How To Convert Your Ideas Into Reality
A CircleMUD Coding Manual

Jeremy Elson and the rest of the CircleMUD Group
<cdev@circlemud.org>
Section 4 written by Luis Pedro Passos Carvalho
<lpcarvalho@SONAE.PT>

November 17, 2002

Abstract

This is a guide to writing C code for use with CircleMUD. Includes a description of commonly used functions, tips on how to add new commands and spells, and other commonly asked coding questions. Good familiarity with both C and UNIX is assumed, although system-level UNIX C programming skill not required. The intended audience is for CircleMUD coders, and this document does not cover area-building. See the building document for details on that.

More information about CircleMUD, including up-to-date versions of this documentation in ASCII and Postscript, can be found at the CircleMUD Home Page `<http://www.circlemud.org/>` or FTP site `<ftp://ftp.circlemud.org/pub/CircleMUD/>`

Contents

1 Introduction

When DikuMUD was first released in 1990, the authors were rightly more concerned with getting their product released than with little cosmetic details. Between writing 25,000 lines of C code and building the entire DikuMUD world, complete with objects, rooms, and monsters, it's understandable that making the code portable or clean was not at the top of the list of their priorities. Most DikuMUD distributions were not portable and had a number of bad bugs and even syntax errors which prevented the code from compiling at all. If a potential MUD implementor wanted to run a Diku, an excellent knowledge of C was necessary, because the MUD simply wouldn't run otherwise.

Now the situation is much different. With the proliferation of user-friendly code bases such as Merc and Circle, any Average Joe can just type "make" and inflict yet another MUD on the world. Therefore, the number of truly unique MUDs as a fraction of the total is dropping drastically because coding experience is no longer a prerequisite to being able to put a MUD up on the 'Net. Some people may tell you that you don't need to know how to code in order to run a MUD – don't believe them. Those people are wrong.

If you want your MUD to succeed and flourish, you'll have to know how to code in C. Otherwise, your MUD will be exactly like every other MUD out there. You're not the only person who knows how to type `make`! Although the quality and originality of your areas is also very important, it is the code that transforms the areas from lifeless text files into a living, breathing virtual world. If you don't know how to code, you won't be able to add new features, respond to requests of your players, add new world flags for your area builders, or even fix the simplest of bugs. Running a MUD without knowing how to code is certainly a recipe for disaster. If you're a great game-player and have some terrific ideas about how a MUD should work, but don't know how to code, you should either learn or find a good coder who can join your team. Don't assume that you can get away with running a MUD without knowing C – you can't. Not for very long, anyway.

This document won't teach you how to program in C; you'll have to learn that on your own. Instead, it will try to familiarize you with the way Circle's code is structured so that you can put your C skills to good use. Even for the best programmers, it takes a while to "get into" a program and feel comfortable enough with the way it works to start modifying it. Hopefully, by reading this manual, your breaking-in period for getting to know Circle will be minimized.

Circle consists of close to 30,000 lines of moderately dense C code, so you shouldn't expect familiarity to come overnight. The best way to learn is to DO. Get your hands dirty! Don't be afraid to tinker with things. Start small by modifying existing functions. Then, work your way up to creating new functions by copying old ones. Eventually you'll be able to write completely original functions, and even tear some of Circle's native functions out as you realize completely new ways of implementing them! But you should learn to walk before you try to run.

Most of all, try to remember that coding for a MUD should be fun. It can sometimes be easy to lose site of the ultimate goal of personal enjoyment that MUDs are supposed to provide, particularly when they start to get crushed under the weight of their own politics or the egos of their administrators. If you enjoy coding, but find yourself spending more time on politics than you are on code, don't be afraid to restructure your MUD or even remove yourself as Imp to a lower wizard position which requires less politics.

People often ask me why I do so much work on CircleMUD. They want to know why I spend so much time writing code *only* for the purpose of letting other people use it, since I don't actually run a MUD myself. After reading the preceding paragraph, the answer to that question should be clear. I've spent considerable time coding for on-line MUDs in the past, but after a while it just wasn't fun any more. I eventually found myself doing nothing but working politically and doing virtually no coding at all. Eventually, I even lost the will to write code completely. By quitting from my various Immortal positions on every MUD I was of which I was a member, and concentrating on my public Circle releases, I have the luxury of enjoying MUD coding in its purest form: all code, no politics. Well, almost none, anyway – my past still comes back to haunt me now and then. :)

A final thought: nothing will turn potential players away from your MUD more than logging in and finding that it's exactly like half the other CircleMUDs out there. Strive to show the world something new and unique. And may the source be with you.

2 Overview and Coding Basics

Before getting down to the details of learning how to write code, we will first examine generally what a MUD is and what it does, to give you an overview of what type of program you're working with.

The first section, "An Internet Server Tutorial", describes how Internet servers such as CircleMUD work. It contains interesting background material if you'd like a deeper understanding of how the MUD actually interacts with the Internet and the computer on which it runs, but little practical coding advice. So, if you're reading this document purely to learn how to write MUD code, you should skip to the second section.

2.1 An Internet Server Tutorial

An Internet "server" is a program which provides some service to Internet users (called "clients"). There are many different types of servers on the Internet. FTP servers allow you to transfer files between a remote computer and your own. Telnet servers allow you to connect to remote machines. News servers allow you to read USENET news. Similarly, CircleMUD is a server which allows you to play a game.

However, MUDs such as CircleMUD differ from most Internet servers in several very important ways. When ten different people connect to CircleMUD, they can all interact with one another. CircleMUD – a single program – must be aware of many users at the same time. On the other hand, most other Internet servers such as FTP servers are written to only be aware of *one* user at a time. If more than one user wants to use an FTP server simultaneously, the operating system runs two copies of the server: one to handle each user. Each individual copy of the FTP server is not aware of anything but the single user it has been assigned to serve.

This approach of making one copy of the program per user works quite well with an FTP server because all the users of an FTP server do not need to interact with one another. However, this approach does not work well at all with MUDs, because it makes the task of allowing users to communicate and interact with each other quite difficult.

In addition, most simple Internet servers do not actually contain any network code – the Internet superserver (`inetd`) contains most of the code to perform the network magic, allowing the individual servers such as FTP and telnet to be network-unaware for the most part, simply reading from standard input and writing to standard output as if a user at a normal terminal was using the program. The Internet superserver is responsible for setting up standard input and standard output so that they are actually network sockets and not a text terminal.

To sum up, a MUD such as CircleMUD does not have the luxury of being able to handle multiple users by allowing the operating system to make many copies of the MUD. The MUD itself must be capable of handling many users. The MUD also doesn't have the luxury of allowing a pre-written program such as `inetd` to set up its network connections. The MUD itself is responsible for setting up and keeping track of all of its own network connections, as well as splitting its time evenly among all of its players. The MUD cannot stop and wait for a player to type something – if it stops and waits for that player, the MUD will appear to have frozen from the point of view of all the other players!

Let's make this idea more concrete with an example. Imagine that your first programming assignment in a C class is to write a simple calculator that gets two numbers from a user as input, multiplies them together, and then prints the product on the screen as output. Your program would probably be quite simple: it would prompt the user for the two numbers, then stop and wait while the user types the numbers in.

Now, imagine that your project is to write a program that lets 10 people simultaneously connect to your calculator and multiply their own numbers together. Forget for a moment the problem of how to write the network code that allows people to connect to your program remotely. There is a more fundamental problem here: your original strategy of stopping and waiting for the user to type input won't work any more. With one user, that worked fine. But what will happen with 10 users?

Let's say your program stops and waits for the first user to type something. Now, what happens if the second user types something in the meantime? The program will not respond to the second user because it is still waiting for a response from the first user. Your simple calculator has suddenly become much more complex – now, it must constantly cycle through all users, asking the operating system if any one of them have typed something, without ever stopping to wait for a single user. When input comes in from any one of the users, your program must immediately process it and move on to the next user.

Let's say that you've written a program which does the cycling among users described in the previous paragraph. Now, imagine that the operating system tells you that User 4 has just typed the number 12. You might be able to see the second problem: what does that 12 mean? Is 12 the first or second multiplicand for your calculator? Should you immediately multiply the 12 with some other number, or store it and wait for another number to multiply by 12?

Your simple calculator has become more complicated again! Now, in addition to cycling through all users to check if any have typed anything, you must remember the STATE each user is in. In other words, each user might start out in a state called "Waiting for First Number." If a user types a number while she's in the "Waiting for First Number" state, you'd store her number somewhere and move her into the "Waiting for Second Number" state. If she types a number while in the "Waiting for Second Number" state, you'd retrieve the first number from memory, multiply it by the number just typed, and print the result. Of course, each user can be in a different state – there is no global state shared by all users.

Now, you might be able to see how this calculator example relates to CircleMUD. Let's say that the MUD receives the string, "Sleep" from a user. What should Circle do with this string? Maybe the user is trying to log in, typing her name which happens to be "Sleep". Maybe the user is typing in her password. Maybe the user is already logged in, and is trying to go to sleep! Just like with our calculator, the MUD knows how to interpret data it receives from users by examining the users' state.

You can see a list of all possible players' states in `structs.h` (they all start with "`CON_`"). All users are put into the `CON_GET_NAME` state when they first connect to the MUD. `CON_GET_NAME` simply means that the MUD is waiting for the user to type her name, at the "By what name do you wish to be known?" prompt. The normal state that most players are in most of the time is the `CON_PLAYING` state, which indicates that they have already logged in and are playing normally.

Now, let's go back to our previous example and trace exactly what happens when you type "Sleep". First, you type "Sleep." Then, your computer sends the string "Sleep" over the Internet to the computer on which the MUD is running. Within one tenth of a second, Circle checks with the operating system to see if any of its users have typed anything. When Circle gets to you, it asks the operating system, "Has this user typed anything?". Since you typed "Sleep", the operating system will respond, "Yes!". The MUD will then ask the operating system to deliver the message and will read your message of "Sleep". (All the magic of talking to the operating system and checking to see whether or not you've typed anything happens in `comm.c`.)

So, now that the MUD now knows that you've typed "Sleep", it has to decide which of several function in `interpreter.c` should get control next. This depends on what state you're in. If you're in the normal `PLAYING` state, it will pass control to a function called `command_interpreter`, which will interpret "Sleep" as a normal command and put you to sleep. If you're in any other state, control goes to a function called nanny, which is responsible for handling all sockets in any state other than `PLAYING`. nanny checks what state you're in and acts accordingly. For example, if you're in the `GET_NAME` state, nanny activates the code to check whether or not "Sleep" is the name of a known player (in which case it puts you into the state asking for your password), or a new player (in which case it'll ask you the question, "Did I get that right, Sleep?".)

In a nutshell, that's how CircleMUD interacts with the Internet. If you don't understand all the details, don't worry – it's not necessary to understand things on this level to be a successful MUD coder. If you are interested, however, there are some excellent references you can read for more information:

"Internetworking with TCP/IP" by Douglas Comer. The canonical text describing Internet protocols; comes in three volumes. Volume 1 gives an excellent description of how Internet protocols, error handling, routing, and nameservice works. Volume 3 describes specifics of writing Internet Servers in C. (Volume 2 describes how Internet protocols are implemented by operating systems and is not as apropos to this discussion.)

"Advanced Programming in the UNIX Environment" by Richard Stevens. An excellent UNIX reference for the serious system programmer. Describes POSIX quite well – worth its weight in gold for anyone trying to write portable UNIX applications. Sections on signal semantics and non-blocking I/O particularly apropos to Internet servers.

"UNIX Network Programming" by Richard Stevens. Similar to Volume 3 of Comer's series, but goes into more detail in several areas, and offers more practical code examples.

2.2 The Way Things Work – Overview

2.2.1 Boot Phase

CircleMUD is a complex system of code, data files, and external input all interacting in fun, unexpected ways. As with any program, it doesn't just spring into existence ready to play, but must cull information from the administrator and the system itself to determine how it should begin.

The first action by CircleMUD on startup is to check for the existence of any command-line parameters, as seen in the `main()` function. These can have radical impact on CircleMUD's operation so it must check them before any other action. For example, CircleMUD might be given the `-d` parameter, specifying an alternate library directory, so it cannot have done any processing on data files prior to the command-line reading.

After finishing the immediate input, the next step is to be able to communicate to the outside world. The communication may be either the "standard error" file descriptor or a file, depending on the command-line options and whichever CircleMUD succeeds in opening.

From here there are two possible branches depending on administrator input. If "Syntax Check" mode is enabled, then we load only the world. Otherwise, we start initializing the game in preparation for loading the world and accepting players. Since syntax checking is a subset of the normal startup phase, this document shall follow only the most common action of a non-syntax-check boot.

A few minor items precede the loading of the world: initializing the random number generator, creating the `.killscript` file, finding (or guessing) the maximum number of players the operating system will allow simultaneously, and opening the file descriptor later to be used to accept connections. This early opening of the "mother" file descriptor is why there is a period of time during startup where a player connection will be accepted but not receive a login prompt. CircleMUD does not check for connection attempts while it loads the world database but the operating system will complete the connection anyway and post notification to be seen later.

The world loading starts by reading in the MUD date and a few simple text files for player reference. If the date file cannot be loaded, then a default date is provided for the calculations. After randomly generating the weather, in the date function, the following user text files are loaded: news, credits, mortal message of the day, immortal message of the day, help default message screen, general MUD info, wizard list, immortal list, policies, immortal handbook, background story, and login greeting screen. These files are reproduced verbatim by various user commands and exist for the players and/or administrators to read. The MUD doesn't interpret these in any way.

Next, the spell definitions are loaded for player use. The `spello()` function gives such important information as casting cost, valid targets (area, self, etc.), spell type, valid casting positions (sitting, standing, etc.), spell name, and wear off message. Any skill or spell that isn't set up via `mag_assign_spells()` will not be usable even if appropriate code exists elsewhere that would make it have an effect. Any spell defined here that does not have appropriate code elsewhere to handle it will do nothing when used.

As far as the game world, the zones must be loaded first to define the structure of the rest of the objects. The `lib/world/zon/index` file is first consulted for the list of zone files that are requested to load. The `index.mini` file is used in the event mini-MUD mode was requested earlier on the command line. Each zone file is read in order to pull in the name, description, reset time, room range, and zone commands. Any error on loading will result in the MUD aborting startup. The rooms are read next from `lib/world/wld/index` in the same manner as the zones, with the added restriction that each room must fall within the ranges defined by the previously-loaded zones. This restriction ensures the code can access the zone record of a room without worrying about any rooms not having an associated zone.

The rooms, as loaded, contain virtual numbers of the rooms they are supposed to exit to. It is slow to do virtual to real number translation while the MUD is running so all room exits are replaced by their real number equivalents during startup. Then the pre-defined starting locations for mortals, immortals, and frozen characters are checked to make sure they exist. If the mortal start room does not exist, the MUD aborts with an error. A missing immortal or frozen start room is redirected to the mortal start room to allow booting to continue. The mortal start room must exist in a room file loaded by `lib/world/wld/index.mini` file for mini-MUD mode to work.

Mobiles from `lib/world/mob/index` and objects from `lib/world/obj/index` are loaded afterward. Mobile and object virtual numbers need not correspond to the zone number ranges as rooms do but it is encouraged. There are various sanity checks done to mobiles and objects that may be printed during startup. Any warnings issued should be fixed but should not adversely affect the MUD itself.

In the same manner as the room number virtual to real translation, the zone reset information is also translated. The zone reset structure contains a variety of different records so it takes special care to find the appropriate numbers to translate from virtual to real. Any virtual numbers that cannot be resolved result in that zone command being disabled. Such entries have their type set to '`*`' to avoid the error in the future.

If special procedures are enabled, as they usually are, the shops are loaded from `lib/world/shp/index` last. Contrary to the rooms and zones, the shops contain lists of virtual numbers to sell so any errors will only show up when attempting to use the shop. The charisma of a shopkeeper is important, as well as the buy/sell rates, as they define what prices the players will pay when shopping.

The entire help database for CircleMUD is loaded into memory after the world has been loaded. This should be about 100kB with the stock CircleMUD help files, but the extra memory used saves the MUD from having to manipulate the help files to find the appropriate entries after the boot. Even though the help index is stored by keyword, any entry having multiple keywords is only stored once for each set.

An index of the player file is built to allow random access to each player as attempt to connect or save. The index stores the player's name, to search by for login, and their ID number, for the mail system to search by. The array index is their position in the player file, used for loading and saving.

Fight messages and socials loaded next are placed by line in their appropriate categories. The messages and socials themselves aren't interpreted beyond their placement but they'll be used extensively in the game. Spells defined earlier from `mag_assign_spells()` get their battle messages from this file. Defaults are provided for the battle messages if none is defined. All socials loaded require a matching entry in the global command table or they will not be accessible by the players.

Special procedures must be associated with their now-loaded object so they're processed now. A virtual number that cannot be resolved for the special procedure elicits a warning message on startup but a missing special procedure function will cause a compiler error. Shopkeepers are assigned via `assign_the_shopkeepers()` instead of the standard `assign_mobiles()` so they can be automatically handled as a result of the shop files loaded earlier.

Since the spells are skills were defined earlier, they can then be assigned to each class upon a certain level. The spells and skills given to the various classes only depend upon a `SPELL_` or `SKILL_` definition so the assignment doesn't need to care if it is a generically handled spell or a custom implemented skill. The spells and skills may be assigned per the whims of the administrator based on their view of the appropriate classes.

The command and spell tables need sorted for special circumstances. In the case of the command table, it is for the `commands` input which displays all known commands for that player's level. The spell table is sorted for the `practice` command for easier visual searching by the player. Neither are critical to the MUD's operation but exist for the players' benefit.

The CircleMUD mail system keeps a binary file of all MUD message sent. It requires an index be built on startup to keep track of which blocks are free and which blocks have mail messages yet to be delivered. It is also important for the code to check the mail file to make sure it hasn't been corrupted somehow. A report on the number of messages present in the system is printed when finished.

The stupid-people prevention code of site banning and invalid name rejection comes next. The site ban code loads a text file list of all sites that have been deemed unworthy to connect to the MUD. Invalid name rejection loads a list of substrings that must not appear in any character's name that is being created. The invalid name list is limited in the number of entries it may hold but any amount of sites may be banned. The length of each banned site name is limited, however.

After deleting any expired rent files, the house code loads up any abodes defined. It must make sure the rooms still exist and the owner is still in the game before setting up the house, atrium, and guest list. Houses aren't loaded in mini-MUD mode since most of the rooms will likely not exist.

The final step of the world startup resets every zone. This populates the world with mobiles, the mobiles with objects, and places other objects that should be on the ground. From here, all zone timers are started and will continue to reset (or not) depending on their settings and the number of people in the zone. The time is recorded after the zone reset to provide a display of the amount of time the MUD has been running.

Once the world has finished being loaded, CircleMUD tells the operating system what sort of signals it wants to receive and which to ignore. It doesn't want to receive a `SIGPIPE`, which would abort the program whenever it tried to write to a player who abruptly disconnected. The user-defined signals (Unix) `SIGUSR1` and `SIGUSR2` are set to re-read the wizard list file and unrestrict the game, respectively. `SIGUSR1` is sent by `autowiz` whenever it finishes writing the file. `SIGUSR2` may be sent by the administrators, using `kill -USR2 pid-of-mud`, if accidentally banning themselves from their own MUD. A 3-minute timer signal prevents the MUD from being stuck in an infinite loop forever. The common shutdown signals of `SIGHUP` (Unix), `SIGTERM`, and `SIGINT` are mapped to a function that prints their reception and then quits the program. The children signal, `SIGCHLD`, is set up to remove finished `autowiz` instances.

From here, the `.killscript` file is removed since if we've made it this far, we can start successfully. The only thing left now is to enter the interactive phase in `game_loop()`, unless the "Syntax Check" option is enabled.

2.2.2 Interactive Phase

Everything that ever happens while the MUD is interactively processing players occurs as a descendant of `game_loop()`. It is responsible for all network I/O, periodic tasks, and executing player actions. Consequently, `game_loop()` has the most concentration of newbie-eating code in the code base. Care should be taken to fully understand the network I/O infrastructure before trying to modify it.

Each second of the game is divided into pulses, or points in time that network I/O is processed and commands run. This works to limit the number of possible commands the player can enter per second, such as speed-walking. If the game falls behind schedule, it will continuously process the pulses until it has caught up with where it is supposed to be. If over 30 seconds have passed, only 30 seconds are processed as it would be computationally expensive to do them all. If there isn't anyone connected to the game at all, then the MUD sleeps until someone connects. ("If a tree falls in the forest, and no one's around to hear it...")

The first task of the pulse is to check for network socket input. First, any pending connections are given descriptors to track them. Then any descriptors with a socket in the exception set is kicked from the game. Incoming socket data is read next, checked for command history or repeat operations, and placed on the appropriate descriptor's command queue.

Having read commands, they are then set up to be executed. A player must first be in a condition to execute those commands, so anyone with a wait state is skipped and people idled are pulled back into the game. Depending on the player's activities, the input may be sent through either the message writing system, the text pager, the `nanny()` login sequence, the alias system, or straight to the in-game command interpreter.

In the message writing system (see `modify.c`), any input, except for an `@` at the beginning of a line, is simply appended to the string the character has decided to edit. An `@` at the beginning of a line finishes the editing and returns the player to their previous state. Typical uses of this are editing the character's description in the menu or writing to boards and notes in the game.

The text pager allows the player to scroll around pages of text the MUD has produced. It allows refreshing the current page, going back a page, or continuing down the page. The text pager returns the player to their previous state upon reaching the end of the text or the user telling it to quit.

The `nanny()` login sequence guides the player through the initial authentication and entering of the game. Here they are prompted for the character name and password. Upon successful login they may take actions such as changing their character's description, changing their password, deleting the character, or entering the game.

The alias system provides a method for player's to shortcut the typing of commands. Each line of input is compared against their existing aliases and, if a match is found, the desired alias result is applied to their input and placed on the command queue. This process applies before the command interpreter so the game need not care what aliases each player may define.

Finally, the command interpreter pulls off a line of input from the player's command queue. It uses the command table in `interpreter.c` to find the appropriate function, if any, to call for the given command. It does various checks to ensure the character is the proper level, in the correct position, and not frozen. Any applicable special procedures are checked before running the function in the command table. The special procedure may override the function completely, as the shopkeepers do with the `buy` and `list` commands, or allow it to execute.

Processing the commands likely generated output for the player so the network output is processed next. As much data is sent from the MUD's output queue as will fit in the operating system's socket buffer for each player. Any output that can't fit in the socket buffer is held until the next pulse when perhaps some of the pending output will have been delivered. If any players decided to exit the game or otherwise disconnected, their descriptor is marked for removal and the connection closed.

Lastly, the periodic tasks are executed via the `heartbeat()` function. Each task may run every minute, every 5 minutes, every pulse, or any other time increment. The list of tasks to run includes:

- Process each zone’s timed updates.
- Disconnect idle descriptors in the login sequence.
- Do basic mobile intelligence and special procedures.
- Determine effects of violence.
- Check the weather.
- Test for magical affect expiration.
- Regenerate health, mana, and movement.
- Auto-save characters.
- Saving the MUD time.
- Extracting dead characters.

If the user-defined signals `SIGUSR1` or `SIGUSR2` have arrived, they are processed at the bottom of the game loop. Doing such work in the signal handler itself is unsafe and could cause unpredictable behavior.

2.2.3 Shutting Down

The first responsibility on shutdown is to save the players’ characters to disk. CircleMUD tracks which characters have been modified with a `PLR_CRASH` flag so it only needs to save those characters which have changed in the period from the last auto-save to the shutdown.

To disconnect the network connections, each player socket is closed in turn. Closing their connection also frees up any memory associated but, unless memory allocation tracing is enabled, it’s not necessary since we’re about to exit anyway. The “mother” descriptor is closed last, preventing any new players from connecting.

Left to do are: closing the player database file, saving the current MUD time for next startup, and logging normal termination of game. The player database is kept open for fast access to loading and saving characters as they come and go. Saving the MUD time tries to maintain some appearance of continuity in your calendar.

The final actions before exiting are only significant if the program is running with a memory allocation tracer. Here it sets about explicitly freeing every known piece of memory previously in use by the MUD. This is done to leave only forgotten allocations that may indicate a long-term memory leak in the program. The operating system will remove even forgotten memory when the program exits but the information may help prevent an ever-increasing memory usage while running. Memory tracing will vary depending on your operating system and may not necessarily be available on your particular platform.

That's it, show's over. "`return 0;`''

2.3 CircleMUD's Global Variables

CircleMUD doesn't use objects in the sense of object-oriented languages so there are various global variables kept that must be manipulated in order to change the game world. Some are stored as arrays; others as lists. The global variables kept as arrays generally have an associated `top_of_...` function to denote its boundary.

This is not an exhaustive list but the most frequently encountered of the ones in use. A large number of global, constant strings are kept in `constants.c` but they're simply read from. Also see `config.c` for a number of configuration global variables.

2.3.1 World Variables

struct room_data *world; This array of `struct room_data` stores all the information for the rooms of the MUD universe. In addition to room titles, flags, and descriptions, it also contains lists of people, mobiles, and objects currently in the rooms.

struct char_data *mob_proto; While rooms are singular entities, there may be many copies of a single mobile running around the game world. The `mob_proto` array of `struct char_data` contains the base information for all mobiles, as well as being used for string sharing amongst all the copies.

mob_rnum top_of_mobt; The highest valid array index of `mob_proto` and `mob_index` is stored here. This is *not* the count of items in the array. That is `top_of_mobt + 1`.

struct index_data *mob_index;

struct index_data *obj_index; The number of instances of each particular mobile or object is tracked through these dynamic arrays of `struct index_data`. This is also the location for reverse-mapping the mobile/object real number back into a virtual number. In addition, the special procedure and several other variables are stored here to avoid keeping redundant data on every mobile/object in the game.

struct obj_data *obj_proto; This is analogous to `mob_proto`, except for objects. It is kept as a dynamic array of `struct obj_data`.

obj_rnum top_of_objt; The highest valid array index of `obj_proto` and `obj_index` is stored here. This is *not* the count of items in the array. That is `top_of_objt + 1`.

struct zone_data *zone_table; A dynamic array of type `struct zone_data` storing all the information for zone resets, titles, and room ranges.

zone_rnum top_of_zone_table; The highest valid array index of `zone_table` is stored here. There are `top_of_zone_table + 1` zones in the array.

2.3.2 Object Instance Lists

struct descriptor_data *descriptor_list; All players connected to the MUD have a descriptor used to send the MUD's output to and receive player input from. Each descriptor does not necessarily have a character (not logged in yet). These are stored as a linked list of `struct descriptor_data` using the `next` field.

struct char_data *character_list; All player characters and mobiles are kept in a linked-list of `struct char_data`. This list uses the `next` field of the structure.

struct obj_data *object_list; As all characters and descriptors are kept in a linked-list, so too are all the objects in the world kept. This list uses the `next` field of `struct obj_data`.

2.3.3 Other

const struct command_info cmd_info[] All user commands are kept in a large array in `interpreter.c`. The `ACMD` functions use this array, along with the command array index variable they are given, to figure out what specific text the user typed to get to this command function. This allows them to handle multiple commands with the same function with `CMD_IS()`. The size of the array is static and determined by the computer at the time of compilation.

struct weather_data weather_info Raining? Snowing? Weather changes occurring in `weather.c` are stored in this structure. The sun's current state (dawn, dusk, etc.) is kept here as well.

struct time_info_data time_info The current date and time of the game world. Used in the shop code for opening and closing the stores on schedule.

2.4 Frequently Used Functions

2.4.1 Basic String Handling

int str_cmp (const char *a, const char *b)

int strn_cmp (const char *a, const char *b, int len) These are portable, case-insensitive versions of the `strcmp()` and `strncmp()` functions. Like their C library counterparts, these functions perform a character-by-character comparison and return the difference of the first mismatching characters or zero, if the two strings are equal. The `strn_cmp()` function only compares the first `len` characters of the first string to the second string.

Many platforms have built-in routines to do case-insensitive string comparisons. Where applicable, `str_cmp()` and `strn_cmp()` are aliases for the platform-specific equivalents. On platforms without these functions built-in, CircleMUD will supply a working implementation. One should prefer the CircleMUD names to ensure portability since it incurs no run-time performance cost.

int isname (const char *str, const char *namelist) Compare a string, `str`, to each of a list of space-delimited keywords, `namelist`, using `str_cmp()`. This is used for matching an argument to an object or mobile, which has its keywords arranged like, `bottle brown beer`.

bool is_abbrev (const char *str, const char *arg2) A case-insensitive substring match. Equivalent to: "`strn_cmp(needle, haystack, strlen(needle)) == 0`" Generally this is user-extended to act like `isname()`, except for abbreviated keywords.

void skip_spaces (char **string) The command interpreter hands off "` cabbage`" if the user types in "`look cabbage`". Since comparisons need to be done without the extra space in the string, this function removes it. It is also used internally for the argument splitting functions to properly handle user input such as: "`put the cabbage in the bag`".

bool is_number (const char *str) Tests if an entire string is an ASCII-encoded, unsigned decimal number by performing `isdigit()` on each character of the string. Only unsigned (zero or positive) numbers are recognized.

char *delete_doubledollar (char *string) The MUD, in processing input, converts a single dollar sign to a double dollar sign. If you want to echo out a user's input through something other than `act()`, you will want to smash '$$' into '$' by using this function.

2.4.2 Argument Processing

char *one_argument (char *argument, char *first_arg)

char *two_arguments (char *argument, char *first_arg, char *second_arg)

char *any_one_arg (char *argument, char *first_arg) These functions are frequently used in MUD commands to parse the arguments to those commands. As their names imply, `one_argument()` will peel off one argument from the string given by the user while `two_arguments()` will peel off two at a time. Note that these functions ignore (and will not return) words such as: "in", "from", "with", "the", "on", "at", and "to". This is so the commands do not need to know the difference between "put the sword in the bag" and "`put sword bag`". If those words are really needed for the command, then use `any_one_arg()` instead. It works just like `one_argument()` in all other respects. All of these functions convert the peeled off argument(s) to lower case as part of the process of storing them in the user-supplied buffer.

char *one_word (char *argument, char *first_arg) Peels an argument off from a string like `one_argument`, but respects grouping via quoting. If the user supplies, '`"moby dick"`', `one_argument()` would return an argument of '"moby', while `one_word()` would return an argument of ‘`moby dick`’. This function converts the peeled off argument(s) to lower case as part of the process of storing them in the user-supplied buffer.

void half_chop (char *string, char *arg1, char *arg2) Apparently a DikuMud relic. Instead of returning the leftover argument bits in the return value, this copies the result (sans leading spaces) into a second buffer.

2.4.3 Character Output (Hello, world!)

void log (const char *format, ...) Whenever a piece of information needs to be sent to the MUD’s logs, this is the function to use. It is especially useful for debugging and supports variable arguments like the `printf()` and `sprintf()` functions. To prevent compilation errors due to a conflict with C’s natural logarithm function of the same name, ‘`log`’ is actually an alias for this function’s real name, `basic_mud_log()`.

void mudlog (const char *str, int type, int level, bool file) In most cases `mudlog()` is better than `log()` because it announces to both the immortals on the MUD and, optionally, the file logs. Chances are the immortals will notice something faster while logged in to the game than in the system logs.

void send_to_char (struct char_data *ch, const char *messg, ...) This is the game’s tether to the players; its mouth; its voice. Most game output goes through this function so it is used very frequently. It supports variable argument formatting like the C library’s `printf()` and `sprintf()` functions.

void act (const char *str, bool hide_invisible, struct char_data *ch, struct obj_data *obj, const void *vict_obj, int
When dealing with character interactions there are frequently three situations to cover: the actor, the target, and the observers. This function takes care of such output along with handy cases for his/her, he/she/it, and other language special cases.

See the ’`act()`’ documentation (`act.pdf`) for more information on what each particular parameter is used for.

void page_string (struct descriptor_data *d, char *str, bool keep_internal) Places the character into a pageable view of whatever string is given to it. Handy for long board messages, opening announcements, help text, or other static information.

2.4.4 File Input

int get_line (FILE *fl, char *buf) Reads one or more lines, if possible, from the given file handle into a user-supplied buffer. This skips lines that begin with an asterisk (*), which are considered to be comments, and blank lines. The returned line has the line terminator(s) removed, and the total number of lines read to find the first valid line is returned. A value of zero indicates that an error occurred or that end of file was reached before a valid line was read.

int get_filename (char *orig_name, char *filename, int mode) Fills in the 'filename' buffer with the name of a file of type '`mode`' for a player with name '`orig_name`'. The mode parameter can be one of:

- `CRASH_FILE`, for player object files,
- `ALIAS_FILE`, for player aliases,
- `ETEXT_FILE`, for the unimplemented e-text system.

The returned filename contains a path to a file in a directory based upon the file type and the first letter of '`orig_name`'.

2.4.5 Utility Functions

int rand_number (int from, int to) Rolls a random number using a (pseudo) random number generator in the range [from, to]. The random number generator is seeded with the time CircleMUD booted as returned by the `time()` system call. This provides a good, difficult to predict sequence of numbers.

int dice (int num, int size) Simulate rolling '`num`' dice, each with '`size`' sides, and return the sum of the rolls.

size_t sprintbit (bitvector_t bitvector, const char *names[], char *result, size_t reslen) Treat an array of strings as if they were descriptions for the individual bits in '`bitvector`' and create a string describing it. This is used by the wizard function `do_stat()` to give human-readable output to the various bitvectors used as storage by the code. This is the approximate reverse of "`PRF_LOG1 | PRF_DEAF`", for example.

size_t sprinttype (int type, const char *names[], char *result, size_t reslen) Retrieves a value from an array of strings. The difference between this and "`array[number]`" is that this will avoid reading garbage values past the end of the array. `sprinttype()` assumes the string arrays are terminated by a "`\n`" entry.

int search_block (char *arg, const char **list, int exact) Searches an array of strings for a match to '`arg`' and returns the index in the array of the match, or -1 if not found. If 'exact' is false, then a prefix match is done akin to `is_abbrev()`. This is useful to map symbolic names to numerical constants. Think of it as the opposite of `array[number]`: "What index has this value?"

2.4.6 Character/Object Manipulation

void char_to_room (struct char_data *ch, room_rnum room)

void char_from_room (struct char_data *ch) Reciprocal, low-level functions to put a character into and remove a character from a given room. The room number must be specified with a "real number" (an index into the room tables) as returned by `real_room()`.

Since a character can only be in one room at a time, you must call `char_from_room()` to remove a character from his current location before placing him in another. After a `char_from_room()` call, the character is in `NOWHERE` and must be moved to another room using `char_to_room()`.

These functions do not check if the character is allowed to enter or leave the room; nor do they provide output to indicate that the character is moving.

void extract_char (struct char_data *ch) Remove the character from the game world and then frees the memory associated with it. Players are saved before removal. Any objects still on the character are dumped on the ground.

void equip_char (struct char_data *ch, struct obj_data *obj, int pos)

struct obj_data *unequip_char (struct char_data *ch, int pos) Takes a free-floating object (i.e., not equipped, in inventory, on the ground, or in another object) and equips it to the character for the specified location. `unequip_char()` does the opposite; it removes the object from the character's equipment list and returns it as a free-floating object. The object being unequipped must be placed elsewhere or destroyed. Note that some objects may not be equipped by characters of certain classes and/or alignments.

void obj_to_char (struct obj_data *object, struct char_data *ch)

void obj_from_char (struct obj_data *object) Reciprocal, low-level functions to put an object into and remove an object from a given character's inventory. Since an object can only be in one location at a time, you must use one of the `obj_from_X()` functions to remove it from its current location before using `obj_to_char()` to place it in someone's inventory. After an `obj_from_char()` call, the object is in `NOWHERE` and must be moved to another location using one of the `obj_to_X()` functions.

These functions do not check if the character is allowed to carry or discard the object; nor do they provide any output to inform anyone that the character has received or discarded the object.

void obj_to_obj (struct obj_data *object, struct obj_data *cont)

void obj_from_obj (struct obj_data *object) Reciprocal, low-level functions to put an object into and remove an object from a given container object. Since an object can only be in one location at a time, you must use one of the `obj_from_X()` functions to remove it from its current location before using `obj_to_obj()` to place it within another object. After an `obj_from_obj()` call, the object is in `NOWHERE` and must be moved to another location using one of the `obj_to_X()` functions.

These functions do not check if the container is allowed to carry or discard the object; nor do they provide any output to inform anyone that the container has received or discarded the object.

void obj_to_room (struct obj_data *object, room_rnum room)

void obj_from_room (struct obj_data *object) Reciprocal, low-level functions to put an object into and remove an object from a given room. Since an object can only be in one location at a time, you must use one of the `obj_from_X()` functions to remove it from its current location before using `obj_to_room()` to place it in a room. After an `obj_from_room()` call, the object is in `NOWHERE` and must be moved to another location using one of the `obj_to_X()` functions.

These functions do not check if the room is allowed to contain or discard the object; nor do they provide any output to inform anyone that the room has received or discard the object.

void extract_obj (struct obj_data *obj) Removes the object from its place in the world, then destroys it.

2.4.7 Object Locating

struct obj_data *get_obj_in_list_num (int num, struct obj_data *list) Get an object from '`list`' with the specified real object number. Only takes first object; no "`2.bread`" support.

struct obj_data *get_obj_num (obj_rnum nr); Find the first object in the world with the real object number given. Does not have "`2.`" support.

struct obj_data *get_obj_in_list_vis (struct char_data *ch, char *name, int *number, struct obj_data *list) Find the '`number`'-th object in '`list`' with keyword 'name' that the character can see. A `NULL` is returned on failure to locate such an object, or if not enough objects to satisfy '`number`' were found. '`number`' is a pointer to an integer so it can be decremented when doing multiple searches, such as room then world. If the first object is desired, '`number`' is left `NULL`.

struct obj_data *get_obj_vis (struct char_data *ch, char *name, int *number) Find the '`number`'-th object in the world with keyword '`name`' that the character can see. A `NULL` is returned on failure to locate such an object, or if not enough objects to satisfy '`number`' were found. '`number`' is a pointer to an integer so it can be decremented when doing multiple searches, such as room then world. If the first object is desired, '`number`' is left `NULL`.

struct obj_data *get_obj_in_equip_vis (struct char_data *ch, char *arg, int *number, struct obj_data *equipmen Find the '`number`'-th object in the character's equipment list with keyword '`name`' that the character can see. A `NULL` is returned on failure to locate such an object, or if not enough objects to satisfy '`number`' were found. '`number`' is a pointer to an integer so it can be decremented when doing multiple searches, such as equipment then inventory. If the first object is desired, '`number`' is left `NULL`.

int get_obj_pos_in_equip_vis (struct char_data *ch, char *arg, int *number, struct obj_data *equipment[]) Return the index of the 'number'-th object in the character's equipment list with keyword 'name' that the character can see. A -1 is returned on failure to locate such an object, or if not enough objects to satisfy '`number`' were found. '`number`' is a pointer to an integer so it can be decremented when doing multiple searches, such as equipment then inventory. If the first object is desired, '`number`' is left `NULL`.

int generic_find (char *arg, bitvector_t bitvector, struct char_data *ch, struct char_data **tar_ch, struct obj_dat
Searches any or all of the character's equipment, inventory, current room, and world for an object with the keyword given in '`arg`'. A 2nd or 3rd object is denoted in "`2.object`" notation. The function's return value specifies where the object was found, or 0, and the '`tar_obj`' value is updated with the object itself, or `NULL`. *NOTE:* This also does characters, either separately or simultaneously.

2.4.8 Character Locating

struct char_data *get_char_room (char *name, int *number, room_rnum room); Find the '`number`'-th character in the room with the keyword '`name`'. A `NULL` is returned on failure to locate such a character, or if not enough characters to satisfy '`number`' were found. '`number`' is a pointer to an integer so it can be decremented when doing multiple searches, such as room then world. If the first character is desired, '`number`' is left `NULL`.

struct char_data *get_char_num (mob_rnum nr); Find the first mobile in the world with real mobile number given. This does not have support for "`2.`" notation.

struct char_data *get_char_room_vis (struct char_data *ch, char *name, int *number); Find the '`number`'-th character in the room with the keyword '`name`' that is visible to the character given. A `NULL` is returned on failure to locate such a character, or if not enough characters to satisfy '`number`' were found. '`number`' is a pointer to an integer so it can be decremented when doing multiple searches, such as room then world. If the first character is desired, '`number`' is left `NULL`.

struct char_data *get_char_world_vis (struct char_data *ch, char *name, int *number); Find the '`number`'-th character in the world, searching the character's room first, with the keyword '`name`' that is visible to the character given. A `NULL` is returned on failure to locate such a character, or if not enough characters to satisfy '`number`' were found. '`number`' is generally a pointer to an integer so it can be decremented when this does both searches. If the first character is desired, '`number`' is left `NULL`.

struct char_data *get_char_vis (struct char_data *ch, char *name, int *number, int where);
When '{texttwhere' is `FIND_CHAR_WORLD`, call '`get_char_world_vis()`'. If '`where`' is `FIND_CHAR_ROOM`, call '`get_char_room_vis()`'. Otherwise, return `NULL`. This is kept for compatibility with various calls in the source code or if people want to easily change a search based on a variable.

int generic_find (char *arg, bitvector_t bitvector, struct char_data *ch, struct char_data **tar_ch, struct obj_dat
Searches the character's current room and/or world for a character with the keyword given in '`arg`'. A 2nd or 3rd character is denoted in "`2.character`" notation. The function's return value specifies where the character was found, or 0, and the '`tar_ch`' value is updated with the character itself, or `NULL`. *NOTE:* This also does objects, either separately or simultaneously.

2.4.9 Violence

void set_fighting (struct char_data *ch, struct char_data *victim); Initiates fighting between '`ch`' and '`victim`'.

void stop_fighting (struct char_data *ch); Removes the character from a fighting posture. Note that if an enemy is still considered fighting this character, the character will revert back to fighting as soon as the enemy hits them again.

void hit (struct char_data *ch, struct char_data *victim, int type); Makes the character attempt to hit the victim. The type determines if it is a skill, backstab in particular, or other type of damage to attempt to hit with. The type is generally left as `TYPE_UNDEFINED` to use the character's natural type.

int damage (struct char_data *ch, struct char_data *victim, int dam, int attacktype); Cause bodily harm to the victim, courtesy of the character. The damage and attacktype determine the message reported. Immortals and shopkeepers (that aren't charmed) may not be injured. Damage is capped at 100 per hit.

3 Adding Features

3.1 Adding Commands

In the course of writing new functionality for your MUD, one of the first projects you may try is to add your own command. Some commands, like socials, are special, but most commands will require you to implement some method to manipulate and parse the user's input. In CircleMUD, this is done with 'command functions' which have a special declaration form:

```
ACMD(do_/* Command name. */)
{
   .
   . /* Command code. */
   .
}
```

The command functions are then registered with the command interpreter by adding them to the `cmd_info[]` table in `interpreter.c`. The order within the command table is significant; entries at the top are substring matched prior to entries lower in the list. So if '`kill`' is before '`kiss`', then '`ki`' will match '`kill`'. Something else to be aware of is that this can render commands unreachable, such as '`goad`' being before '`go`'. The '`go`' can never be matched because '`goad`' will always have a valid substring matched first.

The fields of importance are:

const char *command The name of the command being registered. Remember the substring matching when deciding upon the order in the table.

byte minimum_position One of the `POS_xxx` constants `#define`'d in `structs.h`. This enforces the minimum position, inclusive, the user must be in, in order to execute the command.

ACMD(*command_pointer) This is the name of the function the command will call. It must be defined in the code with the `ACMD()` macro and prototyped above the command table itself.

int minimum_level The minimum level, inclusive, the user must be to execute the command.

int subcmd To allow the same code function to handle multiple, similar commands, this field allows an identifying number to be given to the command's function.

The ACMD declaration form is a C macro that sets up the command function to receive the right arguments. All command functions in CircleMUD receive the same set of parameters:

struct char_data *ch The character that issued the command (or, perhaps, was forced to issue the command): the actor.

const char *argument A string that contains the arguments the user gave to the command, with any leading spaces from the command interpreter still intact. For example, if the user typed "`tell ras Hello, there`" at the prompt, argument would be "` ras Hello, there`". This string is typically processed using one or more of the functions from Sections 2.4.1 (Basic String Handling) or 2.4.2 (Argument Processing).

int cmd The index within the command table where this command was found. Useful when multiple commands can invoke the same command function and you want to identify which command was issued. Since command indices can change when you add new commands, the primary use of this field is in special procedures and with the `IS_CMD` macro. Do not confuse this with the subcmd parameter, which is more general purpose.

int subcmd A special, user-defined integer value passed to select a "subcommand." Usually zero, but sometimes used when multiple commands with similar behavior are implemented with a single command function. Since the subcmd's value is supplied within the command table and has a meaning determined entirely by the command's author, it will not change when you add new commands.

A command with no arguments is very simple to write. For example, here is a simple command that sends a nice, personalized greeting to the user when she runs it:

```
ACMD(do_hello)
{
  act("Hello, $n.", FALSE, ch, NULL, NULL, TO_CHAR);
}
```

To allow the user to access this command, you have to add it to the command table as discussed before. This can be done by adding the `ACMD` prototype above `cmd_info[]` (if necessary) in `interpreter.c`, like:

```
ACMD(do_hello);
```

and then adding to `cmd_info[]` the command's information, as previously discussed:

```
{ "hello", POS_DEAD, do_hello, 0, 0 },
```

Our information specifies this is a command named "`hello`" which anyone can use, regardless of their position (since dead is the minimum) or level (since 0 is the minimum), calls the `do_hello()` function to be executed, and has no subcmd. Note that because `cmd_info[]` does not encode information about the arguments to the command, we don't need to do anything different for commands that take arguments.

Since the command interpreter doesn't process arguments for us (allowing `cmd_info[]` to be simple and general), we have to process them ourselves. Suppose we want to update our "`hello`" command to send a greeting to other users, when its invoked with an argument. First, we need to get the first argument (if any) by using the `one_argument()` function. In order to do this, we need a place to store the first argument (if any), which we declare as a local character buffer of length `MAX_INPUT_LENGTH`. (Note that regardless of what length we *expect* the argument to be, we always make our buffer the maximum input size, so users cannot overflow the buffer and crash the game.)

After this, we need to see what the argument is. If it is nothing, we'll default to our old behavior of just saying hello to our user. Otherwise, we need to look up the character with the given name. If we can't find anyone by that name, we send an error message. If we find someone, we send the greeting to the character we found. We check if the given argument is empty by seeing if its first character is C's end of string marker ('
`0`' or 0). Since `one_argument()` strips leading spaces for us, we don't have to worry about them. If the string is not empty, we need to look up a character in the current room by that name, using `get_char_vis()` (see Section 2.4.8. Character Locating).

Our changes give us:

```
ACMD(do_hello)
{
  char arg[MAX_INPUT_LENGTH]; /* First argument. */
  struct char_data *targ;     /* Who to greet? */

  one_argument(argument, arg);
```

```
    if (!*arg) {                /* Common idiom for empty string test. */
      act("Hello, $n.", FALSE, ch, NULL, NULL, TO_CHAR);
      return;                   /* We’re done for this case. */
    } else if (!(targ = get_char_vis(ch, arg, FIND_CHAR_ROOM))) {
      send_to_char(ch, NOPERSON);
      return;                   /* Done for this case. */
    }

    /* Otherwise we got a target: */
    act("You greet $N.", FALSE, ch, NULL, targ, TO_CHAR);
    act("$n greets you.", FALSE, ch, NULL, targ, TO_VICT);
    act("$n greets $N.", FALSE, ch, NULL, targ, TO_NOTVICT);
  }
```

`NOPERSON` is a constant defined in `config.c` for use an error message when the target of a command cannot be found. Other such constants are `OK` (for when everything goes well) and `NOEFFECT` (as a general failure message for skills, spells, and commands whose success rely on chance).

Of course, this command is little more than an overcomplicated social (we’ll see in the next section that socials don’t require command functions at all). Doing something of interest is up to you, as the programmer. Our "`hello`" command only serves as a starting point and demonstration of some idioms used throughout CircleMUD.

3.2 Adding Socials

Socials are commands that only write messages to the user and possibly his room and/or an optional victim. Examples are typical mud commands like "`nod`" or "`wave`", to let players perform various demonstrative actions. Our "`hello`" command above is an example. However, since socials are very common and superficially simple, there’s a simplified way to write them.

The `lib/misc/socials` file in the CircleMUD directory contains the actual socials in the following format:

```
  <command name> <hide-flag> <minimum position of victim>
  <messg to character if no argument>
  <messg to others if no argument>
  <messg to char if victim found>
  <messg to others if victim found>
  <messg to victim>
  <messg to char if victim not found>
```

```
<messg to others if victim not found>
<messg to char if vict is char>
<messg to others if vict is char>
```

The exact meaning and format of these fields is described in '`socials.pdf`', although much of it corresponds to `act()` (like the hide flag or the format of the messages) and the command interpreter (like the command name and the minimum position the *victim* must be in for the social to work).

Programmatically, the social is still a command and must be registered in `cmd_info[]` like any other command. All socials have the same command function, `do_action()`, which does the specialized social system processing. Thus, to add a social "`foobar`" to the command interpreter, we add the following line to the appropriate place in `cmd_info[]` (taking into account that order is significant, as discussed in Section 3.1):

```
{ "foobar", POS_RESTING, do_action, 0, 0 },
```

Note that the second element in the table entry is the minimum position the user of the social must be in, while the `lib/misc/socials` file stores the minimum position that the *victim* (if any) must be in for the social to work. In this case, the command can only be used by people that are awake and fully conscious (resting, sitting, or standing).

3.3 Adding Spells

CircleMUD improves greatly over standard Diku handling for spells, but how you go about adding them depends on the type of spell you want to make. Damage, affection, group, mass area, area, monster summoning, healing, status removal, and item enchanting spells are all generated in a template format with a touch of special messages and coding effects. More complicated spells such as '`locate object`', '`summon`', or '`identify`' are a combination of the behavior of spells and commands. They are spells in the sense the code checks for mana and requires the '`cast`' syntax but are also commands in the sense that beyond the basic handling, the spell is implemented as a subroutine with given parameters.

All spells require a definition to determine the amount of mana used, how the spell behaves, and what the spell is named. To do that, the function `spello()` is called from `mag_assign_spells()` in `spell_parser.c`. It is called as:

```
spello(
  unique_spell_number = ID# from 0 .. TOP_SPELL_DEFINE for this spell.
  spell_name = Name to be used for 'cast' command.
  max_mana = Mana cost of spell when first learned.
```

```
    min_mana = Minimum mana cost to ever cast the spell.
    mana_change = Reduction in mana cost per level beyond learning it.
    minimum_position = Whether castable sitting, standing, fighting, etc.
    valid_targets = If the spell targets people, yourself, items, etc.
    offensive = Whether casting on someone else is a hostile action.
    spell_routines = One or more of the magic template classifications.
    wear_off_message = Text to display when the spell wears off, or none.
  )
```

A spell with a minimum position value of 0 may not be cast by mere mortal players. For example, the spell "`armor`" is described as:

```
  spello(SPELL_ARMOR, "armor", 30, 15, 3, POS_FIGHTING,
        TAR_CHAR_ROOM, FALSE, MAG_AFFECTS,
        "You feel less protected.");
```

This spell costs 30 mana at the first level it is learned and decreases in cost by 3 mana per level afterward until it reaches the minimum of 15. The spell may be cast either fighting or standing, but not sitting, resting, sleeping, or otherwise incapacitated. Armor will target anyone in the current room as a non-hostile action. It processes through `MAG_AFFECTS` so there will be a lingering affection, after which the "`You feel less protected`" message will display.

To allocate a new spell, create a new `#define` symbol in `spells.h` in the same pattern as the others there. In that header, give the new spell an unused spell number equal to or less than `MAX_SPELLS`. If you run out of spell slots then other means not covered here will be necessary to add more spells.

3.3.1 Template Spells

Similar types of spells have generalized routines that handle multiple spells with very little different code. A damage spell is a damage spell is a damage spell. Even if a spell does multiple actions, such as blinding plus damage plus monster summon, the damage portion of the spell acts identical to a spell that simply does damage. The only difference is how much it does and whether there are special mitigating factors. For example, ’`chain lightning`’ in `mag_damage()` (since it is a `MAG_DAMAGE` spell) is simply:

```
  case SPELL_CALL_LIGHTNING:
    dam = dice(7, 8) + 7;
    break;
```

So the spell does `7d8+7` damage. Simple enough. All checking for saving throws, valid targets, proper mana reserves, etc. is all handled by the generic code with a bit of definition for the code to operate by.

The code fragment in the template sections can use any information about the caster, target, or environment that it chooses to modify the damage, success, or effect done to the target. Some spells do more damage if the caster is a magic user. Others might outright kill lower level targets but only slightly wound more experienced ones. The effect is up to you.

Affection spells require more in their fragment than the simple damage spells. They create affection structures that are then given to the target of the spell for their specified duration if the spell succeeds. More than one affection can be given by a single spell, as shown below in "`bless`":

```
case SPELL_BLESS:
  af[0].location = APPLY_HITROLL;
  af[0].modifier = 2;
  af[0].duration = 6;

  af[1].location = APPLY_SAVING_SPELL;
  af[1].modifier = -1;
  af[1].duration = 6;

  accum_duration = TRUE;
  to_vict = "You feel righteous.";
  break;
```

Any modifier listed in structs.h in the `APPLY_xxx` section may be used as the location field. The modifier's effect will depend on the affection type used. Up to `MAX_SPELL_AFFECTS` values can be assigned to. Although not listed in the above example, a '`.bitvector`' value may be assigned to if the spell should tag the player with an `AFF_` flag. If multiple castings of the same spell should be cumulative in duration, the '`accum_duration`' variable is set to `TRUE`. Likewise, if the modifier is cumulative, the '`accum_effect`' variable should be set to `TRUE`. A string assigned to '`to_room`' will be passed through `act()` for the occupants of the same room as the caster. A '`to_vict`' string will be given to `act()` with the target of the spell as the recipient of the message.

Group spells simply call another spell on everyone in your current group. If you want a '`group fly`' spell, then you make a '`fly`' spell first. Afterward, you make the '`group fly`' definition and then fill in some template areas of the `perform_mag_groups()` function. What you write there will depend on how your spell is designed.

General summoning spells (not '`summon`' itself) deal with the conjuration of mobiles. They require: '`fmsg`', a failure message array index number; '`mob_num`', the virtual mobile number to summon; '`pfail`', the percent chance of failure; and '`handle_corpse`', mostly for the "`animate dead`" spell so it can move the items from the corpse being animated to the mobile being summoned. These spells lend themselves to more customization than some of the other types.

Healing spells in `mag_points()` can restore either health or movement points by default. Just assign the amount of health healed to a ‘`healing`’ variable, the amount of movement points restored to a ‘`move`’ variable, and send the target a message with `send_to_char()`. The general code will handle updating the character’s attributes, position, and make sure a dying character is restored to normal functions if healed sufficiently.

Unaffection spells revert the effects of other spells, such as “`blindness`”, “`silence`”, or “`drunken stupor`.” There are only three variables used in `mag_unaffects()`: `to_vict`, `to_room`, and `spell`. The important variable is ‘`spell`’, which determines which spell effect this unaffection spell will counter. The ‘`to_vict`’ and ‘`to_room`’ messages are optional but sent to the victim and room, respectively, if provided.

Object alteration spells deal with magical modifications to items, such as poisoning, cursing, enchanting, or making them invisible. These spells are all unique by nature so only ‘`to_char`’ and ‘`to_room`’ are expected to be set, as messages to the character and room, respectively. If ‘`to_char`’ is left `NULL`, it is assumed the spell failed and a “`no effect`” message is given.

A creation spell conjures an item out of nothingness. The only variable expected is ‘`z`’, which specifies the object virtual number that should be created. Note that only a single object is created and there is no current mechanism for making multiples.

The last function of note, `mag_materials()`, is not a spell type at all but a helper function which can be used to require up to 3 spell reagents for a particular spell to be cast. The function will return `TRUE` if the caster has the objects, otherwise `FALSE`. If the ‘`extract`’ variable is `TRUE`, then the objects in question will be consumed by the casting. You can also make the function ‘`verbose`’, but it is more of a debugging/funny option than practical.

3.3.2 Manual Spells

Any spell that doesn’t fit one of the template molds is implemented as a manual spell. Adding a manual spell requires a function to be written, generally in `spells.c`, with the `ASPELL()` macro. After the requisite spell identifier macro is added to `spells.h`, add it to the manual spell list in `spell_parser.c`, `call_magic()`. (Search for “`MANUAL_SPELL`”.)

Manual spells are given:

level: The effective character level of the spell being cast. This is NOT the same as the level of the character because the spell could have been case by a wand, staff, or scroll instead of the character.

ch: The character causing the spell.

victim: The target of the spell, if a character.

obj: The target of the spell, if an object.

Other than that, manual spells can do anything. Think of them as being similar to standard commands in power and scope. A useful modification is to add '`argument`' support to spells so that "`locate object`" works properly and a "`change weather`" spell could make it "`better`" or "`worse`."

3.4 Adding Skills

Skills in CircleMUD are usually implemented as commands. The first steps to adding a skill are similar to those of adding a spell. First, make sure you have a clear idea of what your skill is going to do, who you're going to give it to, and how it fits in with the rest of the game. Try to avoid making too many skills that do basically the same thing – having lots of skills isn't a meaningful feature if most of them can be ignored.

After you have a good idea of what you want to do, why you want to do it, and why it's a good idea to do it, then start by adding a `SKILL_xxx #define` to `spells.h` and the corresponding `skillo()` line to `mag_assign_spells()` in `spell_parser.c`. The `skillo()` function takes, as its first argument, the `SKILL_xxx #define` and, as its second, the name of the skill, as a string. This registers the skill as something that can be practiced. As with spells, you have to register the skill's availability with individual classes at the appropriate levels in the `init_spell_levels()` function of `class.c`.

Now your skill can be gained and practiced by players of an appropriate level and class, but it doesn't actually do anything. Most skills, like "`bash`" and "`kick`", are simply commands that perform skill checks. The setup and everything else is the same as in Section 3.1, Adding Commands. The body needs to account for (1) whether the command's user can access the skill and (2) whether they were successful in using the skill. For (1), CircleMUD uses the idiom

```
if (IS_NPC(ch) || !GET_SKILL(ch, SKILL_xxx)) {
  send_to_char(ch, "You have no idea how.\r\n");
  return;
}
```

to check if the skill is available. The `GET_SKILL` macro returns the proficiency (as a percentage) the given character has in the given skill. If the proficiency is 0%, the player does not have the skill (either because his class doesn't have it or he's not learned it, yet). This check is preferred over directly testing if the player is of the right class(es) to use the skill, since that would require you to change several functions across several files to give skills to other classes (instead of just being able to add `spell_level()` calls in `class.c`).

At this point you would do argument processing in the typical manner, as well as any other checks that are necessary (for instance, you might want to check if the room is peaceful, as done in `do_bash()` in `act.offensive.c`). Last, you want to check for the success or failure of the skill by rolling a percentage to compare against the user's proficiency (probability of success). This is typically done with:

```
if (number(1, 101) > GET_SKILL(ch, SKILL_xxx)) {
  /* Failure. */
} else {
  /* Success. */
}
```

where you'd replace the comments with the relevant failure or success code.

For skills that do damage, like "`bash`" and "`kick`", the messages for success and failure are typically not encoded in the skill itself, but instead as damage messages in `lib/misc/messages`, which has the format:

```
M
 <skill number>
<death messg to skill user>
<death messg to skill victim>
<death messg to others>
<miss messg to skill user>
<miss messg to skill victim>
<miss messg to others>
<hit messg to skill user>
<hit messg to skill victim>
<hit messg to others>
<attempt to hit immortal messg to skill user>
<attempt to hit immortal messg to skill victim>
<attempt to hit immortal messg to others>
```

The skill number is the `#define` as appears in spells.h and the rest are single line messages that will be passed to `act()`. This is similar in many respects to `lib/misc/socials`. The format is discussed in more detail in a comment at the beginning of `lib/misc/messages`.

These messages are then displayed by calling `damage()` with the appropriate arguments, as discussed in Section 2.4.9, Violence, with the `attacktype` argument set to the `SKILL_xxx #define`, as in

```
/*
```

```
   * See above and Section 3.1, Adding Commands:
   * ... skill checks, argument processing, etc.
   * ... vict is set to skill’s victim.
   */

  if (number(1, 101) > GET_SKILL(ch, SKILL_FOO)) {
    /* Failure means 0 damage is done. */
    damage(ch, vict, 0, SKILL_FOO);
  } else {
    /* Success means we do some damage. */
    damage(ch, vict, 10, SKILL_FOO);
  }
```

Note that even when the skill succeeds and, thus, our call to do 10 damage to the victim of the skill is made, we’re not guaranteed to do the damage. The hit may miss, in which case `damage()` returns 0. Additionally, the hit may kill the victim, in which case `damage()` returns -1. If we’re going to be modifying `vict` in our skill’s function after the call to `damage()`, it’s important to take these return values into consideration. See `do_bash()` in `act.offensive.c`.

3.5 Adding Classes

Classes are one of the implementors’ most important decisions because the players will constantly deal with them, their advantanges, and their limitations. A good class should be balanced so that it has its own unique perks and flaws, never making other classes pointless to play.

Most changes to be done for classes will be found in the `class.c` file. There may be special quirks for classes implemented in the other files but the basic defining characteristics are all there. The class needs a name, abbreviation, menu entry, unique class number, skill list, guild information, saving throws, combat hit probabilities, an ability priority list, hit/mana/move advancement per level, basic starting kit, opposing item flags, spells, skills, experience charts, and level titles. It’s an exhaustive list, but the actual addition of a class isn’t nearly as complicated as it sounds.

The first change for a class required external of `class.c` is in `structs.h`. There, search for `CLASS_UNDEFINED` and add a new `CLASS_xxx` definition for your class name with the next available number. Remember to bump the value of `NUM_CLASSES`, just below, by 1.

Then search `structs.h` for "`Extra object flags`" so you can add an "`ITEM_ANTI_xxx`" flag for your new class. As before, use the next available number in the sequence for `ITEM_xxx` flags. Note that the limit is "`(1 « 31)`". Beyond that you’ll need special changes (not covered here) to add more flags.

The "`ITEM_xxx`" extra flags have a corresponding text description in `constants.c`, so search it for "`ITEM_x (extra bits)`". Add a string giving a short name for the new `ITEM_ANTI_xxx` flag, in the appropriate order, before the "\n" entry near the bottom.

The shops have a similar "don't trade" setup, so search shop.h for "TRADE_NOGOOD" to add a new TRADE_NOxxx item to the list for the class to be added. Below that (near "NO-TRADE_GOOD"), a line will need to be added for each new class so the 'no trade' status of a shop can be tested for the class.

With the definitions in shop.h, the shop code in shop.c can then be modified to take into account the new classes. In a manner similar to constants.c, there's a table in shop.c with textual names for the TRADE_NOxxx values. Add the new class names to the "trade_letters[]" array in the same order as the TRADE_NOxxx bits were added to shop.h. Also in shop.c, the is_ok_char() function will need modified to add "IS_xyz(...) && NOTRADE_xyz(...)" conditions, to make the above changes take effect.

Lastly for changes beyond class.c, search utils.h for "IS_WARRIOR" and make a similar definition below it for the new class.

Most of the changes to class.c will be straight-forward if going by the existing classes, so only a few items of note:

1. The letters used in parse_class() must be unique and should correspond to the highlighted characters in the 'class_menu' variable.
2. Lower saving throw values are better.
3. Lower 'thaco' values are better.

3.6 Adding Levels

Some people feel the standard 34 levels aren't enough and want to add more. Others feel 34 is too many and want to reduce it. Fortunately, changing the number of levels in the MUD is fairly painless. There are only three important things to remember: adjusting the secondary tables to match your new levels, making over 127 levels requires some additional modifications, and to readjust the mobiles afterward.

The secondary functions that rely on levels directly are: saving_throws, thaco, backstab_mult, level_exp, title_male, and title_female. These must be changed to correctly cover the entire range of new levels for the MUD. If not, the missing levels will have incomplete data and may act in unexpected ways. Fortunately, you'll receive error messages in the logs if such an event happens.

As the number of mortals levels is always one less than the lowest immortal level, changing LVL_IMMORT in structs.h to a new value will give the desired change. Make sure you change the functions described above at the same time. The other immortals levels should be adjusted accordingly.

If you're making more than 127 total levels on the MUD, a little `structs.h` surgery is required. The default range on the '`level`' variable is -128 to 127. CircleMUD doesn't actually use negative levels so changing it to '`ubyte level`' will allow 255 levels. Note that this setting hasn't been tested so test your new level settings to make sure they work as expected. If you need more than 255 levels, you'll need to change the '`byte`' to something larger, like `ush_int` (65,535) or `unsigned int` (4.2 billion). Changing the variable type beyond byte will result in the erasing of your player files and require changes elsewhere in the code where levels are manipulated.

Once you've changed the number of levels on your MUD, the implementor character you may have already created will now have the wrong level to be an implementor. If you've decreased the levels then a '`set self level XX`' command should work to drop yourself to the proper level, since you're considered above the new implementor level still. Those increasing the number of levels will find their implementor is now likely considered a mortal. In that case, you can either erase the player files to recreate yourself or make a command to make yourself the proper level, such as:

```
ACMD(do_fixmylevel)
{
  if (GET_IDNUM(ch) == 1)
    GET_LEVEL(ch) = LVL_IMPL;
}
```

Now remember to change all the mobiles too so they have proper levels. If you added levels, it'll make the mobiles weaker unless fixed. If reducing the levels, then you'll end up with error messages in the logs when those mobiles try to use saving throws or other level-dependent values.

3.7 Adding Color

Color in CircleMUD is handled on a varying scale of color levels the player can choose to display. The possible levels are off, sparse, normal, and complete. If a player has color off, no color must be sent.

To send color to the players, use the `CC*` family of macros:

CCNRM: Normal text color, as defined by player's terminal.

CCRED: Red

CCGRN: Green

CCYEL:	Yellow
CCBLU:	Blue
CCMAG:	Magenta
CCCYN:	Cyan
CCWHT:	White

Each macro takes a pointer to the character and the level at which the color given should be displayed. If the player uses a lower level of color than given to the macro in the code, the color code will reduce to an empty string so it does not appear. See 'color.pdf' for more information on this process.

Now suppose you wish to add high-intensity colors, blinking, or backgrounds for your text. The place to look for the existing color codes is in screen.h, but you'll just see codes like "\x1B[31m" there. So what is "\x1B[31m"? It is an ANSI color code understood by various terminal emulations to display color. There are predefined colors for each code and a special format to use so you can't just make up codes and expect them to work.

In order to compare the low-intensity colors with the high-intensity, an additional color must be known to complete the pattern, black:

```
#define BBLK  "\\x1B[30m"
```

The terminal background color is assumed black by CircleMUD so that particular color definition isn't in screen.h. Now a comparison of red and green with their bright counterparts:

```
#define KRED  "\\x1B[31m" (Dark)
#define BRED  "\\x1B[0;1;31m" (Bright)

#define KGRN  "\\x1B[32m" (Dark)
#define BGRN  "\\x1B[0;1;32m" (Bright)
```

If you want the bright colors, you can extend this pattern to get the other standard colors.

Once the #define is in screen.h, it needs to be usable via the CC* color convention to respect the color level of the players, so for every new color code add a CC* for it, such as:

```
#define CCBRED(ch,lvl)  (clr((ch),(lvl))?BRED:KNUL)
#define CCBGRN(ch,lvl)  (clr((ch),(lvl))?BGRN:KNUL)
```

With a number of colors, making a new `CC*` code for each one may get tedious. You may want to add a new macro, such as:

```
#define CC(ch, color, lvl) (clr((ch),(lvl))?(color):KNUL)
```

Then you can use `CC(ch, KRED, C_NRM)` instead of `CCRED(ch, C_NRM)`. Whether or not you want to use this idiom is up to you. It might come in handy once you get into blinking (use sparingly!) and background colors. The background colors are "`\x1B[40m`" (black), "`\x1B[47m`" (white), and everything in the middle as per the foreground colors.

4 Writing Special Procedures

Special procedures are the way to give life to your world. Through special procedures you can, for instance, make Mobiles react to player actions, fight intelligently,etc.

Using special procedures, your virtual world is not just a bunch of monsters, objects and rooms, reduced to a number of statistics. Just like good descriptions add atmosphere to the world, good use of special procedures adds flesh and life to the world.

Several special procedures are provided with stock CircleMUD which you can use to create your own and get used to the mechanics of special procedures. These special procedures can be found in `castle.c` and `spec_procs.c`. They range from very simple procedures, like `puff` (pulsed special procedure) or `bank` (command-driven special procedure), to very complex procedures like the guild master.

In this chapter, `FALSE` refers to the value 0 and `TRUE` to any non-zero value.

4.1 Overview of Special Procedures

Special procedures are nothing more than C functions, which are associated to mobiles, objects and rooms.

In the standard version of CircleMUD, special procedures are defined and assigned at compile time and cannot be changed or reassigned during runtime.

4.2 Pulsed vs. Command-Driven Special Procedures

Special procedures are called at three points in code: the command interpreter (command-driven special procedures), the game heartbeat and the violence code (pulsed special procedures).

There is no information kept to know if the special procedure is pulsed or command-driven, other than the behavior of the special procedure itself. When creating a special procedure you must keep in mind that it will be called in each of the three places and must therefore be prepared to react to all situations accordingly.

In the next two sub-sections we will present both types of special procedures, and in the third sub-section we'll explain how to differentiate the three types of call.

4.2.1 Pulsed Special Procedures

Every tick, the function heartbeat goes through the list of mobiles and the list of objects, updating everyone of them. If a mobile or an object as a special procedure associated to it, that special procedure is run with the parameters cmd and argument set to 0 and `NULL` respectively. The `NULL` used to be sent as an empty string (""), but this was changed in 3.0bpl19.

When there is a fight, the special procedure of a fighting mobile is called once per round with the same arguments.

4.2.2 Command Driven Special Procedures

Whenever a player issues a command, the command interpreter tries to identify it in the Master Command Table. If it succeeds, before running the command associated in the table, it checks for special procedures in the following order:

- room the player is in;
- objects in the player's equipment (does not enter the containers);
- objects in the player's inventory (does not enter the containers);
- mobiles in the room;
- objects in the room floor;

The first special procedure to succeed and return `TRUE` finishes the interpreting of the command.

If no special procedure returned true or no special procedure was found, then the command interpreter runs the procedure specified in the Master Command Table.

4.2.3 Preparing for all occurrences

In order to make your special procedure react accordingly to all the different places where the special procedure can be called you need to distinguish those places. The way to do it is through the parameters.

Whenever cmd is 0, the special procedure has been called as a pulsed special procedure, otherwise it is a command-driven special procedure.

To detect if the procedure was called through the violence code, it is a pulsed special procedure and `IS_FIGHTING(me)` must be true. Of course, this only has meaning for a mobile special procedure.

4.3 Relating Special Procedures to Objects, Mobiles, and Rooms

The special procedures are assigned to the prototypes of the objects and mobiles, not to the instances themselves. There are functions provided to assign the special procedures:

- `ASSIGNMOB(<vnum>,<special procedure>)`
- `ASSIGNOBJ(<vnum>,<special procedure>)`
- `ASSIGNROOM(<vnum>,<special procedure>)`

Stock CircleMUD also provides a place where to put all the special procedure assignments: the functions `assign_mobiles`, `assign_objects`, and `assign_rooms` in `spec_assign.c`.

4.4 The Special Procedure Function Header

The function header of any special procedure is defined in the macro `SPECIAL(<name>)` and is as follows:

```
int (<name>)(struct char\_data *ch, void *me, int cmd, char *argument)
```

where `<name>` is the name of the special procedure.

The parameters to the function are:

`ch`: Character that issued the command to command interpreter that triggered this special procedure.

me: The mobile, object or room to which the special procedure is given.

cmd: The command that ch issued as recognized by the command interpreter, or zero if the special procedure is being called by the pulse code.

argument: The command that the played has introduced at the prompt or an empty string if the special procedure is being called by the pulse code.

4.5 The Special Procedure Return Value

The return value of a special procedure is looked at only by the command interpreter and has meaning only for command-driven special procedures. Pulsed special procedures should always return false.

A command-driven special procedure can return two values:

FALSE:: The procedure did not process the command given and it should be processed in the standard way by the command interpreter.

TRUE:: The procedure reacted to the command given and so, the command interpreter should ignore it.