NakedMud: A Guide for Programmers

Geoff Hollis
hollisgf@email.uc.edu

November 18, 2006

Contents

1 Introduction

This manual is intended to give you a basic introduction to the concepts found within NakedMud's code. An attempt has been made to explain many of the important aspects of NakedMud in plain english. Additionally, tidbits of code can be found throughout the manual to give concrete examples on how to utilize NakedMud's features. That said, these are very *basic* explanations and examples mostly intended to help new programmers get their bearings when they are just beginning to wade into the code. For more advanced reading and examples of how to use the many aspects of NakedMud, programmers are encouraged to examine the source code. Header files can contain a wealth of information for the more complex aspects of NakedMud.

2 Modules

Almost all new extensions to NakedMud are expected to be added through modules. In its most basic form, a module is a directory that contains src files that are all united in some high-level, conceptual manner. So for instance, you might have a module that contains all of the mechanics for combat, or another module for all the magic mechanics, or maybe a module that adds commands with names that give your MUD the look&feel of a famous codebase like Circle, or ROM. The main point is that modules organize the source code of your mud by concept.

It is good practice to organize your code by concept rather than lumping all your additions into the main src directory. When you need to go back and debug systems within your MUD, things will be easier to find if everything is organized by system or concept. On the same token, code is much easier to maintain and extend if everything you need relating to some concept you are changing is

spatially localized. Furthermore, if you do most of your work with touching the core as little as possible, patching in new versions becomes substantially easier. And as an added benefit, if you would like to distribute new systems you have written, you can simply package up the directory containing your module. No more work is involved.

Modules are very easy to set up. Adding a module is basically like you would normally add code, except you have to make a new directory for everything that will be included in your module, and let the MUD know you are adding a new module. Here, we will write and install your very first module. It will be very simple in design, adding a single command that sends "hello world" to the user. However, it will outline the fundamentals of writing and adding a new module to your MUD - something you WILL have to know how to do to successfully utilize NakedMud. There are six steps to writing a module, which we will walk through. They are:

- Create a directory for your module
- Create a file called module.mk in your directory
- Add an entry for the module to the main Makefile module list
- Add a define for your module in mud.h
- Call your module's init function in gameloop.c
- Write the code for your module

2.1 1. Create a directory for your module

In NakedMud's src directory, you will want to create a new folder for your module. Name it to reflect what your module will do. If you're adding combat to your mud, name the module "combat". If you're adding magic, name it "magic".

2.2 2. Create a module.mk file

In the directory you have just created, you need to add a file called *module.mk*. This will be included by the main Makefile whenever it is executed, which is located in NakedMud's src directory.

In your *module.mk* file, you will list all of the source files in your module that need to be compiled. You will also any compiler flags or libraries that your module requires. This is done by adding the values to one of the variables: C_FLAGS, LIBS, or SRC – whichever is appropriate. What should your *module.mk* file look like? Let's assume you are writing a module, *foo*. It contains a file, *bar.c* which needs to be compiled. Your module also needs library *abc* and the compiler flag, *xyz*. Your module.mk file should look something like:

```
# include our source code
SRC += foo/bar.c

# include our required libraries
LIBS += -labc

# and finally, our compiler flags
C_FLAGS += -xyz
```

2.3 3. List your module in the main Makefile

You now have to let the main Makefile know your module exists. Doing so will make your module compile along with the rest of the MUD whenever the make command is issued. Edit the Makefile in your src directory. Add a listing for your module's directory to the MODULES variable. This is all that needs to be done to your Makefile.

2.4 4. Define your module in mud.h

Although modules are typically *modular* (i.e. they do not touch eachothers' inner workings), modules may sometimes use each other. So, for instance: You may write a general framework for combat that allows players to deal damage and queue attacks. This is your combat module. You may later write a magic module that supports damage spells. Chances are, you do not want to rewrite all of the damage functions that have been set up in your combat module. Thus, you will want to use the combat module. But how does your magic module know the damage module is installed? Well, you could simply assume it is. Chances are, it will be available almost all of the time. However, what if you find a serious bug in your combat module, and would like to temporarily disable combat on a live port while you debug it offline? If your magic module simply assumes your combat module is installed, it is going to be difficult to simply "turn off" your combat module. That is where this step comes in to play. The file, *mud.h* contains a definition for each module that has been installed in your mud. Outside modules can use these definitions to check if a module is currently installed or not, and then disable pieces of their own functionality with preprocessing commands (#ifdef, #endif) if the requisite modules are not installed. When you write a new module, let other modules know it exists by a definition for your module to *mud.h*.

2.5 5. Initialize your module

Modules will often add new variables and commands to your mud. These will have to be initialized. Traditionally, each module has an init_xxx function that is called each time the mud starts up. To make sure your module's init function is called when the mud starts, you will have to call it in the *main* function, which can be found in *gameloop.c*. Edit *gameloop.c*, include the a header in

your module that contains your module's init function prototype, and call it in the *main* function along with the other modules' init functions.

2.6 6. Program your module

The final step to creating a module is programming it. There are no hard and fast rules for programming a module since modules are basically whatever you want them to be. It is good to keep modules modular; a module should be more or less stand-alone, and it should only expose its functionality, never the mechanics behind that functionality. There are some instances when a module may need to depend on one or two other modules (for instance, the magic and combat example given above). These are exceptional circumstances, however, in which it is pretty obvious that a module cannot be completely stand-alone. The source code for a very basic module is provided below. It is the "hello" module, and contains a single player command that sends "hello, world!" to any player who uses it. The module also keeps track of the name of the last person who used the command within the module. This information is exposed to outside modules via a header file that contains a prototype for a function contained within the module. Note *how* and *where* the information is kept is completely hidden from outside modules. Outside modules can only gain access to the module's functionality, and not its mechanics.

```
#ifndef HELLO_H
#define HELLO_H
// hello.h
//
// This header file provides other modules access to the hello module's core
// functionality.

// this function must be called once before the hello module can be used
void init_hello(void);
```

```
// returns the name of the last person who used the "hello" command. Returns an
// empty string if noone has used "hello" yet.
const char *last_hell_user(void);
#endif // #HELLO_H

////////////////////////// END HELLO.H, START HELLO.C ///////////////////////////

// hello.c
//
// implementation of the hello module. Contains a single command that sends
// "hello, world!" to anyone who uses it. The module also keeps track of the
// past person who used this command.
#include "../mud.h"
#include "../character.h"
#include "hello.h"



//*****************************************************************************
// local variables and commands
//*****************************************************************************

char *last_hello_user_name = NULL; // the name of the last person to use hello

//
// this command sends "hello, world!" to anyone who uses it. Also keeps track of
// the name of the last person who used this command.
COMMAND(cmd_hello) {
  send_to_char(ch, "Hello, world!\r\n"); // send the command
  free(last_hello_user_name);            // clear the name of the last user
  last_hello_user_name = strdup(charGetName(ch));
}


//*****************************************************************************
// implementation of hello.h
//*****************************************************************************

void init_hello(void) {
  // make sure we allocate some memory for last_hello_user_name
  last_hello_user_name = strdup("");

  // add cmd_hello to the MUD's command table.
  // Confused? Commands will be explained next.
  add_cmd("hello", NULL, cmd_hello, 0, POS_SITTING, POS_FLYING,
          "player", FALSE, FALSE);
```

```
}
```

You should now understand the basic steps you have to go through in order to write a module. You still need to learn how to write commands, add new variables to things, and utilize NakedMud's many containers and maps. That will all come soon. For now, enjoy the fact that you now have a basic grasp on how NakedMud is structured! Try to keep everything modular. Only interact with outside modules through function calls. Never expose how a module works under the hood - only expose the fact that it *does* work, and explain how to use it. Stick to these rules, and you'll keep your problems isolated, your code clean, and your maintenance duties minimal. For a more rigorous tutorial on designing modules, readers are directed to the manual, *Extending NakedMud: An Introduction to Modules, Storage Sets, and Auxiliary Data*.

3 Commands

Commands are functions that can be executed by players in game (e.g. look, north, inventory). When adding a command to the game, one typically uses the *add_cmd* function, which is prototyped in *mud.h*.

3.1 Defining a new command

Command functions are typically defined by using the COMMAND macro. The COMMAND macro takes one argument – the command's name. The macro will expand out to a function with the following parameters:

```
void cmd_name(CHAR_DATA *ch, const char *cmd, char *arg)
```

ch is the person issuing the command. cmd is the word the a player typed to execute this function arg is the remaining argument that the player supplied to

the command.

3.2 Parsing command arguments

A command is only supplied one argument, which is in string form. In some cases, this is undesirable. For instance, consider the *give* command. It transfers multiple items from your inventory to the inventory of another player in the same room as you. Our "real" arguments are a list of the items to give, and the person to give them to. We have to manually parse them from the string, arg, supplied to the command.

A function called *parse_args* has been written to help aid in the parsing of "real" command arguments from the string argument supplied to a command function. In addition to parsing the real arguments for a command, parse_args can send error messages to players when they enter improper arguments, or do not follow the syntactic rules of a command. Parse args takes at least 5 arguments, and has the form:

```
bool parse_args(CHAR_DATA *ch, bool show_errors, const char *cmd, char *args,
                const char *syntax, ...)
```

ch is the person issuing the command, show_errors should be true if characters should be given messages for improper arguments/syntax, cmd is the name of the command that was issued, args is the argument the character supplied to the command which will now be parsed into the "real" arguments, syntax is a syntactic rule for parsing the real arguments from the arg supplied to the command, and the ellipsis (...) are pointers to variables that will be assigned values as they are parsed from the argument list a character supplied to the command.

A command's syntax is a list of variable types that must be parsed out of the command argument, in the order they should appear. *Flavor* text can be added

to syntactic rules to give commands a more natural-language-like feel. There are 9 types are variables that can be parsed by parse_args. Characters (ch), objects (obj), rooms (room), exits (exit), single words (word), integer values (int), double values (double), boolean values (bool), and full strings (string).

In addition, characters, objects, and exits can have suffixes attached to refine the search process. Valid suffixes are:

Suffix	Description
.world	ch and obj: anything in the world can be returned
.room	ch and obj: the command-issuer's room is searched
.inv	obj: the command-issuer's inventory is searched
.eq	obj: the command-issuer's equipment is searched
.multiple	ch, obj, and exit: returns multiple matches, if they exist
.noself	ch: the command-issuer is not returned if he is a match
.invis_ok	ch, obj, and exit: overrides the command-issuer's need to see matches

There are some circumstances where more than one type of data might be a valid argument. For instance, with the open command, both objects and exits may be valid targets. If multiple types are valid as an argument, they must be surrounded by { and }. Furthermore, an additional pointer to a variable must be supplied as part of the ellipsis arguments in parse_args. It must be supplied right after the multi-type variable, and must be an integer variable. When parse_args finishes, this variable will contain an integer value representing what type of data was found (PARSE_EXIT, PARSE_CHAR, PARSE_OBJ, PARSE_ROOM, PARSE_STRING, PARSE_DOUBLE, PARSE_INT, or PARSE_BOOL).

There are some times where it might be useful to interact with multiple things within a command. For instance, we may want to allow people to get all of the items in their room with a single command. Because of this characters,

objects, and exits can have a .multiple suffix attached to the end of them. This allows multiple matches for an argument to be returned at the same time. Like with multiple valid types (use of { and }), an extra variable must be supplied to the ellipsis arguments in parse_args, which will disambiguate whether a list of matches or a single match was found. It must be supplied after the pointer that will contain the singular data or list of data, as well as after any variable that will disambiguage type when { and } are used. If a list of multiple matches is returned, the list MUST be deleted use (but not its contents).

Flavor syntax can be of two types: mandatory or optional. Mandatory text is surrounded by < and > in the syntax argument supplied to parse_args. Optional text is surrounded by [and]. Flavor text is useful if you would like syntax like "give the object to person" to be just as valid as "give object person". As a concrete example of how parse_args can be utilized, let us look at the give command:

```
COMMAND(cmd_give) {
  CHAR_DATA *recv = NULL;  // the person we're giving stuff to
  void   *to_give = NULL;  // may be a list or a single item
  bool   multiple = FALSE; // are we dealing with a single item or a list?

  // try to give objects from our inventory. We can give multiple items. Give
  // them to a person in the room who is not ourself. The fact we can see the
  // receiver is implied. If we fail to find our items or receiver, parse_args
  // will tell the character what he did wrong, and we will halt the command
  if(!parse_args(ch,TRUE,cmd,arg, "[the] obj.inv.multiple [to] ch.room.noself",
                 &to_give, &multiple, &recv))
    return;

  // just a single item to give...
  if(multiple == FALSE)
    do_give(ch, recv, to_give);
  // we have a list of items to give
  else {
    LIST_ITERATOR *obj_i = newListIterator(to_give);
    OBJ_DATA        *obj = NULL;
    ITERATE_LIST(obj, obj_i) {
```

```
      do_give(ch, recv, obj);
    } deleteListIterator(obj_i);

    // we also have to delete the list that parse_args sent us
    deleteList(to_give);
  }
}
```

3.3 Adding a command to the MUD

After a command is written, a module can add it to the MUD. This is typically done through a call to the *add_cmd* function, which is prototyped in *mud.h*. The add_cmd function can be called from within a module's init function so the module can add new commands to the game without having to edit the core of NakedMud. The add_cmd function takes eight arguments:

```
void add_cmd(const char *name, const char *abbrev, COMMAND(func),
             int min_pos, int max_pos, const char *user_group, bool mob_ok,
             bool interrupts)
```

Name is the full word that a player must type, in order to execute a command (e.g. look, north, inventory). If a person does not supply a full command name, the nearest match will be executed (if one exists). So, for instance, say a player only types "inv". The inventory command will be executed, assuming no closer match exists.

sometimes, we would like certain commands to be matched to certain abbreviations, even if a closer match might exist. Take west and wear as an example. Typically, muds assign the abbreviations of w, we, and wes to the west command. However, wear is a better match for w and we than west is. Thus, we have to over-ride the normal mapping for certain commands by specifying a minimal abbreviation that should always work for the command, regardless of better matches. That is the second argument of add_cmd. If no minimal

abbreviation should exist, this value should be NULL.

Func is the function that is called when a command is successfully executed. There are some common constraints that exist for commands. These are typically matters of character position (e.g. you cannot move north while sleeping), and priviledges (e.g. the "set" command should only be available to game moderators). These are set at the time add_cmd is called. Another constraint is whether or not non-player-characters can use the command. The final value is a boolean value that says whether or not a character action should be interrupted if he issues this command (actions will be discussed momentarily).

4 Hooks

Hooks are functions that attach to the mud and execute whenever the mud broadcasts a specific signal. For instance, we might want to create a *shutdown* signal. Hooks could be attached onto the mud so that whenever the mud issues a shutdown signal, the hooks are executed. Hooks are functions that need to be executed upon some specific event happening, that are not part of the core of NakedMud. The execution of a hook should also be independent of any other hoook's execution. In practice, the hook system is usually used to run functions upon characters performing important events (e.g. wearing or removing a piece of equipment). For example: NakedMud does not support item affects. However, it is conceivable that someone may want to implement such a system. Two hooks can be written (one to execute upon wearing, one to execute upon removing) to ensure a character's affects update properly when equipment is worn or removed. Hooks allow programmers to extend critical pieces of the MUD without actually having to alter the core of the mud.

4.1 Adding a hook

Hooks are attached to the mud through the *hookAdd* function, usually in module init functions. hookAdd takes two arguments: The first is the signal on which the hook executes. The second is the hook itself. Hooks are functions that do not return anything, but take three arguments. The types of arguments depend on the type of signal the hook activates on. NakedMud sends out 11 different types of hook signals, with arguments as follows:

Signal	arg1	arg2	arg3
give	giver	receiver	object
get	taker	object	
drop	dropper	object	
enter	mover	new room	
exit	mover	old room	exit
ask	speaker	listener	speech
say	speaker		speech
greet	greeter	greeted	
wear	wearer	object	
remove	remover	object	
reset	zone		
shutdown			

4.2 Creating new hook types

It is conceivable you may want to create new hook types. For instance, maybe You've implemented combat and death. Maybe you'd like to write a combat monitor module that will track and log all deaths. You could add a new "death" hook signal. Doing this is easy. Whenever the event occurs, simply call *hookRun*.

This function takes 4 arguments: the first is the signal name, and the remaining 3 are any arguments hook functions may need to take. In the death example, one may be the killer and the other might be the person dying. Creating new hook types is extremely simple.

4.3 Unhooking

If for some reason you need to remove a hook that has been attached to the mud, you must call *hookRemove*. It takes two arguments: the first is the signal that causes the hook to execute, and the second is the hook function.

5 Events

Events are temporally delayed function executions. Events can be used for a wide variety of purposes. Examples might include a quest that is scheduled to start in 5 minutes, a disease that will kill someone in time, unless they find a cure for it, a scheduled game reboot, or perhaps a combat loop that runs every second. Events are started with a call to the *start_event* function. This function takes 6 arguments:

```
void start_event(void *owner, int delay, void *on_complete,
                 void *check_involvement, void *data, const char *arg)
```

Events are sometimes attached to *owners* – some thing (whether it be a character, object, room, another function, or maybe a variable) that is critical to the execution of the event. Events tied to an owner can be interrupted by supplying *interrupt_events_involving* with the owner. Owner can be NULL if there is no owner.

Delay is the number of pulses that must pass before the event executes. Because pulses are an odd variable to work with, seconds can instead be used

in conjunction with the SECOND macro, which converts a number in seconds to its pulses counterpart (see *mud.h*).

The on_complete variable is a function that is executed when the required delay has passed. This function must take 3 arguments. The first is a pointer to owner, the second is a pointer to data, and the third is a pointer to arg - all of which were supplied to start_event. The on_complete function is passed to start_event as a void pointer so programmers can define the types of owner and data however they please in their event functions.

check_involvement is a non-mandatory function (can be NULL) that takes two arguments: some pointer that has been passed to interrupt_events_involving, and the data of an event that was supplied along with the check_involvement function. The check_involvement function should return TRUE if the pointer passed into it can be found anywhere in data, which is passed in as the second argument to check_involvement.

Data, conceivably, can be anything. It is supplied to the on_complete function when an event's delay reaches zero. It is expected that on_complete will handle any garbage collection required for data.

Arg is an optional string argument (can be NULL) that is supplied to start_event. When an event's delay expires, it is passed to the on_complete event function. Unlike data, arg does not need to be freed after on_complete is finished with it.

Below is an example of an event in action. It is a delayed-chat event. Someone enters a chat message. 5 seconds after the command is issued, the chat is executed.

```
void dchat_on_complete(CHAR_DATA *owner, void *data, char *arg) {
  communicate(owner, arg, COMM_GLOBAL);
}
```

```
COMMAND(cmd_dchat) {
  if(!*arg)
    send_to_char(ch, "What did you want to delay-chat?\r\n");
  else
    start_event(ch, 5 SECONDS, dchat_on_complete, NULL, NULL, arg);
}
```

6 Actions

Actions are much like events. The main differences, however, are that actions *must* be attached to a character, and actions can be interrupted by use of certain commands. Actions have been added to allow for commands that require prep time before their effect is executed, during which time movement or starting of new actions will disrupt the current action (e.g. spell casting, swinging a sword). Actions are started with the *start_action* function. This function takes 7 arguments:

```
void start_action(CHAR_DATA *actor, int delay, bitvector_t where,
                  void *on_complete, void *on_interrupt, void *data,
                  const char *arg)
```

Currently, *where* is unused. When start_action is called, *where* must always have a value of 1. However, the value has been supplied in case anyone wishes to allow multiple bodyparts to act independently.

The *on_interrupt* function is of the same form of *on_complete* (see the event documentation), and is called if the actor has his action interrupted prematurely. All other parameters work the same as the parameters of the same names explained in the Events section. Below is an example action. It is a delayed-say action. Someone enters a say message. 5 seconds after the command is issued, the say is executed. The action will be interrupted if the speaker attempts to move, or enter another command that would interrupt actions.

```
void asay_on_complete(CHAR_DATA *actor, void *data, char *arg) {
  communicate(actor, arg, COMM_LOCAL);
}

void asay_on_interrupt(CHAR_DATA *actor, void *data, char *arg) {
  send_to_char(actor, "Your delayed say was interrupted.\r\n");
}

COMMAND(cmd_asay) {
  if(!*arg)
    send_to_char(ch, "What did you want to delay-say?\r\n");
  else
    start_action(ch, 5 SECONDS, 1, asay_on_complete, asay_on_interrupt,
                 NULL, arg);
}
```

7 Storage Sets

Storage sets are an integral part of NakedMud. They play an important role in the process of saving data to files, and reading it back out. They simplify the process of saving data from files by eliminating your need to come up with formatting schemes for flat files. They eliminate the need to write file parsers to extract data from files; the process of retrieving information from a file is reduced to querying for the value of some key.

In its basic form, a storage set is a mapping between a key and a value. The twist, of course, is that by converting data to a storage set, the data can be saved to a file and read back out (in the form of a storage set) without any additional work by the programmer. Storage sets map string keys to one of 7 value types: string, int, long, double, boolean, another storage set, or a list of storage sets. The functions available for interacting with storage sets can be viewed in *storage.h*. To solidify the idea behind storage sets, an example will be provided. In this example, we will track the last player to use a command, and the argument they provided. Using the command will display the name of the

previous person to use that command, and their argument.

```
// the name of the file we store the storage set as
#define STORAGE_EXAMPLE_FILE   "../lib/misc/storage_example"

COMMAND(cmd_storage_example) {
  // read in the set containing our info for the previous usage
  STORAGE_SET *set = storage_read(STORAGE_EXAMPLE_FILE);

  // make sure it exists
  if(set == NULL)
    send_to_char(ch, "Noone has used this command before.\r\n");
  // otherwise, show the character who previously used the command
  else {
    send_to_char(ch, "\%s previously used this command. "
                     "Their argument was '%s'\r\n",
                 read_string(set, "name"), read_string(set, "arg"));

    // do our garbage collection
    storage_close(set);
  }


  // now, make a storage set to hold the new info
  set = new_storage_set();
  store_string(set, "name", charGetName(ch));
  store_string(set, "arg",  arg);

  // save the information to disk
  storage_write(set, STORAGE_EXAMPLE_FILE);

  // let the person know the command was used
  send_to_char(ch, "You used the %s command.\r\n", cmd);

  // do our garbage collections
  storage_close(set);
}
```

This is, of course, a very basic example of how storage sets are utilized - so basic you may wonder why they were added in the first place. Their utility becomes more apparent when we consider NakedMud's modular design. NakedMud was designed with the intent on allowing programmers to add functionality to the

major data structures (characters, objects, rooms, accounts, and zones) without having to directly touch the code for them. Information for many of these things need to be saved to disk. So the problem arises: how do we allow this new data to be saved to disk without adding new lines of code to the functions that save these things? Storage sets were the answer. When a programmer uses *auxiliary data* to install new variables onto characters, rooms, objects, etc... they also provide functions that converts these variables into a storage set, and read them back out. This allows us to store the variables as a storage set embedded within another storage set representing the character, room, object, etc... to be stored. It also allows all of this converting to be done through functions that can be placed in a module instead of the core code for NakedMud. For a hands-on demonstration of how to utilize storage sets for such tasks, readers are directed to the manual, *Extending NakedMud: An Introduction to Modules, Storage Sets, and Auxiliary Data*.

8 Auxiliary Data

Auxiliary data is primarily used to add new variables to game entities (rooms, objects, characters, accounts, zones) without actually having to touch the core implementation of these things. The biggest gain from such a feature is the ability to modularize your code by what it is intended to do; all of the code related to combat - including new variables that must be added to characters - can stay in one module, all by itself. Adding new auxiliary data is extremely simple, but does require a bit of effort if you have not done it in the past.

New auxiliary data requires 7 things: a new structure that is the auxiliary data, a function that constructs the auxiliary data, a function that deletes the auxiliary data, a function that copies the auxiliary data, a function that copies the auxiliary data to another instance of the same type of auxiliary data, a

function that reads the auxiliary data from a storage set, and a function that converts the auxiliary data into a storage set.

Let us consider a toy example where we might want to add a new variable to something, and demonstrate how this could be done with auxiliary data. As a very basic example, let's pretend we would like to track how many times a character is read from disk, or stored. If we make the observation that a storage set of a character is only ever created when the character is read, and a character is only ever read from a storage set when he or she is loaded, we can do this by simply incrementing our save/load counts whenever a storage set of the auxiliary data is created, or the auxiliary data is turned into a storage set. The example code is provided below:

```
typedef struct {
  int num_saves;
  int num_loads;
} SL_DATA;

//
// create new save/load data
SL_DATA *newSLData(void) {
  SL_DATA *data = malloc(sizeof(SL_DATA));
  data->num_saves = 0;
  data->num_loads = 0;
  return data;
}

//
// delete the save/load data
void deleteSLData(SL_DATA *data) {
  free(data);
}

//
// copy the information contained in one save/load data to another
void SLDataCopyTo(SL_DATA *from, SL_DATA *to) {
  to->num_saves = from->num_saves;
  to->num_loads = from->num_loads;
}
```

```
//
// make a copy of the save/load data
SL_DATA *SLDataCopy(SL_DATA *data) {
  // make new data, and copy the contents of the old data over
  SL_DATA *newdata = newSLData();
  SLDataCopyTo(data, newdata);
  return newdata;
}

//
// turn the save/load data into a storage set. This must mean we are saving
// a character, so increment its save count
STORAGE_SET *SLDataStore(SL_DATA *data) {
  STORAGE_SET *set = new_storage_set();
  data->num_saves++;
  store_int(set, "saves", data->num_saves);
  store_int(set, "loads", data->num_loads);
  return set;
}

//
// parse save/load data from a storage set. This must mean we are loading a
// character, so increment its load count
SL_DATA *SLDataRead(STORAGE_SET *set) {
  SL_DATA *data = newSLData();
  data->num_saves = read_int(set, "saves");
  data->num_loads = read_int(set, "loads") + 1;
  return data;
}
```

in addition to actually writing the code for representing auxiliary data, we have to specify what types of things you'd like the auxiliary data to be "installed" onto. This has to be done before your mud enters its gameloop. This is typically done from within an init function for whatever module your auxiliary data is part of. Below is an example where we install the previous auxiliary data onto characters. Whenever a new character is created, deleted, read, saved, or copied, the necessary functions will be called to do the same things for the auxiliary data that is on the character:

```
void init_some_random_module(void) {
  auxiliariesInstall("sl_data", newAuxiliaryFuncs(AUXILIARY_TYPE_CHAR,
                                                  newSLData, deleteSLData,
                                                  SLDataCopyTo, SLDataCopy,
                                                  SLDataStore, SLDataRead));
}
```

It will also be useful to know how to gain access to instances of auxiliary that has been installed. Auxiliary data are stored on hash tables within whatever they are installed on. Auxiliary data can be looked up by querying for whatever key the auxiliary data was stored as (e.g. "sl_data" in the previous example). Here is a command that queries for the save/load data on a character, and displays it to them:

```
COMMAND(cmd_show_sl_data) {
  SL_DATA *data = charGetAuxiliaryData(ch, "sl_data");
  send_to_char(ch, "You have been saved %d times and loaded %d times.\r\n",
               data->num_saves, data->num_loads);
}
```

As is probably obvious, this is a toy example. In practice, noone would ever want to add such functionality to their game. For a practical example of how to utilize auxiliary data, readers are directed to the manual, *Extending NakedMud: An Introduction to Modules, Storage Sets, and Auxiliary Data*. It will step through the process of writing a module with auxiliary data that allows players to send mail to each other.

9 Exposing Data to Python

As you extend your mud and add new variables to the various data types housed within it, you will want to give Python access to these data. If you are extending your mud with Python, this is no problem; you can access all of the new variables by querying for the proper piece of auxiliary data and working with it. If you are extending your mud with C, though, you will have to take special actions to ensure Python gets access to the new data. For simply giving Python access to new variables on a character, there are two functions you will need to write. First, you will need to write a *getter* - a function that converts the current value of the variable to a Python object. Second, you may also have to write a *setter* - a function that converts a Python object into an appropriate C value. If you do not want people to be able to set the value of the variable from within a Python script, you do not have to write this setter function. After these two steps are done, you will have to add the new getter and setter to the appropriate Python class (i.e. PyChar, PyRoom, or PyObj). The is done with the corrsponding PyXXX_addGetterSetter() function. The how-to for this part of Python scripts is worthy of a manual of its own, although it is fairly easy to understand by observing previously written code. Thus, you're encouraged to look at how getters and setters are written in the PyChar.c, PyRoom.c, and PyObj.c files. You are also able to add methods to these three datatypes. For examples on how to add methods, also see these three files. If you would like a meatier introduction to embedding Python into a C application (because, this is essentially what you are doing), you are referred to the tutorial on this topic at the Python webpage: http://docs.python.org/ext/ext.html