# Extending NakedMud: An Introduction to Modules, Storage Sets, and Auxiliary Data

Geoff Hollis
hollis@ualberta.ca

October 4, 2005

# Contents

# 1 Introduction

There are three aspects of NakedMud's code that must be understood to efficiently build a mud using the codebase: modules, auxiliary data, and storage sets. Modules and auxiliary data allow programmers to organize their work by concept (e.g. combat-related functions and variables, magic-related stuff, etc...) rather than by data structure (e.g. all character variables, all room variables, etc...). I believe such conceptual organization make the processes of debugging, maintenance, and distribution much easier than they would otherwise be. The third aspect - storage sets - is an attempt to provide a general format for saving and loading data from files, and eliminate much of the legwork that comes with reading and writing to files. It is also (more importantly) an attempt to ensure the addition of new information to data structures in the MUD never results in formatting conflicts within files. This manual is mainly a tutorial for how to use modules, auxiliary data, and storage sets. There are three sections to this manual - one for each topic. The three sections build on top of eachother, but each one can be read and used independently with the help of code supplied in the appendices.

# 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. 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 walk through the creation of a module that allows players to send and receive mail. In later sections, it will be built on to demonstrate how storage sets and auxiliary data work.

## 2.1 Preparing to Program

Before we can begin programming the mail module, we have to make a directory for it. We will also have to change a few things in the makefile and gameloop to ensure the module will be compiled and loaded properly.

Enter your src directory, and make a new folder for the mail module. If you are in a terminal window, you can make the new directory with *mkdir mail.* You will now have to let your Makefile know that the new module exists. Open up Makefile in your src directory, and look for the line where optional modules are added to the variable, MODULES. The line you are searching for will look something like:

```
# optional modules go on this line
```

```
MODULES += time socials alias help
```

To this list, add the name of the new module you just created. Now, when the Makefile compiles your MUD, it will know that you have installed a new module called mail, and it will go into that directory and compile all the files within it. Well, almost. What actually happens is the Makefile goes into the module directory and looks for *another* makefile that lists off all the source files that need to be compiled for that module, along with all the libraries and compiler flags that are required for the new code to work. So, let's make a new makefile in the module directory to let the main makefile know which source files we will be editing. We do not have to worry about adding libraries or compiler flags for this module, as it is going to be very simple. In your module directory, create a file called *module.mk* and edit it. We will only be working with one source file in this directory, and it will be called *mail.c*. To let the main makefile know that this source file will be made, add the following lines of code to your *module.mk* file:

```
# include all of the source files contained in this module
SRC += mail/mail.c
```

In general, the path relative to the main src directory for all source files in your module should be added to the SRC variable.

Now that your MUD knows that your module exists, you will have to take some steps to initialize all of the new features your module will add to the MUD. This is traditionally done by adding an init_xxx() function to your module, and calling it when the MUD first boots up. Let us create an init function and fill it with a nonsense message until we actually have code to initialize. In your new module directory, create and edit a file called *mail.c*. To it, add the following bit of code:

```
// include all the header files we will need from the MUD core
#include "../mud.h"
#include "../utils.h"          // for get_time()
#include "../character.h"      // for handling characters sending mail
#include "../save.h"           // for char_exists()
#include "../object.h"         // for creating mail objects
#include "../handler.h"        // for giving mail to characters

// include headers from other modules that we require
#include "../editor/editor.h" // for access to sockets' notepads

// include the headers for this module
```

```
#include "mail.h"

// boot up the mail module
void init_mail(void) {
  printf("Nothing in the mail module yet!");
}
```

You will notice that we include a header called *mail.h*, which has not yet been created. Let's create the header and add all of the functions that source code outside of the mail module should have access to. In your new module directory, create and edit a file called *mail.h*. To it, add the following bit of code:

```
#ifndef MAIL_H
#define MAIL_H
// this function should be called when the MUD first boots up.
// calling it will initialize the mail module for use.
void init_mail(void);
#endif // MAIL_H
```

Then, let us call the init function where all the other modules' init functions are called. This is a two-step process. We first have to make a define in *mud.h* that informs the rest of the MUD code that the module is installed. So, edit *mud.h* in the main src directory. Near the very start of the file, you will see lists of defined of the form MODULE_XXX. With the rest of your optional modules, add the line:

```
#define MODULE_MAIL
```

We then need to go into *gameloop.c* and call the init function. Edit *gameloop.c* in your main src directory. At the end of the header files, you will see headers for optional modules. Add another entry for your mail module:

```
#ifdef MODULE_MAIL
#include "mail/mail.h"
#endif
```

Now, go down further to where all of the modules are initialized. This will be in the main() function, right before the gameworld is created. Add your init function to the list of other init functions:

```
#ifdef MODULE_MAIL
  log_string("Initializing mail system.");
  init_mail();
```

```
#endif
```

Notice how both the include for our mail.h header, and the call to our init function for the mail module are wrapped around #ifdef and #endif statements? This is to allow us to easily pull out the mail module if we ever want to. If we ever want to turn off the module, all we will have to do is go into mud.h and comment out the line, #define MODULE_MAIL. For all intents and purposes within the code, the mail module no longer exists when this line is commented out. We have now completed all of the prep work needed before we can start writing the mail module.

## 2.2 Programming a Mail Module

To start, we will aim for something simple. The first incarnation of our mail module will allow players to send written messages to one another. Sent mail will not be persistent (it will not save over reboots or crashes). That feature will be postponed until section 3: Storage Sets.

To start, we will have to create a new structure to represent a piece of sent mail. Add this to *mail.c*:

```
typedef struct {
  char *sender;       // name of the char who sent this mail
  char   *time;       // the time it was sent at
  BUFFER *mssg;       // the accompanying message
} MAIL_DATA;
```

Now, let's write some functions for a couple procedures we will be needing down the line - the creation and deletion of mail:

```
// create a new piece of mail.
MAIL_DATA *newMail(CHAR_DATA *sender, const char *mssg) {
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->sender = strdup(charGetName(sender));
  mail->time   = strdup(get_time());
  mail->mssg   = newBuffer(strlen(mssg));
  bufferCat(mail->mssg, mssg);
  return mail;
}

// free all of the memory that was allocated to make this piece of mail
```

```
void deleteMail(MAIL_DATA *mail) {
  if(mail->sender) free(mail->sender);
  if(mail->time)   free(mail->time);
  if(mail->mssg)   deleteBuffer(mail->mssg);
  free(mail);
}
```

Now that the basic steps for creating and deleting mail is complete, we can set up some commands to allow players to send mail:

```
// mails a message to the specified person. This command must take the
// following form:
//   mail <person>
//
// The contents of the character's notepad will be used as the body of the
// message. The character's notepad must not be empty.
COMMAND(cmd_mail) {
  // make sure the character exists
  if(!char_exists(arg))
    send_to_char(ch, "Noone named %s is registered on %s.\r\n",
                 arg, "<insert mud name here>");
  // make sure we have a socket - we'll need access to its notepad
  else if(!charGetSocket(ch))
    send_to_char(ch, "Only characters with sockets can send mail!\r\n");
  // make sure our notepad is not empty
  else if(!*bufferString(socketGetNotepad(charGetSocket(ch))))
    send_to_char(ch, "Your notepad is empty. "
                 "First, try writing something with {cwrite{n.\r\n");
  // the character exists. Let's parse the items and send the mail
  else {
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // if we had some way of storing mail, we'd now do that. But since we
    // don't, let's just delete the mail.
    //***********
    // FINISH ME
    //***********
    deleteMail(mail);
```

```
    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);
  }
}
```

We will also want to add this new command to a list of all the commands in the mud. So, let's go down to init_mail() and do that:

```
// boot up the mail module
void init_mail(void) {
  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
}
```

The first incarnation of our module is just about complete! The only thing we need now is a way to store mail that has been sent. To do this, we will need some way of mapping characters to their received mail, and a command for them to access that mail. Let us create a hashtable to map mail recipients to their mail. We will do this at the top of mail.c, just before we create the MAIL_DATA structure:

```
// maps charName to a list of mail they have received
HASHTABLE *mail_table = NULL;
```

We will also need to create this hashtable in our init_mail() function, so go down to that and perform the initialization for a hashtable:

```
// boot up the mail module
void init_mail(void) {
  // initialize our mail table
  mail_table = newHashtable();

  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
}
```

Earlier, we left cmd_mail unfinished, because we had no way to store mail. Now that we have a hashtable for performing this function, let's go back to cmd_mail and fill in the unfinished part:

```
  // the character exists. Let's parse the items and send the mail
```

```
  else {
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // see if the receiver already has a mail list
    LIST *mssgs = hashGet(mail_table, arg);

    // if he doesn't, create one and add it to the hashtable
    if(mssgs == NULL) {
      mssgs = newList();
      hashPut(mail_table, arg, mssgs);
    }

    // add the new mail to our mail list
    listPut(mssgs, mail);

    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);
  }
```

It's all well and good being able to send mail, but we still need a way for people to receive their mail. Let's write the command for performing that function now:

```
// checks to see if the character has any mail. If he does, convert each piece
// of mail into an object, and transfer them all into the character's inventory.
COMMAND(cmd_receive) {
  // Remove the character's mail list from our mail table
  LIST *mail_list = hashRemove(mail_table, charGetName(ch));

  // make sure the list exists
  if(mail_list == NULL || listSize(mail_list) == 0)
    send_to_char(ch, "You have no new mail.\r\n");
  // hand over all of the mail
  else {
    // go through each piece of mail, make an object for it,
    // and transfer the new object to us
    LIST_ITERATOR *mail_i = newListIterator(mail_list);
    MAIL_DATA       *mail = NULL;
    ITERATE_LIST(mail, mail_i) {
      OBJ_DATA *obj  = newObj();
```

```
      objSetName      (obj, "a letter");
      objSetKeywords  (obj, "letter, mail");
      objSetRdesc     (obj, "A letter is here.");
      objSetMultiName (obj, "A stack of %d letters");
      objSetMultiRdesc(obj, "A stack of %d letters are here.");
      bprintf(objGetDescBuffer(obj),
             "Sender   : %s\r\n"
             "Date sent: %s\r\n"
             "%s", mail->sender, mail->time, bufferString(mail->mssg));

      // give the object to the character
      obj_to_game(obj);
      obj_to_char(obj, ch);
    } deleteListIterator(mail_i);

    // let the character know how much mail he received
    send_to_char(ch, "You receive %d letter%s.\r\n",
                 listSize(mail_list), (listSize(mail_list) == 1 ? "" : "s"));
  }

  // delete the mail list, and all of its contents
  if(mail_list != NULL) deleteListWith(mail_list, deleteMail);
}
```

Like with cmd_mail, we will also have to add this new command to the list of all commands in the mud. Go down to init_mail() and add it in below our other command:

```
  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
  add_cmd("receive", NULL, cmd_receive, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
```

There, we're all done! You now have a great, basic mail module. We could probably add lots to this module, like parcels of items, fees for sending mail, checks to make sure mail is only sent at mailboxes, etc... but for our purposes of demonstrating how modules work, this will suffice. The rest of the features will be left to you as an exercise.

## 2.3 Summary

You have been provided with a hands on demonstration on how new modules are created. This topic covered lots of ground. You are not expected to immediately remember everything that has been covered so far, but going back through the entire tutorial to reread details about the module creation process would definitely be an onerous task. As such, a short reference for the module creation process has been outlined below:

- Create a directory for your module
- Create your directory's module.mk file
- 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

# 3 Storage Sets

Storage sets are a big part of NakedMud. They serve a few important purposes: They simplify the process of saving data from files by eliminating your need to come up with formatting schemes for your flatfiles. They also eliminate your 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. As we will learn later, they also play an integral role in the process of saving and loading auxiliary data.

In this section, we will learn the ropes of storage sets. We'll see how to store and read lists and strings. The other data types storage sets can deal with (ints, bools, doubles, longs) are handled in the exact same way as strings, except with different function names. After this tutorial, you should be able to extrapolate how these other data types interact with storage sets. In the next section on auxiliary data, we will examine how storage sets work in conjunction with auxiliary data.

## 3.1 Getting Up To Speed

If you have not already performed the tutorial on modules, you will need to do a couple things before you can go through the storage set tutorial. If you have already performed the tutorial on modules, disregard this section and move onto the next one.

- In your src directory, make a new directory called *mail*
- Examine appendix A. For each of the 3 files you see, add it and its contents to the *mail* directory you just created
- Edit the *Makefile* in your src directory. To the line listing off your optional modules, add *mail*
- Edit *mud.h* and add *#define MODULE_MAIL* to the list of other module defines you have
- Edit *gameloop.c* and add *#include "mail/mail.h"* to the list of optional module headers in the same fashion it is added for the other module headers
- Still in *gameloop.c*, search for the init functions of your other modules and add your *init_mail()* function in the same way the init() functions for your other modules is added

## 3.2 Storage Set Basics

In the section on modules, we designed a 'proof of concept' for a mail system. One feature we did not add was the ability for unreceived mail to be persistent. If the MUD ever crashed or rebooted, all mail not yet received would be lost. This, of course, is highly undesirable. We will now build on our mail module, and demonstrate how make mail persistent with the aid of storage sets.

The first thing we will need to do is add the header for interacting with storage sets. Let's do this where we include all the other headers we need for interacting with core features of the MUD. We'll add the new header right after we add the *handler.h* header:

```
#include "../handler.h"       // for giving mail to characters
#include "../storage.h"       // for saving/loading mail
```

Next, we will need to define a file where mail will be stored when the mud is down. The MUD's lib directory seems like an ideal candidate directory. Why don't we define where mail will be saved at the top of our *mail.c* file, where we define the hashtable for holding mail:

```
// this is the file we will save all unreceived mail in, when the mud is down
#define MAIL_FILE "../lib/misc/mail"

// maps charName to a list of mail they have received
HASHTABLE *mail_table = NULL;
```

Two things we will need are functions for converting both ways between MAIL_DATA and STORAGE_SETS. By convention, these functions are called xxxStore and xxxRead, where xxx is what we are trying to convert to and from a STORAGE_SET. Let's get those functions set up next, just below the *newMail* and *deleteMail* functions:

```
// parse a piece of mail from a storage set
MAIL_DATA *mailRead(STORAGE_SET *set) {
  // allocate some memory for the mail
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->mssg      = newBuffer(1);

  // read in all of our values
  mail->sender = strdup(read_string(set, "sender"));
  mail->time   = strdup(read_string(set, "time"));
```

```
  bufferCat(mail->mssg, read_string(set, "mssg"));
  return mail;
}

// represent a piece of mail as a storage set
STORAGE_SET *mailStore(MAIL_DATA *mail) {
  // create a new storage set
  STORAGE_SET *set = new_storage_set();
  store_string(set, "sender", mail->sender);
  store_string(set, "time",   mail->time);
  store_string(set, "mssg",   bufferString(mail->mssg));
  return set;
}
```

Now that we have the ability to store and read mail, let's create two functions for actually doing the storing and reading:

```
// saves all of our unreceived mail to disk
void save_mail(void) {
  // make a storage set to hold all our mail
  STORAGE_SET *set = new_storage_set();

  // make a list of name:mail pairs, and store it in the set
  STORAGE_SET_LIST *list = new_storage_list();

  // iterate across all of the people who have not received mail, and
  // store their names in the storage list, along with their mail
  HASH_ITERATOR *mail_i = newHashIterator(mail_table);
  const char      *name = NULL;
  LIST            *mail = NULL;
  ITERATE_HASH(name, mail, mail_i) {
    // create a new storage set that holds each name:mail pair,
    // and add it to our list of all name:mail pairs
    STORAGE_SET *one_pair = new_storage_set();
    store_string    (one_pair, "name", name);
    store_list      (one_pair, "mail", gen_store_list(mail, mailStore));
    storage_list_put(list, one_pair);
  } deleteHashIterator(mail_i);
```

```
  // make sure we add the list of name:mail pairs we want to save
  store_list(set, "list", list);

  // now, store our set in the mail file, and clean up our mess
  storage_write(set, MAIL_FILE);
  storage_close(set);
}


// loads all of our unreceived mail from disk
void load_mail(void) {
  // parse our storage set
  STORAGE_SET *set = storage_read(MAIL_FILE);

  // make sure the file existed and wasn't empty
  if(set == NULL) return;

  // get the list of all name:mail pairs, and parse each one
  STORAGE_SET_LIST *list = read_list(set, "list");
  STORAGE_SET  *one_pair = NULL;
  while( (one_pair = storage_list_next(list)) != NULL) {
    const char *name = read_string(one_pair, "name");
    LIST       *mail = gen_read_list(read_list(one_pair, "mail"), mailRead);
    hashPut(mail_table, name, mail);
  }

  // Everything is parsed! Now it's time to clean up our mess
  storage_close(set);
}
```

This code may be a bit ugly to the untrained eye. However, there are some very useful nuggets of knowledge buried within it. If you are having troubles understanding what is going on, it is highly suggested that you take a few minutes to trace through these two functions and figure out what is going on. Once we are completely done this section, it may also help to write a couple mails to yourself and examine what the mail file looks like. The file's structure might help elucidate many of the things that are going on in these two functions.

Ok, we are on the home stretch. Now that our save and load functions are written,

we have to make sure they are called appropriately. We will want to load up unread mail when the mail module initialized, and we will want to make sure we update the contents of the mail file whenever mail is sent or received. Why don't we add those bits of code.

At the end of cmd_mail, make sure mail is saved:

```
    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);

    // save all unread mail
    save_mail();
```

At the end of cmd_receive, make sure mail is saved:

```
    // let the character know how much mail he received
    send_to_char(ch, "You receive %d letter%s.\r\n",
                 listSize(mail_list), (listSize(mail_list) == 1 ? "" : "s"));

    // update the unread mail in our mail file
    save_mail();
```

Finally, ensure we load up all unread mail when we initialize the module:

```
// boot up the mail module
void init_mail(void) {
  // initialize our mail table
  mail_table = newHashtable();

  // parse any unread mail
  load_mail();
```

That's it! Unreceived mail will now be persistent across reboots and crashes. You may have noticed that saving of mail is rather inefficient; every time someone receives a mail or sends a mail, we have to re-save all unreceived mails. Ideally, we would like to change it so we have to re-save as little information as possible. That is the problem we will tackle in the section on auxiliary data.

## 3.3 Summary

In this section, we have learned the basics of storage sets. Storage sets can be a bit tricky to understand at first. If you are having troubles completely understanding the code presented in this section, it is strongly suggested you spend some time reading over it and understanding how it works. It may also help looking at the files that storage sets generate after one is written to disk. If you plan on adding anything new to NakedMud, it is almost mandatory that you understand how storage sets work. However, Once you figure out how they work, you will be glad you did. They are very helpful data structures.

# 4 Auxiliary Data

Auxiliary Data is, perhaps, the most important part of NakedMud's design. Auxiliary Data allows you to add new variables to the various datatypes NakedMud handles within the game (objects, rooms, mobiles, accounts, sockets) without even touching the files that house those data structures. The biggest gain from this 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 created - can stay in one module. As was mentioned in the introduction the modules section, this will undoubtedly help with your ability to debug, maintain, and distribute pieces of your code in the future. Designing new auxiliary data is very simple, but it does require a bit of effort if you have not done it in the past. This tutorial will walk you through the steps of writing and installing new auxiliary data.

## 4.1 Getting Up To Speed

If you have already completed the tutorial on modules, and that is the only tutorial in this guide you have performed, you can skip this subsection, as it does not apply to you.

If you have completed the tutorial on storage sets, you will have to make some modifications to your code. As we discussed at the end of the storage sets section, the way saving is performed is rather inefficient, and we will attempt to address that problem within this section. *You will need to replace your mail.c file with the mail.c found in appendix A.*

If you have not yet completed any of the tutorials in this guide, you will need to do a couple things before you can go through the auxiliary data tutorial. All of the steps you must take are outlined in section 3.1 - the *Getting Up To Speed* section for storage sets.

Once you have figured out which set of changes apply to you, carry on with the next subsection in the auxiliary data tutorial.

## 4.2 Auxiliary Data Basics

In the section on modules, we designed a 'proof of concept' for a mail system. One feature we did not add was the ability for unreceived mail to be persistent. If

the MUD ever crashed or rebooted, all mail not yet received would be lost. This inconvenience was addressed in the section on storage sets. However, we ran into another problem with the saving procedure being inefficient: to save any change to someone's unreceived mail status, we effectively had to re-save everyone's unreceived mail. This section will address the problem by attaching a character's unread mail to that character's actual data structure. Now, whenever we want to the unread mail for a character, we will only have to save that character, and not all unread mail in existence.

The first thing we will need to do is add the headers for interacting with auxiliary data and storage sets. Let's do this where we include all the other headers we need for interacting with core features of the MUD. We'll add the new headers right after we add the *handler.h* header:

```
#include "../handler.h"       // for giving mail to characters
#include "../storage.h"       // for saving/loading auxiliary data
#include "../auxiliary.h"     // for creating new auxiliary data
#include "../world.h"         // for loading offline chars receiving mail
```

Because we will save unread mail as auxiliary data within character data, we will no longer need a hashtable for keeping everything stored. Therefore, let us search and destroy all references to the mail_table:

Right after we finish including all of our headers, delete:

```
// maps charName to a list of mail they have received
HASHTABLE *mail_table = NULL;
```

In cmd_mail, delete the entire section within the last else statement:

```
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // see if the receiver already has a mail list
    LIST *mssgs = hashGet(mail_table, arg);

    // if he doesn't, create one and add it to the hashtable
    if(mssgs == NULL) {
      mssgs = newList();
      hashPut(mail_table, arg, mssgs);
    }

    // add the new mail to our mail list
```

```
    listPut(mssgs, mail);

    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);
```

At the very start of cmd_receive, remove the reference to hashRemove, and for the time being, set mail_list to NULL:

```
  // Remove the character's mail list from our mail table
  LIST *mail_list = hashRemove(mail_table, charGetName(ch));
```

Finally, remove our creation of the mail_table in init_mail:

```
  // initialize our mail table
  mail_table = newHashtable();
```

Before we start writing our auxiliary data, we are going to have to provide a couple functions for handling the saving, reading, and copying of mail. Right after newMail and deleteMail, add these 3 new functions:

```
// parse a piece of mail from a storage set
MAIL_DATA *mailRead(STORAGE_SET *set) {
  // allocate some memory for the mail
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->mssg      = newBuffer(1);

  // read in all of our values
  mail->sender = strdup(read_string(set, "sender"));
  mail->time   = strdup(read_string(set, "time"));
  bufferCat(mail->mssg, read_string(set, "mssg"));
  return mail;
}

// represent a piece of mail as a storage set
STORAGE_SET *mailStore(MAIL_DATA *mail) {
  // create a new storage set
  STORAGE_SET *set = new_storage_set();
  store_string(set, "sender", mail->sender);
  store_string(set, "time",   mail->time);
  store_string(set, "mssg",   bufferString(mail->mssg));
  return set;
```

```
}

// copy a piece of mail. This will be needed by our auxiliary
// data copy functions
MAIL_DATA *mailCopy(MAIL_DATA *mail) {
  MAIL_DATA *newmail = malloc(sizeof(MAIL_DATA));
  newmail->sender = strdup(mail->sender);
  newmail->time   = strdup(mail->time);
  newmail->mssg   = bufferCopy(mail->mssg);
  return newmail;
}
```

Now we're ready to start writing our auxiliary data. Auxiliary Data require 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 auxiliary data, a function that reads the auxiliary data from a storage set, and a function that writes the auxiliary data to a storage set. Below, we lay out all of those things. There is quite a bit of code but, as you will notice, it is all very simple (perhaps with the exception of the read and write functions). Right below the the mailCopy function, add the following bit of code:

```
// our mail auxiliary data.
// Holds a list of all the unreceived mail a person has
typedef struct {
  LIST *mail; // our list of unread mail
} MAIL_AUX_DATA;

// create a new instance of mail aux data, for us to put onto a character
MAIL_AUX_DATA *newMailAuxData(void) {
  MAIL_AUX_DATA *data = malloc(sizeof(MAIL_AUX_DATA));
  data->mail = newList();
  return data;
}

// delete a character's mail aux data
void deleteMailAuxData(MAIL_AUX_DATA *data) {
  if(data->mail) deleteListWith(data->mail, deleteMail);
  free(data);
```

```
}

// copy one mail aux data to another
void mailAuxDataCopyTo(MAIL_AUX_DATA *from, MAIL_AUX_DATA *to) {
  if(to->mail)   deleteListWith(to->mail, deleteMail);
  if(from->mail) to->mail = listCopyWith(from->mail, mailCopy);
  else           to->mail = newList();
}

// return a copy of a mail aux data
MAIL_AUX_DATA *mailAuxDataCopy(MAIL_AUX_DATA *data) {
  MAIL_AUX_DATA *newdata = newMailAuxData();
  mailAuxDataCopyTo(data, newdata);
  return newdata;
}

// parse a mail aux data from a storage set
MAIL_AUX_DATA *mailAuxDataRead(STORAGE_SET *set) {
  MAIL_AUX_DATA *data = malloc(sizeof(MAIL_AUX_DATA));
  data->mail          = gen_read_list(read_list(set, "mail"), mailRead);
  return data;
}

// represent a mail aux data as a storage set
STORAGE_SET *mailAuxDataStore(MAIL_AUX_DATA *data) {
  STORAGE_SET *set = new_storage_set();
  store_list(set, "mail", gen_store_list(data->mail, mailStore));
  return set;
}
```

Finally, we will want to ensure this new auxiliary data exists on characters. When our mail module boots up, we will want to install this new bit of auxiliary data:

```
// boot up the mail module
void init_mail(void) {
  // install our auxiliary data
  auxiliariesInstall("mail_aux_data",
     newAuxiliaryFuncs(AUXILIARY_TYPE_CHAR,
                               newMailAuxData, deleteMailAuxData,
```

```
                                mailAuxDataCopyTo, mailAuxDataCopy,
                                mailAuxDataStore, mailAuxDataRead));

  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
  add_cmd("receive", NULL, cmd_receive, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
}
```

Our mail auxiliary data is installed and completely functional! Now, we will want to go back to cmd_mail and cmd_receive to ensure they use the auxiliary data in replacement of the hashtable we used before. Let's start with the new code for cmd_mail. In the last else block, add the following bit of code:

```
    // create the new piece of mail
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // get a copy of the player, send mail, and save
    CHAR_DATA *recv = get_player(arg);
    send_to_char(recv, "You have new mail.\r\n");

    // let's pull out the character's mail aux data, and add the new piece
    MAIL_AUX_DATA *maux = charGetAuxiliaryData(recv, "mail_aux_data");
    listPut(maux->mail, mail);
    save_player(recv);

    // get rid of our reference, and extract from game if need be
    unreference_player(recv);

    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);
```

Now, we will want to make it so players can receive their unread mail. At the very start of cmd_receive, add the following bit of code:

```
COMMAND(cmd_receive) {
  // Remove the character's mail list from our mail table
  MAIL_AUX_DATA *maux = charGetAuxiliaryData(ch, "mail_aux_data");
  LIST *mail_list     = maux->mail;
```

```
// replace our old list with a new one. Our old one will be deleted soon
maux->mail = newList();
```

There, we're done! Players can now send and receive mail.

## 4.3 Summary

In this section, we have learned the basics of using auxiliary data. In our tutorial on storage sets, we were able to make unreceived mail persistent, but loading and saving was very inefficient. In this tutorial, we addressed the efficiency problem by saving unread mail on the character the mail belongs to. Auxiliary Data does require a bit of programming, but once you get the hang of it, you will realize it is very routine programming - the bulk of which you can probably copy from a template like the one provided in this tutorial or any other module that employs the use of auxiliary data.

# 5 Conclusion

Throughout this manual, we have learned the basics of modules, storage sets, and auxiliary data - three fundamental aspects of NakedMud. With a bit more practical experience, you will know all there is to know about these three topics. Hopefully this tutorial has helped you understand enough of the basics so that you will not have too much trouble extending NakedMud with new game content.

# A Mail Module Code, First Draft

```
##############################################################################
# module.mk
##############################################################################

# include all of the source files contained in this module
SRC += mail/mail.c
```

```
//****************************************************************************
// mail.h
//****************************************************************************
#ifndef MAIL_H
#define MAIL_H
// this function should be called when the MUD first boots up.
// calling it will initialize the mail module for use.
void init_mail(void);
#endif // MAIL_H
```

```
//****************************************************************************
// mail.c
//****************************************************************************

// include all the header files we will need from the MUD core
#include "../mud.h"
#include "../utils.h"         // for get_time()
#include "../character.h"     // for handling characters sending mail
#include "../save.h"          // for char_exists()
#include "../object.h"        // for creating mail objects
#include "../handler.h"       // for giving mail to characters

// include headers from other modules that we require
#include "../editor/editor.h" // for access to sockets' notepads
```

```
// include the headers for this module
#include "mail.h"

// maps charName to a list of mail they have received
HASHTABLE *mail_table = NULL;

typedef struct {
  char *sender;       // name of the char who sent this mail
  char   *time;       // the time it was sent at
  BUFFER *mssg;       // the accompanying message
} MAIL_DATA;

// create a new piece of mail.
MAIL_DATA *newMail(CHAR_DATA *sender, const char *mssg) {
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->sender = strdup(charGetName(sender));
  mail->time   = strdup(get_time());
  mail->mssg   = newBuffer(strlen(mssg));
  bufferCat(mail->mssg, mssg);
  return mail;
}

// free all of the memory that was allocated to make this piece of mail
void deleteMail(MAIL_DATA *mail) {
  if(mail->sender) free(mail->sender);
  if(mail->time)   free(mail->time);
  if(mail->mssg)   deleteBuffer(mail->mssg);
  free(mail);
}


// mails a message to the specified person. This command must take the
// following form:
//   mail <person>
//
// The contents of the character's notepad will be used as the body of the
// message. The character's notepad must not be empty.
COMMAND(cmd_mail) {
```

```
  // make sure the character exists
  if(!char_exists(arg))
    send_to_char(ch, "Noone named %s is registered on %s.\r\n",
                 arg, "<insert mud name here>");
  // make sure we have a socket - we'll need access to its notepad
  else if(!charGetSocket(ch))
    send_to_char(ch, "Only characters with sockets can send mail!\r\n");
  // make sure our notepad is not empty
  else if(!*bufferString(socketGetNotepad(charGetSocket(ch))))
    send_to_char(ch, "Your notepad is empty. "
                 "First, try writing something with {cwrite{n.\r\n");
  // the character exists. Let's parse the items and send the mail
  else {
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // see if the receiver already has a mail list
    LIST *mssgs = hashGet(mail_table, arg);

    // if he doesn't, create one and add it to the hashtable
    if(mssgs == NULL) {
      mssgs = newList();
      hashPut(mail_table, arg, mssgs);
    }

    // add the new mail to our mail list
    listPut(mssgs, mail);

    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);
  }
}


// checks to see if the character has any mail. If he does, convert each piece
// of mail into an object, and transfer them all into the character's inventory.
COMMAND(cmd_receive) {
  // Remove the character's mail list from our mail table
  LIST *mail_list = hashRemove(mail_table, charGetName(ch));
```

```
  // make sure the list exists
  if(mail_list == NULL || listSize(mail_list) == 0)
    send_to_char(ch, "You have no new mail.\r\n");
  // hand over all of the mail
  else {
    // go through each piece of mail, make an object for it,
    // and transfer the new object to us
    LIST_ITERATOR *mail_i = newListIterator(mail_list);
    MAIL_DATA       *mail = NULL;
    ITERATE_LIST(mail, mail_i) {
      OBJ_DATA *obj  = newObj();
      objSetName      (obj, "a letter");
      objSetKeywords  (obj, "letter, mail");
      objSetRdesc     (obj, "A letter is here.");
      objSetMultiName (obj, "A stack of %d letters");
      objSetMultiRdesc(obj, "A stack of %d letters are here.");
      bprintf(objGetDescBuffer(obj),
             "Sender   : %s\r\n"
             "Date sent: %s\r\n"
             "%s", mail->sender, mail->time, bufferString(mail->mssg));

      // give the object to the character
      obj_to_game(obj);
      obj_to_char(obj, ch);
    } deleteListIterator(mail_i);

    // let the character know how much mail he received
    send_to_char(ch, "You receive %d letter%s.\r\n",
                 listSize(mail_list), (listSize(mail_list) == 1 ? "" : "s"));
  }

  // delete the mail list, and all of its contents
  if(mail_list != NULL) deleteListWith(mail_list, deleteMail);
}


// boot up the mail module
```

```
void init_mail(void) {
  // initialize our mail table
  mail_table = newHashtable();

  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
  add_cmd("receive", NULL, cmd_receive, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
}
```

# B Mail Module Code, Second Draft

```
##############################################################################
# module.mk
##############################################################################

# include all of the source files contained in this module
SRC += mail/mail.c
```

```
//****************************************************************************
// mail.h
//****************************************************************************
#ifndef MAIL_H
#define MAIL_H
// this function should be called when the MUD first boots up.
// calling it will initialize the mail module for use.
void init_mail(void);
#endif // MAIL_H
```

```
//****************************************************************************
// mail.c
//****************************************************************************

// include all the header files we will need from the MUD core
#include "../mud.h"
#include "../utils.h"         // for get_time()
#include "../character.h"     // for handling characters sending mail
#include "../save.h"          // for char_exists()
#include "../object.h"        // for creating mail objects
#include "../handler.h"       // for giving mail to characters
#include "../storage.h"       // for saving/loading mail

// include headers from other modules that we require
#include "../editor/editor.h" // for access to sockets' notepads
```

```
// include the headers for this module
#include "mail.h"



// this is the file we will save all unreceived mail in, when the mud is down
#define MAIL_FILE "../lib/misc/mail"

// maps charName to a list of mail they have received
HASHTABLE *mail_table = NULL;

typedef struct {
  char *sender;       // name of the char who sent this mail
  char   *time;       // the time it was sent at
  BUFFER *mssg;       // the accompanying message
} MAIL_DATA;

// create a new piece of mail.
MAIL_DATA *newMail(CHAR_DATA *sender, const char *mssg) {
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->sender = strdup(charGetName(sender));
  mail->time   = strdup(get_time());
  mail->mssg   = newBuffer(strlen(mssg));
  bufferCat(mail->mssg, mssg);
  return mail;
}

// free all of the memory that was allocated to make this piece of mail
void deleteMail(MAIL_DATA *mail) {
  if(mail->sender) free(mail->sender);
  if(mail->time)   free(mail->time);
  if(mail->mssg)   deleteBuffer(mail->mssg);
  free(mail);
}

// parse a piece of mail from a storage set
MAIL_DATA *mailRead(STORAGE_SET *set) {
```

```
  // allocate some memory for the mail
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->mssg      = newBuffer(1);

  // read in all of our values
  mail->sender = strdup(read_string(set, "sender"));
  mail->time   = strdup(read_string(set, "time"));
  bufferCat(mail->mssg, read_string(set, "mssg"));
  return mail;
}

// represent a piece of mail as a storage set
STORAGE_SET *mailStore(MAIL_DATA *mail) {
  // create a new storage set
  STORAGE_SET *set = new_storage_set();
  store_string(set, "sender", mail->sender);
  store_string(set, "time",   mail->time);
  store_string(set, "mssg",   bufferString(mail->mssg));
  return set;
}

// saves all of our unreceived mail to disk
void save_mail(void) {
  // make a storage set to hold all our mail
  STORAGE_SET *set = new_storage_set();

  // make a list of name:mail pairs, and store it in the set
  STORAGE_SET_LIST *list = new_storage_list();

  // iterate across all of the people who have not received mail, and
  // store their names in the storage list, along with their mail
  HASH_ITERATOR *mail_i = newHashIterator(mail_table);
  const char      *name = NULL;
  LIST            *mail = NULL;
  ITERATE_HASH(name, mail, mail_i) {
    // create a new storage set that holds each name:mail pair,
    // and add it to our list of all name:mail pairs
    STORAGE_SET *one_pair = new_storage_set();
```

```
    store_string    (one_pair, "name", name);
    store_list      (one_pair, "mail", gen_store_list(mail, mailStore));
    storage_list_put(list, one_pair);
  } deleteHashIterator(mail_i);

  // make sure we add the list of name:mail pairs we want to save
  store_list(set, "list", list);

  // now, store our set in the mail file, and clean up our mess
  storage_write(set, MAIL_FILE);
  storage_close(set);
}

// loads all of our unreceived mail from disk
void load_mail(void) {
  // parse our storage set
  STORAGE_SET *set = storage_read(MAIL_FILE);

  // make sure the file existed and wasn't empty
  if(set == NULL) return;

  // get the list of all name:mail pairs, and parse each one
  STORAGE_SET_LIST *list = read_list(set, "list");
  STORAGE_SET  *one_pair = NULL;
  while( (one_pair = storage_list_next(list)) != NULL) {
    const char *name = read_string(one_pair, "name");
    LIST       *mail = gen_read_list(read_list(one_pair, "mail"), mailRead);
    hashPut(mail_table, name, mail);
  }

  // Everything is parsed! Now it's time to clean up our mess
  storage_close(set);
}


// mails a message to the specified person. This command must take the
// following form:
//   mail <person>
```

```
//
// The contents of the character's notepad will be used as the body of the
// message. The character's notepad must not be empty.
COMMAND(cmd_mail) {
  // make sure the character exists
  if(!char_exists(arg))
    send_to_char(ch, "Noone named %s is registered on %s.\r\n",
                 arg, "<insert mud name here>");
  // make sure we have a socket - we'll need access to its notepad
  else if(!charGetSocket(ch))
    send_to_char(ch, "Only characters with sockets can send mail!\r\n");
  // make sure our notepad is not empty
  else if(!*bufferString(socketGetNotepad(charGetSocket(ch))))
    send_to_char(ch, "Your notepad is empty. "
                 "First, try writing something with {cwrite{n.\r\n");
  // the character exists. Let's parse the items and send the mail
  else {
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // see if the receiver already has a mail list
    LIST *mssgs = hashGet(mail_table, arg);

    // if he doesn't, create one and add it to the hashtable
    if(mssgs == NULL) {
      mssgs = newList();
      hashPut(mail_table, arg, mssgs);
    }

    // add the new mail to our mail list
    listPut(mssgs, mail);

    // let the character know we've sent the mail
    send_to_char(ch, "You send a message to %s.\r\n", arg);

    // save all unread mail
    save_mail();
  }
}
```

```
// checks to see if the character has any mail. If he does, convert each piece
// of mail into an object, and transfer them all into the character's inventory.
COMMAND(cmd_receive) {
  // Remove the character's mail list from our mail table
  LIST *mail_list = hashRemove(mail_table, charGetName(ch));

  // make sure the list exists
  if(mail_list == NULL || listSize(mail_list) == 0)
    send_to_char(ch, "You have no new mail.\r\n");
  // hand over all of the mail
  else {
    // go through each piece of mail, make an object for it,
    // and transfer the new object to us
    LIST_ITERATOR *mail_i = newListIterator(mail_list);
    MAIL_DATA       *mail = NULL;
    ITERATE_LIST(mail, mail_i) {
      OBJ_DATA *obj  = newObj();
      objSetName      (obj, "a letter");
      objSetKeywords  (obj, "letter, mail");
      objSetRdesc     (obj, "A letter is here.");
      objSetMultiName (obj, "A stack of %d letters");
      objSetMultiRdesc(obj, "A stack of %d letters are here.");
      bprintf(objGetDescBuffer(obj),
             "Sender   : %s\r\n"
             "Date sent: %s\r\n"
             "%s", mail->sender, mail->time, bufferString(mail->mssg));

      // give the object to the character
      obj_to_game(obj);
      obj_to_char(obj, ch);
    } deleteListIterator(mail_i);

    // let the character know how much mail he received
    send_to_char(ch, "You receive %d letter%s.\r\n",
                 listSize(mail_list), (listSize(mail_list) == 1 ? "" : "s"));
```

```
    // update the unread mail in our mail file
    save_mail();
  }

  // delete the mail list, and all of its contents
  if(mail_list != NULL) deleteListWith(mail_list, deleteMail);
}


// boot up the mail module
void init_mail(void) {
  // initialize our mail table
  mail_table = newHashtable();

  // parse any unread mail
  load_mail();

  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
  add_cmd("receive", NULL, cmd_receive, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
}
```

# C Mail Module Code, Third Draft

```
##############################################################################
# module.mk
##############################################################################

# include all of the source files contained in this module
SRC += mail/mail.c
```

```
//****************************************************************************
// mail.h
//****************************************************************************
#ifndef MAIL_H
#define MAIL_H
// this function should be called when the MUD first boots up.
// calling it will initialize the mail module for use.
void init_mail(void);
#endif // MAIL_H
```

```
//****************************************************************************
// mail.c
//****************************************************************************

// include all the header files we will need from the MUD core
#include "../mud.h"
#include "../utils.h"         // for get_time()
#include "../character.h"     // for handling characters sending mail
#include "../save.h"          // for char_exists()
#include "../object.h"        // for creating mail objects
#include "../handler.h"       // for giving mail to characters
#include "../storage.h"       // for saving/loading auxiliary data
#include "../auxiliary.h"     // for creating new auxiliary data
#include "../world.h"         // for loading offline chars receiving mail
```

```
// include headers from other modules that we require
#include "../editor/editor.h" // for access to sockets' notepads

// include the headers for this module
#include "mail.h"

// maps charName to a list of mail they have received
HASHTABLE *mail_table = NULL;

typedef struct {
  char *sender;       // name of the char who sent this mail
  char   *time;       // the time it was sent at
  BUFFER *mssg;       // the accompanying message
} MAIL_DATA;

// create a new piece of mail.
MAIL_DATA *newMail(CHAR_DATA *sender, const char *mssg) {
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->sender = strdup(charGetName(sender));
  mail->time   = strdup(get_time());
  mail->mssg   = newBuffer(strlen(mssg));
  bufferCat(mail->mssg, mssg);
  return mail;
}

// free all of the memory that was allocated to make this piece of mail
void deleteMail(MAIL_DATA *mail) {
  if(mail->sender) free(mail->sender);
  if(mail->time)   free(mail->time);
  if(mail->mssg)   deleteBuffer(mail->mssg);
  free(mail);
}

// parse a piece of mail from a storage set
MAIL_DATA *mailRead(STORAGE_SET *set) {
  // allocate some memory for the mail
  MAIL_DATA *mail = malloc(sizeof(MAIL_DATA));
  mail->mssg      = newBuffer(1);
```

```
  // read in all of our values
  mail->sender = strdup(read_string(set, "sender"));
  mail->time   = strdup(read_string(set, "time"));
  bufferCat(mail->mssg, read_string(set, "mssg"));
  return mail;
}

// represent a piece of mail as a storage set
STORAGE_SET *mailStore(MAIL_DATA *mail) {
  // create a new storage set
  STORAGE_SET *set = new_storage_set();
  store_string(set, "sender", mail->sender);
  store_string(set, "time",   mail->time);
  store_string(set, "mssg",   bufferString(mail->mssg));
  return set;
}

// copy a piece of mail. This will be needed by our auxiliary
// data copy functions
MAIL_DATA *mailCopy(MAIL_DATA *mail) {
  MAIL_DATA *newmail = malloc(sizeof(MAIL_DATA));
  newmail->sender = strdup(mail->sender);
  newmail->time   = strdup(mail->time);
  newmail->mssg   = bufferCopy(mail->mssg);
  return newmail;
}


// our mail auxiliary data.
// Holds a list of all the unreceived mail a person has
typedef struct {
  LIST *mail; // our list of unread mail
} MAIL_AUX_DATA;

// create a new instance of mail aux data, for us to put onto a character
MAIL_AUX_DATA *newMailAuxData(void) {
  MAIL_AUX_DATA *data = malloc(sizeof(MAIL_AUX_DATA));
```

```
  data->mail = newList();
  return data;
}

// delete a character's mail aux data
void deleteMailAuxData(MAIL_AUX_DATA *data) {
  if(data->mail) deleteListWith(data->mail, deleteMail);
  free(data);
}

// copy one mail aux data to another
void mailAuxDataCopyTo(MAIL_AUX_DATA *from, MAIL_AUX_DATA *to) {
  if(to->mail)   deleteListWith(to->mail, deleteMail);
  if(from->mail) to->mail = listCopyWith(from->mail, mailCopy);
  else           to->mail = newList();
}

// return a copy of a mail aux data
MAIL_AUX_DATA *mailAuxDataCopy(MAIL_AUX_DATA *data) {
  MAIL_AUX_DATA *newdata = newMailAuxData();
  mailAuxDataCopyTo(data, newdata);
  return newdata;
}

// parse a mail aux data from a storage set
MAIL_AUX_DATA *mailAuxDataRead(STORAGE_SET *set) {
  MAIL_AUX_DATA *data = malloc(sizeof(MAIL_AUX_DATA));
  data->mail          = gen_read_list(read_list(set, "mail"), mailRead);
  return data;
}

// represent a mail aux data as a storage set
STORAGE_SET *mailAuxDataStore(MAIL_AUX_DATA *data) {
  STORAGE_SET *set = new_storage_set();
  store_list(set, "mail", gen_store_list(data->mail, mailStore));
  return set;
}
```

```
// mails a message to the specified person. This command must take the
// following form:
//   mail <person>
//
// The contents of the character's notepad will be used as the body of the
// message. The character's notepad must not be empty.
COMMAND(cmd_mail) {
  // make sure the character exists
  if(!char_exists(arg))
    send_to_char(ch, "Noone named %s is registered on %s.\r\n",
                 arg, "<insert mud name here>");
  // make sure we have a socket - we'll need access to its notepad
  else if(!charGetSocket(ch))
    send_to_char(ch, "Only characters with sockets can send mail!\r\n");
  // make sure our notepad is not empty
  else if(!*bufferString(socketGetNotepad(charGetSocket(ch))))
    send_to_char(ch, "Your notepad is empty. "
                 "First, try writing something with {cwrite{n.\r\n");
  // the character exists. Let's parse the items and send the mail
  else {
    // create the new piece of mail
    MAIL_DATA *mail = newMail(ch, bufferString(socketGetNotepad(charGetSocket(ch))));

    // get a copy of the player, send mail, and save
    CHAR_DATA *recv = get_player(arg);
    send_to_char(recv, "You have new mail.\r\n");

    // let's pull out the character's mail aux data, and add the new piece
    MAIL_AUX_DATA *maux = charGetAuxiliaryData(recv, "mail_aux_data");
    listPut(maux->mail, mail);
    save_player(recv);

    // get rid of our reference, and extract from game if need be
    unreference_player(recv);

    // let the character know we've sent the mail
```

```
    send_to_char(ch, "You send a message to %s.\r\n", arg);
  }
}


// checks to see if the character has any mail. If he does, convert each piece
// of mail into an object, and transfer them all into the character's inventory.
COMMAND(cmd_receive) {
  // Remove the character's mail list from our mail table
  MAIL_AUX_DATA *maux = charGetAuxiliaryData(ch, "mail_aux_data");
  LIST *mail_list     = maux->mail;
  // replace our old list with a new one. Our old one will be deleted soon
  maux->mail = newList();

  // make sure the list exists
  if(mail_list == NULL || listSize(mail_list) == 0)
    send_to_char(ch, "You have no new mail.\r\n");
  // hand over all of the mail
  else {
    // go through each piece of mail, make an object for it,
    // and transfer the new object to us
    LIST_ITERATOR *mail_i = newListIterator(mail_list);
    MAIL_DATA       *mail = NULL;
    ITERATE_LIST(mail, mail_i) {
      OBJ_DATA *obj  = newObj();
      BUFFER   *desc = newBuffer(1);
      objSetName      (obj, "a letter");
      objSetKeywords  (obj, "letter, mail");
      objSetRdesc     (obj, "A letter is here.");
      objSetMultiName (obj, "A stack of %d letters");
      objSetMultiRdesc(obj, "A stack of %d letters are here.");

      // make our description
      bprintf(desc,
             "Sender   : %s\r\n"
             "Date sent: %s\r\n"
             "%s", mail->sender, mail->time, bufferString(mail->mssg));
      objSetDesc(obj, bufferString(desc));
```

```
      // clean up our mess and give the object to the character
      deleteBuffer(desc);
      obj_to_game(obj);
      obj_to_char(obj, ch);
    } deleteListIterator(mail_i);

    // let the character know how much mail he received
    send_to_char(ch, "You receive %d letter%s.\r\n",
                 listSize(mail_list), (listSize(mail_list) == 1 ? "" : "s"));
  }

  // delete the mail list, and all of its contents
  if(mail_list != NULL) deleteListWith(mail_list, deleteMail);
}


// boot up the mail module
void init_mail(void) {
  // install our auxiliary data
  auxiliariesInstall("mail_aux_data",
     newAuxiliaryFuncs(AUXILIARY_TYPE_CHAR,
       newMailAuxData, deleteMailAuxData,
       mailAuxDataCopyTo, mailAuxDataCopy,
       mailAuxDataStore, mailAuxDataRead));

  // add all of the commands that come with this module
  add_cmd("mail", NULL, cmd_mail, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
  add_cmd("receive", NULL, cmd_receive, POS_STANDING, POS_FLYING,
          "player", FALSE, TRUE);
}
```