LPmud

LPmud, a programmable multi user game.

Lars Pensjö
`lars@cd.chalmers.se` *

Pre-release, last update: 2-7-93

*The text of this manual was taken from Lars' LPmud.texinfo and other files written by Amylaar, Ardanna, Demos, Macbeth, Marion and some unknown authors. This manual describes the tubmud driver, which is Amylaar's version of the original gamedriver written by Lars and many others. Descriptions of parts of the tubmudlib have been added. This document was crufted together by Foslay. Special thanks go to Bumblebee and Ignatios for TEXnical advice.

Contents

1 General

1.1 Idea of the game

LPmud is a multi user adventure game. That means that several players can be playing the game at the same time, using the same object database. It also means that the players will meet each other, and affect the game for other players.

1.2 History of LPmud

In the beginning, I[1] played a lot of Abermud and some Tinymud, and wanted to do something better, combining the two systems. I made the first version of LPmud as some kind of argument, to show that my ideas were possible. Luckily, I didn't know at that time how that would impact my near future.

1.3 Objects, files and programs

The programs defining the behaviour of objects are stored in files. Every object has exactly one file defining the program, but every file may be used for more than one object. When an object that is not loaded is referenced, it will automatically be compiled and loaded. More than one instance of an object can be created using the function *clone_object() (See also section ??)*. The function *file_name()* returns a string "filename#number" for a cloned object and just "filename" for the blueprint. When a blueprint has an environment (is inside another object) it cannot be cloned any longer, this enforces the rule that a blueprint is either an object in the game or a source of clones. *clone_object()* should not be used if only one instance of an object is wanted. Cloned objects may be configured differently after creation, hence enabling different behaviour.

[1] Lars Pensjö

2 File Hierarchy in Tubmud

All files which form the object database reside in the '`mudlib`'. The tubmudlib looks like this:

`basic:`	Basic modules of the new tubmudlib (unfinished)
`complex:`	Configurable objects of the new tubmudlib (unfinished)
`global:`	New directory of global objects
`secure:`	Security related objects
`sys:`	System files, include files
`lib:`	Library objects
`obj:`	All 2.4.5 standard objects and more.
`room:`	All standard 2.4.5 rooms, some include file
`domains:`	*(See also section ??)*
`players:`	The castle of wizards reside here
`save:`	Various save files (for player objects etc.)
`log:`	Various log files
`doc:`	Manual pages for functions, configurable objects etc.
`open:`	Readable and writable for all wizards
`std:`	Obsolete (taken from another new mudlib)

3 Installation and Setup

This chapter describes how to get LPmud running from scratch. It works for SUN SparcStation running SunOS 4.1.1, silicon graphic IRIS, Atari st/tt running MiNT and equipted with gcc/mintlibs/bash, Commodore Amiga with DICE compiler, 80386/80486 machines running linux 0.98.5, DEC 5000/125 with Ultrix 4.2 and Next. If you are using any other platform you may have some trouble getting the game up. We don't have time or equipment needed for porting the game to other systems.

3.1 Getting LPmud Sources

LPmud can be retrieved with anonymous FTP from host alcazar.cd.chalmers.se, IP-number 129.16.79.30. Log on as `ftp`, and give your email address as password.

The archive holding the source can be found in '`/pub/lpmud`', and is called '`LPmud-3.1.2.tar.Z`'. It is a binary file, which means that you have to set FTP in binary mode.

This is how a ftp session might look:

```
% ftp alcazar.cd.chalmers.se
Connected to alcazar.
220 alcazar FTP server (Version 5.53 Sun May 27 01:43:44 MET DST 1990) ready.
Name (alcazar.cd.chalmers.se:arne): ftp
331 Guest login ok, send ident as password.
Password: arne@cd.stanford.edu
230 Guest login ok, access restrictions apply.
ftp> cd pub/lpmud
250 CWD command successful.
ftp> bin
200 Type set to I.
ftp> get LPmud-3.1.2.tar.Z
200 PORT command successful.
150 Opening BINARY mode data connection for LPmud-3.1.2.tar.Z (510193 bytes).
226 Transfer complete.
local: LPmud-3.1.2.tar.Z remote: LPmud-3.1.2.tar.Z
510193 bytes received in 1e-06 seconds (1.4e+06 Kbytes/s)
```

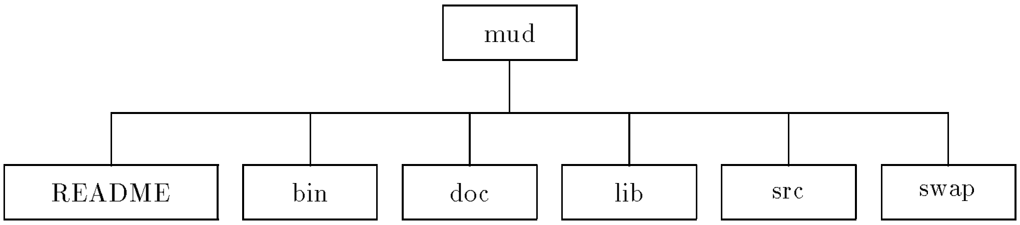
```
ftp> bye
221 Goodbye.
```

The tubmudlib, some other mudlibs and versions of the gamedriver can be found on ftp.cs.tu-berlin.de (quepasa.cs.tu-berlin.de). Tubmud has its own ftp server (morgen.cs.tu-berlin 7681) which accepts your wizard name and password if you are a wizard in tubmud, there is also a mtp server with port number 7682.

3.2 Contents of the Archive

The archive contains all you need to set up an LPmud on your host. However you need a `mudlib`. You can write one yourself or get one by ftp. The software you got in this archive contains the preprocessor, communication stuff, efuns and lots of other functions needed in the game. The other major part is what we call '`mudlib`'. Everything accessible from within the game is in '`mudlib`', such as save files, log files and LPC source code. The LPC source code is the code defining what your LPmud will be like. It is the game itself, while the '`game driver`' is a tool for running the game. There are also some other things included in the archive, among them documentation and some useful utilities.

This is what the directory structure in the archive looks like:



This is a description of what some of the directories are. *(See also section* ??*)*, for more information on the contents of the '`lib`' directory.

README	A file describing where to find the documentation.
bin	Directory that will contain all the executables after installation.
doc	All the documentation as a text file, a texinfo file and a PostScript file.
lib	The mudlib directory.
src	The source code for the game driver.
src/util	Source code for some utility programs and shell scripts.
swap	When the game is running it creates a swap file here.

3.3 Unpacking the Archive

When you unpack the source archive, a directory called '`mud`' will be created in your current directory. All the files in the archive will be put in the '`mud`' directory. Before unpacking the archive you should decide where in the filesystem you want to have the game and change directory to that location.

The archive is a `compressed tar` archive. To unpack it you need one of the commands `uncompress` or `zcat` and the program `tar`, or gnu tar, which can do all the unpacking with gtar xzf <archive>.

Here is an example of how the archive can be unpacked:

```
% cd /usr/src/games
% zcat LPmud-3.0.22.tar.Z | tar xf -
```

3.4 Compiling the game driver

The first thing you have to do is to configure LPmud to suit your system. Change directory to '`mud/src`'. Edit '`config.h`' and change any defines needed. They are well commented in the file, but those that it is likely that you want to change are explained here.

`TIME_TO_SWAP`	Tells how long the game driver should wait before swapping out an unused object. The time should be high if you have much memory.
`TIME_TO_RESET`	Sets the intervall between calls to *reset()* in objects.
`PORTNO`	This is the port number that the games uses. Before setting it, you should ask your system manager what number to use.
`DOMAINS`	If this is defined you enable **`domains`**, i.e. groups of wizards working together, with a special domain directory for each group.
`SWAP_FILE`	Should be a full path to a file used for swapping objects. It is typically set to '`mud/swap`' as mentioned previously.
`LOG_SHOUT`	If LOG_SHOUT is defined all shouting in the game is logged in a file, '`mud/lib/log/SHOUTS`'.
`MAX_PLAYERS`	The maximum number of simultaneous players in the game. The more powerfull system you have, the higher should it be.

Next, you should edit the '`Makefile`' - you probably need to pick the appropriate one from hosts/* first - and make changes suitable for your system. These are some of the parameters that you might want to change.

`MALLOC`	There are three different versions of malloc to choose from. The '`Makefile`' tells the difference between them.
`CC`	What C compiler to use.
`BINDIR`	Where to put all the executables. Typically set to '`mud/bin`'.
`MUD_LIB`	The path to `mudlib`. Usually set to '`mud/lib`'.

After editing '`config.h`' and '`Makefile`' you are ready to compile the game driver. Type

```
make
```

if the compilation is completed without errors, an executable file '`debug`' should have been created.

3.5 Installing the Game

After compiling the game driver, several files have to be installed in their proper places. If you have changed the path for `MUD_LIB` in the '`Makefile`' to something other than '`mud/lib`' in the directory where you extracted the software, you have to move the contents of '`mud/lib`' to the new location. If you have changed `BINDIR` in the '`Makefile`' or `SWAP_FILE` in '`config.h`' to be other than those that were extracted, you must make those directories in the new location. After doing that, give the command

```
make install
```

Now the game driver should be installed in `BINDIR`. We provide a shell script that restarts the game every time it crashes or is shut down. It restarts the game at most 50 times. After that the script has to be rerun. The script is called '`restart_mud`' and is located in '`mud/src/util`'. To install it in `BINDIR` type

```
make install.restart_mud
```

If you want the game to be restarted every time your computer is rebooted, ask your system manager to run '`restart_mud`' from '`/etc/rc.local`'. Here is an example of what to put in '`/etc/rc.local`'.

```
if [ -f /usr/games/mud/bin/restart_mud ]; then
   /bin/su arne -c /usr/games/mud/bin/restart_mud 2>&/dev/null
   echo 'Starting LPmud'
fi
```

3.6 Starting the game

The simplest way of starting the game is to issue the command

```
parse
```

that after a successfull installation resides in '`bin`'. That command will start the game in 3.0 mode, accepting connections on the port configured in '`src/config.h`'.

These are the command line options that are available:

- `-c` Print a message to stdout every time a file is compiled.
- `-d` Debug information. Repetition increases verbosity.
- `-D` Define a preprocessor macro for all LPC files.
- `-e` Start the game without loading any wizard or domain files.
- `-f` The string following the option is passed to master::flag();
- `-m` Set the mudlib directory, overriding the path in Makefile. There must be no space after the 'm'
- `-M` Set the path name of the master object inside the mudlib. If not given, /obj/master is used for COMPAT_MODE and /secure/master for native mode.
- `-r` Set the size of the reserved memory, overriding the value in config.h

4 Commands tied to Functions in Objects

All commands except a very few special cases are defined by the objects. All commands have a simple basic way to be recognized. The first word of the sentence is supposed to be the verb. Every command defined is tied to a special function in an object. Commands are defined with the function *add_action()*, which specifies the verb to be recognised, and the name of the local function to be called. *(See also section ??)*. When a living object gives a command which matches a verb with a command defined by an object, then the corresponding function will be called in the specified object. If this function returns 0, then next command with the same verb is tried. If the function returns 1, then the search is terminated. This enables several objects to define commands with the same verbs, but still behave different if the rest of the sentence differs. For example, there might be two armours. One is named `leather jacket`, and one is named `plate mail`. Both objects will have defined a command with the verb *wear*. If the player now gives the command '`wear jacket`', then we can't know which defined command is called first. Suppose that the wear function in the plate mail is called first. It will then detect that the argument to wear is `jacket`, not `plate mail`. It would then return 0, which would enable the game driver to call the command in the leather jacket that defines the wear verb. This function would accept the command, and execute some appropriate code, followed by a return of 1. Every time an object `O` comes in contact with a living object `L` then `O` will be asked to define commands. *(See also section ??)*

5 Call of clean_up()

The function *clean_up()* is automatically called now and then. When an object is loaded, it is checked for existence of a function *clean_up()*. If found, a flag O_WILL_CLEAN_UP is set. If an object hasn't been used (any function called) for a certain time and O_WILL_CLEAN_UP is set, then *clean_up()* is called. If this function returns a non-zero value, then O_WILL_CLEAN_UP is set again (which means that *clean_up* can be called again).

O_WILL_CLEAN_UP will also be set again after un-swapping of an object. This way, if the object gets touched and the reason why clean_up wasn't applicable may have vanished, the object is asked again what to make of the changed situation.

The idea of *clean_up()* is that objects can self-destruct, which is much more space effective that being swapped out. It will also work for cloned objects.

It can be a good idea to define a default *clean_up()* in the "much used" room.c, which will destruct the room when it is empty. If a wizard wants to save important rooms, he will have to redefine *clean_up()*.

It is of course possible to do other types of cleaning than destruction. The administrator has to define time until clean up in config.h. The time should be much longer than the time to *reset()*.

6 How to handle when a player enters a box

Suppose there is a big box in the room, which makes it possible for players to enter. Players entering the box should not be transfered into the inventory of the box, but rather into a new room, which represents the interior of the box.

There is a specific reason for this. When a player arrives in the room with the box, the function `init` will be called, which will define the `enter` command. But, if the player would be transfered into the box, he would again have an enter command defined. Similar problems exist for the `short()` and `long()` functions. They should give different messages depending on if the player is on the inside or the outside of the box.

7 LPC Reference Manual

The language used to program objects is called `LPC`. It is syntactically modelled after C. As it is important that objects be loadable "on the fly" in a game, I choose to make it an interpreted language. If it would be compiled for real, there would be big problems of portability when moving to different machines. The security of the program is also very important. Under no circumstances should a LPC programmer crash the game by doing a mistake. That rules out standard C.

Several ideas has been borrowed from object oriented languages, like inheritance. However, as performance is very important, I have not hesitated to use "impure" language constructs when needed, which will conflict with the concepts of object oriented languages.

An LPC programs consists of several building blocks:

- inheritance specification
- preprocessor
- variables
- functions

7.1 Type Declarations

Types can be used at four places:

1. Declaring type of global variables.
2. Declaring type of functions.
3. Declaring type of arguments to functions.
4. Declaring type of local variables to functions.

Normally, the type information is completely ignored, and can be regarded purely as documentation (the internal representation differs, according to the type of the value assigned to a variable but type casting is done automatically, if possible). However, when the basic type of a function is declared, then a more strict type checking will be enforced. That means that the type of all arguments must be defined. And, the variables can only be used to store values of the declared type. The function *call_other* is defined to return an unkown type, as the compiler can't know the type. This value must always be casted (when strict type checking is enabled). Type `unknown` is only returned from `call_other`.

Note that casting in LPC is not the same as casting in C. Initially it was only used as information for the compiler, and could only be used to cast values of type unknown or mixed.

There are now two different forms of type casting in lpc:

A real casting operator will be compiled if the casting syntax with '(' <type> ')' is used on a value that is **not** "mixed" or "unknown". You can also explicitely use runtime type-casting with the to_int() etc. efuns.

`x = ({ int }) y)` merely re-declares the value to be an int while `x = (int) y` does "real" type casting for all types except "unknown" and "mixed" which are just re-declared as with the first method, example: `(int *)"s" = ({ 115, 0 })` .

A type declaration for a variable may include an initialization. If the gamedriver is configured to create an `__INIT()` function (this is done by defining `INITIALIZATION_BY___INIT`) efuns may be used to initialize variables, in tubmud you may only use constant values, like that: `int a = 42;` If the constant value is an array it will be shared between all clones of the object. (Note that operations which change the size of an array create a new array).

When a function is compiled with strict type testing, it can only call other functions that are defined. If they are not yet defined, prototypes can be defined:

```
string func(int arg);
```

Note the ';' instead of a body to the function. All arguments must be given by names, but do not have to have the same names as in the real definition. All types must of course be the same.

There are two kinds of types. Basic types, and special types. There can be at most one basic type, but any number of special types. The strict type checking is only used by the compiler, not by the runtime. Hence, it is actually possible to store a number in a string variable even when strict type checking is enabled.

Why use strict type checking? It is really recommended, because the compiler will find many errors at compile time, which will save a lot of hard work. It is in general much harder to trace an error occuring at run time. I recommend, that when a wizard is having problem with an object and wants help, that he first must make all functions have declared types.

The basic types can be divided in to groups. Those that are referenced by value, and those that are referenced by address. The types `int` and `string` are always representing different entities. But the type `object` is a pointer to an object. If a value of this type is assigned to a variable or passed as an argument, both will point to the same object. The same goes for arrays and mappings, which implies that a change to an element of an array changes all variables pointing to the same array. Changing the size of the array will always allocate a new one, though. The comparation operator, ==, will compare the actual value for the group of types above. But for arrays and objects, it will simply check if it is the same object (or array). That has the very important implication that the expression '`({x }) == ({x })`' will always evaluate to false. The array construction operator-pair `({})`, if used to create an array of size zero, always returns the same pointer.

7.2 Basic Types

All uninitialized variables have the value 0. A pointer to a destructed object will always have the value 0.

`int`	An integer 32 bit number.
`status`	Boolean, either 0 or 1 (same as int).
`float`	A floating point variable.
`object`	Pointer to an object. An object pointer can mainly be used for two things. Either giving as argument to functions, or used for calling functions defined by that object with its specific instance of variables.
`string`	An unlimited string of characters. A lot of operators are allowed for strings, like + and [ ] etc.
`mapping`	This type allows indexing with strings: `value = map["index_name"];` Please note that an uninitialized mapping, like any other variable, contains an "integer zero". Since 0["string"] = "something" is an invalid assignment a mapping must be initialized before usage. You can declare a mapping like that: `mapping map = ([ ]);` Mappings cannot be initialized in declarations, except for the empty mapping, as shown above. Mappings can be initialized like that: `m = ([ "key" : "value", "key2" : "value2" ]);`
`mixed`	This type is special, in that it is valid to use in any context. Thus, if everything was declared `mixed`, then the compiler would never complain. This is of course not the idea. It is really only supposed to be used when a variable really is going to contain different types of values. This **should** be avoided if possible. It is **not** good coding practice, to allow a function for example to return different types.
`void`	This type is only usable for functions. It means that the function will not return any value. The compiler will complain (when type checking is enabled) if the return value is used.

7.3 Arrays

Arrays are declared using a '*' with a basic type. For example, declaring an array of numbers: '`int *arr;`'. Use the type mixed if you want an array of arrays, or a `mixed` combination of types.

Arrays can be allocated dynamically with the external function `allocate()`.

Arrays are stored by reference, so all assignments of whole arrays will just copy the address. The array will be deallocated when no variable points to it any longer, unless there is a circular reference, which should be avoided.

When a variable points to an array, items can be accessed by indexing, e.g., `arr[3]`. The name of the array being indexed can be any expression, even a function call, e.g. `func()[2]`. It can also be another array, if it contains pointer to arrays :

```
arr    = allocate(2);
arr[0] = allocate(3);
arr[1] = allocate(3);
```

Now `arr[1][2]` is a valid value.

The external function `sizeof()` (which is not a function in the C language) will give the number of elements in an array.

Arrays can be constructed with a list inside '`({`' and '`})`'. E.g., when you write `({1, "xx", 2})`, a new array with size 3 will be constructed, its elements will be initialised with `1, "xx"` and `2` respectively.

Partial arrays or strings can be cut out or inserted with the range operator `a[n..n+x]`. If the second index is lower than the first the range is empty. The indices in ranges may exceed the boundaries of a string/array without an error (they are adapted before indexing there, unlike in arr[a]). The size of an array or string range doesn't have to fit the size of the destination, the assignment may change the size of the range and thus of the whole string/array. You can prepend a $<$ sign to denote counting from the end. The second index of a range defaults to the size of the string or array (-1) and can be omitted. `x[<3..]`, for example, addresses the last three elements of x.

The operator `-` returns the first array excluding all members of the second array. The operator `+` concatenates two arrays and the operator `&` returns the set intersection of two arrays.

Using the ',' operator inside index brackets, whithout enclosing it by round brackets, is no longer allowed. The ',' is reserved for new types of indexed data access.

7.4 Type Modifier

There are some special types, which can be given before the basic type. These special types can also be combined. When using special type `T` before an *inherit* statement, all symbols defined by inheritance will also get the special type `T`. The only special case is public–defined symbols, which can not be redefined as *private* in a private inheritance statement.

`varargs`	A function of this type can be called with a variable number of arguments. Otherwise, the number of arguments is checked, and can generate an error.
`private`	Can be given for both functions and variables. Functions that are private in object `A` can not be called through *call_other* from another object. And, they are not accessible to any object that inherits `A`.
`static`	This special type behaves different for variables and functions. It is similar to private for functions, in that they can not be called from other objects (even though they may be called from an object that inherits `A`). static variables will be neither saved nor restored when calling *save_object()* or *restore_object()*.
`public`	A function defined as public will always be accessible from other objects, even if private inheritance is used.
`nomask`	All symbols defined as nomask can not be redefined by inheritance or shadowing *(See also section ??)*. They can still be used and accessed as usual.
`virtual`	This keyword is not implemented as of this writing. An object which is inherited 'virtual' can be inherited in several places in the inheritance tree of an object and it will still be the same object with the same set of internal variables. This keyword is meaningless for functions and variables.

There can only be one non-static variable for a given name in an object, others are silently converted to static. This ensures a saner save_object / restore_object operation.

7.5 Access of data and programs in other objects

There is a function *call_other()*, that can be used to call functions in other objects. All functions can be called except those declared `static` or `private`. *(See also section ??)* There is another syntax that will do the same thing: '`ob->func(args)`'. This will call function `func` in object `ob`. It is a much more good-looking way to do it.

There has been a lot of questions why this syntax hasn't been used to allow for accessing variables in other objects. There are some good reasons for that.

1. It conflicts with the idea of programming in an object-oriented way.
2. It makes the programming less structured, as there are more dependencies.
3. If a variable name is changed, code can be broken.
4. Sometimes, you don't even want to keep the variable at all any longer.

7.6 Comments and Preprocessor Commands

Comments are either enclosed in a `/* */` pair or start with `//` and end with the end of the line.

Preprocessing happens before the compilation of a program, hence the name. Available preprocessor commands are explained below.

7.6.1 Macros

A macro is defined with the `#define MACRO <value>` command, it sets a macro MACRO to value <value>; below the line of the definition you can use MACRO instead of <value> in the program code, example: `#define MACRO(x,y) (clone_object(x)->move(y))`. Macro names are always in uppercase by convention.

Macro definitions can be split into several lines (just like strings) with the escape character \ followed by the newline character.

The command `#undef MACRO` erases the macro definition `MACRO`.

7.6.2 Conditionals

In a macro processor, a conditional is a command that allows a part of the program to be ignored during compilation, on some conditions. In the C preprocessor, a conditional can test either an arithmetic expression or whether a name is defined as a macro.

A conditional in the C preprocessor resembles in some ways an `if` statement in C, but it is important to understand the difference between them. The condition in an `if` statement is tested during the execution of your program. Its purpose is to allow your program to behave differently from run to run, depending on the data it is operating on. The condition in a preprocessor conditional command is tested when your program is compiled. Its purpose is to allow different code to be included in the program depending on the situation at the time of compilation.

The `#ifdef` and `#ifndef` commands just verify if the following macro is (un)defined, no matter what the value of the macro is.

It is a good practise to put the expression of the matching `#if` as a comment behind the `#else` or `#endif` in long or nested conditionals.

The following conditionals exist:

1. #if
2. #else
3. #endif
4. #ifdef
5. #ifndef

7.6.3 Unintended Grouping of Arithmetic

You may have noticed that in most macros each occurrence of a macro argument name has parentheses around it. In addition, another pair of parentheses usually surrounds the entire macro definition. Here is why it is best to write macros that way:

The operator-precedence rules *(See also section ??)* may change the meaning of a macro which is not enclosed in brackets. A macro `A(a,b)` defined as `a + b`, if multiplied with c would turn into `a + (b * c)` instead of `(a + b) * c`. `A(a & b,c)` would turn into `a & (b + c)` instead of `(a & b) + c`.

7.6.4 Swallowing the Semicolon

Often it is desirable to define a macro that expands into a compound statement (which is enclosed in curly brackets *(See also section ??)*).

Strictly speaking, the macro expands to a compound statement, which is a complete statement with no need for a semicolon to end it. But it looks like a function call. So it minimizes confusion if you can use it like a function call, writing a semicolon afterward, as in `MACRO (x);`

But this can cause trouble before `else` statements, because the semicolon is actually a null statement. The presence of two statements—the compound statement and a null statement—in between a `if` condition and the `else` makes invalid C code.

The definition of a macro can be altered to solve this problem, using a 'do ... while' statement: `do {...}while (0);`

7.6.5 Including files

The `#include <file>` command inserts the given file in the position of the command. Include files have the extension `.h` by convention. `#include "file"` searches the file in the directory of the including file. `#include <file>` searches the file in the system include files, which are '`/sys/`' and '`/room/`' in tubmud. (The master object defines a function `define_include_dirs` which specifies the system include files *(See also section ??)*).

7.7 Operators

These are the operators available in LPC. They are listed in the order of precedence (low priority first):

<table>
<tr><td>expr1 , expr2</td><td>Evaluate expr1 and then expr2. The returned value is the result of expr2. The returned value of expr1 is thrown away.</td></tr>
<tr><td>var = expr</td><td>Evaluate expr, and assign the value to var. The new value of var is the result.</td></tr>
<tr><td>var += expr</td><td>Assign the value of expr+var to var. This is equivalent to var = var + expr.</td></tr>
<tr><td>var -= expr</td><td>ditto with -.</td></tr>
<tr><td>var &= expr</td><td>ditto with &.</td></tr>
<tr><td>var |= expr</td><td>ditto with |.</td></tr>
<tr><td>var ^= expr</td><td>ditto with ^.</td></tr>
<tr><td>var <<= expr</td><td>ditto with <<.</td></tr>
<tr><td>var >>= expr</td><td>ditto with >>.</td></tr>
<tr><td>var *= expr</td><td>ditto with *.</td></tr>
<tr><td>var %= expr</td><td>ditto with %.</td></tr>
<tr><td>var /= expr</td><td>ditto with /.</td></tr>
<tr><td>expr1 || expr2</td><td>The result is true if expr1 or expr2 is true. expr2 is not evaluated if expr1 was true.</td></tr>
<tr><td>expr1 && expr2</td><td>The result is true if expr1 and expr2 is true. expr2 is not evaluated if expr1 was false.</td></tr>
<tr><td>expr1 | expr2</td><td>The result is the bitwise 'or' of expr1 and expr2.</td></tr>
<tr><td>expr1 ^ expr2</td><td>The result is the bitwise 'exclusive or' of expr1 and expr2.</td></tr>
<tr><td>expr1 & exp2</td><td>The result is the bitwise 'and' of expr1 and expr2. It is also defined as a set intersection on arrays.</td></tr>
<tr><td>expr1 == exp2</td><td>Comparison. Valid for strings and numbers.</td></tr>
<tr><td>expr1 != exp1</td><td>ditto.</td></tr>
<tr><td>expr1 > exp2</td><td>ditto.</td></tr>
<tr><td>expr1 >= exp2</td><td>ditto.</td></tr>
<tr><td>expr1 < exp2</td><td>ditto.</td></tr>
<tr><td>expr1 <= exp2</td><td>ditto.</td></tr>
<tr><td>expr1 << exp2</td><td>Shift expr1 left expr2 bits.</td></tr>
<tr><td>expr1 >> exp2</td><td>Shift expr1 right expr2 bits.</td></tr>
<tr><td>expr1 + exp2</td><td>Add expr1 and expr2. If numbers, then arithmetic addition is used. If one of the expressions is a string, then that string is concatenated with the other value as a string. Two arrays can also be concatenated.</td></tr>
<tr><td>expr1 - exp2</td><td>Subtract expr2 from expr1 (int, float or arrays only).</td></tr>
<tr><td>expr1 * exp2</td><td>Multiply expr1 with expr2, yields type float if one or both operands are of type float.</td></tr>
</table>

expr1 `%` *exp2*	The modulo operator of numeric arguments.
expr1 `/` *exp2*	Division, yields type float if one or both operands are of that type.
`++` *var*	Increment the value of variable *var*, and return the new value.
`--` *var*	Decrement the value of variable *var*, and return the new value.
`-` *expr*	Compute the negative value of *var*.
`!` *expr*	Compute the logical 'not' of an integer.
`~` *expr*	The boolean 'not' of an integer.
var `++`	Increment the value of variable *var*, and return the old value.
var `--`	Decrement the value of variable *var*, and return the old value.
expr1`[`*expr2*`]`	The array given by *expr1* is indexed by *expr2*.
expr1`->`*name*`(...)`	*expr1* gives either an object or a string which is converted to an object, and calls the function *name* in this object.
`(` *expr* `)`	Expressions can be enclosed in parentheses to bypass the opererator precedence and thereby to change the order of evaluation.

7.8 Statements

Variables, functions or variables and/or functions combined by operands form expressions. A statement is an expression followed by a semicolon.

7.9 Conditions

The if-else statement is used to select a statement depending on the value of an expression; `statement1` is executed when `expr` is non-zero otherwise `statement2` is executed (if it wasn't omitted):

```
if (expr)
 statement1;
else
 statement2;
```

Instead of:

```
if (n == n1)
 statement1;
else if (n == n2)
 statement2;
else if (n == n3)
 statement3;
else
 statement4;
```

you can use the following statement:

```
switch(n) {
case n1:
 statement1; break;
case n2:
 statement2; break;
case n3:
 statement3; break;
default:
 statement4; break;
}
```

You may also use a range `n..n+k` or `'A'..'Z'` behind a `case`.

7.10 Blocks

A block is a group of statements enclosed in curly brackets, which form a new statement; local variables can be defined in a block:

```
if (b >= 0)
{
 int a;

 a = random(5);
 a += b;
 write("Result: " + a + ", " + b + "\n");
}
```

> **Note:** Currently, the local variables are *visible* until the end of the function. This is different to C and will be fixed later (any year ;-)

7.11 Loops

The do–while loop takes the form

`do {`*statement*`;} while (`*expr*`);`

Its purpose is to execute *statement* until *expr* evaluates to 0.

The while loop takes the form

`while (`*expr*`)` *statement*`;`

While *expr* evaluates to a non-zero value, *statement* will be executed.

The for statement takes the from

`for (`*expr1* `;` *expr2* `;` *expr3*`)` *statement*`;`

Execute *expr1* once. Then, while *expr2* has a non-zero value, execute *statement.* Every time *statement* has been executed, or a `continue` statement has been executed, execute *expr3* before next loop.

> **Note:** In all loops a `break` command in the statement will terminate the loop and a `continue` command will continue the execution from the beginning of the loop.

7.12 Functions

The head of a function definition consists of the (optional) return type, the function name and a list of arguments. The body is a block in which the arguments are local variables:

```
static string
my_function (int is_a_player, object somebody)
{
 string desc;

 if (is_a_player)
  desc = somebody->query_name();
 else
  desc = somebody->short();

 return desc;
}
```

A locally defined function can have any number of arguments. All basic types can be sent in the argument. A return value is sent with the 'return' statement. All seven data types can be used in the return statement. The type of the function must not be declared but if a basic type is declared strict type checking is enabled in the body of the function.

Uninitialized arguments are set to 0.

Arguments which are prefixed with & are passed "call-by-reference", which means that they "become" the local variables in the called function, instead of merely initializing them. The difference is, of course, that an assignment to the local variable also changes the associated variable of the calling function. If you want to pass the member of an array by reference it has to be in parenthesis: `&(a[n])`.

The type of the arguments must not be declared when strict type checking is disabled. It is illegal to have a function with the same name as a function in the same object, or local variable. Functions are reentrant.

If there is no return statement, the number 0 will be returned.

If a function has the type 'static', then it will not be possible to call it with `call_other()` from another object.

There are two kinds of functions:

efun
: The hard coded functions, which are defined by the game driver. They can be redefined by a local function of the same name, which will then be used instead. Redefinition of efuns and the definiton of efuns which have been removed or are to be added later is done in the file '`simul_efun`'. Functions defined in this object are available to all other objects except the master object and except objects inherited by master and/or the simul_efun object.

lfun
: Functions that can be defined by objects. These functions will be called by other lfuns, and sometimes by the game driver. They control the behaviour of an object. An example is *get()*, which if defined and returning 1, will enable the object to be picked up by players. If returning 0, then the player will get a message that says that the object can't be picked up.

7.13 Saving and Restoring Objects

7.14 Inheritance

An object can inherit all variables and functions from another object. This is done with the declaration

`inherit "`*file*`";`

This declaration must precede the declaration of any local variables or functions. When an inherited local function *fun* is redefined, it can still be accessed with '`::`*fun*`(...)`'. The name of the module which contains the function can pe prepended like that: `door::open()`. `door` may even be a string literal containing the name of the module. An example, defining a monster:

```
inherit "obj/monster";

    reset(arg) {
      ::reset(arg);
      set_name("troll");
      set_level(9); set_hp(100); set_wc(12); set_al(-60);
      set_short("A troll");
      set_long("It is a nasty troll that looks very aggressive.\n");
      set_aggressive(1);
    }
```

7.15 Shadowing

8 External Functions (efun)

The following efuns have no separate entries below and should be self explanatory: sin, cos, tan, asin, acos, atan, exp, log, sqrt. The only argument and return type of these efuns is float.

8.1 add_action

```
void    add_action(fun,cmd,flag)
string  fun,cmd;
int     flag;
```

Set up a local function `fun` to be called when user input matches the command `cmd`. Functions called by a player command will get the arguments as a string. It must then return 0, if it was the wrong command, otherwise 1. If it was the wrong command, the parser will continue searching for another command, until one returns true, or give an error message to the player.

For example, there can be a wand and a rod. Both of these objects defines `add_verb("wave")`. One of them will be randomly called first, and it must look at the argument, and match against `"wand"` or `"rod"` respectively.

If the second argument (`cmd`) is not given, it must be given by `add_verb()`. The function `add_verb()` is still supported, because of historical reasons.

If argument `flag` is 1, then only the leading characters of the command have to match the verb `cmd`.

Always have `add_action()` called only from an `init()` routine. The object that defines commands must be present to the player, either being the player, being carried by the player, being the room around the player, or being an object in the same room as the player. Never define an action that will call the function `exit()`, because it is a special function.

See also efun/query_verb, efun/query_actions, lfun/init, lfun/exit.

8.2 add_verb

```
void    add_verb(str)
string  str;
```

This function is connected to the `add_action()` function. It will set up the command `str` to trigger a call to the function set up by the previous call to `add_action()`.

This function is now obsolete as the verb can be given directly as a second parameter when the function `add_action()` is called. `add_verb()` remains for compatibility.

See also efun/add_action, efun/query_verb.

8.3 add_worth

```
void    add_worth();
```

Used by the shop to update the wizlist. Has argument(s). May not be called if creator() is non-zero. This efun should be considered obsolete, and is to be replaced with a simul_efun.

8.4 add_xverb

```
void    add_xverb(str)
string  str;
```

Same as `add_verb()` but only the leading characters of the command have to match `str`. This function differs from `add_action` with three parameters because the string which is passed to the function (which has been added by `add_action("function")`) is the remainder of the commandline while `add_action(,,1)` removes the first word of the command line and the added verb just has to match the beginning of this word. `query_verb` always returns the first word, not the added verb.

See also: efun/add_action, efun/add_verb.

8.5 all_environment

This function is defined in /obj/simul_efun.c

```
object  *all_environment(ob)
object  ob;
```

Gives an array of all containers which an object is in, i.e. for match in matchbox in bigbox in chest in room all_environment(match) would return `({matchbox, bigbox, chest, room })`

8.6 all_inventory

```
object  *all_inventory(ob)
object  ob;
```

Returns an array of the objects contained in the inventory of `ob`.

See also: efun/first_inventory, efun/next_inventory.

8.7 allocate

```
mixed   *allocate(size)
int     size;
```

Allocate an array of `size` elements. The number of elements must be non-negative and it must not be bigger than the system maximum (1000 by default; 3000 in tubmub, as of this writing).

See also efun/sizeof.

8.8 assoc

```
mixed   *assoc(key,keys,data_or_fail,fail)
mixed   key,*keys,*data_or_fail,fail;
```

Searches a key in an alist.

Three modes of calling:

1. With exactly two arguments, the second being an array which's first element is no array. In this case the entire array is searched for the key; -1 is returned if not found, else the index (like member˙array, but faster).
2. With two or three arguments, the second being an array which's first element is an array. The array has to have a second element of the same size; the key is searched in the first and the associated element of the second array that is element of second argument is returned if succesful; if not, 0 is returned, or the third argument, if given.
3. With three or four arguments, the second being an array of keys (first element no array) and the second is a matching data array. returns 0 or fourth argument (if given) for failure, or the matching entry in the array given as third argument for success.

Complexity : O(lg(n)) , where n is the number of keys. Return value is undefined if another list is given in place of a presorted key list.

See also: LPC/alists order_alist insert_alist

8.9 break_string

This function is defined in /obj/simul_efun.c

```
string  break_string(str,width,indent)
string  str;
int     width,indent;
```

Breaks a continous string without newlines into a string with newlines inserted at regular intervalls replacing spaces. Each newline separated string can be indented with a given number of spaces.

8.10 caller_stack This function is not yet implemented

`object  *caller_stack()`

Return a list of all previous objects.

See also: previous_object, caller_stack_depth.

8.11 caller_stack_depth

`int     caller_stack_depth();`

Returns the depth (size) of the caller stack.

See also: previous_object, caller_stack.

8.12 call_other

```
unknown call_other(ob,str,arg)
object  ob;
string  str;
```

Call a function in another object with one or more arguments. The return value from the function of the other object is returned. The type of the returned variable is undefined, when strict type checking is enabled type casting must be used with every call_other.

Note: The return type is really `unknown`. This matters when type checking is enabled; the return value has to be casted in this case: `value = (<type_name>) call_other(...);`.

See also efun/present, efun/find_living.

8.13 call_out

```
void    call_out(fun,delay,arg)
string  fun;
int     delay;
```

Set up a call of function `fun` in `this_object()`. The call will take place in `delay` seconds, with the argument `arg` provided. `arg` can be of any type. call'out can go to static functions.

See also efun/remove_call_out.

8.14 call_out_info

`mixed   *call_out_info()`

Get information about **all** pending call outs. An array is returned, where every item in the array consists of 4 elements:

1. The object
2. The function
3. The delay to go
4. The optional argument

See also: call_out, remove_call_out.

8.15 call_resolved

```
int     call_resolved(&result,ob,func,arg1)
mixed   result,ob,arg1;
string  func;
```

This function is similar to call_other. Returns 1 if the functions exists, 0 otherwise. The return value is stored in `result`. Note that call_resolved calls functions which are defined in shadows (like call_other) while function_exists ignores shadows. The first argument result has to be passed as a reference (prefixed with a &); it will be overwritten if the function exists and left unchanged otherwise.

See also efun/function_exists, efun/call_other.

8.16 capitalize

```
string  capitalize(str)
string  str;
```

Convert the first character in `str` to upper case, and return the new string.

8.17 cat

```
int     cat(path,start,num)
string  path;
int     start,num;
```

List the file found at `path`. It is not legal to have periods or spaces in the path. This command is normally connected to the `cat` command that wizards have. It is also used by the `help` command. The optional arguments `start` and `num` specify the number of the first line and the number of lines to print. If they are not given, the whole file is printed from the beginning.

The total number lines will not exceed a system limit, which is normally set to 40 lines.

`cat()` returns 1 if success, 0 if no such file or no such lines.

See also efun/ls, efun/file_size.

8.18 catch

```
mixed   catch(expr)
```

Evaluate `expr`. If there is no error, 0 is returned. If there is a standard error, a string (with a leading '`*`') will be returned.

The function `throw(value)` can also be used to immediately return any value, except 0.

The `catch()` is somewhat costly, and should not be used everywhere. Rather use it at places where an error would destroy consistency.

8.19 cindent

```
void    cindent(file)
string  file;
```

This function is rumoured to run an external C beautifier over `file`. Use at your own risk ;) The external program is not installed in tubmud but the "I" command of ed can be used instead.

8.20 clear_bit

```
string  clear_bit(str,n)
string  str;
int     n;
```

Return the new string where bit `n` is cleared in string `str`. Note that the old string `str` is not modified.

See also efun/set_bit, efun/test_bit.

8.21 clone_object

```
object  clone_object(name)
string  name;
```

Load a new object from definition `name`, and give it a new unique name. Return the new object. (The original used for cloning should not be used in the game. It should only be used for cloning.) Objects which are to be used as blueprints may not have an environment.

See also efun/destruct, efun/move_object, efun/clonep.

8.22 clonep This function is defined in /obj/simul_efun.c

```
void    clonep(ob)
object  ob;
```

Return 1 if `ob` is a cloned object. The argument is optional and defaults to this_object().

See also efun/stringp, efun/intp, efun/floatp, efun/pointerp.

8.23 command

```
int     command(str,ob)
string  str;
object  ob;
```

Execute `str` as a command given directly by the player. Any effects of the command will apply to the current object. If the second optional argument is present, then the command is executed for that object.

The return value is the evaluation cost (0 on failure).

Note: the evaluation cost returned may be wrong if command() is used from inside a catch().

See also efun/enable_commands.

8.24 create_wizard This function is defined in /obj/simul_efun.c

```
string  create_wizard(name,domain)
string  name,domain;
```

Create the environment and castle for a wizard. *Do not use this if you are not sure about what you are doing!* It will create a new directory for the wizard with the name '`name`', and copy a definition of a castle to this directory. It will also set up automatic loading of this castle for the start of the game.

It returns the name of the new castle. In case of error, false is returned.

8.25 creator

```
string  creator(ob)
object  ob;
```

Return as a string the name of the wizard that created object `ob`. If the object was not created by a wizard, 0 is returned.

8.26 creator_file This function is defined in /obj/simul_efun.c

```
string  creator_file(str)
string  str;
```

creator_file takes a file name as its argument and derives the name of the wizard/domain that created it.

```
Example:  creator_file("/players/macbeth/castle") == "macbeth"
       creator_file("/domains/banking/rcs/shop") == "Banking"
       creator_file("/open/test") == "/nil/"
       creator_file("/obj/player") == 1    /* mudlib backbone */
```

```
        creator_file("/ftp/test") == 0      /* can't be loaded or cloned */
```

Please note that the leading slash isn't obligatory, i.e.
creator_file("/"+file) == creator_file(file);

8.27 crypt

`string  crypt(str,seed);`

Crypt the string `str` using two characters from `seed` as a seed. If `seed` is 0, then a random seed is used. The result has the first two characters as the seed.

8.28 ctime

```
string  ctime(clock)
int     clock;
```

Give a nice string with current date and time, with the argument `clock` that is the number of seconds since 1970.

See also efun/time.

8.29 debug_info

```
void    debug_info(flag,arg)
int     flag;
mixed   arg;
```

Gives some debug information depending on `flag`. Flag may currently be 0, 1 or 2, in the first two cases arg has to be an object. (case 2: get object from specified object list pos)

8.30 deep_inventory

This function is defined in /obj/simul_efun.c

```
object  *deep_inventory(ob)
object  ob;
```

Recursivly collects the contents of an object and all objects inside it.

8.31 destruct

```
void    destruct(ob)
object  ob;
```

Completely destroy and remove object `ob`. The argument can also be a string. After the call to `destruct()`, no global variables will exist any longer, only local, and arguments.

The master object of tubmud calls `notify_destruct` in the object to be destructed and `notify_leave` in the environment of the object.

There are few things guranteed for destruct:

- All references to the object will be 0, including this_object() in the destructed object itself.
- If an object selfdestructs, it's executing function is _NOT_ terminated as in former versions.
- If an object is destructed and a call in a function of this object returns, it will be executed. This means: if A calls a function in B, which destructs A and returns, the function calling B will continue execution (in A).
- The values of global object variables _may_ vanish.
- The destructed object fails to call other objects. (!)

- Everything pending on call_out and input_to will be cleared.
- A destruct is not reversible (unless the delinquent was the master and its successor can't be loaded :-)
- If someone carried the object that was destructed, his local weight will be updated, if master.c chooses to do so.

The update of weight happens in '`/obj/master::prepare_destruct()`'. There won't be an update of weight in native mode.

See also efun/clone_object.

8.32 disable_commands

```
void    disable_commands()
```

Reverses enable_commands().

8.33 dump_array

This function is defined in /obj/simul_efun.c

```
void    dump_array(var)
mixed   var;
```

Dumps a variable with write() for debugging purposes.

8.34 ed

```
int     ed(file,exithandler)
string  file,exithandler;
```

This is a funny function. It will start a local editor on an optional file. This editor is almost 'ed' compatible.

The optinal second parameter defines a function to be called upon exiting ed.

8.35 enable_commands

```
void    enable_commands()
```

Enable this object to use commands normally accessible to players. This also marks the current object as 'living'. Commands defined by `player.c` will not be accessible, of course.

This function must be called if the object is supposed to interact with other players.

Avoid to call this function from other places than `reset()`, because the `command_giver` will be set to this object, this means that `this_player()` will return the object which called `enable_commands` until the end of the execution.

See also efun/command, efun/living, efun/disable_commands.

8.36 environment

```
object  environment(obj)
object  obj;
```

Return the surrounding object to `obj`. If no argument is given, it returns the surrounding to the current object.

See also efun/first_inventory, efun/this_player, efun/this_object.

8.37 exclude_array

This function is defined in /obj/simul_efun.c

```
mixed   *exclude_array(arr,from,to)
mixed   *arr;
int     from,to;
```

Deletes a section of an array.

8.38 exclude_element

This function is defined in /obj/simul_efun.c

```
mixed   *exclude_element(arr,index)
mixed   *arr;
int     index;
```

Deletes an element from an array.

8.39 exec

```
void    exec(new,old)
object  new,old;
```

This function replaces the player object of an interactive player with another object, preferably a player object.

8.40 explode

```
mixed   explode(str,del)
string  str,del;
```

Return an array of strings, created when the string `str` is splitted into substrings as divided by `del`. The `str` must end with `del` if the last part is wanted too.

Example: `explode(str," ")` will split the string `str` into an array of words as separated by spaces in the original string. The array is returned.

In tubmud implode(explode(str,del),del) is the same as str and if del is "" the string is splitted into single-char-strings.

You can have the old behaviour when you define OLD_EXPLODE_BEHAVIOUR in config.h. It stripped off all empty strings at the beginning of the returned array and at most one at its end.

See also efun/sscanf, efun/extract.

8.41 export_uid

This function is defined in native mode only

```
void    export_uid(ob)
object  ob;
```

Set the uid of object `ob` to `this_object()`'s effective uid. It is only possible when object `ob` has an effective uid of `0`.

See also efun/seteuid efun/getuid

8.42 extract

```
string  extract(str,from,to)
string  str;
int     from,to;
```

Extract a substring from a string. Character 0 is the first character. `extract(str,n)` will return a substring from characer number '`n`' to the end. `extract(str,i,j)` will return a string from character '`i`' to character '`j`'. Indexes may be greater than the string length; they will be reduced automagically.

Extract will now accept array, too.

See also efun/sscanf, efun/explode.

8.43 file_name

```
string  file_name(ob)
object  ob;
```

Get the file name of an object. If the object is a cloned object, then it will not have any corresponding file name, but rather a new name based on the original file name.

Example: `find_object(file_name(ob))==ob` is guaranteed to be true for all objects `ob`.

See also efun/program_name, efun/find_object.

8.44 file_size

```
int     file_size(file)
string  file;
```

Give the size of a file. Size −1 indicates that the file either does not exist, or that it is not readable by you. Size −2 indicates that it is a directory.

See also efun/save_object, efun/load_object, efun/write_file, efun/cat.

8.45 file_time This function is defined in /obj/simul_efun.c

```
int     file_time(file)
string  file;
```

Return the date of the last write access to the file in seconds since 1970.

See also efun/time, efun/ctime

8.46 filter_array

```
mixed   *filter_array(arr,fun,ob,extra)
mixed   *arr;
string  fun;
object  ob;
mixed   extra;
```

Returns an array holding the items of `arr` filtered through '`ob->fun()`'. The function `fun` in `ob` is called for each element in `arr` with that element as parameter. A second parameter `extra` is sent in each call if given. If '`ob->fun(arr[.index.], extra)`' returns non-zero the element is included in the returned array.

If `arr` is not an array, then 0 will be returned.

The last two arguments are optional, `ob` defaults to `this_object`.

See also efun/map_array.

8.47 filter_mapping

```
mapping filter_mapping(map,fun,ob,extra)
mapping *map;
string  fun;
object  ob;
mixed   extra;
```

Similar to filter_array but takes a mapping instead of an array.

8.48 filter_objects

```
object  *filter_objects(arr,fun,extra)
object  *arr;
string  fun;
mixed   extra;
```

Similar to filter_array but calls `arr[n]->fun(extra)`. If the call returns non-zero the object arr[n] is included in the returned array. You can give an arbitrary number of arguments to pass to `fun` after `fun`.

Note: `filter_objects` has been the original name for `filter_array` which was renamed when the other `*_array` functions were implemented. This is no longer an alias for `filter_array`!

8.49 find_living

```
object  find_living(str)
string  str;
```

Find first 'living' object named `str`. The id() function is not used, only the 'living name' is recognised. A living object is an object that has done `enable_commands()`. The object must have set a name with `set_living_name()`. There is a special hash table that speeds up the search for living objects.

See also efun/find_player, efun/enable_commands, efun/set_living_name.

8.50 find_object

```
object  find_object(str)
string  str;
```

Find an object with the file name `str`. If the file isn't loaded, it will not be found.

8.51 find_player

```
object  find_player(str)
string  str;
```

Find a player with the name `str`. The string must be lowercase. Players are found even if they are invisible or link dead. Monsters are not found.

This function uses the name that was set by `set_living_name()`. This is done automatically in `player.c`.

See also efun/find_living, efun/set_living_name.

8.52 first_inventory

```
object  first_inventory(ob)
object  ob;
```

Get the first object in the inventory of `ob`.

See also efun/next_inventory.

8.53 floatp

```
int     floatp(arg)
mixed   arg;
```

Return 1 if `arg` is a floating point value.

See also clonep, stringp, pointerp, objectp, referencep, mappingp, intp.

8.54 function_exists

```
string  function_exists(str,ob)
string  ob;
object  ob;
```

Return the file name of the object that defines the function `str` in object `ob`. The returned value can be other than '`file_name(ob)`' if the function is defined by an inherited object. 0 is returned if the function was not defined.

Shadows are ignored by function_exists.

See also efun/call˙resolved.

8.55 get_dir

```
string  *get_dir(path,mask)
string  path;
int     mask;
```

Return the directory specified by `path`; accepts wildcards. the (optional) second parameter is a bit mask:

1	return name
2	return size
4	return date
8	reserved for data
16	reserved for data
32	give unsorted output
64	reserved for array format
128	reserved for array format

A flat array is returned, no matter which bits are set in the mask. With all bits set the array would look like this: ({ file1, size1, date1, file2 ... })

8.56 get_error_file

```
string  get_error_file(name,flag)
string  name;
int     flag;
```

Returns the file where a compilation error occured. If the flag is set the error will be forgotten.

8.57 get_extra_wizinfo

```
wizinfo get_extra_wizinfo(wiz)
mixed   wiz;
```

Get the extra wizinfo field for `wiz`. `wiz` may be a string denoting the name of a wizard, or an object created by the wizard. Arrays are not copied. The call causes a privilege violation. *(See also section ??)*

See also efun/set_extra_wizinfo.

8.58 getuid This function is defined in native mode only

```
string  getuid(ob)
object  ob;
```

Get the name of the wizard that is set to the user of this object. That name is also the name used in the wizlist.

See also efun/seteuid.

8.59 heart_beat_info

```
object  *heart_beat_info()
```

Return an array with all objects which have a heart_beat running.

8.60 implode

```
string  implode(arr,del)
```

Concatenate all strings found in the array `arr` with the string `del` between each element. Only strings are used from the array.

See also efun/explode.

8.61 inherit_list

```
string  *inherit_list(ob);
object  ob;
```

Return a list of filenames of objects inherited by `ob`. The object itself is in slot 0 of the returned array.

See also: efun/replace_program

8.62 input_to

```
void    input_to(fun,flag)
string  fun;
int     flag;
```

Enable next line of user input to be sent to the local function **`fun`** as an argument. The input line will not be parsed.

Note that **`fun`** is not called immediately. It will not be called until the current execution has terminated, and the player has given a new command.

If **`input_to()`** is called more than once in the same execution, only the first call has any effect.

If optional argument **`flag`** is non-zero, the line given by the player will not be echoed, and is not seen if snooped.

See also efun/call_other, efun/sscanf.

8.63 insert_alist

```
mixed   insert_alist(key,data_or_key_list,...,alist)
mixed   key,data_or_key_list,*alist;
```

inserts an entry into an alist, or shows the place where this is to be done.

- When called with the last argument being an alist: The first argument is a key to be inserted, the second and all the following but the last are data to associate it with. The last has to be an array with as much elements as key and data arguments are given, the matching key and data arrays; this should be already an alist, or the return value will neither be an alist. Return value is the enlarged assoc list (array of two arrays). If the key is already in the list, the data is simply replaced in the returned list.
- When called with the last argument beinig a list of non-lists: The call has to be done with exactly two arguments. The first argument is a key to be inserted in the presorted key list (first element of an array that is an alist) that has to be given as second argument. Return value is the index where the key has to be inserted to preserve the structure of a presorted alist, or the index where the key has been found. Return value is an int.
  CAVEATS: when called with certain string keys, the correct place might change after another call to insert_alist in this mode, so use the index while it is fresh.

Complexity O(lg(n) + a $\times$ n) Where n is the number of keys and a is a very small constant (for block move);

See also: LPC/alists efun/order_alist efun/assoc efun/intersect˙alist

8.64 intersect_alist

```
mixed   *intersect_alist(alist1,alist2)
mixed   *alist1,*alist2;
```

Does a fast set intersection on alists.

8.65 intp

```
int     intp(arg)
mixed   arg;
```

Return 1 if `arg` is an integer number.

See also clonep, stringp, pointerp, objectp, referencep, mappingp, floatp.

8.66 interactive

```
int     interactive(ob)
object  ob;
```

Return true if `ob` is interactive.

8.67 living

```
int     living(ob)
object  ob;
```

Return true if `ob` is a living object (i.e., `enable_commands()` has been called by `ob`).

8.68 localcmd

This function is defined in /obj/simul_efun.c

```
void    localcmd()
```

Prints all currently defined actions.

See also efun/add_action.

8.69 load_object

This function is defined in /obj/simul_efun.c

```
object  load_object(filename);
string  filename;
```

Loads file `filename` and returns the object pointer.

8.70 log_file

This function is defined in /obj/simul_efun.c

```
void    log_file(file,message)
string  file;
string  message;
```

Append a message to a log file. All log files are in the directory `mudlib/log`. '`/log`' is automatically prepended to the file name.

See also efun/write_file.

8.71 lower_case

```
string  lower_case(str)
string  str;
```

Convert the all characters in `str` to lower case, and return the new string.

8.72 ls

This function is defined in /obj/simul_efun.c

```
void    ls(path)
char    path;
```

List files in an optional path. It is not allowed to use periods or spaces in the path. This function is normally connected to the `ls` command that wizards have.

See also efun/cat.

8.73 map_array

```
mixed   *map_array(arr,fun,ob,extra)
mixed   *arr;
string  fun;
object  ob;
mixed   extra;
```

Returns an array holding the items of `arr` mapped through '`ob->fun()`'. The function `fun` in `ob` is called for each element in `arr` with that element as parameter. A second parameter `extra` is sent in each call if given. Principal function:

```
foreach (index) arr[index] = ob->fun(arr[index],extra);
```

The value returned by '`ob->fun(arr[.index.], extra)`' replaces the existing element in the array. If `arr` is not an array, then 0 will be returned.

The last two arguments are optional, `ob` defaults to `this_object`.

See also efun/filter_array.

8.74 map_mapping

```
mapping map_mapping(map,fun,ob,extra)
mapping map;
string  fun;
object  ob;
mixed   extra;
```

Similar to map_array but takes a mapping instead of an array.

8.75 map_objects

```
object  *map_objects(arr,fun,extra)
object  *arr;
string  fun;
mixed   extra;
```

Similar to map_array but calls `arr[n]->fun(extra)`. The return value replaces the object in the returned array. Can pass any number of arguments to `fun` like filter_objects() .

8.76 member

```
int     member(arr,item)
mixed   arr,item;
```

Takes an array, mapping or string as first argument. In case of mappings it returns 1 if `item` is a member of the mapping, otherwise it returns the index of `item` in `array`.

See also efun/member_array.

8.77 member_array

```
int     member_array(item,arr)
mixed   item,*arr;
```

Returns the index of the first occurence of `item` in array `arr`. If not found, then -1 is returned. `arr` may be a string if item is a single character `'c'`.

See also efun/member.

8.78 m_delete

```
mapping m_delete(map,index)
mapping map;
mixed   index;
```

Remove the entry with index `index` from mapping `map`, and return the changed mapping. If the mapping does not have an entry with index `index`, the first argument is returned.

See also mappingp, mkmapping, m_indices, m_values, m_sizeof

8.79 m_indices

```
mixed   *m_indices(map)
mapping map;
```

Return an array containing the indices of mapping `map`.

See also: mappingp, mkmapping, m_values, m_delete, m_sizeof

8.80 m_sizeof

```
int     m_sizeof(map)
mapping map;
```

Return the number of indices (or values) in mapping `map`. This function is obsolete (it is merely an alias for sizeof() now).

See also: mappingp, mkmapping, m_indices, m_values, m_delete

8.81 m_values

```
mixed   *m_values(map)
mapping map;
```

Return an array with the values of mapping `map`.

See also: mappingp, mkmapping, m_indices, m_delete, m_sizeof

8.82 mappingp

```
int     mappingp(arg)
mixed   arg;
```

Return 1 if the argument `arg` is a mapping, or 0 if it is not.

See also intp, clonep, stringp, pointerp, objectp, referencep, floatp, mkmapping, m_indices, m_values, m_delete, m_sizeof

8.83 mapping_contains

```
int     mapping_contains(&result,map,key)
mapping map;
mixed   key,
```

result;

Returns 1 if `key` is in `map`. `result` contains the value associated with the key. Note that a reference to `result` has to be passed to mapping_contains, that's what the & is good for. The result won't be overwritten if the mapping doesn't contain a value for the index.

8.84 mkmapping

```
mapping mkmapping(arr1,arr2)
mixed   *arr1,*arr2;
```

Return a mapping with indices from `arr1` and values from codearr2. `arr1[0]` will index `arr2[0]`, `arr1[1]` will index `arr2[1]`, etc. If the arrays are of unequal size, the mapping will only contain as much elements as are in the smallest array.

See also: mappingp, m_indices, m_values, m_delete, m_sizeof

8.85 mkdir

```
int     mkdir(file)
string  file;
```

Create a directory `file`. Return 1 for success and 0 for failure.

8.86 move_object This function is defined in /obj/simul_efun.c

```
void    move_object(item,dest)
object  item,dest;
```

Move the object `item` to the object `dest`. Currently, both arguments can be strings. Usually, `transfer()` should be used instead of `move_object()`.

In native mode an object may only move itself.

When you call `move_object` the call is intercepted by the simul_efun `move_object` which checks if your request to move an object is accepted by the object's current environment, the new environment and the object itself. The function `prevent_leave(ob,to)` is called in the current environment (if it exits); `prevent_enter(from,ob)` is called in the new environment and `prevent_move(ob,to)` is called in the object itself, if one of them returns non-zero move_object() will fail. The functions `notify_leave(ob,to)`, `notify_move(from,to)` & `notify_enter(from,ob)` are called like their cousins when the object has been moved. Errors in the notify functions will be caught (see efun/catch). When an object is destructed `notify_leave` is called in its environment (and `notify_destruct` in the object itself).

See also efun/transfer, efun/first_inventory, efun/this_object, efun/this_player, efun/catch.

8.87 next_inventory

```
object  next_inventory(ob)
object  ob;
```

Get next object in the same inventory as `ob`.

Warning: If the object `ob` is moved by `move_object()`, then `next_inventory()` will return an object from the new inventory.

See also efun/first_inventory.

8.88 notify_fail

```
void    notify_fail(str)
string  str;
```

Store `str` as the error message given instead of the default message '`What ?`'.

If `notify_fail()` is called more than once, only the last call of will be used.

The idea of this function is to give better error messages instead of simply '`What ?`'.

8.89 objectp

```
int     objectp(arg)
```

Return 1 if `arg` is an object.

See also intp, clonep, stringp, pointerp, referencep, floatp, mappingp.

8.90 order_alist

```
mixed   *order_alist(keys,data,...)
mixed   *keys,*data;
```

Creates an alist. Keys have to be of type integer, string or object. Types can be mixed.

Either takes an array containing keys, and others containing the associated data, where all arrays are to be of the same length, or takes a single array that contains as first member the array of keys and has an arbitrary number of other members containing data, each of wich has to be of the same length as the key array. Returns an array holding the sorted key array and the data arrays; the same permutation that is applied to the key array is applied to all data arrays.

Complexity is O(n $\times$ lg(n) $\times$ m), where n is the number of elements in the key array and m is the number of data arrays + 1;

Note that the the dimensions of the arrays are used the other way than in lisp to allow for faster searching.

Global variables can be initialized with results from order_alist()

See also LPC/alists, efun/insert_alist, efun/assoc.

8.91 parse_command

```
int     parse_command(str,source,pattern,var1,var2...)
string  str;
object  source;    /* Object or array holding objects */
string  pattern;
```

This function parses a command given in `str` against the pattern in `pattern` and returns 1 if it matches. `source` is either an object or an array of objects. This is essentially a 'hotted' `sscanf()` and it has a similar syntax, although `parse_command()` works on word basis whereas `sscanf()` works on character basis.

`str`: The given command.

`source`: If this parameter is an array, it holds the accessible objects. If it is an object, it is the object from which to recurse and create the list of accessible objects, normally

```
ob = environment(this_player())
```

`pattern`: The parse pattern as list of words and formats with the following syntax:

'word'	obligatory text (one word);
[word]	optional text (one word);
/	alternative marker;
%o	single item, object;
%l	single living object;
%s	any text (multiple words);
%w	any word;
%p	preposition;
%i	any items;
%d	number, $0 \ldots \infty$ or textual $0 \ldots 99$.

An example parse pattern: `" 'get' / 'take' %i "` Items as in `%o` and `%i` can have many forms, some examples:

apple, two apples, twentyfirst apple apples, all apples, all green apples, all green ones

`var`*n*: This is the list of result variables as in `sscanf()` One variable is needed for each %... The return types for the different %... are:

- `%o` Returns an object.
- `%l` Returns an object.
- `%s` Returns a string of words.
- `%w` Returns a string of one word.
- `%p` The variable can hold a list of words in an array on function entry, or it can hold an empty variable. If an empty variable was given it will contain a string afterwards. If an array was given `array[0]` will be the matched word afterwards.
- `%i` Returns a special array of this form:

  Element [0] (int) is the given numeric prefix:

  - $= 0$: 'all' or a plural form given
  - > 0: numeral given: two, three, four, ...
  - < 0: order given: second, third, ...

  The elements $[1] \ldots [n]$ (object) are object pointers that constitute a list of all *possible* objects that can match the given `%i`. No choosing of 'third' or such will take place.
- `%d` Returns a number.

Example:

```
a=parse_command("take apple",environment(this_player()),
                                      " 'get' / 'take' %i ",items);
```

8.92 pointerp

```
int     pointerp(arg)
```

Return 1 if `arg` is an array.

See also intp, clonep, stringp, objectp, referencep, floatp, mappingp.

8.93 present

```
object  present(str,ob)
string  str;
object  ob;
```

If an object that identifies to the name `str` is present, then return it. An object that wishes to identify as `str` has to define a function `id` that returns 1 if called with `str` as argument. By default `str` is searched in the inventory and environment of this_player(). The first argument may also be an object pointer (which makes searching faster). If the second argument is given, search takes place only in its inventory and not in its environment.

`str` can also be an object.

The object is searched for in the inventory of the current player, and in the inventory of the environment of the current player.

A second optional argument `ob` is the enviroment where the search for the `str` is done. Normally `this_player()` is a good environment.

See also efun/move_object, efun/environment.

8.94 previous_object

```
object  previous_object(depth)
int     depth;
```

Returns an object pointer to the last object that called the current function, if any. If the optional argument `depth` is supplied it scans back the trace by depth external calls. previous_object(1) returns what previous_object() would have returned, if called by the previous object. Destructed objects are zero.

See also efun/call_other, efun/call_resolved, efun/caller_stack, efun/caller_stack_depth.

8.95 printf

```
void    printf(fmt,arg,...)
string  fmt;
mixed   arg;
```

(See also section ??)

8.96 process_string

This function is going to be removed

```
string  process_string(str)
string  str;
```

Returns a string where occurences of '`@@function[:filename][|arg|arg]@@`' are replaced the return code of the specified function. Note that process˙string does not recurse over returned replacement values. If a function returns another function description, that description will not be replaced. Note that both object and arguments are optional.

Example (added after reading TMI docs :-)

```
"You are chased by @@query_name:/players/myself/orc@@ eastward."
```

is replaced by (if `query_name` in `/players/myself/orc` returns `"Orc"`):

```
"You are chased by Orc eastward."
```

8.97 program˙name

This function is defined in /obj/simul_efun.c

```
string  program_name(ob)
object  ob;
```

program_name takes an object and returns the file name of the class it was cloned from or the file name of the object itself if it is a blueprint.

`program_name(load_object(file)) == program_name(clone_object(file))`

See also efun/load_object, efun/clone_object, efun/file_name.

8.98 program_time

```
int     program_time(ob)
object  ob;
```

Returns the creation time of the program struct which belongs to object ob.

8.99 query_actions

```
mixed   *query_actions(ob,mask_or_verb)
mixed   ob,mask_or_verb;
```

`query_actions` takes either an object or a filename as first argument and a bitmask or string as a second argument. If the second argument is a string query˙actions will return an array containing information (see below) on the verb or zero if the living object `ob` cannot use the verb. If the second argument is a bitmask `query_actions` will return a flat array containing information on all verbs added to `ob`. The second argument is optional (default is the bitmask 1).

1	the verb
2	typ
4	short_verb
8	object
16	function

`typ` is one of the values defined in '`/sys/sent.h`', which is a gamedriver include file:

SENT_PLAIN	added with `add_action(fun,cmd);`
SENT_SHORT_VERB	added with `add_action(fun,cmd,1);`
SENT_NO_SPACE	added with `add_action(fun); add_xverb(cmd);`
SENT_NO_VERB	just an `add_action(fun);` without a verb
SENT_MARKER	internal, won't be in the returned array

8.100 query_editing

```
int     query_editing(pl)
object  pl;
```

Returns 1 if `pl` currently uses ed.

See also efun/query_input_pending.

8.101 query_idle

```
int     query_idle(ob)
object  ob;
```

Query how many seconds a player object has been idle.

8.102 query_imp_port

```
int     query_imp_port()
```

Returns the port number for incoming imps. *(See also section ??)*

See also efun/query_mud_port, efun/send_imp.

8.103 query_input_pending

```
int     query_input_pending(pl)
object  pl;
```

Returns 1 if the gamedriver is waiting for `input_to` from player `pl`.

See also efun/query_editing, efun/input_to.

8.104 query_ip_name

```
string  query_ip_name(ob)
object  ob;
```

Give the ip-name for player `ob`. An asynchronous process '`hname`' is used to find out these name in parallell. If there are any failures to find the ip-name, then the ip-number is returned instead.

8.105 query_ip_number

```
string  query_ip_number(ob)
object  ob;
```

Give the ip-number for player `ob`.

See also: query_ip_name.

8.106 query_is_wizard

This function is defined in /obj/simul_efun.c

```
int     query_is_wizard(ob)
object  ob;
```

True if `ob` is a wizard.

8.107 query_load_average

```
string  query_load_average()
```

Returns a string describing the performance of the driver.

8.108 query_mud_port

```
port    query_mud_port()
```

See also efun/query_imp_port.

8.109 query_once_interactive

```
int     query_once_interactive(ob)
object  ob;
```

True if object `ob` is interactive or has been interactive once.

8.110 query_verb

```
string  query_verb()
```

Give the name of the current command, or 0 if not executing from a command. This enables `add_action()` of several commands to the same function.

See also efun/add_action.

8.111 random

```
int     random(n)
int     n;
```

Return a random number in the range $[0 \ldots \mathtt{n} - 1]$.

8.112 read_bytes

```
string  read_bytes(file,offset,number)
string  file;
int     offset,
```

number;

Returns the contents of the file 'file', beginning at offset `offset`, reading `number` bytes. There is a maximum limit of bytes for reading. It is determined by the constant `MAX_BYTE_TRANSFER` (50k) in config.h.

See also: efun/file_size, efun/file_date, efun/write_bytes

8.113 read_file

```
string  read_file(file,offset,number)
string  file;
int     offset,
```

number;

Returns the contents of the file 'file', beginning at offset `offset`, reading `number` lines. There is a maximum limit of bytes for reading. It is determined by the constant `READ_FILE_MAX_SIZE` (50k) in config.h. If `number` is `-1` the maximum number of bytes will be read.

See also: efun/file_size, efun/file_date, efun/write_bytes

8.114 referencep

```
int     referencep(&ref)
mixed   ref;
```

Returns true if ref is a reference. Note that there has to be a leading &. References behave just like other local variables but if their value is changed the associated variable in the calling functions is changed, too. `referencep()` gives no information about the actual type of the variable.

See also intp, clonep, stringp, objectp, pointerp, floatp, mappingp.

8.115 regexp

```
string  *regexp(list,pattern)
string  *lists,pattern;
```

Match the pattern *pattern* against all strings in *list*, and return a new array with all strings that matched. Special characters are:

```
. ^ $ & * ? < > { } |
```

See the manual page of the unix command egrep for more information.

8.116 remove_call_out

```
int     remove_call_out(fun)
string  fun;
```

Remove next pending call out for function `fun` in this object. The time left is returned. The number -1 is returned if there was no call out pending to this function.

See also efun/call_out.

8.117 remove_interactive

```
void    remove_interactive(ob);
object  ob;
```

Removes an interactive object.

8.118 rename

```
int     rename(from,to)
string  from,
```

to;

The efun *rename* will move '`from`' to the new name '`to`'. If '`from`' is a file, then '`to`' may be either a file or a directory. If '`from`' is a directory, then '`to`' has to be a directory. If '`to`' exists and is a directory, then '`from`' will be placed in that directory and keep its original name.

It is only possible to change name of a directory within a directory on machines running System V, i.e it is not possible to move it to another directory. It is not possible to move a directory across filesystems on any system.

8.119 replace_program

```
void    replace_program(name);
string  name;
```

Substitutes a program with an inherited one. This is useful if you consider the performance of the driver. A program which doesn't need any additional variables and functions (except during creation) can call `replace_program` to increase the function-cache hit-rate of the driver which decreases with the number of programs in the game. Rooms are a good example for the application of this function, as many rooms just consist of an inherit statement and the `configure` function. Any object can call `replace_program` but looses all extra variables and functions which are not defined by the inherited program. `name` is the actual filename of the program (without the path and without .c). The array returned by inherit_list will not contain the name of the program which used replace_program, but the program which replaced it in position 0. file_name will, however, still return the old filename. This function will also reduce memory requirements for the object.

8.120 restore_object

```
int     restore_object(name)
string  name;
```

Restore values of variables for current object from file `name`. It is illegal to have periods or spaces in the name; '.o' is appended to the filename. The function returns true if it was successful.

Variables that have the type modifer `static` will not change.

If inheritance is used, then it might be possible that a variable will exist with the same name in more than one place. When restoring, only one of these variables will be restored if encountered in the save file. A good practice is to have verbose and unique name on non-static variables, which also will make it more easy to read or patch the save file manually. private variables which are not static will be overwritten.

restore_object can restore recursive data structures. However, make sure to deallocate such structures by hand lest permanent loss of memory may incurr.

See also efun/save_object.

8.121 rm

```
int     rm(file)
string  file;
```

Remove file `file`. Returns 0 for failure and 1 for success.

See also: mkdir, rmdir.

8.122 rmdir

```
int     rmdir(dir)
string  dir;
```

Remove directory `dir`.

See also: rm, mkdir.

8.123 rusage

```
void    rusage()
```

This efun is linked to the unix function getrusage. The first entry will be the usertime im milliseconds and the second will be the system time spent in milliseconds; the other entries are not supported by all systems. (read the unix manpage for more information).

8.124 save_object

```
void    save_object(name)
string  name;
```

Save values of variables of this object in the file `name`. It is illegal to have periods or spaces in the filename. Wizards that call this function can only save to files in their own directories.

Variables that have the type modifer `static` will not be saved.

Example: `static int xxx;`

Save files escape all c++ special characters with a backslash. The new save file format is denoted by a leading line with a hash mark at the start and no spaces. The new file format allows to save recursive data structures; the use of circular structures is not suggested, however, since they occupy memory which can only be regained by a reboot once the object which created them has been destructed.

See also efun/restore_object.

8.125 say

```
void    say(str,obj)
string  str;
object  obj;
```

Send a message `str` to all players in the same object (usually a room). This function is also used by the `say` command.

If second argument `obj` specified, messages is sent to all except `obj`.

This command behaves differently if called from a `heart_beat()` or otherwise. When called from a `heart_beat()`, the message will reach all players in the same environment of the object that calls `say()`.

If called from a living object (an object which called enable'commands()) the living object will be excluded instead of this'player().

See also efun/write, efun/shout, efun/tell_object, efun/tell'room, lfun/catch_tell.

8.126 send_imp

```
void    send_imp(host,port,message)
string  host,message;
int     port;
```

Send an imp to the specified host and port which carries the specified message. Note that imps travel slightly faster than internet packets even though the nasty little creatures are difficult to tame. send_imp is a privilege violation. *(See also section ??)*

See also efun/query_imp_port.

8.127 set_auto_include_string

```
void    set_auto_include_string(s)
string  s;
```

If called, the string s is prepended to every file compiled after that. If there are errors within it, line numbers will range from -n to 0. updating the master will clear the string. Thus it has to be called from `reset()/create()`in the master **and** `reactivate_destructed_master`. maximum length for s is 25k. calling it from outside thre master will cause a privilege_violation.

8.128 set_bit

```
string  set_bit(str,n)
string  str;
int     n;
```

Return the new string where bit `n` is set in string `str`. Note that the old string `str` is not modified and the new string will automatically be extended if needed. Bits are packed 6 per byte in printable strings.

The maximum value of `n` is limited to MAX_BITS (in config.h) In tubmud the limit is 1200 bits, 200 chars.

See also efun/clear_bit, efun/test_bit.

8.129 set_extra_wizinfo

```
void    set_extra_wizinfo(wiz,value)
mixed   wiz,value;
```

Set the wizinfo entry for wiz, which can be a name or object pointer. Note that arrays are not copied; the call causes a privilege violation. *(See also section ??)* set_extra_wizinfo can ignore the size set by set_extra_wizinfo_size. The later only sets up a default entry for get_extra_wizinfo which can then be manipulated without needing set_extra_wizinfo in case of an array.

See also efun/get_extra_wizinfo, efun/set_extra_wizinfo_size.

8.130 set_extra_wizinfo_size

```
void    set_extra_wizinfo_size(s)
int     s;
```

Set the size of the additional wizinfo entry. A positive size `s` means that it is an array with `s` entries; a size of `-1` means that it is a single variable.

See also efun/set_extra_wizinfo, efun/get_extra_wizinfo.

8.131 set_heart_beat

```
int     set_heart_beat(flag)
int     flag;
```

Enable or disable heart beat. If the heart beat is not needed for the moment, then do disable it. This will reduce system overhead.

Return true for success, and false for failure. Specifically, it will fail if the heart beat function has been disabled, which it will be if there is a run time error in it.

See also lfun/heart_beat.

8.132 set_is_wizard

```
int     set_is_wizard(ob,flag)
object  ob;
int     flag;
```

Sets the wizard flag for a player object with flag = 1, queries it with flag = -1 and resets it with flag = 0.

See also query¨is¨wizard.

8.133 set_light

```
int     set_light(n)
int     n;
```

An object is by default dark. It can be set to not dark by calling `set_light(1)`. The environment will then also get this light. The returned value is the total number of lights in this room.

Note that the value of the argument is added to the light of the current argument. Don't confuse this with the lfun set_light defined by '`/complex/room`' which actually **sets** the light.

See also /basic/light::set_light, add_light, light

8.134 set_living_name

```
void    set_living_name(name)
string  name;
```

Set a living name on an object that must be living. When this is done, the object can be found with `find_living()`.

An object can only have one name that can be searched for with `find_living()`.

See also efun/find_living, efun/find_player.

8.135 seteuid This function is defined in native mode only

```
int     setuid(str)
string  str;
```

Set effective uid to 'str'. It is not possible to set it to any string. It can always be set to getuid(), the creator of the file for this object or 0. When this value is 0, then current objects uid can be changed by export˙uid, and only then. But, when the value is 0, no objects can be loaded or cloned by this object.

See also efun/export˙uid, efun/getuid.

8.136 set_modify_command

```
mixed   set_modify_command(ob)
object  ob;
```

Call the function `modify_command` in an interactive object `ob` to modify each command given by `this_object()`. This is very convenient for alias/history tools. Upon disconnectiong a set_modfiy_command(0) is done automatically.

Valid return values are a modified string, 0 or 1. 1 indicates that the function doesn't wish further processing.

8.137 set_this_object

```
void    set_this_object(object_to_pretend_to_be)
object  object_to_pretend_to_be;
```

This is a privleged function, only to be used in the master object or in the simul_efun object. It changes the result of this˙object() in the using function, and the result of previous_object() in functions called in other objects by call_other(). Its effect will remain till the return of the using function.

Use it with extreme care to avoid inconsistencies. After a call of set_this_object(), some LPC-constructs might behave in an odd manner, or even crash the game. In particular, using global variables or calling local functions (except by call_other) is illegal.

Allowed are call˙other, map functions, access of local variables (which might hold array pointers to a global array), simple arithmetic and the assignment operators.

8.138 set_prompt

```
string  set_prompt(new,ob)
string  new;
object  ob;
```

Sets the prompt to `new` if it is non-zero and returns the old one.

8.139 shadow

```
object  shadow(ob,flag)
object  ob;
int     flag;
```

If flag is 1, then current object will shadow ob. If flag is 0, then either 0 will be returned, or the object that is shadow for ob.

An object that defines the funtion query_prevent_shadow() to return 1 can't be shadowed, and the shadow() function will return 0 instead of ob.

If an object a shadows an object b, then all call'other() to b will be redirected to a. If object a has not defined the function, then the call will be passed on to b.

There is only one object that can call functions in b with call'other(), and that is a. Not even object b can call_other() itself.

All normal (internal) function calls inside b will however remain internal to b.

There are three ways to remove the shadow. Either destruct it, the object that was shadowed or use unshadow() from the shadow.

In the later case, the shadow will also be destructed automatically. The result is that it is possible to hide an object behind another one, but everything can be totally transparent.

See also efun/unshadow.

8.140 shout

```
void    shout(str)
string  str;
```

Send a string `str` to all players. This function is also used by the `shout` command.

See also efun/write, efun/tell_object, efun/say.

8.141 sizeof

```
int     sizeof(arr)
```

Return the number of elements of an array `arr` or the number of indices (or values) in mapping `arr`. `sizeof(0)` is 0, this allows to test the size of uninitialized arrays.

See also efun/allocate.

8.142 snoop

```
object  snoop(snoopee)
object  snoopee;
```

Forward all messages given to a living object to the current player.

The correct syntax for efun::snoop is: snoop(snooper[, snoopee]) but the simul_efun snoop() maintains compatibility.

Return values are: -1: snooping loop, failed. 0: already snooped, failed. 1: success.

See also efun/living.

8.143 sort_array

```
mixed*  sort_array(arr,greater_fun,ob)
mixed*  arr;
string  greater_fun;
object  ob;
```

Returns an array sorted by the ordering function ob->greater'fun()

The function 'greater_fun' in the object 'ob' is continously passed two arguments which are two of the elements of the array 'arr'. It should return true or a positive number if the first argument is greater than the second.

8.144 sprintf

```
string  sprintf(fmt,arg,...)
string  fmt;
mixed   arg;
```

Most of the characters in the format string (FMT) get passed straight through to the output (ie: printed or put in the return string), to format the arguments into the string it's nessasary to include an argument format string (AFS) in the FMT. An AFS is a series of characters starting with a percent sign "argument type specifier. To include a "nessasary to include a double percent sign "

Valid argument type specifiers are:

- "s" the argument is a string.
- "d" the argument is an integer to be included in decimal representation.
- "i" same as "d".
- "o" the argument is an integer to be included in octal representation.
- "x" the argument is an integer to be included in hexidecimal representation.
- "X" as "x" except letters are capitalised.
- "O" the argument is an LPC datatype to be printed in an arbituary format, this is for debugging purposes. If the argument is an object then the function object˙name() on the master object is called with the object as a parameter, the string returned is included in brackets at the end of object file name. If 0 is returned then nothing is appended after the file name.

Between the percent sign and the argument type specifier in the AFS, the following modifiers can be included to specify the formatting information. Order is not important unless otherwise specified. "n" is used to specify a integer, which can be a "*" in which case the next argument is used as the number.

Modifiers:

- n specifys the field size, if prepended with a zero then the pad string is set to "0".
- "."n specifies the presision, for simple (not columns or tables) strings specifies the truncation length.
- ":"n n specifies the fs _and_ the presision, if n is prepended by a zero then the pad string is set to "0".
- "'X'" the pad string is set to the char(s) between the single quotes, if the field size is also prepended with a zero then which ever is specified last will overrule.
  NOTE: to include "' " in the pad string, you must use "\\' " (as the backslash has to be escaped past the interpreter), similarly, to include "\" requires "\\\\".
- " " pad positive integers with a space.
- "+" pad positive integers with a plus sign.
- "–" left adjusted within field size. NB: std (s)printf() defaults to right justification, which is unnatural in the context of a mainly string based language but has been retained for "compatability" ;)
- "—" centered within field size.
- "=" column mode. Ignored unless the argument type specifier is s. Field size must be specified, if presision is specified then it specifies the width for the string to be wordwrapped in, if not then the field size is. The field size specifies the width of the column.
- "# " table mode. Ignored unless the argument type specifier is s. Field size must be specified, if presision is specified then it specifys the number of columns in the table, otherwise the number is "optimally" generated. Table mode is passed a list of slosh-n separated 'words' which are put in a format similar to that of ls.
- "@" the argument is an array. the corresponding AFS (minus all "@") is applied to each element of the array.

8.145 sscanf

```
int     sscanf(str,fmt,var1,var2...)
string  str;
string  fmt;
```

Parse a string `str` using the format `fmt`. `fmt` can contain strings separated by '`%d`' and '`%s`'. Every '%d' and '%s' corresponds to one of `var1`, `var2`, ... '`%d`' will give a number and '`%s`' will give a string.

Number of matched '%d' and '%s' is returned.

See also efun/extract, efun/explode.

8.146 stringp

```
int     stringp(arg)
```

Return 1 if `arg` is a string.

See also intp, clonep, objectp, pointerp, referencep, floatp, mappingp.

8.147 strlen

```
int     strlen(str);
string  str;
```

Returns the length of a string `str`.

See also efun/sizeof, efun/strstr, efun/stringp.

8.148 strstr

```
int     strstr(str,str2,pos);
string  str,str2;
int     pos;
```

Returns the index of str2 in str searching from position pos. The third argument is optional.

See also efun/strlen, efun/sscanf, efun/sprintf, efun/explode.

8.149 tell_object

```
void    tell_object(ob,str)
object  ob;
string  str;
```

Send a message `str` to object `ob`. If it is an interactive object (a player), then the message will go to him, otherwise it will go to the local function `catch_tell` (which can be defined by player objects nowadays).

See also efun/write, efun/shout, efun/say, efun/tell_room, lfun/catch_tell.

8.150 tell_room

```
void    tell_room(ob,str,exclude)
object  ob;
string  str;
mixed   exclude;
```

Send a message `str` to object all objects in the room `ob`. `ob` can also be the name of the room (i.e., a string).

The optional parameter `exclude` is an array of the objects which should not receive `str`.

See also efun/write, efun/shout, efun/say, efun/tell_object, lfun/catch_tell.

8.151 test_bit

```
int     test_bit(str,n)
string  str;
int     n;
```

Return bit `n` of `str` (i.e., return 1 if bit `n` was set in string `str`, otherwise return 0).

See also efun/set_bit, efun/clear_bit.

8.152 this_interactive

```
object  this_interactive()
```

Return the object representing the player at the beginning of the execution chain.

See also efun/this_player, efun/previous_object.

8.153 this_object

```
object  this_object()
```

Return the object pointer of this object. This is not to be confused with the internal name of an object, which is used by the `id()` function.

See also efun/this_player, efun/previous_object.

8.154 this_player

```
object  this_player()
```

Return the object representing the current player.

See also efun/this_object.

8.155 time

```
int     time()
```

Return number of seconds since 1970.

See also efun/ctime.

8.156 to_array

```
int     to_array(arg)
mixed   arg;
```

Converts `arg` into an array.

See also efun/to_float, efun/to_int, efun/to_string.

8.157 to_float

```
int     to_float(arg)
mixed   arg;
```

Converts `arg` into a float.

See also efun/to_array, efun/to_int, efun/to_string.

8.158 to_int

```
int     to_int(arg)
mixed   arg;
```

Converts `arg` into an int, example: `to_int( "42" ) = 42`

See also efun/to_array, efun/to_float, efun/to_string.

8.159 to_string

```
int     to_string(arg)
mixed   arg;
```

Converts `arg` into a string, example: `to_string( ({ 'a', 'b', 'c' }) ) = "abc"`

See also efun/to_array, efun/to_float, efun/to_int.

8.160 trace

```
int     trace(traceflags)
int     traceflags;
```

Sets the trace flags and returns the old trace flags. When tracing is on a lot of information is printed during execution.

The trace bits are:

1	Trace all function calls to lfuns.
2	Trace all calls to *call_other*.
4	Trace all function returns.
8	Print arguments at function calls and return values.
16	Print all executed stack machine instructions (produces a lot of output!).
32	Enable trace in heart beat functions.
64	Trace calls to apply.
128	Show object name in tracing.

8.161 traceprefix

```
string  traceprefix(prefix)
string  prefix
```

If the the traceprefix is set (i.e. not 0) tracing will only occur in objects having a name with the set prefix.

8.162 transfer This function is defined in compat mode only

```
int     transfer(item,dest)
object  item;
object  dest;
```

Move the object `item` to the object `dest`. All kinds of tests are done, and a number is returned specifying the result:

0: Success.
1: To heavy for destination.
2: Can't be dropped.
3: Can't take it out of it's container.
4: The object can't be inserted into bags etc.
5: The destination doesn't allow insertions of objects.
6: The object can't be picked up.

If an object is transfered to a newly created object, make sure that the new object first is transfered to it's destination.

The returns values of `transfer()` are defined in the include file '`/basic/move.h`'.

When you call `transfer()` in your code the call will be intercepted by the simul_efun transfer which adds some additional checks to see if your request to move an object is accepted by the current environment, the new environment and the object itself, see `move_object` to see how it is done.

This function is **not** going to be removed in tubmud.

See also efun/move_object, lfun/drop, lfun/get, lfun/prevent_insert, lfun/can_put_and_get, lfun/add_weight.

8.163 transpose_array

```
mixed   *transpose_array(arr)
mixed   *arr;
```

Transpose array `arr`. This can be used to convert alists into a format that can be handled by map_array.

8.164 unique_array

```
mixed   unique_array(obarr,separator)
object  obarr;
string  separator;
```

Groups objects together for which the `separator` function returns the same value. `obarr` should be an array of objects, other types are ignored. The `separator` function is called only once in each object in `obarr`. The return value is an array of arrays of objects on the form:

```
({
   ({Same1:1, Same1:2, Same1:3, .... Same1:N }),
   ({Same2:1, Same2:2, Same2:3, .... Same2:N }),
   ({Same3:1, Same3:2, Same3:3, .... Same3:N }),
              ....
              ....
   ({SameM:1, SameM:2, SameM:3, .... SameM:N }),
})
```

8.165 unshadow

```
void    unshadow()
```

This function can be called by a shadow to abort shadowing.

See also efun/shadow.

8.166 update_actions

This function is defined in /obj/simul_efun.c

```
void    update_actions()
```

Updates this_object()'s actions in all living objects.

8.167 users

```
object  *users()
```

Return an array of objects, containing all interactive players.

8.168 wizlist

```
void    wizlist()
```

Prints the wizlist. Often mistaken for a Top-10 list but it is just statistic. Takes creator() as its optional argument.

See also efun/wizlist_info.

8.169 wizlist_info

```
mixed*  wizlist_info()
```

Returns an array of arrays, where every subarray represents an entry in the wizlist which consist of seven variables.

```
w[0] = wizard's name.
w[1] = total number of commands executed by his objects.
       Saved between reboots and decaying 1% per hour.
w[2] = total eval_cost. Decaying 10% per hour.
w[3] = total heart_beats. Decaying 10% per hour.
w[4] = reserved.
w[5] = total size of used arrays.
w[6] = the extra svalue. If it is an array, it will be copied.
```

See also efun/wizlist, efun/set_extra_wizinfo, efun/get_extra_wizinfo

8.170 write

```
void    write(str)
string  str;
```

Write a message **str** to current player. **str** can also be a number, which will be translated to a string.

See also efun/say, efun/tell_object, efun/shout, lfun/catch_tell

8.171 write_bytes

```
int     write_bytes(file,start,bytes)
string  file,bytes;
int     start;
```

Writes the bytes of the string `bytes` into the file specified by `file`, beginning at `start`. The return code is 1 for success, 0 otherwise.

Note: The bytes in your file will be overwritten. The file has to be existent to perform write_bytes.

See also: efun/file_size, efun/log_file, efun/write_file, efun/read_bytes

8.172 write_file

```
int     write_file(file,str)
string  file,str;
```

Append the string `str` to the file `file`.

See also efun/file_size, efun/cat, efun/log_file.

8.173 version

This function is defined in /obj/simul_efun.c

```
string  version()
```

Returns the gamedriver version string.

9 Local Functions (lfun)

9.1 General Lfuns

9.1.1 add_weight

```
int     add_weight(w)
int     w;
```

An object that can contain other objects must define this function. It is called with the extra weight of the new object. If this is ok, then it has to increment the local weight count, and return true. Otherwise, return false, and the new object can not be entered into this object.

9.1.2 apply_action

```
int     apply_action(skill,level,arg)
string  skill;
int     level;
mixed   arg;
```

This function is called when a player uses the skill `skill` on the object which defines it. The success of the action should depend on the skill and the level which is a percentage relative to the possible skill maximum. A return value of 0 means failure to use the skill, a value greater than 0 means success, whereupon value−1 is added to the current skill in the skill path.

(See also section ??)

9.1.3 can_put_and_get

```
int     can_put_and_get(str)
string  str;
```

Define this function if you want to make it possible to put something into the current object. Return true if this is ok, otherwise 0. That means that the default is that it is not possible to put something into an object.

When a player does '`look at xxx`', then '`xxx`' will be sent to `can_put_and_get()`, to test if the player can look at the inventory. Otherwise, `str` will be 0. This is trivial for containers. If the are open, they return 1. If `id()` accepts other things, like '`lock`' (e.g., in a chest). Then '`lock`' will be sent to `can_put_and_get()`, which should return false, because the lock has no inventory, of course.

See also lfun/id, lfun/long.

9.1.4 catch_tell

```
void    catch_tell(str)
string  str;
```

When `tell_object()` sends a message to a noninteractive player, it will get to the function `catch_tell()`. This will enable communications between NPC's and from a player to an NPC. The only exception is 'shout', which the monster won't hear. The monster must be living, i.e., the function `enable_commands()` must have been called. It will be used in interactive objects if present. In this case, only the player object itself sends messages to the internet. When remove_interactive sets the closing flag, it will also disable all catch_tell functions for the affected interactive, thus avoiding errors in these functions.

See also efun/enable_commands, efun/disable_commands.

9.1.5 clean_up

```
void    clean_up(arg)
int     arg;
```

clean_up() is called when an object hasn't been used for long, to give it a possibility to self-destruct. The reference_count passed as argument will be 0 for clone objects, 1 for a simple loaded object, and greater when the object is cloned or inherited by some existing object. It is recommended not to self-destruct the object when the reference count is greater than one. When clean_up returns 0 or no value, it won't be called in a swapped object again till the object is loaded again from swap. Returning a non-zero value is only recommended when the reason why the object can't self-destruct is likely to vanish without the object being touched, that is, when no local function is called in it, it isn't searched for by find_object, nor function_exists is applied to it.

/obj/drink uses clean_up to self-destruct when it is empty, not carried by a living being and not touched for long.

/room/room uses clean_up to let the room self-destruct if neither inherited nor used as blueprint, and if it is empty and has no environment too.

A typical mud configuration defines the time to wait for clean_up so long that you can assert reset(1) has been called since the object has been touched last time.

(See also section ??).

9.1.6 drop

```
int     drop(silently)
int     silently;
```

This function must be defined by all objects that want to control when they can be dropped. if `silently` is true, then don't write any messages.

`drop()` should return 1 to prevent dropping. This is the opposit of the `get()` function. That is because if `drop()` is not defined, it will always be possible to drop an object.

If the object self-destructs when `drop()` is called, be sure to return 1, as the destructed item surely cannot be dropped. Similarly, if `drop()` is called in another object, always test if the object did self-destruct, as the object variable will turn to 0.

9.1.7 exit

```
void    exit(ob)
object  ob;
```

This function is called in rooms everytime a living object `ob` leaves. The function `this_player()` will return a random value, don't use it at this point.

See also: lfun/init.

9.1.8 extra_long

```
string  extra_long()
```

If this function returns a string, and the object is inside a /complex/room, then the string returned by this function will be printed after the long description but before the list of exits.

See also lfun/extra_look.

9.1.9 extra_look

```
string  extra_look()
```

If this function returns a string, and the object is carried by a player, then the string returned by this function will be printed after the character data, but before the list of what the character is carrying. This can be used to introduce curses for players, that gives some visual result.

See also lfun/short, lfun/query_auto_load lfun/extra_long

9.1.10 get

```
int     get(str)
string  str;
```

If an object wants control the possibility of picking it up, then it must define `get()`, and return 1 if it is ok to pick it up.

The argument `str` comes from the syntax '`get str`' from the player command. The local function `id()` has been called before this function call to identify the object.

9.1.11 heart_beat

```
void    heart_beat()
```

This function will be called automatically every 2 seconds. The start and stop of the heart beat is controlled by `set_heart_beat()`.

Be careful not to have objects with heart beat running all the time, is it uses a lot of resources. If there is an error in the heart beat routine, the heart beat will be turned off until this object is recompiled, and can not be restarted with `set_heart_beat()`.

The function `this_player()` will return this object, but only if it is living. Otherwise the function `this_player()` will return 0.

See also: efun/set_heart_beat, efun/call_out, efun/enable_commands.

9.1.12 id

```
int     id(str)
string  str;
```

This function is used to identify an object. If it identifies with the string `str`, then return 1, else return 0.

Note: If you want to give an object an id player should not be able to use prepend "\n" to the id string (e.g. "\nmy_invisible_object")

See also efun/present.

9.1.13 init

```
void    init()
```

This function is called everytime a living objects can 'see' the object. It is good to the set up of `add_action()` and `add_verb()` in the `init()` routine.

See also lfun/exit.

9.1.14 long

```
void    long(str)
string  str;
```

This function prints out an elaborate description of itself. The minimum requirement is to print the same description as the local function `short()`.

If there is an argument, then print the description of that argument. An argument can only be passed to `long()` if the local function `id()` has agreed. For example, a room with a door can allow `id("door")` to be true. Then it is possible for a player to do '`look at door`'. The function `long()` will then have to print information about the door. To prevent the listing of all things in this room when the player does '`look at door`', let `can_put_and_get("door")` return 0.

See also lfun/short, lfun/can_put_and_get, lfun/id.

9.1.15 notify_petrification

```
void    notify_petrification(player)
object  player;
```

This function is called inside a room when a player in it turns into a statue.

9.1.16 notify_resurrection

```
void    notify_resurrection(player)
object  player;
```

This function is called inside a room when a player statue in it turns alive again.

9.1.17 query_auto_load

```
string  query_auto_load()
```

An object that wants to be loaded automatically when the player logs in should define this function. There are some important rules for the usage of this feature.

1. The object must not have any weight.
2. The object must prevent the player from dropping it.
3. `query_auto_load()` must return a string of the format
   file: *argument*
   The '*file*' is the definition that will be cloned. The '*arg*' is a string that will be sent as argument to the function `init_arg()`. The '*argument*' can be an empty string.
4. The object must not be an actively usable item, like weapon or armour.
5. The object must not help the player in combats.

The idea with this feature is that a player can get a curse or membership, that will stick with him, even if he quits. The idea is not that the player will save his weapons etc.

Look at '`/obj/shout_curse.c`' for an example.

See also lfun/extra_look.

9.1.18 query_info

```
string  query_info()
```

Declare this function if the object has some information that is hidden. A scroll of identify would call `query_info()` to find out.

The standard objects `weapon.c`, `armour.c` and `treasure.c` all have a function `set_info()`, to enable setting an information string.

9.1.19 query_name

```
string  query_name()
```

All living objects, weapons and armour must return the name of itself.

See also lfun/short.

9.1.20 query_no_teleport

```
int     query_no_teleport(from,to,player)
mixed   from,to;
object  player;
```

Every room can define this function. It has to return a nonzero value if teleporting to and from this room is not allowed. In order to ensure that it is working properly, every teleportation item available to players has to check for this.

The correct way to do this is to inherit '`/basic/teleport`' and call `teleport_ok(from, to, player)`, where `player` defaults to this_player.

Teleporting can also be disabled by the properties `P_NO_TELEPORT`, `P_NO_TELEPORT_FROM` & `P_NO_TELEPORT_TO`.

9.1.21 query_value

```
int     query_value()
```

Return the value of this object. If it is not possible to sell this object, then the value 0 should be returned. One gold coin corresponds to one experience point, as a reference.

9.1.22 query_weight

```
int     query_weight()
```

This function is called to query the weight of this object. It is a design choice of this game that objects inside another object do not change the weight of that object. This makes it possible for a player to carry more if he puts it into a bag or something. Note that an object doesn't have a shape or a size, only a weight.

9.1.23 reset

```
void    reset(flag)
```

`reset()` is called everytime the object is resetted. The first time is when the object is loaded. If a room creates things when `reset()`, it should check that these objects don't exist any longer (at least in this room) creating new.

When `reset()` is called for the first time, a null argument is passed. The second time `reset()` is called `flag` will be 1. Every object will repeatedly get resetted by the game driver. The game wouldn't be fun if no challenges remained.

9.1.24 short

```
string  short()
```

All objects must have a `short()` function. This function returns a short message describing what it is. Invisible objects will return the value 0.

See also lfun/long.

9.2 Functions of /obj/cron

9.2.1 time

```
int     time()
```

Mud time in seconds. See <timezone.h>.

9.2.2 timeofday

```
int     timeofday()
```

Mud time in seconds of the day. See <timezone.h>.

9.2.3 crontab_add

```
void    crontab_add(when,obj,fun,arg)
int     when;
string  obj,fun;
mixed   arg;
```

Set up a cron job to call `obj->fun(arg)` every mud day at time `when`. The cron table is saved immediately and the entry will remain until `crontab_remove(when,obj,fun)` is called!

9.2.4 crontab_list

```
void    crontab_list()
```

List the cron job table.

9.2.5 crontab_remove

```
void    crontab_remove(when,obj,fun)
int     when;
string  obj,fun;
```

Remove a cron job.

9.3 Functions of /obj/living

These are most of the query_*() functions of living.c:

int query_ac()	The armour class
object query_age()	the age of the player in seconds / 2
object query_attack()	the attacker object or zero
int query_con()	The constitution value
string query_current˙room()	Filename of the current environment
int query_dex()	The dexterity value
int query_gender()	0 (neuter), 1 (male) or 2 (female)
string query_gender_string()	({ "neuter", "male", "female" })[gender]
int query_hp()	The current number of hit points
int query_int()	The intelligence value
int query_level()	The level of a player or monster. All mobile (living) objects must define this. The lowest level is 1. An apprentice wizard has level 20, and a full wizard with a castle has level 21
int query_local_weight()	The local weight
int query_max_hp()	Maximum number of hit points
int query_money()	The amount of money
int query_npc()	True for Non-player-characters (monsters etc.)
string query_objective()	({ "it", "him", "her" })[gender]
string query_possessive()	({ "its", "his", "her" })[gender]
string query_pronoun()	({ "it", "he", "she" })[gender]
int query_sp()	The number of spell points
int query_spell_points()	Same as query_sp()
int query_str()	The strength value
int query_wc()	The weapon class
string query_wield()	the name of the wielded weapon
int query_wimpy()	The whimpy value, yes 'whimpy'

9.3.1 add_money

```
void    add_money(m)
int     m;
```

Objects that can pick up the special '`money`' object should have this function. It will be called with the amount of gold coins.

9.3.2 attack

```
status  attack()
```

This function is called from `heart_beat()`. It returns true if there is still a fight.

9.3.3 attacked_by

```
void    attacked_by(ob)
object  ob;
```

This routine sets the attacker_ob if it is zero or the alt_attacker_ob if it is zero. This is called in the opponent when starting to attack it.

9.3.4 clear_flag

```
void    clear_flag(n)
int     n;
```

Clear a bit in the flags string of an object that inherits /obj/living. When you want to use a flag you have to request it from the flag dispenser first.

See also lfun/set_flag lfun/test_flag efun/clear_bit

9.3.5 heal_self

```
void    heal_self(h)
int     h;
```

This routine is called when the object is allowed to heal by '`h`' points. Of course, this is only interesting for living objects.

9.3.6 hit_player

```
int     hit_player(dam,who)
int     dam;
object  who;
```

If the object can fight and get hit by other objects, it must have a `hit_player` function. The `dam` is the maximum damage the other object wants to give. Return the actual damage.

This function should also tell other players in the current room that someone got hit (using `say()`).

The second, optional, argument is the attacker. If it is a living object a fight will start. Your average non-living room can drain hitpoints without starting a fight.

See example in '`player.c`'.

See also efun/enable_commands, efun/living, lfun/attacked_by.

9.3.7 move_living

```
void    move_living(dir_dest,dest)
string  dir_dest,dest
```

This function moves the living object in direction `dir` to `dest`. The argument `dir_dest` may be a string of the format `"dir#dest"`, this remains from a time when lfuns had only one argument. The first argument should just be a string like "north" or "up" and the second argument should be the filename of the room to move to. `move_living()` uses `move_player` but additionally calls `leave_inv` in the old environment and `enter_inv` in the new environment. When `dir` is `"X"` the player teleports (mmsgout and mmsgin are printed instead of msgin and msgout) and if it is `0` no messages are printed.

9.3.8 receive_object

```
void    receive_object(ob,pl)
object  ob,pl;
```

This function gets called in a living object whenever player 'pl' gives item 'ob' to the object.

9.3.9 run_away

```
void    run_away()
```

Let the player or npc run away in a random direction (only standard directions).

9.3.10 set_flag

```
void    set_flag(n)
int     n;
```

Set a bit in the flags string of an object that inherits /obj/living. When you want to use a flag you have to request it from the flag dispenser first. You are not supposed to do this arbitrarily. Every wizard can allocate a few bits from the administrator, which he then may use. If you manipulate bits that you don't know what they are used for, unexpected things can happen.

See also lfun/clear_flag lfun/test_flag efun/set_bit

9.3.11 show_stats

```
void    show_stats()
```

Living objects should define this function. It should print all important stats about the object.

9.3.12 stop_fight

```
void    stop_fight()
```

This function must be defined by all monster and player objects. If you call this function, that player or monster will stop fighting. If you want to stop a fight, you have to call `stop_fight()` in both opponents.

9.3.13 stop_wearing

```
void    stop_wearing(type)
string  type;
```

Stop wearing a piece of armour of the specified type. This function is called by the armour, which should define the verb "remove".

9.3.14 stop_wielding

```
void    stop_wielding()
```

Objects able to wield weapons should have this function. It is called by the weapon when it is not possible to continue wielding the weapon. The function should adjust the weapon class of the current object.

9.3.15 test_flag

```
int     test_flag(n)
int     n;
```

Test a bit in the flags string of an object that inherits /obj/living . When you want to use a flag you have to request it from the flag dispenser first.

See also lfun/set_flag lfun/clear_flag efun/test_bit

9.3.16 test_if_any_here

```
int     test_if_any_here
```

For monsters. Call this one if you suspect no enemy is here any more. Returns 1 if anyone there, 0 if none.

9.4 Functions of /obj/player

/obj/player inherits /obj/living, /basic/time, /basic/view and the following objects from /obj/player/: autoload, guild, (less), quest, scar, shell, snoop, wizline

These are most of the query_*() functions of player.c:

query_al_title	the alignment title
query_brief	the brief/verbose flag
query_channels	the channels of the wizline
query_domains	the domains a wizard belongs to
query_guilds	A string, set bits represent memberships
query_intoxication	intoxication (how much alco_drink was consumed)
query_mailaddr	the e-mail address
query_pretitle	the pretitle
query_quests	the quest string (quests are separated by #)
query_real_name	the login name of a player (in lower case)
query_stuffed	stuffed (how much was eaten)
query_soaked	soaked (how much soft_drink was consumed)
query_title	the title
query_vis_name	the login name of a player

9.4.1 add_alignment

```
void    add_alignment(a)
int     a;
```

`alignment = alignment * 9 / 10 + a;`

9.4.2 add_exp

```
void    add_exp(exp)
int     exp;
```

`experience += e`

9.4.3 add_intoxination

```
void    add_intoxination(i)
int     i;
```

`intoxicate += i;` There are also `add_stuffed` and `add_soaked`.

9.4.4 clear_intoxication

```
void    clear_intoxication()
```

Clears the intoxication.

9.4.5 command_less

```
void    less::command_less(files)
string  files;
```

This function allows to read files in blocks of about 20 lines. Don't call this function in another object than this_player() because it relies on this_player().

9.4.6 compute_values

```
int     compute_values(ob)
object  ob;
```

Recursively compute the values of the inventory of `ob`.

9.4.7 normalize_path

```
string  valid::normalize_path(path)
mixed   path;
```

Substitutes the strings "~", "." and ".." in a path which may be given as a string or an array (`explode(path,"/")`). An array is returned.

9.4.8 recompute_armour_class

```
void    recompute_armour_class()
```

Recalculates the total ac of the worn armour. When a piece of armour is destructed or changes its ac this function has to be called in order to adjust the player's ac.

9.4.9 second_life

```
int     second_life(corpse)
object  corpse;
```

This function is called when a living object dies, in case of player objects it starts the death sequence.

9.4.10 valid_read

```
string  valid::valid_read(string)
```

Returns a normalized path if the file may be read, 0 otherwise.

9.4.11 valid_write

```
string  valid::valid_write(string)
```

Returns a normalized path if the file may be written to, 0 otherwise.

9.4.12 trusted

```
int     valid::trusted(ob)
object  ob;
```

Check whether object `ob` is trusted. This is done by means of another object, which is loaded via the wizard's castle. This may be the castle itself, for instance. This function produces a warning when the pointer is zero, for security reasons.

The castle has to define a function `valid_trusted` which returns the object pointer to the actual object which defines the function `trusted`. The return value of `castle->valid_trusted()->trusted(ob)` is the return value of `valid::trusted()`.

Additionally '`/secure/access`' holds a list of objects which are always trusted objects.

9.5 Functions of /obj/monster

/obj/monster inherits /obj/living

The following functions can be used to configure a monster; all except set_name and set_level are optional:

set_name(n)	string n. Sets the name and short description to n. Sets long description to "You see nothing special.\n"
set_level(l)	int l. The monster gets the level l. Hit points and ep is set as the same as player of level l. Armour class to 0 and weapon class to that of hands.
set_hp(hp)	int hp. Sets hit points to hp.
set_ep(ep)	int ep. Sets ep to ep.
set_al(al)	string al. Stets the alignment to al, negativ is evil, pos good.
set_alias(n)	string n. Adds and alternate name for the monster.
set_alt_name(n)	string n. Adds another alternate name for the monster.
set_race(r)	string r. Adds an alternate generic name for the monster.
set_short(sh)	string sh. Sort description is set to sh. Long to short + "\n"
set_long(long)	string long. Long description is set to long.
set_wc(wc)	int wc. Sets the weapon class, how much the damage it will do, to wc. The damage inflicted is in the range 0..wc-1
set_ac(ac)	int ac. Armour class is set to ac.
set_aggressive(a)	int a. 0 means peaceful until attacked. 1 that it will attack everyone it sees.
set_move_at_reset()	If this routine is called the monster will do a random move at every reset.
set_frog()	If anyone kisses the monster he will turn into a frog.
set_whimpy()	When monster get low on hp it will flee.
init_command(string cmd)	Force the monster to do a command. The force_us() function isn't always good, because it checks the level of the caller, and this function can be called by a room.
set_dead_ob(ob)	object ob. The function 'monster_died' in 'ob' will be called just before the monster dies. The first argument to 'monster_died' will be the nearly dead monster object. The second argument to 'monster_died' will be the monster's corpse. The return value from 'monster_died' determines the fate of the monster. A 1 means the monster will survive, 0 that it will die.
set_init_ob(ob)	object ob. The function 'monster_init' in 'ob' will be called from init in the monster. The argument to 'monster_init' will be the the monster object. The return value from 'monster˙init' determins if the monster will attack, if it's aggressive. A 1 means that the monster will not attack, 0 that it will function as usually.
set_give_ob(ob,func)	object ob, string func. Calls func(item,pl) in ob whenever player 'pl' gives item 'item' to the monster. func is optional and defaults to "receive_object".
set_spell_mess1(m)	string m. This is the message that the other players in the room get when the monster cast's a spell.
set_spell_mess2(m)	string m. This is the message that the victim of the monster's spell get.
set_chance(c)	int c. This is the percent chance of casting a spell.
set_spell_dam(d)	int d. How much damage the spell will do if it hits. The damage will be randomly 0 .. d-1

set_death_mess(m)	string m. The message which is output when the monster dies instead of <monster_name> died. m="" will output nothing at all, while m=0 will give you the normal message. Don't forget the "\n".
set_corpse_weight(i)	int i. When our monster dies, the corpse will weight i units. The default is 5. Think of a good weight for several monsters. For example a fly should have a set_corpse_weight(1);, to prevent the corpse from looking 'very heavy'.
load_chat(chance, strs)	This enables the monster to say something every heart beat. chance is the probability that something will be said. strs is an array of strings which will be printed one at a time.
load_a_chat(chance, strs)	This enables the monster to chat while fighting.

9.5.1 pick_any_obj

```
void    pick_any_obj()
```

The monster picks up everything it can carry when this function is called.

9.5.2 query_npc

```
int     query_npc()
```

This function is defined by all living objects and monsters. It will return 1 for monsters, and 0 for players. (NPC means Non Player Character.)

9.5.3 receive_object

```
void    receive_object(ob,pl)
object  ob,pl;
```

This function is called when the monster receives the object `ob` from the player `pl`.

9.6 Functions of /complex/door

/complex/door inherits /complex/item and the following basic objects: exit, open, lock, orientation, doortuple

An object that wants to create a door should inherit /basic/makedoor. Descriptions of the door object itself might be added later.

9.6.1 make_door

```
object  make_door(verb,dest,description)
string  verb,dest,description;
```

Creates a door which moves a player to `dest` when he enters `verb`. The long description of the door is `description`. The door uses '`/basic/exit`' to move the player; the exit hook which prevents movement when the door is closed is `door::door_hook()`, another hook can be installed if the door object is accessed on a lower level. The function `doortuple::set_destination(verb, dest)` tries to create a counterpart of the door, facing the opposite direction. `verb` should be a standard direction like "north", "east" or "up" because `orientation::orientation_reverse()` has to generate the other direction automagically.

9.7 Functions of /complex/item

/complex/item inherits the following basic objects: description, id, move, value.

9.7.1 id

```
int     id::id(str)
string  str;
```

Return true if the object identifies as `str`. This is used by present() if the first argument is a string.

9.7.2 move

```
int      move::move(dest)
mixed    dest;
```

Move this object to the destination given by string/obj dest.

This function will remain but don't rely on move_check, move_prolog, move_epilog, enter_inv & leave_inv since move_object now already provides notify functions for exactly the same purpose.

The return codes of move are defined in '`basic/move.h`' (??). They are identical with the returns codes of the efun `transfer()`.

In tubmud `move` is not defined in all objects, don't rely on it if you are not sure what kin of item you are moving, `move_object` and `transfer` are always safe, however.

9.7.3 query_encumbrance

```
int      move::query_encumbrance
```

Query the bulkiness of the object. This replaces the 2.4.5 object weight; it is a meassure for volume and weight.

9.7.4 query_id

```
string   *id::query_id()
```

Query the array of names which identifies the object.

9.7.5 query_long

```
string   description::query_long()
```

Query the long description (For 2.4.5 compatibility this is also available from 'long'). query_description_long() also returns the long description (this can be useful if you can't access description::query_long() because description.c is inherited by an object which is inherited by your object and not directly).

9.7.6 query_short

```
string   description::query_short
```

Query the short description (For 2.4.5 compatibility this is also available from 'short').

9.7.7 query_value

```
int      value::query_value()
```

Query the value.

9.7.8 query_weight

```
int      move::query_weight()
```

Query the weight of the object. The encumbrance value is used to calculate a weight for the object (for 2.4.5 compatibiliy).

9.7.9 set_encumbrance

```
void     move::set_encumbrance(enc)
int      enc;
```

Set the encumbrance value. When the object already has an environment move_prolog and move_epilog are called.

9.7.10 set_id

```
void    id::set_id(new_id)
string  *new_id;
```

Set array of names which will identify the object.

9.7.11 set_long

```
void    description::set_long(str)
string  str;
```

Set a long description (with a '\n' at the end).

9.7.12 set_short

```
void    description::set_short(str)
string  str;
```

Set a short description (without a '\n' or '.' at the end).

9.7.13 set_value

```
void    value::set_value(v)
int     v;
```

Set the value.

9.8 Functions of /complex/room

This is a generic room object, it inherits the following basic objects: description (see item.c), timedep*, exit, light, property, extralong. Only some functions from light.c and exit.c are listed below:

9.8.1 light

```
int     light::light()
```

Get the current light in the environment.

9.8.2 add_light

```
int     light::add_light(l)
int     l;
```

This has the old efun::set_light functionality: the light value is increased by l.

9.8.3 set_light

```
int     light::set_light(l)
int     l;
```

Actually set the light, don't add to it.

9.8.4 query_light

```
int     light::query_light()
```

Query the light given by the current object.

9.8.5 set_exits

```
void    exit::set_exits(exits,commands)
string  *exits,*commands;
```

Set the exits for this room. There are two ways to call this function. The first way is to pass two arrays, the first containing the filenames of the destinations and the second containing the associated commands. The second method is to pass a single array of arrays. Every sub-array in the array represents an exit ({ `filename, command, hook, flag` }); If flag is 1 (bit 0 is set) the exit won't be listed as an obvious exit. Destination filenames may be given as "./file", the '.' will be replaced by the path of the room in which the exit is defined.

Example:

```
set_exits( ({
  ({ "room/church", "south" }),
  ({ "players/foo/workroom", "workroom", "test_access" }),
  ({ "players/foo/treasury", "abracadabra", 0, 1 })
}) );
```

9.8.6 remove_exits

```
void    exit::remove_exit(command)
string  command;
```

Remove the exit used with `command`.

9.8.7 add_exit

```
void    exit::add_exit(exit,command,hook,flag)
string  exit,command,hook,flag;
```

Add or replace one exit. If bit 0 of the integer is set the exit won't be listed in the "obvious exits list".

9.8.8 set_move_hooks

```
void    exit::set_move_hooks(hooks)
string  *hooks;
```

Set the move hooks for this room. (A hook is the name of a function to be called, the function decides whether the exit is usable or not).

Example: '`/complex/door`' defines a function `door_hook`:

```
int
door_hook (str)
{
   if (!query_open()) {
      write ("The door is closed.\n");
      return 1;
   }
}
```

9.9 Functions of /basic/action

Generic skill logic for all skills which use the brain object. The include file '`/basic/action.h`' is explained in section ??. This object defines an example `create()` function:

```
void create(){
   skillpath=({"skill","miscellaneous","curiosity"});
   reward = LEARNING_1;
   default_objects = ({});
   specific = "skill";
   command = "use";
   function = "command_default";
}
```

9.9.1 set_default_objects

```
void    set_default_objects(what)
string  *what;
```

9.9.2 set_function

```
void    set_function(str)
string  str;
```

9.9.3 set_skillpath

```
void    set_skillpath(what)
string  *what;
```

9.9.4 query_skillpath

```
string  *query_skillpath()
```

9.9.5 query_skill

```
string  query_skill()
```

9.9.6 set_reward

```
void    set_reward(amount)
int     amount;
```

9.9.7 query_reward

```
int     query_reward()
```

9.9.8 set_command

```
void    set_command(str)
string  str;
```

9.9.9 set_specific

```
void    set_specific(str)
string  str;
```

9.9.10 query_info

```
string  query_info(str)
string  str;
```

9.9.11 init_living

```
string  *init_living(ob)
object  ob;
```

9.9.12 command_default

```
int     command_default(arg)
string  arg;
```

9.9.13 query_actions

```
string  *query_actions
```

9.10 Functions of /basic/alias

9.10.1 id_alias

```
int     id_alias(subject,space)
string  subject,*space;
```

9.10.2 query_alias

```
string  query_alias(subject,space)
string  subject,*space;
```

9.10.3 set_single_alias

```
void    set_single_alias(name,description,space)
string  name,*space;
mixed   description;
```

9.10.4 set_alias

```
void    set_alias(names,description,space)
mixed   names,description;
string  *space;
```

9.10.5 set_aliases

```
void    set_aliases(names,descriptions,space)
mixed   names,descriptions;
string  *space;
```

9.10.6 remove_single_alias

```
void    remove_single_alias(name,space)
string  names,*space;
```

9.10.7 set_alias_name

```
void    set_alias(str)
string  *str;
```

9.11 Functions of /basic/autoload

Inherit this to make an object autoloading.

9.11.1 query_auto_load

```
string  query_auto_load()
```

This function puts the filename of the subclass (the object which inherits it) into the autoload string of a player who carries it.

9.12 Functions of /basic/fakeitem

This object allows to add 'faked' items to an object, an example is a wardrobe which identifies itself as 'wardrobe', 'door', 'lock' and 'keyhole'. It is not necessary to include this file into '`/complex/room`' because '`/basic/timedepdesc`' already provides a similar mechanism for time dependent descriptions. Item names may be strings or array of strings, arrays will be treated as synonyms, using the same description.

9.12.1 set_item_description

```
void    set_item_descriptions(names,descriptions)
mixed   *names,*descriptions;
```

Set the items and their descriptions.

9.12.2 set_one_item_description

```
void    set_one_item_description(name,description)
mixed   name,description;
```

Set one item description.

9.12.3 id

```
int     id(str)
string  str;
```

Return true if we describe an item with this id.

9.12.4 query_long

```
string  query_long(str)
string  str;
```

Give long description of an item if we describe it.

9.12.5 query_fakeitem_long

```
string  query_fakeitem_long(str)
string  str;
```

Same as query_long.

9.12.6 query_items

```
mixed   *query_items()
```

Return an array `({names, descriptions })`.

9.12.7 add_item

```
void    add_item(name,description)
mixed   name,description;
```

Set one item description (same as `set_one_item_description`).

9.12.8 remove_item

```
void    remove_item(str)
string  str;
```

Remove the item named `str`.

9.13 Functions of /basic/fsm

9.13.1 fsm_transition

```
string  fsm_transition(state,input,space)
string  state,input,*space;
```

9.13.2 fsm_set_transition

```
void    fsm_set_transition(state,input,action,space)
string  state,input,*space;
mixed   *action;
```

9.13.3 fsm_set_transitions

```
void    fsm_set_transitions(state,transitions,space)
string  state,*space;
mixed   *transitions;
```

9.13.4 fsm_test_transition

```
string  fsm_test_transition(ident,space)
string  ident,*space;
```

9.14 Functions of /basic/grammar

9.14.1 vocal

```
int     vocal(c)
int     c;
```

Returns true if c is a vocal. Note that there is no type `char`, a character `'c'` has the type `int`.

9.14.2 article

```
string  article(str)
string  str;
```

Returns the correct article for `str`.

9.14.3 articalize

```
string  articalize(str)
string  str;
```

Prepends the correct article to `str`.

9.15 Functions of /basic/namespace

This object manages a hierarchy of associative lists. Data entries in the alists are again alists up to the leaves of this tree structure of alists, which contain the data stored in it. To access data in this tree you have to know an array of keys. Key[1] has to be a key in the alist which is stored as the data associated with Key[0].

9.15.1 add_namespace

```
int     add_namespace(key)
mixed   key;
```

Insert a new key in the root list. Nothing is associated with it.

9.15.2 set_namespace

```
int     set_namespace(keys,data)
mixed   *keys,data;
```

Inserts `data` in the tree. The `keys` array is a list of keys which are inserted into the tree if they do not exit. `data` is associated with `keys[sizeof(keys)-1]` in the alist associated with `keys[sizeof(keys)-2]`.

9.15.3 get_namespace

```
mixed   get_namespace(keys)
mixed   *keys;
```

Returns the data selected by `keys`.

9.15.4 export_namespace

```
mixed   *export_namespace()
```

Returns the root of the tree. (static function)

9.15.5 import_namespace

```
void    *import_namespace(import)
mixed   *import;
```

Imports a namespace (all alists in it are sorted with `order_alist`). (static function)

9.16 Functions of /basic/open

Generic open/close logic

9.16.1 open

```
int     open()
```

Set state to open and return true if state has changed.

9.16.2 close

```
int     close()
```

Set state to closed and return true if state has changed.

9.16.3 query_open

```
int     query_open()
```

Return the state (true if open).

9.16.4 toggle

```
int     toggle()
```

Toggle the state and return new state.

9.17 Functions of /basic/lock

Generic lock/unlock logic

9.17.1 lock

```
int     lock()
```

Set state to locked and return true if state has changed.

9.17.2 unlock

```
int     unlock()
```

Set state to unlocked and return true if state has changed.

9.17.3 query_lock

```
int     query_lock()
```

Return the state (true if locked).

9.17.4 toggle

```
int     toggle()
```

Toggle the state and return new state.

9.18 Functions of /basic/property

Valid properties can be found in the include files in '/sys/prop/'. Please use the #defined names of properties and **NOT** the string names of properties because the strings might change (that's what include files are for, after all).

9.18.1 set_property

```
void    set_property(key,value)
string  key;
mixed   value;
```

Set the property key to value. If value is zero the property is set to one. Use remove_property to zero a property.

9.18.2 query_property

```
mixed   query_property(key)
string  key;
```

Query the value of property key. If key is zero a mapping with all properties of the object is returned.

9.18.3 remove_property

```
void    remove_property(key)
string  key;
```

Remove the property key.

9.19 Functions of /basic/reinit

9.19.1 reinit

```
void    reinit()
```

call init() on all livings in this object.

9.20 Functions of /basic/time

All functions of '/basic/time' exist in two version: time_* and heart_* - the second version takes the return value of player->query_age(); as an argument.

9.20.1 time_val

```
int     *time_val(time)
int     time;
```

Returns an array ({ sec, min, hours, days, year, dayofyear, month, weekday, dayofmonth }) converted from time, which should be a time in seconds. (e.g. "/obj/cron/"->time(), which is the tubmud time).

9.20.2 time_long

```
string  time_long(time)
int     time;
```

Returns a string formatted as the output for the player age from the score command. The argument is the same as to *_val.

9.20.3 time_short

```
string  time_short(time)
int     time;
```

Returns a string formatted as the output for the player age from the people command. The argument is the same as to *_val.

9.21 Functions of /basic/timedep

9.21.1 set_timedep

```
string  *set_timedep(when,what,space)
int     *when;
mixed   *what;
string  *space;
```

9.21.2 query_timedep

```
mixed   query_timedep(space)
string  *space;
```

9.21.3 add_timedep

```
string  *add_timedep(when,what,space)
int     when;
mixed   what;
string  *space;
```

9.21.4 query_timedep_table

```
mixed   *query_timedep_table(space)
string  *space;
```

9.21.5 set_timedep_space

```
void    set_timedep_space(space)
string  *space;
```

9.22 Functions of /basic/timedepdesc

Inherit this object if you want time dependant long descriptions. All you have to do is inherit it, include "/basic/timdep.h", call one of the predefined macros set_timedep2(), set_timdep4() or set_timedep8() and pass an array of long descriptions to set_long(). The array size corresponds to the numerical part of the previously used macro name. You can set multiple time dependencies, either with the provided macros or by creating new ones:

```
set_timedep ( ({ 8 * 3600, 20 * 3600 }), ({ "day","night" }), ({"day2"}) );
```

This creates a depspace named `({"day2"})` which defines the time from 8 am to 8 pm as "day" and the time from 8 pm to 8 am as "night". (It is the predefined macro set_timedep2(), in fact.)

9.22.1 set_timedepdesc

```
int     set_timedepdesc(descriptions,depspace,space)
string  *descriptions,*depspace,*space;
```

9.22.2 add_timedepdesc

```
int     add_timedepdesc(key,description,space)
mixed   key;
string  description,*space;
```

9.22.3 query_timedepdesc

```
string  query_timedepdesc(space,depspace)
string  *space,*depspace;
```

9.22.4 query_timedepdesc_table

```
string  *query_timedepdesc_table(space)
string  *space;
```

9.22.5 set_timedepdesc_space

```
void    set_timedepdesc_space(space)
string  *space;
```

9.22.6 set_long

```
void    set_long(desc,depspace)
string  *desc,*depspace;
```

Sets the long description of the object. If depspace is omitted a default space is choosen, depending on the size of the array (2, 4 and 8 are allowed as defaults). The default space must have been created before with one of the predefined macros set_timedep2(), set_timedep4() or set_timedep8().

9.23 Functions of /basic/timedepitem

This object provides a similar mechanism for "fakeitems" as /basic/fakeitem. Items defined by timedepitem can have more than one long description. Which description is returned by query_long (subject) depends on the depspace associated with 'subject'. '`/basic/timedep.h`' defines three default spaces. All specified item names may be arrays of strings. '`/basic/alias`' turns them into aliases for the same description.

9.23.1 set_item_descriptions

```
void    set_item_descriptions(names,descriptions,when,depspace,space)
mixed   *names;
string  *descriptions,when,*depspace,*space;
```

Set descriptions[x] for the item(s) names[x].

9.23.2 set_one_item_description

```
void    set_one_item_description(name,description,when,depspace,space)
mixed   name;
string  description,when,*depspace,*space;
```

Set the description of a single item at a specified time.

9.23.3 set_one_item_descriptions

```
void    set_one_item_descriptions(name,descriptions,depspace,space)
mixed   name;
string  *descriptions,*depspace,*space;
```

Set all descriptions for a single item. The size of `descriptions` has to match the size of the depspace specified by `depspace`.

9.23.4 id

```
int     id(subject)
string  subject;
```

The usual id function.

9.23.5 query_long

```
string  query_long(subject)
string  subject;
```

Query the long description of an item.

9.23.6 query_items

```
mixed   *query_items(space,when)
string  *space,when;
```

Query the array of all items.

9.23.7 add_item

```
void    add_item(name,description,depspace,when)
mixed   name,description;
string  *depspace,when;
```

If `description` is an array of strings its size has to match the size of the depspace specified as the third argument. If no third argument is provided a default space is selected; existing default spaces are defined in '`/basic/timedep.h`', they have to be set with the macros set_timedep2, set_timedep4 or set_timedep8 from the same include file before they can be used. If `description` is a string and no depspace is given the item will always be visible, otherwise the item will be visible during the period named `when` of `depspace`.

9.23.8 remove_item

```
void    remove_item(subject,when,space)
string  subject,when,*space;
```

Remove an item.

9.23.9 set_timedepitem_space

```
void    set_timedepitem_space(space)
string  *space;
```

10 Include Files

This is not a complete reference of all include files, missing include files are either for internal usage, belong to the /std/ mudlib or are just too mysterious.

10.1 /basic/

10.1.1 action.h

10.1.2 fsm.h

10.1.3 macros.h

BLUEPRINT(ob)	the blueprint of object or string `ob`
CLONED(ob)	true if object `ob` is a cloned object
LOAD(file)	loads `file` and returns 1 on sucess, 0 on failure
CONTAINS(ob, what)	true if object `ob` contains `what`, also if `what` is inside one or more container(s) inside `ob`
INSIDE	`CONTAINS(this_object(), this_player())`

10.1.4 move.h

The return codes of move() in /basic/move (they are the same as the return codes of the efun transfer()):

- MOVE_OK (zero)
- MOVE_NO_ROOM
- MOVE_NO_DROP
- MOVE_BAD_SOURCE
- MOVE_NO_INSERT
- MOVE_BAD_DEST
- MOVE_NO_GET

10.1.5 timedep.h

There are three pre-defined groups of time arrays for time dependent descriptions: TD_DAY<n>_WHAT & TD_DAY<n>_WHEN, where <n> is 2, 4 or 8.

Two phases of the day		**Four phases of the day**			
Day	Night	Dew	Day	Dawn	Night
8 - 20	20 - 8	6 - 7	7 - 18	18 - 19	19 - 6

Eight phases of the day							
Midnight	Late Night	Morning	Forenoon	Noon	Afternoon	Evening	Night
0 - 1	1 - 5	5 - 8	8 - 12	12 - 13	13 - 18	18 - 20	20 - 24

You can set one of the predefined time dependencies with the three macros: set_timedep2, set_timedep4 & set_timedep8.

10.2 /sys/

10.2.1 ctype.h

The following macros determine if a character `'c'` is ...

isalpha(c) ... a letter

isupper(c) ... uppercase

islower(c) ... lowercase

isdigit(c) ... a digit

isspace(c) ... a whitespace

ispunct(c) ... punctuation

isalnum(c) ... a digit or letter

isprint(c) ... printable

toupper(c), tolower(c), tostring(c) transform chars to uppercase chars, lowercase chars and strings.

10.2.2 stdlib.h

10.2.3 time.h

This file requires you to inherit '`/basic/time.c`'. It defines the arrays `WEEKDAYS`, `MONTHS`, `SEASONS`, all weekdays, month & seasons by name (e.g. `#define DECEMBER 11`) and the following macros:

```
query_hour
query_weekday
query_month
query_season
query_year
query_day_of_month
query_day_of_year
```

All these macros call `time_val`; instead of calling several of these macros you can call `time_val` once and use the macros `TV_HOUR`, `TV_MONTH`, etc. which are indices in the returned array.

10.2.4 timezone.h

The macros localtime() and localdaytime() return the local mud time as a total and as seconds per day.

10.2.5 turncoat.h

Provide an easy way to use the 3.0 create() strategy in compatibility mode. This defines reset to __turncoat_reset. /basic/turncoat, which is inherited by turncoat.h, provides a dummy __turncoat_reset function.

11 Object Properties

12 The Master Object

12.1 Startup

The master object is the second object loaded after void.c. Everything written with `write()` at startup will be printed on stdout.

1. reset() will be called first.
2. flag() will be called once for every argument to the flag -f supplied to the driver.
3. epilog() will be called.
4. The game will enter multiuser mode, and enable log in.

12.2 Differences between COMPAT and NATIVE mode

One thing is different in the file access permission system in game driver 3.0. The master object can read and write any file. That means that all occurences of such manipulations below must be carefully guarded against calls from elsewhere. Always make a comment at the start of a function that is doing such things, to make it easy for other to quickly assert security. The master object can of course not access files above the mudlib directory.

12.3 Functions of master.c

12.3.1 flag

```
void    flag(str)
string  str;
```

To test a new function xx in object yy, do `parse "-fcall yy xx arg" "-fshutdown"`

12.3.2 connect

```
object  connect()
```

This function is called every time a player connects; it returns the player object which is to be used for the new player. input_to() can't be called from here.

12.3.3 verify_create_wizard

```
int     verify_create_wizard(ob)
object  ob;
```

This function is called for a wizard that has dropped a castle. The argument is the file name of the object that called `create_wizard()`. Verify that this object is allowed to do this call.

12.3.4 get_wiz_name

```
mixed   get_wiz_name(file)
string  file;
```

Get the owner of a file. This is called from the game driver, so as to be able to know which wizard should have the error.

12.3.5 log_error

```
void    log_error(file,message)
string  file,message;
```

Write a compile time error message into a log file. The error occured in the object `file`, giving the error message `message`.

12.3.6 runtime_error

```
void    runtime_error(error,program,current_object,line)
string  error,program,current_object;
int     line;
```

Runtime errors will be sent to the function runtime_error().

12.3.7 save_ed_setup

```
int     save_ed_setup(wiz,setup)
object  wiz;
int     setup;
```

`save_ed_setup()` and `restore_ed_setup()` are called by the ed to maintain individual options settings. These functions are located in the master object so that the administrators can decide what strategy they want to use.

12.3.8 retrieve_ed_setup

```
int     retrieve_ed_setup(wiz)
object  wiz;
```

See `save_ed_setup()`.

12.3.9 master_create_wizard

```
string  master_create_wizard(owner,domain,caller)
string  owner,domain;
string  caller;
```

Create a home directory and a castle for a new wizard. It is called automatically from `create_wizard()`. We don't use the 'domain' info. The `create_wizard()` efun is not really needed any longer, as a call could be done to this function directly.

This function can create directories and files in '`/players/`'. It is guarded from calls from the wrong places.

12.3.10 destruct_environment_of

```
void    destruct_environment_of(ob)
object  ob;
```

When an object is destructed, this function is called with every item in that room. We get the chance to save players!

12.3.11 define_include_dirs

```
string  *define_include_dirs()
```

Define where the '#include' statement is supposed to search for files. "." will automatically be searched first, followed in order as given below. The path should contain a 'searched for.

12.3.12 query_allow_shadow

```
int     query_allow_shadow(ob)
object  ob;
```

The master object is asked if it is ok to shadow object ob. Use `previous_object()` to find out who is asking.

12.3.13 parse_command_*

Default language functions used by parse_command() in native mode.

- `string *parse_command_id_list()`
- `string *parse_command_plural_id_list()`
- `string *parse_command_adjectiv_id_list()`
- `string *parse_command_prepos_list()`
- `string parse_command_all_word()`

12.3.14 get_ed_buffer_save_file_name

```
string  get_ed_buffer_save_file_name(file)
string  file;
```

Give a file name to save the ed buffer of a player who loses his connection. Argument is the path name remembered so far by ed.

12.3.15 get_simul_efun

```
mixed   get_simul_efun()
```

Give a path to a simul_efun file. Observe that it is a string returned, not an object. But the object has to be loaded here. Return 0 if this feature isn't wanted.

`master::get_simul_efun()` can return an array, first item should be the simul_efun_name, the others names for files to call obsolete simul_efuns in.

It is a good idea to have a spare simul_efun file. In case of trouble (the original isn't loadable) get_simul_efun can load the spare object and return its name.

12.3.16 query_player_level

```
int     query_player_level(what)
string  what;
```

There are several occasions when the game driver wants to check if a player has permission to specific things.

These types are implemented so far:

```
"error messages":    If the player is allowed to see runtime error messages.
"trace":             If the player is allowed to use tracing.
"wizard":            Is the player considered at least a "minimal" wizard ?
"error messages":    Is the player allowed to get run time error messages ?
```

12.3.17 valid_exec

```
int     valid_exec(name)
string  name;
```

Checks if a certain 'program' has the right to use codeexec() `name` id the name of the 'program' that attempts to use `exec()` Note that this is different from file_name(), programname is what 'function_exists' returns; there is no leading slash in the name. Returns true if `exec()` is allowed.

12.3.18 valid_write

```
mixed   valid_write(path,eff_user,call_fun,caller)
mixed   path;
string  eff_user,call_fun;
object  caller;
```

Return the path to be used or deny write access (with return value 0) for the calling function.

call_fun may be: save_object, write_file, ed_start, mkdir, rmdir, write_bytes, remove_file, cindent, do_rename.

12.3.19 valid_read

```
mixed   valid_read(path,eff_user,call_fun,caller)
string  path,eff_user,call_fun;
object  caller;
```

Return the path to be used or deny read access (with return value 0) for the calling function.

call_fun may be: restore_object, ed_start, read_file, read_bytes, file_size, get_dir, tail, print_file, make_path_absolute, do_rename.

12.3.20 make_path_absolute

```
mixed   make_path_absolute(path)
string  path;
```

12.3.21 creator_file

```
mixed   creator_file(object_name)
string  object_name;
```

12.3.22 move_or_destruct

```
void    move_or_destruct(what,to)
object  what,to;
```

An error in this function can be very nasty. Note that unlimited recursion is likely to cause errors when environments are deeply nested.

12.3.23 valid_snoop

```
int     valid_snoop(snooper,snoopee)
object  snooper,snoopee;
```

Returns 1 if `snooper` is allowed to snoop `snoopee`.

12.3.24 valid_query_snoop

```
int     valid_query_snoop(wiz);
object  wiz;
```

Return 1 if `wiz` is allowed to find out who's snooping who.

12.3.25 prepare_destruct

```
mixed   prepare_destruct(ob)
object  ob;
```

12.3.26 privilege_violation

```
int     privilege_violation(cause,object,program)
string  cause;
mixed   object,program;
```

`privilege_violation` is called when objects try to do illegal things, or files being compiled request a privileged efun.

A call to one of the following functions is a privilege violation: wizlist_info, set_extra_wizinfo, get_extra_wizinfo, set_extra_wizinfo_size, set_this_object, send_imp.

If `what` is "nomask simul_efun" and the master denies access for `who != SIMUL_EFUN` simul_efuns can't be circumvented by efun:: outside the simul_efun file.

Return values:

- 1: The caller/file is allowed to use the privilege.
- 0: The caller was probably misleaded; try to fix the error.
- -1: A real privilege violation. Handle it as error.

12.3.27 reactivate_destructed_master

```
void    reactivate_destructed_master(flag)
int     flag;
```

If reloading of the master fails, the old master will be reactivated. The function `void reactivate_destructed_master(int flag)` will then be called in it, with flag indicating if the old variable values are preserved.

12.3.28 external_master_reload

```
void    external_master_reload()
```

When the master is too buggy to be updated from inside the game, it can be forced to update from outside by sending `SIGUSR1`. The function `external_master_reload()` will then be called in the new master. If an object is blocking the driver with a time consuming evaluation, you might need to send the signal several times, it will increment the eval_cost by MAX_COST/8.

12.3.29 slow_shut_down

```
void    slow_shut_down(minutes)
int     minutes;
```

This function is called when memory is getting low. The parameter is the suggested time until shutdown, the time for Armageddon to trans players to the shop etc...

12.3.30 heart_beat_error

```
int     heart_beat_error(object,error,program,current_object,line)
string  object,error,program,current_object;
int     line;
```

This function is called when an `error` in `current_object` has stopped the heart_beat in `object`. Return non-zero if heartbeat is to be restarted.

12.3.31 receive_imp

```
void    receive_imp(foreign_host,message)
```

The mud has received `message` from `foreign_host`.

13 Skills

13.1 Implementing Skills

Oh yes, if you have no brain object, go to /room/vill_green, there you will get one. The brain object facilitates the init() function to add_action to the player for the skill verbs. More than that, the brain holds the skill scores for the player, is autoloading and saves the skill scores on every save (including autosave).

I'll explain the mechanism of the brain. The explanation is sort of the reverse of the flow control in the actual verb execution.

Every object which you want to be affected by a certain skill (or more than one) defines a function:

```
int apply_action (string skill, int level, mixed arg)
```

which is called when the skill is used by a player on the specific object. 'skill' is identifying the skill which is used by the player for the action. The identifyer is unique for the skill and is defined when the skill is first added to the brain logic. The apply˙action of the object in question decides whether it is affected by the skill in a way similar to the normal add_actions.

The success should be depend on the 'skill' and 'level' parameter. Of course, if the 'skill' does not match, the attempt should fail. Depending on how complicated the intended action is, the success should depend on the level, which is a percentage relative to the possible skill maximum.

If there is no success in using the skill, the object must return 0 to enable others to be tried. If the value returned is 1, the player succeeded and the object already performed the nessecary action. This is also true for return values greater than 1, but the use of the skill is rewarded with value-1 added to the current skill score in the skill path.

The skill identifiers may be listed with your default skill 'action', which is only given to wizards. 'action list' gives a list of skill identifiers and the objects holding the logic and their skill path. If there is more than one skill with a similar verb, the identifier is split into a top-level verb and sub-specifier. For apply_action, these two parts are concatenated, thus 'use' 'skill' yields 'use skill'. Since the first part may not contain blanks, this method is unambiguous.

The apply_action is called by an object holding the skill parsing and score adding logic. This object may be global or created by a wizard. Never create a skill just for fun, since the skill structure is changed when a player uses it and then saved to the individual player skill save files. A skill logic may be registered to the brain by 'action add'. Please keep in mind, that this registration only affects the players brains loaded after then. This means, that the currently active players will not notice the change until they log out and back in again.

The skill logic object is called by the players brain when the appropiate verb is used. It is not cloned neither present to the player. I will explain the details of creating this object in a different documentation. The global skills may serve as an example, but most of them don't really add to the players skill score. Every skill logic object defines a function query_info to return a short explanation of the purpose.

The object /obj/actions is a sort of administration object, which knows the currently registered skill logic objects. It is queried by the brain for the list of verbs and the corresponding skill logic object file names. See also the FILES listing at the end of this document.

The magic skills are a bit special. Since spells of the same type are more likely to be related closely than defensive/offensivene spells of a very different type, the spell type is the root for spells of similar type but different character. The magic skills are quite deep since they are likely to be very specialized. We don't want the generic skills affect the magic skills very much.

13.2 Skills in the sunrise mudlib

Children in the skill-tree add to the value of their parents, so that the value of the root (called skill) always represents the sum of all skills (and as such experience). In calculating the percentage-values of a skill, it's parents are taken in account with weight decreasing as distance increases.

```
Example:
    The short sword skill is calculated from the
    absolute value of short sword, 50% of sword,
    25% of sharp weapons, 12.5% of offensive and
    6.25 % of skill in the current tree. From
    this absolute value a percentage is calculated
    with a curve where 5000 points are 50% and
    500000 are 100%.
```

When creating new skills, try to put them pretty far down in the skill-tree as they influence other sibling-skills less thet way.

A skill is an object which is cloned once at the start of the mud and that players hold references to via Marion's brain. For an example of a programmed skill, look at basic/action/miscellaneous/climb.c You should put new basic skills in directories there (create them if needed). Weapon-Skills are defined in the weapon, look at /w/ardanna/weapons for two cloneable examples (quite trivial ones).

An object which allows a skill to be used on it should define a procedure "int apply_action(string type, int level)" It gets the type and level(in percent) of the skill which the player tries to use on the object as parametres and should react accordingly. It returns success.

13.3 An Example Skill

Example (from /outside/sunrise/w/ardanna/castle.c, which is a cactus):

```
int apply_action(string type, int level){
   switch(type){
       case "climb":
              return climb_me(level);
              break;

   /* insert other skills here, like

       case "fireball":
           return fry_cactus(level);
           break;
       case "biology":
           write cactus_info;
           return 1;
    */
           }
           return 0;
}
```

The cactus then has a function called climb_me which is called when somebody tries to climb it.

```
int climb_me(int level){
   if(level<50){
      write("You are hurt badly by the sharp cactus needles!\n");
      ....
      this_player()->add_hp(-20);
      return 1;
   }
   ....
}
```

Experience is given by the climb-skill when sucess is returned. If you want to add some skillpoints on your own (which you shouldn't do until you are sure of what you are doing) use the macro's from /basic/action/miscellaneous/skills.h

13.4 The Skill Tree

Suggested outline of general skill tree:

```
skill
     miscellaneous
          curiosity
          climb
     magic
          detect
               defensive
                    alignment
                    magic
          comprehend
               neutral
                    readmagic
                    locateobject
          affect
               defensive
                    holdperson
               neutral
                    light
                    heal
               offensive
                    levitate
          create
               neutral
                    fire
                    monster
               offensive
                    fireball
          curse
               defensive
                    remove
               offensive
                    turn
                    disspell
                    initiate
```

FILES

/obj/actions	central skill logic administration
/obj/saver	helper object to save players skill scores
/global/player/brain	the brain every player gets to define the verbs
/basic/action	generic skill logic functions
/global/actions/add_action	skill logic doing skill logic administration
/global/actions/show_skill	skill logic to display skill score
/save/action/[a-z]/.o	save files for skill scores

14 Domains

Technically spoken, domains are directories in the /domains part of the file system. They are used for multi-wizard projects by allowing more than one wizard to write to them. Every domain is run by a so called domain lord who is responsible for the area. There is no restriction on the number of domains you may join or the number of wizards that are members of a single domain.

14.1 Rules for Domains

If you have to change code you didn't write yourself follow these rules:

1. Comment all changes with a line containing reason of change, your name and the date
2. Send a mail to the person whose code you changed, explaining the change.
3. If the change is likely to affect other parts of the domain, post it on the domain board.
4. **Never** delete code another person wrote, use `#ifdef 0` and `#endif` to disable it.

14.2 How to create a new domain

Ask a wizard with level ≥ 25 to create a new domain for you, if you need one. If he/she thinks your request is justified, he/she will ask you for the name of the domainlord and install it.

14.3 Domain Management

The domainlord has the exclusive right to grant other wizards access to the domain. He/She may also revoke once granted permissions and deny other wizards access to the domain. There is no guideline, though, how building within the domain is organized. This is left to the members of the domain. Any domain member may leave the domain at any time (details described below).

14.4 Domain Commands

Every wizard has the builtin command 'domains' to list the domains he/she is a member of. More functionality is provided by additional commands in /room/domain_room. Type

'list domains'	to list the domains in existence.
'list member of <domain>'	to list the members of domain <domain>.
'leave <domain>'	to leave domain <domain>.

Domain lords may grant and deny access to their domain by using the commands

```
'grant <wizard> access to <domain>' and
'deny <wizard> access to <domain>'.
```

Elders and wizards of higher level may use these commands for any domain. They may also create and destroy domains by typing:

'create <domain> for <wizard>' to create a domain called <domain> with <wizard> as domain lord. It will allocate a directory called /domains/<domain> and create a castle file /domains/<domain>/castle.c. The castle.c file will be loaded at every reboot. If there is already a directory /domains/<domain>, only internal information will be updated. This might be useful after a crash, if the information got corrupted but the file still exist.

'destroy <domain>' will destroy the given domain by revoking write access for all members of the domain and then removing the domain from the internal list. Removing the files and updating /room/init_file will need additional attention.

'promote <wizard> to lord of <domain>' will replace the domain lord by <wizard>. Please note that there may be only one domain lord per domain at any time.

14.5 Technical Information

The domain information is held in several arrays in /save/domains.o Every transaction is written to /log/DOMAINS in order to be able to restore domains after a crash. Domains objects have the domain name as their creator with the first letter capitalized. Compilation errors are written to /log/<domainname> with the first letter of the domainname capitalized.

15 The LPC Implementation

The language is defined by three files. '`lex.c`' defines the lexical elemements and takes care of preprocessor directives. '`func_spec`' defines the stackmachine codes and function prototypes. '`lang.y`' defines the grammar.

15.1 Moving Objects

When the stackmachine comes across a F_MOVE_OBJECT instruction, after having obtained object pointers for the destination, it calls the move_object routine in simulate.c. The main function of this routine is to transfer an object to a different environment, hereby making sure that the sentences, making up the command definitions are updated correctly. (Sentences are structures that contain the player's commands, which associate a command string, with an object and function.) There are also differences in the execution of a F_MOVE_OBJECT instruction depending on if the driver has been compiled with COMPAT_MODE defined or not, I shall explain what happens in the non-compat mode.

The first part involves performing some checks to see if it legal to move the object in the first place; if the object to be moved 'item' is the current_object, if the object has been "approved" and if the object is shadowing another object. The next step is to update the light level in the object's new environment.

If the item has an environment (not true if it has been freshly loaded or cloned), we need to remove its sentences that have been defined by its environment, or objects in its environment, not forgetting those that exist in objects in its environment defined by the item itself. We start by checking to see if the object has had commands enabled (that is able to perform commands, and thereby having a list of command sentences). If this is so, then we remove all sentences defined by item in its environment. Depending of whether item's environment has commands enabled, the sentences defined in the environment, by item need to be removed too. The next step is to scan through the objects in item's environment, removing sentences defined by these objects from their and our sentence lists.

The next part actually "moves" the item into its new environment by adjusting the object's and destination's next_inventory and environment pointers

The last part of the move requires the local function `init()` to be called in various objects. (The init function is mainly used to set up commands). This is rather tricky, and I have found that novices tend to get confused as to when exactly init gets called in their object, and with which object as `this_player()`. Naturally the order in which init gets called in the objects is important as command overlaying is a valuable feature.

Init is called in objects according to the following scheme: If an object `item`, with commands enabled is moved to a destination 'dest' then all objects in dest's inventory will have `init` called. In addition, regardless of whether an object has commands enabled, its own init function will be called once for each living object in its new environment, including the environment itself, should it be living.

Furthermore, during these calls, the value of the command_giver is set to the living object for which it is called. (The value of command_giver is obtained by a call to `efun::this_player()`.)

Things in the COMPAT_MODE run similarly. The main differences being that it is legal for an object to move another object and that the local function `exit()` is called in the object's environment.

15.2 Single-Threadedness

The execution mechanism of the game driver is single threaded, meaning that there is no parallel execution of lpc code (or any other code for that matter). The driver takes each input event at a time and executes until that thread has finished (the routine called by the add_action does a 'return', or a run time error crops up. This means that every time the driver is executing one of your (or anybody else's) commands, it cannot execute anything else. This means that if I write some code which does not do a return (for (;;) ; for example) the driver will block and nothing else will happen in the whole mud. In order to prevent this happening, (a reboot would be required to alleviate the problem) the driver does some run time accounting of the current execution thread, and terminates the execution of the current thread if at some point it becomes too costly, with the error eval_cost too big. The efun command can be used to see how costly a certain command was in terms of execution time.

There are two main ways to get time delays. Firstly use efun::call_out if you want to have a function in this_object() called after a certain space of time. Or define an lfun heart_beat, and use the efun::set_heart_beat(1) to arrange for the driver to call the function every two seconds. However, it must be said that having a heart_beat running for an object is very costly in terms of driver resources, so do **NOT** use the heart beat mechanism for polling etc.. I strongly advise you to turn off an object's heart beat when you no longer require it. You can do this with the use of the efun set_heart_beat(0).

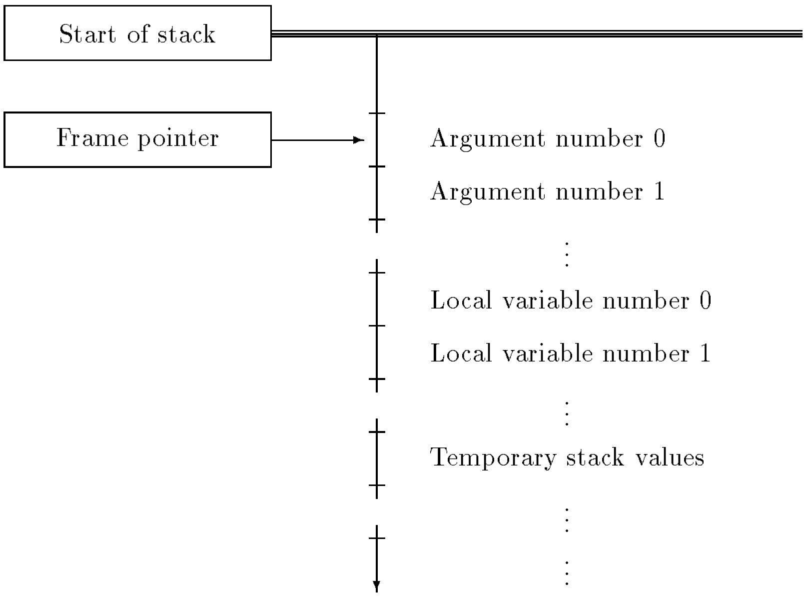
In 2.4.5 mud drivers, the call_out mechanism had a slightly different semantic. It used to be the case that this_player() was not defined in routines called by the call_out mechanism. In my opinion this was sensible, but baffled many wizards who used efuns dependant on this_player(), efun::write for example). However, now when a efun::call_out() is called, the values of this_player() (and maybe even this_interactive() as well) are saved, and restored when the function is called. However, this also means that the saved value of this_player() might no longer exist, and strange things indeed may happen.

15.3 The Virtual Stack Machine

This section describes how a virtual stack machine has been defined to execute compiled lpc code. There are two stacks:

15.3.1 Control stack and value stack

The control stack contains return addresses, frame pointer etc. The stack of values is used for evaluation, local variables and arguments. Note that arguments are treated as local variables. Every element on the value stack will have the format "struct svalue", as defined in interpret.h. The value stack is stored in an array, with limited size. The first push operation will store a value into element 0. Access of arguments and local variables are supposed to be fast, by simply indexing in the value stack using the frame pointer as offset.



Calling local functions All arguments are evaluated and pushed to the value stack. The last argument is the last pushed. It is important that the called function gets exactly as many arguments as it wants. The number of arguments will be stored in the control stack, so that the return instruction not needs to know it explicitely.

```
Instruction format:

b0 b1 b2 b3

b0     = F_CALL_FUNCTION_BY_ADDRESS
b1, b2 = The number of the function to be called.
b3     = Number of arguments sent.
```

The F_FUNCTION instruction will also initiate the frame pointer to point to the first argument.

The number of arguments are stored in the 'struct function' which is found using the number of the function and indexing in ob->prog->functions[]; The number of arguments will be adjusted to fit the called function. This is done by either pushing zeroes, or poping excessive arguments. F_CALL_FUNCTION_BY_ADDRESS will also initiate local variables, by pusing a 0 for each of them.

The called function must ensure that exactly one value remains on the stack when returning. The caller is responsible of deallocating the returned value.

When a function returns, it will use the instruction F_RETURN, which will deallocate all arguments and local variables, and only let the top of stack entry remain. The number of arguments and local variables are stored in the control stack, so that the evaluator knows hoh much to deallocate.

If flag 'extern_call' is set, then the evaluator should return. Otherwise, the evaluator will continue to execute the instruction at the returned address.

```
Format:

b0

b0 = F_RETURN.
```

Calling predefined functions Arguments are pushed to the stack. A value is always returned (on the stack).

```
Instruction format:

b1

b1 = The F_ code of the called function.

or

F_ESCAPE b1   ; b1 = The F_ code of the called function, minus 256
```

If a variable number of arguments are allowed, then an extra byte will follow the instruction, that states number of actual arguments.

The execution unit will parse number of arguments immediately, regardless of which instruction it is when it is stated that a variable number of arguments are allowed. It will also check soem of the types of the arguments immediately, if it is possible. But never more than the types of the first two arguments.

F_SSCANF The function sscanf is special, in that arguments are passed by reference. This is done with a new type, T_LVALUE. The compiler will recognize sscanf() as a special function, pass the value of the two first arguments as normal rvalues and pass the rest as lvalues. The total number of arguments is given as a one byte code supplied to the F_SSCANF instruction.

F_CALL_OTHER This command takes one argument, a byte which gives the number of arguments.

```
b1, b2

b1 = F_CALL_OTHER, b2 = number of arguments.
```

F_AGGREGATE This command takes one argument, the size of the array. The elements of the array are picked from the top of stack.

```
b1, b2, b3

b1 = F_AGGREGATE, (b2,b3) = Size of the array (max 0xffff).
```

F_CATCH The compiler constructs a call to F_CATCH before the code to evaluate the argument of F_CATCH. After the code, a call to F_END_CATCH is made. Thus, it will look like a function call on the control stack.

```
F_CATCH will when executed do setjmp() and call eval_instruction()
recursively. That means that a new frame has to be set up.

F_THROW will do a longjmp().

format:

F_THROW, b1, b2, (instructions...), F_RETURN

Where b1,b2 is the address of the instruction after the return instruction.
```

F_RETURN Will deallocate the current frame, and restore the previous. If the flag extern_call is set, then a return from eval_instruction() will be done.

15.4 How to Add Your Own Functions

The functions that returns the *void* value don't have to return anything (as in earlier implementations, where void functions returned their first argument) If you have no value to return, it is a good idea to set the function return type to void, so there needn't be a pop of the unused value.

Many 'stack instructions' exists only to be called by the compiler. Like 'pop', which obviously must not return a value on the stack. But, all those instructions are only generated explicitely by the compiler, which knows what it does.

If you want to add a new function of you own, you will have to change:

func_spec: add a prototype in the 1-byte-codes, the latter for 2-byte-codes.

Functions that only allow a constant number of arguments are best, as the compiler always knows how many arguments there are, and won't generated code information about that.

interpret.c: add a case statement in eval_instruction(). If you have a prototype in the for a prototype in the You will have to check the types of arguments. If different types are allowed, then they will have to be checked. Also, if the number of arguments are constant, then you can assume they are correct. Otherwise, use the macro GET_NUM_ARG so the variable num_var will tell you the actual number of arguments.