l'
```

And, the second to last line of the file:

```
  ./netmuck $DBIN $DBOUT $PORT >& logs/stdouterr.log &
```

Change `'netmuck'` here too.

```
./VanFair $DBIN $DBOUT $PORT >& logs/stdouterr.log &
```

Compiling the Source Code

Now it's time to compile the server.

A very brief rundown on what's happening here: In addition to information files like `README` and `INSTALLATION,` and the server you just downloaded consists of quite a few files of 'source code'... human-readable text files written in the C programming language. You'll use the 'make' and 'configure' utilities to create machine-executable code from the source files.

Change directories to `game/src,` the directory holding the source code. Type `'configure'.` This determines what flavor of `UNIX` you're running under, the location of certain files and executables, and so forth. This part should be quite straightforward. If it's not — if you get messages such as `'No processor installed'` or `'Welcome to Macintosh'` — log onto a large `MUCK` and find someone who can help.

Now, the (first) moment of truth: while still in `game/src,` type `'make'.` Watch arcane messages scroll by. Then, when they seem to have all done so, type `'make install'.` Watch more arcane messages. You may see some labled `'Warning'.` As long as the warning isn't followed by a `'Fatal error',` you should be OK. (Fatal errors are bad.)

If you get other compilation errors at this point, you'll need to track down a guru to help you out. Again, wizzes, helpstaff, and `@shouts` on large `MUCK's` are a good place to start looking for one.

Once you have the server compiled, you should tidy up a bit. In the compilation process, a number of 'object files' were created... Object files are intermediate files created as the compiler generates the executables. After compiliation, you don't need them any more; they just take up disk space. In the game/src directory, type `'make clean'` to get rid of them.

Specifying Database Files

You need to provide the server with a database to use at start up. You can use either the minimal database included with the server, or the standard start-up database included in `basedb.tar.gz.` The minimal database includes only two objects: Room `#0,` and God, player `#1.` The start-up database includes about 70 objects, including important programs and a couple rooms.

The start-up database is of course more convenient, but you may wish to use your own versions of the programs or to set aside some low dbrefs for players and other important objects. If you use the start-up database, the first objects you create will have dbrefs in the high 60's.

The minimal database is, well, minimal. You'll need to do a bit more work to get the place going, but you'll have greater control.

The files to be used for the database are defined in the `'restart'` script. We didn't change those: the server will expect to find files `'std-db.db',` `'std-db.old',` and `'std-db.new'.` The default file names are fine, but we need to create the files themselves.

In directory `game/data...` If you're using the start-up database

```
gunzip basedb.gz
```

This should create a file named `'basedb.db'.` Type `ls` to make sure. If it created something else, use that file name below:

```
cp basedb.db std-db.db
cp basedb.db std-db.old
cp basedb.db std-db.new
```

If you're using the minimal database:

```
cp minimal.db std-db.db
cp minimal.db std-db.old
cp minimal.db std-db.new
```

These commands will make copies of the database file where the server expects to find them.

Starting and Logging onto the Server

You're now *almost* ready to start the `MUCK.` One last thing... If you're on a public site, and edited the restart script changing `'netmuck'` to something like `'VanFair',` you need to either rename `'netmuck',` or create an alias for it.

In the `'game'` direcectory, type `ls.` You should see file `'netmuck'.` Rename that file with the following command:

```
  mv netmuck VanFair
```

If you made a copy of the `restart` file, you may need to make it executable at this point:

```
  chmod +x netmuck
```

...in the final column.

Now, type `'./restart'.` You should see some messages like `'restarting at <time>'`.

Type `'ps -aux'.` This will list all the processes running on the machine. If all is right with your world, you should see a process for it, a line with something like...

```
  ./VanFair data/std-d
```

...in the final column.

If you see too much stuff, and can't tell if the server process is in the list, filter it with the `grep`:

```
  ps -aux | grep VanFair
```

Quick and Dirty Trouble-Shooting

If the `MUCK` compiled correctly (you didn't get errors when compiling, and you have a file named `'netmuck'` in your game directory), but you don't have a process running, then something went awry in the restart script. You should confer with your sysadmin or some other knowledgable `UNIX`-type about it. Meanwhile, though, you can use a simplified restart script. If the problem truly is in the restart script, and not in the server, this should get you up and running while you enlist outside help.

Move your current restart script to another file for safe keeping:

```
  mv restart restart.cp2
```

Edit a new, simplified restart script, that includes the following three lines:

```
mv data/std-db.db data/std-db.old
mv data/std-db.new data/std-db.db
./VanFair data/std-db.db data/std-db.new 8000 >& logs/stdouterr.log &
```

(`'VanFair'` and `'8000'` are specific to our example: use the `MUCK` name and port number for your `MUCK.`)

Make the new script executable:

```
chmod +x restart
```

This script won't provide all the error checking and logging that the full-blown one will, but there's also less to go wrong in it. With the new script in place, type `'./restart'` again.

(A common reason for the restart script to fail: You are running your `MUCK` on a server that is also hosting other `MUCK`s, and neither you nor the other `MUCK` admins renamed '`netmuck`'. Because the restart script uses this file name to see if the server is already running when you try to restart, and bails out if so, your restart script fails because it sees someone else's `MUCK` running as a `netmuck` process. The fix: rename `netmuck` and edit the restart script appropriately.)

We'll assume that you now have a `MUCK` process running. Congratulations.

Now you need to log on. There is only one character on the `MUCK... #1,` who is probably named 'One'. `#1's` initial password is 'potrzebie'. Connect to the `MUCK` with your normal client, and log in:

```
connect #1 potrzebie
```

If you're running the minimal database, type `@stats:` savor this pristine and perfect universe.

```
***************************************
*** IMPORTANT: CHANGE #1's PASSWORD ***
***************************************
```

It is imperative to change God's password from the default.

```
@password potrzebie = <whatever>
```

You're done!

5.1.3 Setting Up the Database

Growing the database into a lively and attractive world is of course an on-going, never-ending process. When you're just starting out, the primary concerns are installing the necessary programs and setting up some kind of organization.

Initial Tidiness:

If you used the minimal database with an eye toward setting aside low dbrefs for Very Important Players and the like, `@create` some things to hold the dbrefs until needed, before you create other objects.

```
@create 2
@create 3
```

```
  @create 4
  @create 5
  .
  .
  .
  etc.
```

Then, when you want to use a low dbref for a player, `@recycle` the appropriate object and immediately `@pcreate.`

```
> @rec #2
  Thank you for recycling.
> @pcreate Agamemnon = bartlegast
  Player Agamemnon created as object #2.
```

Setting up at least a minimal environment tree now, while things are manageable, also saves effort later on. Most `MUCKs` end up with a storage room for `MUFs,` a room for new players to start in, and a guest room. Most also have a central meeting place. Use whatever structure you like, but something like the following should work well while you're getting started:

```
                         Room #0
                            |
                  Main Environment Room
                  |                   |
         IC Environment Room    OOC Environment Room
                  |                   |
            Village Well              |_ Guest Room
                                      |_ Player Start
                                      |_ MUF Vault
```

(See Section 2.2 for information on parenting rooms.)

Porting Global Programs:

The files provided in the `fbmuf.tar` are a genuine help. These are upload scripts that not only provide necessary programs, but also set up necessary exits, macros, and properties. To install them, uncompress and extract the files (`'gunzip fbmuf.tar.gz'`, `'tar -xvf fbmuf.tar'`), and quote, paste, or upload the resulting files onto the `MUCK,` using whatever method works best for you.

The file for `cmd-@register` should be uploaded first: the other upload scripts will use the `@register` command in installing the programs. Next, install the libraries, in the following order:

- `lib-strings`
- `lib-stackrng`
- `lib-props`
- `lib-lmgr`
- `lib-edit`
- `lib-editor`

- `lib-match`
- `lib-mesg`
- `lib-mesgbox`
- `lib-reflist`
- `lib-index`
- `lib-case`
- `lib-look`

(*Note*: The README file in fbmuf lists 'lib-case' as 'lib-cases'. You will need to quote it by the name given here, 'lib-case'.)

The remaining programs can be installed in any order.

If you're not uploading scripts from `fbmuf.tar,` you'll need to install all the programs 'by hand'. Even if you are using `fbmuf.tar` or the standard database, you will need to install some programs without the benefit of a script. The following gives an example of porting a library; the same techniques can be used for any program. First, get the code, perhaps by `@listing` and logging the program on an established `MUCK.` The example uses the logging and quoting syntax for TinyFugue; other clients will have different syntax.

```
On the established MUCK...
> /log lib-props
  % Logging to file lib-props
> @list $lib/props

  <output output output>

> /log off

On your MUCK...
> @prog lib-props
  Program created with number 26.
  Entering editor.
> i
  Entering insert mode.
> /quote -0 'lib-props
> .
> c
  Error in line 78: Unrecognized word lines.
> 78 l
  77 lines displayed.
> 78 d
  1 lines deleted.
> c
  Compiler done.
> q
  Editor exited.
```

As the example indicates, listing and quoting a program picks up an extra, unwanted line: the `@list` command follows its output with a line indicating how many lines of program code were listed... 77 in this case. You need to remove that line, either with a text editor on your system, or — as in this example — by uploading, compiling, noting and deleting the offending line, and then re-compiling.

Once you have the program compiled, you need to set its flags appropriately. All libraries should be set `M3` and `L`; `lib-lmgr`, `lib-props`, and `lib-reflist` should also be set `S`, `B`, and `H`. `Lib-look` should be set `S`. Other programs should be set with whatever flags they have on the `MUCK` you're porting from.

Libraries — and other programs frequently used by players, such as `do-nothing` and `obv-exits` — will need to be registered. You can do this either with the `@register` command, or by setting the property directly.

```
> @reg lib-props = lib/props
  Now registered as _reg/lib/props: lib-look(#26FLM3) on Room Zero(#0R)

or...

> @propset #0 = dbref:_reg/lib/props:#26
  Property set.
```

Some programs will also need properties set. For libraries, this information is readily available with the `@view` command; for other programs, you will probably need to get a wizard or the program owner to help you view the props.

```
> @view $lib/look
  Command to view: @list $lib/props=1-20
  Run this command? (y/n)
> n
  Read definitions? (y/n)
> y
  .envprop = "$lib/props" match "envprop" call
  .envsearch = "$lib/props" match "envsearch" call
  .locate-prop = "$lib/props" match "locate-prop" call
  .setpropstr = "$lib/props" match "setpropstr" call
  envprop = "$lib/props" match "envprop" call
  envsearch = "$lib/props" match "envsearch" call
  locate-prop = "$lib/props" match "locate-prop" call
  setpropstr = "$lib/props" match "setpropstr" call
```

The first bit of output in this series...

```
  Command to view: @list $lib/props=1-20
```

...tells you that the program documentation appears in lines 1 - 20 of the program. To set things up so that players on your `MUCK` can view the program, set the `_docs` property on the program:

```
> @set lib-props = _docs:@list $lib/props=1-20
  Property set.
```

The definitions are stored in propdir `_defs/` on the program object. They provide information needed for calling programs to communicate with the library. The definition...

```
  .envprop = "$lib/props" match "envprop" call
```

... means 'Where a program that uses this library contains the word '.envprop', use the function `'envprop'` in this library'.

Set the `_def` properties for each definition.

```
> @set lib-props = _defs/.envprop:"$lib/props" match "envprop" call
  Property set.
  <Etc. Copy, find/replace, and paste are your friends.>
```

Setting up libraries is also discussed in Section 3.2.2, `MUF` Libraries.

Other Porting Considerations:

Permission: all programs in `fbmuf.tar` and the start-up database are in the public domain. For other programs, you should make a good faith effort to honor the conditions for porting programs. Read the program's header comment, looking for notes about permission and/or the author of the program. If the program says it can't be ported without permission, don't: instead, contact the program's author and ask for permission, or find a similar program that can be freely ported.

Sometimes you just can't get in contact with the author, usually because he is no longer `MUCK`'ing. In this case, contact the `MUCK's` `MUF` or Globals wizard, and let her know that you want to port the program. Usually there is no problem with this. And, even if the program says it can be ported freely, sending `page #mail` that you've done so is a good idea: people make programs freely available because they would like to see them widely used; letting them know you are using their program is a thoughtful gesture.

Connect and Disconnect Calls: Programs called from these queues (and related queues such as `_arrive` and `_listen`) must be set `Link_OK.` Again, when uploading from scripts, this is handled for you, but when you're manually porting the program, you will need to set the calling property yourself.

```
> @set con-announce = L
  Flag set.
> @propset #0 = dbref:_connect/announce:#82
  Property set.
> ex #0 = _connect/announce
  ref _connect/announce:con-announce(#82FWLM3)
```

Sometimes a program needs to be called from a prop like `_connect`, `_disconnect`, or `_listen` — and so should be set `Link_OK` — but also needs to be kept *not* `Link_OK` for security reasons. This is handled by creating a small, `Link_OK` stub program that calls the real program. An example of this is provided below, in the discussion of setting up Guest characters.

Macros: Programs uploaded from scripts or provided with the start-up database will set up any macros they need. For programs that you install yourself, you will sometimes need to define macros. If a program that works fine on another `MUCK` will not compile on yours because of an unrecoginized word...

```
  Error in line 273: Unrecognized word otell.
```

... the problem is probably an undefined macro. On the `MUCK` from which you are porting the program, go into the program editor, type `'s'` to see a list of defined global macros, and look for the offending word... `'otell'` in this example. Copy the macro definition. Then, on your `MUCK,` define the macro.

```
> @edit lib-look
  Entering editor.
  Line not available for display.
> def otell ( s -- ) loc @ me @ rot notify_except
  Entry created.
> c
  Compiler done.
> q
  Editor exited.
```

`MUF` Macros are also discussed in Section 3.2.3, `MUF` Macros.

`MPI` Macros: You will probably want to set up global `MPI` macros as well. These are stored in the `_msgmacs/` directory of Room `#0.` Set a property in this directory, with the name of the macro as the prop name, and the `MPI` to be used in its place as the stored value. For example, to set up a simple `{look-notify}`...

```
> @set #0=_msgmacs/look-notify:{null:{tell:[ {name:me} looked at
  you. ],this}}
  Property set.
```

Wizards on an established `MUCK` may well be willing to email you a file containing their `MUCK's MPI` macros. `MPI` macros are also discussed in Section 3.1.1.

You may also copy sample `MUF` macros and sample `MPI` macros from here.

Tuning System Parameters:

The `@tune` command lets you adjust a number of system parameters. Type `'@tune'` without arguments to view the parameters and their current settings. Most can be left at their default settings. Of those that you may want to change, some only make cosmetic or formatting changes, while others significantly affect the operation of the `MUCK.` The following are some parameters you may want to change. The syntax for resetting a parameter is `'@tune <parameter> = <value>'.` The value must be compatible with the data type indicated in the left column of `@tune` output: `(str), (bool),` etc.

`dumpwarn_mesg`
`dumpwarn_mesg`
`deltawarn_mesg`
`dumpdeltas_mesg`
`dumping_mesg`
	These parameters can be freely changed to alternate save messages. Note, however, that some players configure clients to be triggered by the ## characters in a save message. So, it's a good idea to keep this format: enclose your new save messages in ## double octothorpes ##

`penny`
`pennies`
`cpenny`
`cpennies`
	Declaring new currency denominations is a time-honored wiz pastime.

`muckname`	This you should change. Set it to the name of your `MUCK.` It's a good idea to use a `muckname` setting that does not include any spaces.
`huh_mesg` `leave_mesg`	You may want to supply something more imaginative for these.
`dump_interval`	Four hours is the default interval between saves. This is quite workable, though it's often shorter on new `MUCKs` and longer on large, established ones.
`max_pennies`	If money is signifcant on your `MUCK,` you may want to lower this. Note that `M2` Muckers can make very simple programs that give pennies freely. (This will eventually be changed in fb6.0+.)
`penny_rate`	Lower this if you want fewer `'You found a penny!'` messages; raise it if you want more.

`command_burst_size`
`commands_per_time`
`command_time_msec`
`max_delta_objs`
`max_loaded_objs`
`max_process_limit`
`max_plyr_processes`
`max_instr_count`
`instr_slice`
`mpi_max_commands`
`pause_min`
`free_frames_pool`
	It is unlikely that you will need to change these. However, if for some reason you believe your server is performing poorly, you may wish to try adjusting these system-performance parameters.

`playermax_limit`	The default limit is 64. Higher limits allow more players, but setting an unreasonably high limit wastes memory.
`listen_mlev`	This parameter controls the minimum Mucker level of programs which can be called by `_listen.` The default is `M3,` which is a good choice. The decision whether to raise or lower it should be based on how closely player programming is monitored.
`player_start`	This should be changed. Set it to the dbref of the room where you want new players to start.

`use_hostnames` The default for this parameter is `no,` but tuning it to `yes` is convenient. Wizards see connection information when typing `'WHO*'` (use `'WHO'` without the asterix for normal formatting). With `use_hostnames` set to `'yes',` domain names rather than numeric IP addresses will be shown for all players logged on. Note: the change won't be immediately apparent. Host names will be supplied for players who log on after you tune the parameter to `'yes';` those who are logged on now will still have numeric addresses.

`log_commands`
`log_failed_commands`
`log_programs`

Whether or not to log and what you do with the logs are fairly significant decisions. See also Security Concerns and Privacy Issues. As for the technical aspects of logging, note the following:

- All commands (including says and poses) entered by wizards are logged, regardless of the logging parameters. If `log_commands` is off, and you want to have a private discussion that won't go into the logs, set yourself `Quell.`

- Log files pile up. You will need to follow some schedule for deleting old log files. (The files are in the server directory `game/logs.`) Either by hand, or with a script, or with a crontab script that runs automatically, follow a routine for copying the current logs to storage files. For example:

  ```
  mv com2 com3
  mv com1 com2
  mv commands com1
  ```

  Doing this once per day, say, will keep one-day- old log_commands files in `'com1',` two-day-old logs in `'com2',` three-day-old logs in `'com3'.` After three days, they're gone.

  Raw logs are difficult to read. If you do need to review log files, mastering the `UNIX` `'grep'` command will be very helpful.

`dbdump_warning`
`deltadump_warning`

Until your `MUCK` grows to past several thousand objects, full saves will take only a couple seconds, and delta saves will be practically instantaneous. You might want to turn these off, out of anti-spam sentiment.

`realms_control` `The` Realms Wizard system has advantages and disad- vantages. See Using the Realms Wizard System.

The remaining parameters can safely be left at their default settings. For a (terse) description of what each parameter does, see the entry for `SYSPARM` in Section 3.2.5, `MUF` Reference.

Editing Server Files:

The screen shown at log in, and the information provided in `'info', 'news', 'help', 'man', 'mpi',` and `'motd'` (Message of the Day) are all stored in the server directory `game/data.`

One of your first tasks will be providing a new log in screen, to replace the *TygressMUCK* screen supplied with the server. To do so, edit the file game/data/welcome.txt. Note: Though many players currently have large screens, it's a good idea to limit the file to 78 characters by 20 rows. This is all that can be seen at one time by someone running TinyFugue in 'visual' mode on a standard 80 x 24 monitor, or a similarly limited terminal program on some other platform.

You can add to or edit the documentation provided in `'news'`, `'info'`, `'help'`, `'man'`, and `'mpi'` by editing the files `'news.txt'`, `'info.txt'`, `'help.txt'`, `'man.txt'`, and `'mpihelp.txt'` respectively. Or, you can add files to the directories `'news'`, `'info'`, `'help'`, `'man'`, and `'mpihelp'`. Adding a file called `'join'` to the `'news'` directory would create an entry that players can read by typing `'news join'`.

The `MOTD` will usually be updated online, with the `'motd'` command (syntax: `'motd <message>'` to add an entry; `'motd clear'` to clear all entries). The `motd` command gives no control over formatting, however, and you can only clear *all* the entries, not one or some. If you want to make more precise changes to the `MOTD`, edit the file `game/data/motd.txt`.

Setting Up Guest Characters:

(Note: Setup for the `con-multiguest`, the Guest program supplied in the `fbmuf` package, described here, is more difficult than it really ought to be. As an alternative to the process described below, you may want to make a copy of `MultiGuest.muf`, edit the line that specifies the guests password to something unique, go to the room that the Guests will start at, and paste the file into your client window. This will set up your Guest characters automatically. You need to have a $nothing action set up before quoting the file.)

You will probably want to allow Guest characters. A program that shuffles incoming connections to the next available Guest character is available in fbmuf.tar and the start up database: `con-multiguest`. There is also a small 'stub' program — `con-callmultiguest` — that calls the main program when players connect as Guests. The reason for this is that programs called from `_connect` must be set `Link_OK`, but the password creation scheme for Guests should be kept private, for security reasons. Rather than calling `con-multiguest` directly (which would require it to be set `Link_OK`), `con-callmultiguest` is set `Link_OK` and called on connection. Then, this small program calls `con-multiguest` itself.

If you don't have `con-multiguest` and `con-callmultiguest` ported and registered yet, do so now. Then set both program Wizard. Set `con-callmultiguest` `Link_OK` and `Set_UID`.

```
> @prog con-multiguest
  Program created with number 88.
  Entering editor.
> i
  Entering insert mode.
  < quote or paste program >
> .
> c
  Compiler done.
> q
  Editor exited.

> @reg con-multi = con/multiguest
```

```
  Now registered as _reg/con/multiguest: con-multiguest(#88FM3) on
  Room Zero(#0R)

> @set #88 = W
  Flag set.

> @prog con-callmultiguest
  Program created with number 89.
  Entering editor.
> i
  Entering insert mode.
  < quote or paste program >
> .
> c
  Compiler done.
> q
  Editor exited.

> @set #89 = W
  Flag set.
> @set #89 = L
  Flag set.
> @set #89 = S
  Flag set.
```

If you haven't created a room for the Guests to start in, do that now as well.

```
> @dig Guest Room
  Guest Room created with number 90.
```

`@Pcreate` the proto-Guest character.

```
> @pcreate Guest = guest
  Player Guest created as object #91.
```

Then `@pcreate` the actual Guests. By default, the `con-multiguest` program is set up to handle eight Guests. If you want more or less, find the following line in the program...

```
  $def NumGuests 9 (Max guests allowed connected + 1)
```

... and edit it appropriately. That is, change `9` to some other number.

```
> @pcreate Guest1 = guest
  Player Guest1 created as object #92.
  .
  .
  .
> @pcreate Guest8 = guest
  Player Guest1 created as object #99.
```

`@Teleport` the Guests to the Guest Room, and `@link` them there.

```
> @tel *guest = #90
  Teleported.
> @link *guest = #90
  Home set.
  <etc.>
  <etc.>
  <etc.>
```

`@Set` the `_connect/multiguest` property on each Guest to call `con-callmultiguest.`

```
> @propset *guest=dbref:_connect/multiguest:#89
  Property set.
> @propset *guest1=dbref:_connect/multiguest:#89
  Property set.
  <etc.>
  <etc.>
  <etc.>
```

This version of the multi-guest program changes Guest passwords by forcing a wizard (not the most elegant approach). You'll need to edit line 14 of the program, designating a wizard to do the forcing. In the original file, #1 is designated, but God cannot be forced if the `MUCK` is compiled with `GOD_PRIV.` (If you don't have other wiz characters yet, you need to make one.)

You should probably also change the password creation scheme. This is in line 20. It's not really crucial what you change it too: just make it something that returns a string.

```
> @edit con-multiguest
  Entering editor.
  Line not available for display.
> 14 l
  $def Wizard #1 (Change this to a wizard's dbref. Preferably not #1)
> 14 d
  1 lines deleted.
> 14 i
  Entering insert mode.
> $def Wizard #2 (Change this to a wizard's dbref. Preferably not #1)
> .

> 20 l
  $def PassWdMake (i -- s) 3 + 9 * intostr "TimE" swap strcat "tWINe"
strcat
> 20 d
  1 lines deleted.
> 20 i
  Entering insert mode.
> $def PassWdMake (i -- s) "snerk" 5 * 7 intostr swap strcat "aVast"
strcat
> .
> c
```

```
  Compiler done.
  Editor exited.
```

Creating a 'Gohome' Exit:

A `'gohome'` action that will let players return to their homes without losing things they're carrying is a worthwhile and easy-to-install convenience.

```
> @act gohome = #0
  Action created with number #100 and attached.
> @link gohome = home
  Linked to HOME.
```

Using the Realms Wizard System:

Realms Wizards are players who have extended control over objects within a certain area of the MUCK... that is, in rooms parented to a common environment room. There are advantages and disadvantages to the realms wizards system, discussed below. Whether you are going to use the system is not something you must decide immediately, but if you are going to use realms wizards, there a few advantages to doing so from the outset.

In order to use the realms wizards system, the system parameter `realms_control` must be tuned to `'yes'.` The effect is to give the owners of rooms set `Wizard` extended control over objects in that room, and over objects in rooms parented to that room. A realm wizard can examine, set properties on, and link objects within his realm as though he owns them. He cannot chown objects belonging to others, unless the objects are set `Chown_OK,` and he can only force objects that are set `xForcible` and force locked to him, but அ since he can `@force_lock` and set flags on objects in the realm அ he can essentially chown and force anything in the realm as well. He can `@teleport` anywhere within the realm, just as a wizard can.

There are decided advantages to making your staff builders realms wizards. The extended control is a genuine help as they construct the area, and their realm wizard status may well be a motivating factor: they are lord in their realms, but that will only be meaningful if people use the areas, so they have good reason to make the areas as well constructed and attractive as possible. Once the area is built, they will be able to help players with matters such as linking homes. Also, making staff builders realms wizards rather than wizards avoids the problem of having several surplus wizards once the world is constructed.

The one significant disadvantage — or at least consideration — is security. Wizards have virtually unlimited control over the database, but they also have built-in accountability: all wizard commands are logged, regardless of whether `log_commands` is tuned to 'yes' or 'no'. Within their realms, realms wizards have comparable control, but their commands are not automatically logged. A clever and malicious realms wizard could set locks and flags on a true wizard who enters his area, and use these to directly or indirectly force the true wizard to do something destructive or harmful elsewhere. If (im)properly handled, the properties could erase themselves once they have done their job, leaving no evidence, and the logs would show the true wizard as the guilty party. The only real safeguard is the same safeguard provided against true wizards: accountability. If you want to use the realms wizard system, turn on command logging.

The realms wizard system works best if you set up a consistent, geographically structured environment tree. This is much easier if done when the `MUCK` is new. The following schematic shows how a science fiction

`MUCK` might be set up to use realms wizards. Rooms `'Starport Env. Room'` and `'Earth Env. Room'` would be chowned to staff builders, and set `Wizard.` These players who own these two rooms would then have primary responsibility for and control over those two areas. Additional areas could be parented to `IC Environment Room,` and the two realms control rooms could also include environment rooms, perhaps with other players holding realms wizard control over those smaller areas.

```
                            Room #0
                               |
                     Main Environment Room
                     |                    |
            IC Environment Room    OOC Environment Room
              |              |                |
   Starport Env. Room     Earth Env. Room    |_ Guest Room
              |              |                |_ Player Start
              |_ Some room   |_ Some room     |_ Staff Offices
              |_ Some room   |_ Some room     |_ Help Room
              |_ Some room   |_ Some room     |_ MUF Vault
              |_ etc.        |_ etc.
```

5.1.4 Security Concerns

Security problems are, fortunately, rather rare on `MUCKs.` Most `MUCKs` function — quite successfully — in a climate of trust, but giving thought to security issues is one of your responsibilities as an administrator. The issues are magnified if you become an administrator on a large `MUCK,` or if your `MUCK` attracts a sizeable player base.

In this section, 'security' is meant as 'protection against unauthorized modification or destruction of the database itself, or of database objects'. This is somewhat different than the related issue of privacy. Privacy issues are discussed in Section 5.2.4.

Access to the MUCK Account:

Being able to log onto the server account the `MUCK` is run from is the trump card of all security concerns. With access to this account, someone bent on mischief or destruction can do anything: change God's password, alter logs, or destroy the database and all backups. Someone bent on safeguarding the `MUCK` can undo any damage: no matter what has gone wrong or been destroyed on the `MUCK,` the world can be restored to sane and healthy condition by restoring from back ups.

So, control over and awareness of who has access to this account is the foremost security concern. If you own the machine the `MUCK` runs on, and are able and willing to do all site administration tasks, the issue is considerably simplified: don't give anyone else the password to the account.

For most `MUCKs,` the situation is a bit more murky. Often someone else owns the machine. Sometimes the site administrator is a wizard on the `MUCK,` but only marginally involved in its day-to-day affairs. Sometimes several staff members end up having access to the account: God has access because it is, after all, her world; the site admin has access because it is, after all, his computer; the player relations and security wiz ends up getting access so he can review logs in situations where one player has charged another with some kind of offense; the wiz responsible for creating new characters gets access so she can check email for char requests; perhaps another wizard is given access so he can update news, info, and help files. (For some things, you can enlist server-side help from staff members without giving them access to the account. For example, a separate account could be set up for the security wiz, and you could copy the log back ups to a directory in that account, rather than `game/logs.` A separate account can be set up for the wizard who does character creation, with char requests to be mailed there, or mail can be forwarded to her.)

If this is a team of trustworthy people, then things are grand. Work is shared among several people, and any one of them could log on to deal with an emergency, such as restarting the `MUCK` after a power failure. But, be aware: if you share access to the `MUCK` account, you no longer have total assurance that the `MUCK` is safeguarded. It is conceivable that one of these people will get mad at you, or at life, the universe, and everything, and destroy the `MUCK,` or that one will give the account password to someone you don't know.

Keeping Backups

The server does a certain measure of backup on its own. At restart (assuming you are using the standard database filenames) the database used at the previous restart, `std-db.db,` is copied to `std-db.old,` and the most current form of the database (`std-db.new,` created from the most recent dump) is copied to `std-db.db.` So, at any given moment, you have three versions of the database on hand.

You may want to take this a step further. Copying the most recent database file to another file and compressing it gives you a safe copy. Depending on your energy level and disk space, you may want to do a rotating swap of database back ups similar to the one suggested for logs.

```
mv backup2.gz backup3.gz
mv backup1.gz backup2.gz
cp std-db.new backup1
gzip -9 backup1
```

The reason for keeping older backups is that it's possible for a database to become corrupt during normal operation: some bug in the server or on-line programs creates inconsistencies in the database file. In this situation, the most recent backup file is not necessarily the best one: it too could be affected.

Also, to safeguard against hardware or access problems, you may want to do an offsite back up. With File Transfer Protocol, this is a simple (but potentially time-consuming) process. `FTP` your most recent backup to some other host. If your server goes down, or one of your wizards goes bonkers and deletes the backups, you can set up on an alternate site.

Access to #1:

Next in the hierarchy of security issues is access to the God character. If the `MUCK` is compiled with God privileges, only God can set players `W` or `!W,` or change wizards' passwords. Also, only God can use the utilities for analyzing and repairing the database, `@sanity` and `@sanfix.`

On most established `MUCKs,` God is not used as an actual player. The primary wizard of the `MUCK` will create a wizard character for daily use, and only log onto `#1` to only set or unset players' wizbits or to use the database utilities. In part this is for security: just as `UNIX` sysadmins do their normal work from an account other than root, `MUCK` Gods do their normal work from a different character. In part it is simply a convenience: wizards sometimes need to force themselves, or use programs that force them, and `#1` can't be forced. Also, `#1` always gets the full-data `WHO` screen (which wizards see when typing `WHO*`), whether she wants it or not. And in part it's a matter of facilitating administration of the `MUCK.` Sometimes the *de facto*

God of the MUCK (the overall administrator who appoints wizards, determines policy, etc.) and the most technically accomplished staff member (the person who would handle matters such as running @sanfix) are two different people, so both have reason for using #1. Or — a rather common situation — there is no single wizard with total control; rather, leadership is shared among a core group of wizards, each having access to the God character.

Whatever set up you use, the password to #1 is an important piece of information, and should only be given out on a need-to-know basis.

Wizards:

Most wizards are committed to their worlds, and would never directly harm it, but the possibility exists. It's not unheard of for an immature wizard to get angry and destroy a world. The MUCK can recover from this by restoring from back ups, but obviously this is an unpleasant chore.

More common problems come about inadvertently. Most are minor: a wizard with wayward fingers mistypes #321 for #312, and recycles some important room or command. So, you fix it.

Sometimes players @force lock themselves to each other. Wizards shouldn't. When you appoint a new wizard, check his force lock. When he was a player, it was fine for his roommate to be able to force his sleeping body downstairs when she had visitors, but it's not fine for her to be able to force him once he is a wizard.

Wizards should also be aware of a specific line of attack that can be used to modify objects with the wizard's permissions. You can set properties on objects you control (you control everything). Also, objects you control can set properties on other objects you control (and you control everything). And, properties can include MPI that causes programs or other properties to execute. (This is true for players too, but the implications are less threatening.) For wizards, these events are carried out with wizard permissions. In short, and without providing specific recipes, a naughty player can make objects which — if they are owned by a wizard — can do naughty things. Whenever you @chown an object, examine its properties, with particular attention to MPI that stores values or calls properties or programs ({store}, {exec}, {lexec}, {muf}, and the like).

Programs:

Wizard programs can do pretty much anything a wizard can do, and M3 programs are quite powerful as well. As with security issues regarding wizards, the trick is to identify and support trustworty people (and programs), and to be aware of the issues, rather than trying to anticipate and safeguard against every possible threat.

Your best defense against security breaches through programs is to appoint a good MUF wizard. This staff member determines who gets Mucker bits, with special attention to M3's, and reviews global and public programs, with special attention to programs that players ask to port from elsewhere. If you are not particularly MUF savvy, or if you are a good coder but lack the sneaky mindset necessary to be a good hacker or anti-hacker, you should recruit someone with these qualities. (Surprisingly, it is possible to find honest sneaky people.)

Some general points...

The apparent purpose of a program is no guarantee of its potential threat or lack thereof. If someone asks you to port a 2000-line Poker program that needs to be set Wizard so it can keep scores in a protected .scores/

directory, approach it as a wizbitted program large enough to hide some Trojan horse feature, and not as 'just a card game'.

*

Programs set `Link_OK` are public: they can be listed, linked to, and *called by other* programs. Many are a little too trusting: they assume that you (the person using the program) are who you're supposed to be. And, if you use them in the normal way, you are. But it is a trival matter to concoct a small stub program that stores an incorrect dbref in the `'me'` variable, puts an argument string on the stack, and then calls a public program. An `M2` morphing program may seem perfectly safe — after all, people won't put harmful things in their own descs — but consider: `JoeHacker(#666PBJM2)` might be able to store `#1` or the dbref of some other wizard in `'me'`, put a string holding naughty `MPI` onto the stack, and then call the morph program, creating a situation where the next time someone looks at the wizard, untoward things will happen.

Fortunately, there is a rather simple way to plug this hole. Check all `Link_OK` programs; if they don't guard against such assualts, edit them, adding a line that does so. Before the first line of the last function in the program, insert the following line:

```
 "me" match me !
```

This explicitly matches the dbref of the triggering player or puppet, and stores it in the `'me'` variable.

*

In general, any program should be run with with the lowest permissions level that will work. If a program will run at `M2`, set it `M2`, not `M3` or `W`. If a program doesn't need to be owned by a wizard, consider chowning it to some mortal, possibly an `NPC` you use for just this purpose.

*

You owe it to yourself and your players to make your policies on security and privacy concerns as they pertain to programming explicit. Providing a document stating this policy, readable through the `'news'` or `'info'` command, is a very good idea. A sample policy is provided in Appendix B. The sample policy may be freely copied and edited to meet your `MUCK's` needs.

5.2 Nontechnical Issues

The following discussions of nontechnical issues are necessarily subjective and editorial. They should be read with a critical eye, but not ignored: new `MUCKs` succeed or fail for nontechnical reasons, not technical ones.

5.2.1 Conceptualizing the World

The historical bias of `MUCK` toward socializing presents the would-be administrators of a new world with a special challenge. On platforms such as `MUSH`, with its emphasis on role playing, creating a successful new world is not intrinsically difficult: one develops a well-written geography and sound command structure, plants `TP` seeds with timelines and background events, advertises the `MUSH`, and the world has a good chance to survive. If it's well done, and if it taps into a new or popular genre, it should attract players. But, `MUCKs` tend to be primarily places to socialize rather than `RP.` There is no technical reason why `MUCK` is not as good as or better than `MUSH` for `RP,` but the trend is there. So long as players' motivation for `MUCKing` is socializing, established worlds will have a powerful advantage over new worlds in attracting players: if one goes to `MUCKs` in order to meet and be with people, it makes good sense to go to a world where there are 300 people online rather than a struggling new place where there are three. Unless the new world offers something that established worlds do not, the administrators will, after a great deal of time and effort, find themselves lords of a very lonely domain.

Dealing with this problem is beyond the scope of this manual, but it is mentioned and even stressed here because it is very easy for new administrators to lose sight of. Their new world seems grand to them — at least in part — because now they are *wizards*, now they are *in charge*. It's a grand new world because, in short, it is their world. But if players are to make it their world as well, the administrators will need to give thought to a fundamental question: Why should anyone come here?

Does the `MUCK` put a new genre online? Are there intriguing adventures or quests, or well-wrought gaming programs? Does it appeal to some audience beyond that of current `MUCKs?` Are players brought together by a common and heretofore unserved interest? The answer to at least one of these questions needs to be 'yes'.

5.2.2 Recruiting a Staff

Recruiting and keeping a good staff of wizards can be surprisingly difficult. The best wizards already *are* wizards somewhere, and have little incentive to take on another world and stick with it through the long haul.

Putting and keeping a `MUCK` online requires the sustained efforts of several people. At least one person on the staff should have solid technical skills, and all staff members should have a thorough understanding of the difference between a `#dbref` and a hole in the ground. But it is not necessary nor even advisable to recruit wizzes exclusively from a pool of technical hot-shots. Technical expertise is one (and only one) of several skills required among staff members. Interpersonal, organizational, and writing skills are important as well. A strong staff will have a balance of these skills among them, plus time, energy, and commitment.

Time, energy, and commitment are in fact the most scarce and valuable of `VR` resources. And, wizards who become inactive after a first flush of enthusiasm are the most common management problem. There are no magic bullets for these problems, just hard-to-follow advice:

- Try first to find the ideal candidates for wiz positions. There are two flavors:
  - Good, active wizards who would like to move on from a world where there is friction among staff, or their area of responsibility is limited.
  - Good, energetic players who demonstrate maturity and skill in one of the areas a new `MUCK` needs. Helpstaffers on large `MUCKs` are a good pool: these are people with a demonstrated willingness to spend their online time making things work.
- Recruit friends if you want, but don't do so on the basis of friendship. Recruit friends who have something to offer, and know from the outset that one day you may need to ask your friend to resign, or be stuck with an inactive wizard.
- Be willing to perform the unpleasant task of firing wizards who don't work out.

5.2.3 Sharing Responsibility

Most `MUCK` administrators find it necessary or at least helpful to recruit a staff, to share responsibilities for building, programming, technical maintenance, player relations, and so forth, with a team of people with aptitude for a specific area of responsibility.

Use whatever pigeon holes and job desciptions you like. The following are common:

- Site Administration: This wizard is responsible for maintaining Net access. Sometimes the site admin is only marginally involved in the `MUCK's` day-to-day affairs, acting essentially as a landlord. Even in this situation, there are reasons for making the site admin a wizard: for example, he can log onto the `MUCK` and do a clean `@shutdown` if it's necessary to shutdown the server for maintenance or other RL reasons. The site admin may also be responsible for server-side chores such as keeping back-ups, updating news and info files, and answering email addressed to the `MUCK` staff.

- Programming: This wizard sets and unsets Mucker bits, based on players' programming expertise, trustworthiness, and committment to the world. She also reviews programs to be ported, checking security and efficiency considerations. She may also be responsible for creating and maintaining a listing of publically available programs.

- Player Relations: This wizard arbitrates grievances between players and charges of `AUP` violations. He is empowered to act as appropriate in such situations, using disciplinary tools such as recording warnings and charges in a log kept on the character, temporary `@newpasswording,` `@toading,` and site banning.

- New Characters: This wizard `@pcreates` new characters. She needs a way to read emailed character requests, and she needs information about any players, sites, and addresses that have been the source of player-relations problems. For a small staff, it makes sense to combine this job and that of Player Relations.

- Building: This wizard is, in the `MUCK's` early days, charged with building the world, or coordinating the efforts of a team of builders. Once the world is built, the Building wizard will have the most complete understanding of how everything is set up, and is the best person to decide matters such as planning or incorporating new areas. Often, the Building wizard also determines policy on quota issues.

The exact form of your organizational set up isn't crucial, but it is important that the staff has a good understanding of who's responsible for what, when it's OK to do something that affects another staffer's area, and that God or the core administrators keep a sense of proportion. Silly, unproductive 'turf wars' are a common reason for staffs failing to gel as a team, and worlds burdened with squabbling wizards seldom thrive. Some guidelines...

- God should refrain from micromanagement. If you've given a wizard authority to handle Player Relations, let him handle it.

- Good communication solves most problems before they become problems. Set up a global page alias for wizards and staff. All staff should page mail each other when they do something that affects the `MUCK` as a whole, or affects others' areas of responsibility.

- If instant action isn't required (and it seldom is) let the wizard who's responsible for an area handle that area. The Builder wizard may think that JoeHacker's actions obviously call for toading, and she may be right, but it's better for her to simply `@newpassword` JoeHacker, and let the Player

Relations wiz handle the actual toading. Not only is it his job, but he may have worked out a procedure for `@chowning` the character's objects, notifying the player, and so forth.

- And the flipside of the same coin... If you and the players frequently find that something needed just doesn't get done because a key wizard is unavailable (the Programming wiz hasn't logged in in three weeks, say), then something is amiss with staff organization. God should step in, handle the immediate situation, and consider making changes to the staff roster.

5.2.4 Privacy Issues

Privacy on `MUCKs` is a charged issue, and making good decisions about how to handle privacy issues requires making mature judgments in the face of seemingly contradictory considerations.

On the one hand, there is no guarentee of privacy in an online environment. On the other hand, there is no excuse for administrators who abuse their position to pry into the online lives of players on their worlds.

On the one hand, the subjective experience of `VR` as a 'safe' environment often leads people to let down their emotional guard... to reveal intimate or painful details of their lives to others online. Administrators should take every precaution against violating the implied trusts that lead these players to make such revelations. On the other hand, people have organized illegal activities online, and the administrators of worlds where this has taken place have been embroiled in legal issues as a result. Administrators owe it to themselves and to other players on the `MUCK` to take reasonable precautions against such cases.

Doing a good job with privacy issues is primarily a matter of following an important list of DON'Ts:

- DON'T read log files unless you have a specific, valid reason for doing to. Filter the output so you only read what you need to read.
- DON'T set yourself `Dark` and move about the `MUCK` to populated areas, eavesdropping on conversations.
- DON'T use `Darked` puppets to eavesdrop on rooms that you would otherwise not be able to hear.
- DON'T `@teleport` into private rooms without warning.
- DON'T `@teleport` online players without warning them.
- DON'T `examine` players' properties unless you have a specific, valid reason for doing so.

Religiously abiding by these "don'ts" in conjunction with keeping a log of commands issued by players is a reasonable compromise. Be aware that if you do log commands, and players on your world are investigated for illegal online activities (which includes using online worlds to organize illegal `RL` activities), your command logs may be subpoenaed. It is not absolutely necessary for your `AUP` to state that you log commands, but you will be on firmer ground if you do so: players will not be able to claim that they did not know that commands were logged, and would have acted differently if they had known.

Your `MUCK` should almost certainly have an Acceptable Use Policy, an explicit statement of what constitutes acceptable online activity for the `MUCK` and — more importantly — what does not. The policy serves two purposes:

- It provides an objective basis for resolving disputes.
- It provides some measure of protection should players engage in illegal activities online.

Resolving online disuputes well requires judgment, maturity, and a certain degree of empathy... none of which are really 'objective' factors. But the more you can found your judgment in objective, verifiable factors, the less you will open yourself to charges of playing favorites, or being a hothead, or letting yourself be swayed by personalities. Any of these can contribute to an unhealthy online environment, so do what you can to put objective factors into place. The Acceptable Use Policy is a primary tool for this.

Copyright and privacy laws as they relate to the Internet are complex and rapidly evolving fields. The authors of *The MUCK Manual* make no claim to expertise in this area. That being said, a general premise stands: if you have put in place a policy stating the site's position on copyright, privacy, and related issues, and have put in place mechanisms to make players aware of the policy and a way for them to indicate their agreement to abide by the policy, it will be much easier for you to show that responsibility for any illegal acivity that occurs on your site rests with an offending individual rather than you, the site administrator.

A sample Acceptable Use Policy, which may be freely ammended to fit the needs of your `MUCK`, is provided in Appendix B. You may also wish to install cmd-aup, a program that lets players indicate explicit agreement with the `MUCK`'s Acceptable Use Policy, and provides a way to lock exits such that only players who have done so may use them.

5.2.6 Toading

The command for getting rid of players is `@toad`. The syntax is:

```
@toad <player1> [ = <player2> ]
```

This turns the Player object for `player1` into an object of type Thing, and transfers ownership of all objects owned by `player1` to `player2`. If `player2` is not supplied, all objects owned by `player1` will be `@chowned` to the wizard issuing the `@toad` command.

The Thing object will retain all properties that were set on the player.

Occassionally you will need to toad players: sometimes for particularly grave violations of the AUP, more often because the player wants to leave the world or has simply stopped showing up and you want to reclaim unused dbase space.

Some considerations:

As discussed in Section 5.1.4, Security Concerns , Wizards should examine the properties of objects they `@chown`. If you are `@toading` a large number of players, or `@toading` a player with a large number of objects, it's a very good idea to supply a non-wiz player for objects to be `@chowned` to when `@toading` (perhaps an `NPC` player object used just for this purpose). This is of course espcially important if the player is being `@toaded` as a disciplinary measure for security violations.

If you are `@toading` idle players to reclaim database space, you will probably want to `@recycle` objects they own as well. It is advisable to proceed cautiously here. The player may have created rooms, actions, or things that are in public use. So, examine objects as you proceed. It will be much easier to keep track of which objects should be checked and potentially recycled if — again — you use a dedicated `NPC` character to `@chown` objects too when `@toading`. If you are consistent about supplying this character as the second argument to the `@toad` command, you can periodically review objects owned by this player (`@owned <player name>`), and `@recycle` private objects previously owned by `@toaded` players.

Players often ask to be `@toaded` when something online upsets them. And, such players often change their minds. And, such players aren't likely to be in a receptive frame of mind if you grill them about whether they *really* want to leave. So... You can make your life and theirs easier if you institute a "silent grace period" policy. When players ask to be `@toaded`, just say "OK", and let them leave with a minimum of fuss. But, instead of actually `@toading` them at that point, give them a new password and new name, and move them to a holding area. This way, if they change their minds, they can come back easily... all you will need to do is rename and repassword the character object. And, you can take care of the database-related details of `@toading` at your leisure. Periodically, you can review characters in the holding area, and do an actual `@toad` on those who have been gone for an appreciable period.

Fixing a mistake:

It is not all that hard to accidently `@toad` someone. If you do, you can recreate the character exactly as it was, except for one thing: You'll have to give it a new password. Hopefully, you have access to the player's email address (see Record Keeping), in which case, you can simply recreate the character and send email letting the player know what happened and telling him or her the new password. If you don't have their email address, recreate the character and wait for a grumpy Guest to show up, demanding to know what happened to his or her character.

To recreate an accidentally slain character, copy all properties from the slimy toad object to a temporary holding object (so that you can reapply them to the new character), then recycle the slimy toad object and immediately `@pcreate` a new character with the same name (so that the new player will have the same dbref, which might be used in props and programs). Then, copy the properties back from the holding object to the new character.

The `'cp'` and `'mv'` commands are the easiest way to shift the properties around, but, be aware that they will not move wizard or restricted properties unless the program is set `Wizard`... And, be aware that if you are using the 'stock' version of `cmd-mv-cp`, you should *not* leave it set `Wizard`: doing so creates a significant security hole. Two options: You can temporarily set `cmd-mv-cp Wizard`, make your changes, and then set `cmd-mv-cp not-Wizard`, or... you can install a modified version of `cmd-mv-cp` (available here) that can safely be left at a `Wizard` setting.

5.2.7 Record Keeping

Your `MUCK` is itself a large, complex set of records, and most information you would need to record is automatically recorded, either in log files or the database itself. There is, though, one record that most administrators find extremely useful that `MUCK` itself has no mechanism for: player's email addresses. Life will be much easier if you religiously follow a practice of recording email addresses each time you run `@pcreate.` This will let you notify players of things like downtime and address changes, and — if you follow a policy of not accepting character requests from anonymous email sites such as Hotmail, Rocketmail, Yahoo, etc. — gives you some chance of identifying alternate characters controlled by the same player.

You can store this online (on the player object, in a property such `as @/email),` but it is more valuable to keep the record offline, so that you can access it if the `MUCK` goes down... which is a prime example of when you would need the players' email addresses. A flat text file works fine for this.

One other kind of record keeping may prove useful. It is occasionally necessary to issue warnings to players about AUP infringements, and the like. In order for such warnings to be a useful administrative measure, other wizards need to know about warnings you have issued. A common and workable practice is to use a program that stores administrative notes about such matters in a wizard-only propdir on player objects. With it, you can see if any other wizards have noted AUP problems with a player before you decide what an appropriate response to an incident is. (An example of such a program is provided here.)

5.2.8 The Last Word

Throughout, *The MUCK Manual* has discussed the numerous technical effects of the `Wizard` bit being set on a player. But wizbits often have another, nontechnical effect: they can greatly aggravate any tendency the player has toward arrogance and pomposity. Do your players a favor... Do *yourself* a favor: guard constantly against thinking that being a wizard makes you special. The best MUCK wizards treat the job as "glorified janitor", with the emphasis on "janitor".

Clients

A `M*` client is a program running on your computer that, like telnet, can connect to `M*` servers, but — unlike telnet — has numerous features intended for use with `M*'s`. Some clients, naturally, are more full featured than others. Common features include macro capability, logging, automated logon, hiliting, multi-worlds, and recall. Below is a list of a few popular clients... try one of these to get started.

AmiMUD
: An Amiga client.

MacMUSH
: A Macintosh client.

MUDCaller
: A DOS Client.

MudRunner
: A DOS client.

MUDSOCK
: A Windows 3.1 and Win32 client.

MUSHClient
: A Win32 client.

Phoca
: A Win32 client.

Pueblo
: A Win32 client with multimedia and hypertext capabilities.

Rapscallion
: A Mactintosh client.

SimpleMU
: A Win32 client. (Recommended)

TinyFugue
: The definitive UNIX M* client. Ports to OS/2, Windows, MacOS, VAX/VMS, and BeOS are also available here. (Recommended)

WyrmNet
: A console-based Win32 client.

zMUD
: A Windows 3.1 and Win32 client.

Appendix B: Sample Policy Files

You may freely modify and use these on any `MUCK`

The `@AUP` command mentioned in the Acceptable Use Policy can be copied from here. This command sets the restricted property `~aup` to `"yes"`. You may wish to `@lock` important commands and exits (such as the exit leading out of the `player_start` room to this property, to ensure that players are apprised of the AUP's existence and are (more or less) forced to enter an explicit command indicating their agreement, or stop using the `MUCK`.

```
> @lock out = ~aup:yes
> @fail out = Before entering GenericMUCK, you must first read the
Acceptable
  Use Policy (type 'info aup'), and indicate that you agree to abide by
its
  terms (by typing '@aup').
```

```
                 GenericMUCK's Acceptable Use Policy

The staff of GenericMUCK respects your privacy and intellectual
property rights, and requires that you extend the same respect to others.
Please read this file completely, and enter the command @AUP to
acknowledge that you have read the file and agree to abide by its terms.

'Residents' refers to all players and staff. Where the distinction
between staff and players is important, this will be explicitly noted.

1.0  Access and Legal Responsibility

Access to GenericMUCK, and the computing resources required to operate
GenericMUCK, is granted on a revokable basis, and at no time is this
access guaranteed.

GenericMUCK services may be used only for lawful purposes. Transmission
or solicitation for reception of material which violates US Federal or
Hawaii State Law, any Policy of the site where the MUCK operates, or
state or local law of the area in which you reside is prohibited. This
includes, but is not limited to, material that is threatening, libelous,
```

or violates trade-secret, patent, or copy-right protections.

You agree to indemnify and hold harmless GenericMUCK, its staff, the owners/operators of the computing resources, and all other parties connected with the administration of GenericMUCK from all claims which are a result of your usage, without limitation. You agree that any traffic which originates from your character or connection is your legal responsibility.

Notwithstanding the above, you agree that should GenericMUCK or its staff be found liable in a court of law for any action or lack of action related to your use of GenericMUCK, our liability is limited to $0.00.

2.0 Privacy

As a resident of GenericMUCK, you have a right to reasonable privacy.

2.1 Residents may not use programs, zombie objects, or other methods to monitor your conversations or activities inappropriately.

> 2.1.1 Residents may use zombies, broadcasting exits, programs, etc., as detailed in 'news programming'.
>
> 2.1.2 Staff may monitor intrusively, by setting the Dark flag on themselves or referring to logs, if they have reason to believe that a player or group of players is actively conspiring to destroy or damage the MUCK's database, or is engaged in illegal activities.
>
> 2.1.3 Electronic Communications Privacy Act (ECPA) NOTICE: The staff of GenericMUCK reserves the right to monitor any and all communications through or with GenericMUCK computing facilities. You agree that GenericMUCK facilities are NOT to be considered a secure communications medium for the purposes of the ECPA.

2.2 Residents may not use programs or other methods to retrieve information stored in properties on your character or belongings inappropriately.

> 2.2.1 Residents may create and use programs that retrieve information in ways that meet the the guidelines detailed in 'news programming'.
>
> 2.2.2 Staff may use the 'examine' command and other methods of retrieving information remotely as needed.

2.3 Residents may not harrass you.

> 2.3.1 If you indicate that you are not receptive to intimate or sexual contact with another resident, that resident is obligated to respect your wishes. If someone persists in making unwanted advances, repeat your demand that he or she stop, and page a staff member immediately.
>
> Residents guilty of sexual harrassment are liable to disciplinary action up to and including @toading and site-banning.
>
> 2.3.2 Repeated attempts to talk to, page, or otherwise interact with a resident who has clearly requested to be left alone are a violation of Acceptable Use.
>
> 2.3.3 Racial, ethnic, and religious slurs directed at a player are a violation of Acceptable Use.

2.4 Information about your life outside GenericMUCK is privileged. Your

RL name, gender, age, phone number, email address, and place of residence are priviledged information. That is, unless you explicitly indicate that this information may be dispersed, residents are obligated to treat the information as confidential.

2.5 Privacy as regards alternate characters is left to residents' discretion.

2.5.1 You may have alternate characters, and you are not obliged to inform others of the relationships between your characters.

2.5.2 If you do not wish others to know of the relationships between your characters, do not give out this information. If you do not mind other residents knowing about the relationships between your characters, it would be helpful if you indicated this, by conversation, pinfo, or other means.

3.0 Intellectual Property Rights

The staff of GenericMUCK honors intellectual property rights as they pertain to the MUCK, and requires that residents do so as well. The staff claims no particular expertise in the complex and evolving field of international copyright law and the Internet. This does not, however, invalidate or limit the AUP as it pertains to intellectual property rights: GenericMUCK policies on intellectual property rights stand as detailed below.

3.1 Residents may not create programs, rooms, or other objects incorporating copyrighted material without verifiable permission from the copyright holder.

3.2 The staff of GenericMUCK become joint holders of intellectual property rights for public rooms or other creations which become an integral part of GenericMUCK and are used by a substantial portion of the residents. The owners of such creations may improve and add to them as they see fit, but may not @recycle the object or remove the properties or code that make it integral to the MUCK without permission from the staff.

In the event that a player inappropriately destroys or changes public creations, the staff reserves the right to restore such creations from back ups. Should such occassions arise, intellectual property rights for that creation as the pertain to the MUCK will from that point lie solely with the staff of GenericMUCK. That is, the creating player forfeits rights to determine if and how the creation is used on the MUCK.

3.3 Programs which include notices that permission to port is requested or required may not be ported to or from GenericMUCK without such permission. If you wish to port such a program and are unable to contact the author, notify a staff member. The staff will attempt to contact the author, and if unable to do so, will determine ona case-by-case basis whether the port meets AUP requirements.

3.4 The staff will not modify or remove any of your programs without your permission, with the following exceptions: in the case of programs which pose a threat to the integrity of the database or violate privacy as detailed in 'news programming' and the AUP, the staff will either request that you modify or remove the program, or will do so themselves.

4.0 Violations of Acceptable Use.

Residents who violate the Acceptable Use Policy, or who commit acts not explicitly covered by this policy but deemed a vilotation of the policy's spirit by the staff, are liable to consequences including:

- Suspension of access to the Muck and its computing resources for
  a time to be determined by the administrators.
- Removal of your character and termination of access to the Muck
  and its computing resources without prior notice.
- Notification of your site and/or system administrators
- Notification of civil and/or law enforcement authorities.

GenericMUCK's Mucker Policy

1.0 How can I be a Mucker?

We welcome both experienced and neophyte Muckers. We ask only that (a) you help the general public, not just yourself, by writing useful programs of some social benefit or aid to building, and (b) that you abide by the spirit and letter of our Mucker policy.

If a program you would like to write or use falls into a "gray area" of these guidelines, please ask us for clarification. We may be able to suggest ways to accomodate your needs.

2.0 What does a Mucker do?

2.1 Learning to Program

Start by reading the MUF tutorial and the MUCK Manual, also the CHANGES files. Find them with the INFO command. If you need more help than this, we may be able to put you in touch with a Mucker willing to tutor you. If you write small programs to test something, or no longer need a program, please remember to clean up after yourself. A clean database is an effcient database, with less lag.

2.3 Public Programming

Whenever you write a program, and especially when a program is publicly accessible (Link_OK), please make sure it adheres to our guidelines:

1. It should be useful in some way, social or building.
2. It should not violate guidelines of privacy, respect, or honesty. (see section 3.0 of this policy)
3. It should not be wasteful-- do not use more space or CPU time than is reasonable, or duplicate things that already exist.

If one of your programs meets the guidelines above, you may ask to have it made publicly available. Such a program must be set link_ok in order for other people to use it. A program that would be useful as a command for everyone may be installed as a global. Type 'globals' to see some commands available.

Many programs are already publicly available; type 'programs' for a list, then list these programs for more information, or go to the Programming Room off the Administrative Nexus and look through the files. If you are looking for a program that would be a modification of an existing one, you might find it more convenient to ask the programmer to change it appropiately.

3.0 Guidelines to Proper Programming Etiquette

3.1 Privacy

Players of GenericMUCK are entitled to privacy. If you couldn't find something out by normal means or without the permission of the users involved, you should not find it out with a program.

***** Normal means does NOT include abuse of the powerful MPI command language. That means one does not use MPI to violate the Privacy of others, or attempt to manipulate or look at things or rooms you do not own. Please see NEWS MPI for more information. ****

However, when you use certain programs, it is understood that the program may store, relay, or use *reasonable* information for *reasonable* purposes. If you aren't sure a program would fit this guideline, please ask.

(A) Bugs relay information to another person, room, or stores it for later reading, without your implicit consent.

Examples of legal programs:

1. Public programs such as page and spoof, which only use information for administrative purposes, and are documented as to this use.

2. Programs that broadcast messages for the purposes of "virtual reality" such as a program to let you be heard from a stage. However it is possible to disguise the use of such programs. This is unethical.

3. Bulletin boards, mailing systems, and other programs meant to record and display messages for the public. By entering messages into such a program, the user gives implicit consent that they be stored and displayed.

Examples of illegal programs:

1. Programs that duplicate "page", "whisper", "say", or "pose" in order to record or relay information without the player's knowledge.

(B) Scanners find information about other players, their properties, or belongings, that would not normally be available, and which the player does not wish revealed. This includes programs written in MPI.

Examples of legal programs:

1. A program that only shows you messages or properties which are set specifically for that program. E.G. a smell program might show you people's smell messages, but it would be unethical to write programs to read people's smell messages at a distance, without their consent.

2. Programs that show you information which you could obtain another way, e.g. a program to tell you which of a selected group of people are on the WHO list, or a program to tell you what exits go from rooms that you own. Owners can always examine anything they own.

3. In reasonable circumstances, programs may show you information such as which players are in another room if it is known to the people in this room that they may be observed. E.G. a transparent exit description for windows.

Examples of illegal programs (in MUF or MPI):

1. Programs that show you properties or messages on players, objects, and so forth that you do not own and could not normally find out.

2. Programs that reveal private exits in rooms without the permission of the room's owner. (similarly, programs to locate players without their permission)

3.2 Respect

Certain programs may, while not invading a player's privacy, harass that player, or make it possible to do so in a way that is undetectable. Players are entitled to respect and dignity.

(a) Spoofers that allow a player to simulate another player's actions. This would allow players to forge incriminating or insulting messages under that player's name. Messages that could potentially be spoofed should be made apparent in some plain manner or changed to remove this danger.

(b) Markers are programs that change, add to, or remove player properties without their implicit permission.

Examples of legal programs:

1. Programs may set temporary properties, or properties that obviously belong to the program, and do not interfere with other programs or the user's convenience. E.G. the page program sets a number of properties for records keeping.

2. Programs that explicitly give the user full knowledge of what changes are about to be made. E.G. role-playing systems and shape-shifting programs that modify your description.

3. Programs for building assistance, which change your properties and/or objects in an approved way.

Examples of illegal programs (in MUF or MPI):

1. Programs to overwrite another player's descriptions, messages, etc. without their consent.

2. Programs meant to harass them by insulting, annoying, or otherwise inconveniencing them, e.g. programs to send a spoof message directly to a player without the player's permission, or to whisper to all but one player.

3.3 Other Prohibited Programs (MUF or MPI)

GenericMUCK prohibits general teleporter programs that would allow players to invade private rooms or that are intended to violate virtual reality, e.g. encouraging people to teleport directly to a room, ignoring intervening areas. We also dislike programs that add pennies for no valid reason. Pennies are spent and used for things, and available at the GenericMUCK bank.

Teleporters may be allowed in limited cases, but only for very specific reasons and purposes. They must follow these restrictions:

1. Players must agree explicitly, or implicitly by entering a vehicle, following another player, being picked up by that player, etc. to be moved.

2. Programs to move players can only move players from or to rooms

for which they have permission, and under such circumstances as are appropiate, e.g. a taxi might have stops in different rooms with the permissions of these rooms' owners.

3. Such programs must be appropiate to the circumstances, the "virtual reality" of the situation in question, etc.

3.4 What Happens If I Make A Mistake?

There are three steps, depending on seriousness of the offense. Accidents and oversights happen; on the other hand, a deliberate infraction may well lose you your Mucker bit immediately. You will be informed of the reasons for any actions we take as regards you or your programs. If you misuse MPI, your BUILDER bit will be removed.

1. If you write a program considered abusive or forbidden, you will be talked to about why you need such a program.

    (a) If the reasons are acceptable, the program may be allowed as is, or we may suggest how it can be modified to make it fit the guidelines.

    (b) If the reasons are unacceptable or inadequate, you may be warned to remove the program, or else modify it so that it will be acceptable.

2. If you don't comply with requests to modify or remove a program that is particularly abusive, then the program may be removed. Programs that may crash the server may also be confiscated or removed. You will be told how and why the program crashes the server

3. If you repeatedly upload programs that were removed as abusive, or crash the server, or otherwise abuse your Mucker bit, then you may be deMuckered and offending programs will be removed. Violations with MPI will result in the loss of your BUILDER bit as well.

Appeals will be allowed under extenuating circumstances.

4.0 Program Libraries and Macros

4.1 Documentation

Please document your programs where possible, so that those who are meant to use them can do so. If programs are Link_OK, therefore publicly usable, you should put comments at the top. Public programs should also follow the convention of including a #help function.

We suggest that programs have comment headers describing the basic code (two or three lines to synopsize the program's purpose, how it should be used, and unusual things that users should know), naming the owner, and providing any copyrights you wish to include.

Please don't complicate your code more than MUF normally reads; deliberately obfuscating your code beyond a capable Mucker's ability to read is considered impolite. Macros should be commented, or else obvious from name or definition as to what they are intended to do.

4.2 Information Ownership

Many programs are set Link_OK so that they can be readily used. This does not mean that you are free to copy them to other systems, or for your own uses. Please ask the permission of the creator before you copy programs!

5.0 The Final Word

If you have any questions about this policy, MUCKing, or other related issues, please ask a knowledgeable player, Mucker, or Administrator. They may be able to offer you answers, suggest where you can look quickly and conveniently for information, or suggest how Mucker policy applies to your question.

In all disputes related to Mucker policy and programming on GenericMUCK, the judgement of the MUF Administrators is final, and supercedes the guidelines of this policy.

6.0 Mucker bit levels and getting your mucker bit.

There are 3 Mucker levels.

```
        M1 - Starter, or Apprentice
        M2 - Standard, or Journeyman
        M3 - Special, or Master
```

When you ask for a Mucker bit on GenericMUCK, you will be set M1. This is where all Muckers will start. This gives you access to the @program and @edit commands and allows 75% of MUF functions. This is a safety feature to allow you to learn MUF without fear of damaging anything.

To get to Level M2, you should have written, tested, and debugged some useful programs and then ask the MUF Administrator or hir assistants to look at them for programming style and usefulness. The Administrators may then grant you the M2 level. This allows 95% of MUF features to work.

Level M3 is needed only in very special programs that must directly manipulate sensitive data from the database. This level can only be granted by the MUF Administrators.

TO GET A Mucker BIT: Page or page #mail a wizard. Any wizard can give you an M1 bit. M2 and M3 bits are set by the MUF Administrator.