This guideline is an attempt to give a better understanding of how the spells in CircleMUD work and a step-by-step method on how to create them. This is based on bpl15, with some reference to previous patchlevels or codebases.  
 **[Part 1:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 1) Categories of Spells  
[Part 2:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 2) Basic Steps to Create A Spell  
[Part 3:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 3) Damage Spells - Killing Enemy 101  
[Part 4:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 4) Area Spells - Make Thy Enemies Suffer!  
[Part 5:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 5) Affect Spells - Affect Them for Better or Worse…  
[Part 6:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 6) Unaffect Spells - "First You Have It, Now You Don't!"  
[Part 7:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 7) Alter Object Spells - Bend It to Your Will.  
[Part 8:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 8) Points Spells - Measuring the Beat of Our Life…  
[Part 9:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 9) Group-based Spells - Defend Thy Neighbors!  
[Part 10:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part10) Masses Spells - Let Them See Your Way…  
[Part 11:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part11) Summon Spells - "Come and Follow Me!"  
[Part 12:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part12) Creation Spells - Magical Sleight of Hand.  
[Part 13:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part13) Manual Spells - Making It Special…  
[Part 14:](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part14) Giving NPCs Immunity.**

A lot of people have benefited from the contributions made by the CircleMUD community. I, for one, have gained a lot and would like to contribute back. This, err… rather long guideline was supposed to be short at first. That is, until my housemates who also mud asked a simple question: "How to know what spell is under which category?" So, one thing led to another, after a while it became this long in my attempt to make it as comprehensive as possible, especially for those who just started to code.

Anyway, I hope someone will benefit from my first attempt…. :)

- Fizal ([amustapa@hotmail.com](mailto:amustapa@hotmail.com))

**Part 1: Categories of Spells**

All the spells in Circle, whether it's a magical or clerical spell, or the more exotic druidic or elemental spell, they can be categorized into the following types:

**Type 1:** Damage spells

* The spells that fall under this category are offensive spells that will inflict damage to the victim.
* Example: Fireball and Magic Missile spells.
* Identified by: MAG\_DAMAGE
* Target: A character in the room.

**Type 2:** Area spells

* Spells under this category will affect a wide area or a room. However, the only currently implemented area spell is an offensive (damaging) spell.
* Example: Earthquake spell.
* Identified by: MAG\_AREAS
* Target: No target is needed, but will affect characters in the room at casting time. However, the caster, Immortals, other players if no-pkill is set, and charmed mobs are excluded.

**Type 3:** Affect spells

* Affect spells are spells that will attach affect bits or flags on the victim, apply a certain effect on the victim, or both. Affect flags are used if you need to check whether a character has a certain flag attached to him elsewhere in the codes.
* STRUCT.H has a list of currently implemented affect flags and applications. Affect flags starts with AFF\_xxx (xxx is the name of the spell or skill that utilizes it) and affect application starts with APPLY\_yyy (yyy is the victim's characteristic that is changed).
* Example: Strength spell (apply an effect), Detect Magic spell (attach an affect flag) and Poison spell (does both).
* Identified by: MAG\_AFFECTS
* Target: A character in the room.

**Type 4:** Unaffect spells

* While affect spells set affect flags, unaffect spells will remove that flag and the spell that is associated with it.
* Example: Remove Curse and Remove Poison spells (when cast on a character)
* Identified by: MAG\_UNAFFECTS
* Target: A character in the room.

**Type 5:** Alter Object spells

* If affect spells are for characters, spells under this category will set or remove object flags on objects. Object flags can be used as a form of a check to allow a character or stop him from manipulating the object.
* STRUCT.H has a list of currently implemented object flags, which start with ITEM\_xxx (xxx is the characteristic of the object that is altered), under Extra Object Flags.
* Example: Curse and Remove Curse spells (when cast on an object)
* Identified by: MAG\_ALTER\_OBJ
* Target: An object in the room, in caster's inventory or equipped by caster.

**Type 6:** Points spells

* Points spells are those that changes the victim's hit, mana and movement points. So far, spells under this category are the curative type which increase (or cure) those points.
* Example: Heal and Cure Light Wounds spells.
* Identified by: MAG\_POINTS
* Target: A character in the room.

**Type 7:** Group-based spells

* These spells can only be cast if the caster is in a group and only affect the caster's group members who are in the same room at the time. These spells are usually defensive and benign. No one wants to kill their own group members :P
* Example: Group Armor and Group Heal spells.
* Identified by: MAG\_GROUPS
* Target: No target needed, but will affect characters who are grouped and in the same room as the caster.

**Type 8:** Masses spells

* Masses spells will affect the mass in the room. In another words, everyone in the room will be affected by these spells, except for the caster. It is slightly similar to area spells. Currently, no spell has been implemented for this type
* Example: -none-
* Identified by: MAG\_MASSES
* Target: No target is needed, but will affect all characters in the room except for the caster.

**Type 9:** Summon spells

* Summon spells will summon or create NPCs for you. The concept of "summon" is not to transfer an NPC from another place in the realm, but to actually load a new one based on the prototype in the database. However, you need to actually create and place the mobile prototype that you want to be used in the database beforehand.
* Example: Animate Dead spell.
* Identified by: MAG\_SUMMONS
* Target: No target needed with possibility for an object in room, in caster's inventory or equipped by caster.

**Type 10:** Creation Spells

* Spells under this type will do the same thing as summon spells, except for objects. It will create the object based on the prototype in the database and place it in the caster's inventory. Again, you need to create and place the object prototype that you want to be used in the database beforehand.
* Example: Create Food spell.
* Identified by: MAG\_CREATIONS
* Target: No target needed with possibility for an object in room, in caster's inventory or equipped by caster.

**Type 11:** Manual Spells

* Spells that do not fall under any of the categories above tend to fall under this type. Instead of using a predefine function to do the checks, you actually need to code the whole function yourself. Hence, the term "manual" and a bit more complex. Most manual spells are those that have a lot of conditions to check, manipulates the location of caster, victim and/or object, or do more then just a simple attachment of flags.
* Example: Locate Object, Recall and Charm spells
* Identified by: MAG\_MANUAL
* Target: Characters and/or objects depending on the nature of the spell.

As you can see, although most spells tend to be under a single category (Fireball - damage spell), some spells falls under several (Curse - affect spell and alter object spell) or make use of other spell types as its own (Group Heal - group-based spell that uses points spell). With this flexibility, think of various possibilities that can be added in to make a spell interesting. Something like Glitterdust spell, an affect spell that will attach AFF\_GLITTERDUST to a person but at the same time, an unaffect spell that will remove AFF\_HIDE and AFF\_INVISIBLE. Every time someone affected by Glitterdust tries to hide or become invisible, you can do a check and make him unable to do so!

**Part 2: Basic Steps to Create A Spell**

Now, let's say you have loads of idea for a new spell, and you want to implement them. How? It's rather easy if you follow the basic steps below. As an example, we'll use the Glitterdust spell from before:

**Step 1:** Define your idea

* Give the spell a name, and think about what the spell is supposed to do. It's much easier if you list them on a piece of paper so that you can refer back from time to time.
* Identify which category, or categories, the spell will fall into. This can be determined by looking at what the spell is supposed to do. Then crosscheck with the information above to make sure that it does fall under those categories.
* List out detail information about the spell, like the conditions that must be met before the spell can be cast, a basic formula to use if calculation is needed, whether an affect flag is needed and identifying how the spell will affect the caster, victim and/or object.

**Step 2:** Define and name the spell

* First and foremost, you need to define the spell number for the new spell. This is done inside SPELLS.H under the heading Player Spells. A prefix SPELL\_xxx is used to differentiate whether it's a spell or a skill.

/\* PLAYER SPELLS -- Numbered from 1 to MAX\_SPELLS \*/

#define SPELL\_ARMOR 1 /\* Reserved Skill[] DO NOT CHANGE \*/

#define SPELL\_TELEPORT 2 /\* Reserved Skill[] DO NOT CHANGE \*/

#define SPELL\_BLESS 3 /\* Reserved Skill[] DO NOT CHANGE \*/

#define SPELL\_BLINDNESS 4 /\* Reserved Skill[] DO NOT CHANGE \*/

... [ snipped to make it shorter ]

#define SPELL\_INFRAVISION 50 /\* Reserved Skill[] DO NOT CHANGE \*/

#define SPELL\_WATERWALK 51 /\* Reserved Skill[] DO NOT CHANGE \*/

/\* Insert new spells here, up to MAX\_SPELLS \*/

#define MAX\_SPELLS 130

* A stock Circle has a MAX\_SPELLS limit set at 130. So, you can define the new spell with any number from 52, which is exactly after SPELL\_WATERWALK, to MAX\_SPELLS. The number will uniquely identify that spell, and will be used throughout the codes. However, if the new spell is not castable by players but by objects like scrolls and NPCs only, you have to look further down the list for a special section specific for these type of spells. An example is located in [Part 6](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 6).

#define SPELL\_GLITTERDUST 52 ç Our spell is defined as 52

* Now, you need to give the name to the spell. Those who are using codebases prior to bpl15, you'll need to take a look at SPELL\_PARSER.C and find the table that has the list of spell/skill names. The table will look something like below. Notice the number at the side? That is to indicate the slot number in the table. Since we defined our Glitterdust spell as number 52, we must put the spell name inside slot 52 as well. The "!UNUSED!" label is to denote that the slot is still empty.

char \*spells[] =

{

"!RESERVED!", /\* 0 - reserved \*/

/\* SPELLS \*/

"armor", /\* 1 \*/

"teleport",

"bless",

"blindness",

... [ snipped again to make it shorter ]

"infravision", /\* 50 \*/

"waterwalk",

"glitterdust", ç The name for the new spell in slot 52.

"!UNUSED!",

* However, the above method of placing the spell name is only applicable for those using codebases prior to bpl15. Those using bpl15 will find it much easier to place the spell name, which is done together when defining the characteristics of the new spell [Step 4].

**Step 3:** Code the spell in

* As different spell category will have different method of implementation, the way the spell is coded in will be explain in detail in their respective sections later on. This is the only place where the various spell categories differ greatly from one another.

**Step 4:** Defining the spell characteristics

* The characteristics of a spell are defined at the end of SPELL\_PARSER.C under MAG\_ASSIGN\_SPELLS(). A special function called SPELLO() is used for this purpose with the following arguments:
  + Spell number - the unique number that identifies the spell.
  + Spell name - the name of the spell. (\*\*Note: Only applicable under bpl15.)
  + Max mana - the amount of mana needed for casting this spell the first time.
  + Min mana - the minimum amount of mana needed for casting.
  + Change in mana - the amount that will reduce the mana cost for casting as the caster's level increases.
  + Min position - the minimum position of the caster before this spell can be cast (usually it's fighting or standing).
  + Target - a list of valid targets for the spell, joined with bitwise OR ('|'). This is clearly explained in SPELLS.H under Possible Target, and the list of targets starts with TAR\_xxx.
  + Violent - a TRUE or FALSE check to indicate whether the spell should be considered as aggressive or offensive.
  + Routines - to identify the categories that this spell is under. Also joined with bitwise OR ('|').
* For the Glitterdust spell, we mentioned that it's both an affect spell as well as an unaffect spell. It's not violent, at least I don't think so, and the target is someone in the room. Just to make sure, we set it so the caster will not be a valid target.
* Note again that if the spell is not-castable by players but by objects like scrolls and NPCs only, you'll have to go further down for a special section specific for these type of spells. An example is located in [Part 6](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part 6).

*Under codebases prior to bpl15*

spello(SPELL\_ARMOR, 30, 15, 3, POS\_FIGHTING,

TAR\_CHAR\_ROOM, FALSE, MAG\_AFFECTS);

spello(SPELL\_CURSE, 80, 50, 2, POS\_STANDING,

TAR\_CHAR\_ROOM | TAR\_OBJ\_INV, TRUE, MAG\_AFFECTS | MAG\_ALTER\_OBJS);

spello(SPELL\_GLITTERDUST, 80, 50, 2, POS\_STANDING, ç Our spell characteristic.

TAR\_CHAR\_ROOM | TAR\_NOT\_SELF, FALSE, MAG\_AFFECTS | MAG\_UNAFFECTS);

*Under codebase bpl15*

spello(SPELL\_ARMOR, "armor", 30, 15, 3, POS\_FIGHTING,

TAR\_CHAR\_ROOM, FALSE, MAG\_AFFECTS);

spello(SPELL\_CURSE, "curse", 80, 50, 2, POS\_STANDING,

TAR\_CHAR\_ROOM | TAR\_OBJ\_INV, TRUE, MAG\_AFFECTS | MAG\_ALTER\_OBJS);

spello(SPELL\_GLITTERDUST, "glitterdust", 80, 50, 2, POS\_STANDING, ç

TAR\_CHAR\_ROOM | TAR\_NOT\_SELF, FALSE, MAG\_AFFECTS | MAG\_UNAFFECTS);

**Step 5:** Create appropriate spell messages

* What being displayed when a spell is cast, an offensive spell blasted a victim to bits and pieces, or when a spell starts to lose its effect, still needs to be added in. Fortunately, the message for when a spell is being cast is done together while coding the spell in. Unfortunately, the other two are in separate files.
* Messages for spells losing their effect can be found close to the end of CONSTANTS.C under the heading of spell\_wear\_off\_msg[] which look something like below. This is what we call the "wear off message" which will be seen only by the affected person. Again, we must use the same slot number as the one that we used to define our new spell:

const char \*spell\_wear\_off\_msg[] = {

"RESERVED DB.C", /\* 0 \*/

"You feel less protected.", /\* 1 \*/

"!Teleport!",

"You feel less righteous.",

"You feel a cloak of blindness dissolve.",

... [ snipped to make it shorter ]

"Your night vision seems to fade.", /\* 50 \*/

"Your feet seem less buoyant.",

"The wind finally blew away the glittering dust on you.", ç added.

"!UNUSED!"

};

* Since this is mostly for affect spells, other types like damage and area spells will simply denote their slot with "!Spellname!" for reference. However, every spell **must** have an entry in here, no matter which category the spell falls into.
* Messages for damaging spells or "battle messages" are located in MESSAGES file inside CIRCLEMUD/LIB/MISC directory. Unlike wear off message, only damaging spells will have an entry here. The messages are divided into 4 groups, with 3 messages per group for different audience -- the person doing the damage, the person receiving the damage, and those enjoying the fight. A detail explanation is located at the very top of that file.

\* Burning Hands ç A comment

M ç Start message

5 ç Spell number

You have burned $N to death! ----|

You have been burned to death by $n! Death messages (G1)

$n has burned $N to death! ----|

You miss $N with your burning hands. ----|

$n reaches for you with burning hands, but misses! Miss messages (G2)

$n tries to burn $N, but $E escapes! ----|

You burn $N with your hot little hands! ----|

You cry out in pain as $n grabs you with burning hands! Hit messages (G3)

$N cries out as $n burns $M! ----|

Your attempt to burn $N nears BLASPHEMY!!! ----|

The pitiful creature before you tries to burn you. Attacking God (G4)

Unaware of the risks, $n tries to burn $N. ----|

**Step 6:** Setting class and level

* Even after all the above steps have been taken, your players still can't make use of the new spell yet. This is because all spells and skills are initially been set for Immortal use only. So, you must set it so that players of certain class and certain level will get a chance at practicing this new spell.
* To do this, look inside CLASS.C and search for INIT\_SPELL\_LEVELS(). A special function called SPELL\_LEVEL() is used which accept the following arguments:
  + Spell number - the unique number that identifies the spell.
  + Class - the unique number that identifies the class. Look inside STRUCTS.C under PC Classes heading.
  + Level - the minimum level before player can practice and use the new spell.
* If two or more classes have a chance at learning the same spell, SPELL\_LEVEL() needs to be called for each of those classes.

spell\_level(SPELL\_MAGIC\_MISSILE, CLASS\_MAGIC\_USER, 1);

spell\_level(SPELL\_DETECT\_INVIS, CLASS\_MAGIC\_USER, 2);

spell\_level(SPELL\_GLITTERDUST, CLASS\_MAGIC\_USER, 10); ç for Mages

spell\_level(SPELL\_ARMOR, CLASS\_CLERIC, 1);

spell\_level(SPELL\_DETECT\_INVIS, CLASS\_CLERIC, 6);

spell\_level(SPELL\_GLITTERDUST, CLASS\_CLERIC, 15); ç for Clerics

After going through all these 6 basic steps, your spell will be complete and ready for the players to use. However, with all the changes done to the codes, you'll need to recompile the source code so that the changes will be incorporated and restart the mud so that it will be based on the newly compiled version. Now, the last thing you need to do is test the spell to see if it works, and do some modifications or adjustments so that the spell will do exactly as you planned it to.

But of course, I've oversimplified the whole process since I omitted the coding of the spell itself. That happens to be the backbone of the spell where the bulk of the codes will be. So now, let's slowly take a look at how to spells are actually coded in. So get your spell information and outline ready because if you plan and write down everything properly, they can be very helpful.

**Part 3: Damage Spells - Killing Enemy 101**

The first few spells that most people will think up are those for fighting purposes. Those spells will fall under this category. Now, let's try to code a clerical spell called "Flamestrike" where a column of holy flame will burst forth from the ground to engulf the victim. For details, let's make it a level 22 Cleric spell, obviously a violent one, can be cast while fighting and the target will be someone in the room or the person who the caster is fighting with. Beside that, since the holy fire needs to be "bursting forth" from the ground, let's limit it so that it can’t be cast on air, on watery surface and underwater. Let's make it cause quite a damage since it's high level: 12d8 + (caster's level) / 2

Functions or files involved in this spell:

* MAG\_DAMAGE() in MAGIC.C, where the specific conditions for the spell are checked, the damage is calculated and the test for saving vs. spell for half damage is done. DAMAGE() from FIGHT.C is called from within to actually inflict the damage on the victim.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Victim - pointer to the victim of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.

Inside mag\_damage(), a switch/case statement is use to determine which spell is being cast. So, let's add in a case for our spell:

case SPELL\_HARM:

dam = dice(8, 8) + 8;

break;

case SPELL\_FLAMESTRIKE: ç Case statement for our spell.

/\* Is caster on air, on watery surface or underwater? \*/

if (SECT(IN\_ROOM(ch)) >= SECT\_WATER\_SWIM) {

send\_to\_char("Your feet needs to be on firm ground!\r\n", ch);

return (0);

}

dam = dice(12, 8) + level/2;

break;

We don't need to worry neither the test for saving nor how it will inflict the damage to the victim. As mentioned, the function will do all those on our behalf. Now, all we need to do are define the characteristics of the spell, create some wonderful battle messages about how victim is being burned alive in the column of holy flames, and set the class and level for the spell.

spello(SPELL\_FLAMESTRIKE, "flamestrike", 80, 50, 5, POS\_FIGHTING,

TAR\_CHAR\_ROOM | TAR\_FIGHT\_VICT, TRUE, MAG\_DAMAGE);

**Part 4: Area Spells - Make Thy Enemies Suffer!**

Since there's only 1 area spell, let's make more and this time it's for the Mages. This time, let's code in a spell called "Reverse Gravity" which let the Mage temporarily reverse the gravitational pull around an area to go up, instead of down. Victims will be thrown up into the air for a while, before they come crashing back down. This will be a level 25 spell, violent and can be cast while fighting. Since this is an area spell, no target needs to be specified. For limitation, let's make it so that it can't be cast if the caster is on air, and only quarter damage when cast underwater. Let's take the victim's weight into consideration when calculating the damage. The heavier they are, the harder they fall: 3d8 + (victim's weight) / 2

Functions or files involved in this spell:

* MAG\_AREAS() in MAGIC.C, where the general message when casting the spell is located and the list of characters in the room is being evaluated. Then MAG\_DAMAGE() is being called to calculate and inflict the damage to each valid characters.
* MAG\_DAMAGE() in MAGIC.C as stated above.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Victim - pointer to the victim of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.

First, let's take a look inside mag\_areas(). Again a switch/case statement is used to determine to find out which spell is being cast. Inside this function we only need to add in another case for our spell, and create a general message for when the spell is being cast, one for the caster and another for the rest in the room.

switch (spellnum) {

case SPELL\_EARTHQUAKE:

to\_char = "You gesture and the earth begins to shake all around you!";

to\_room ="$n gracefully gestures and the earth begins to shake violently!";

break;

case SPELL\_REVERSE\_GRAVITY: ç Case statement for our spell

/\* Check if the caster is on air \*/

if (SECT(IN\_ROOM(ch)) == SECT\_FLYING) {

send\_to\_char("You are defying gravity yourself!\r\n", ch);

return;

}

to\_char = "You chant the spell softly, concentrating on reversing the gravitational pull."

to\_room = "$n softly chants. Suddenly, the gravitational pull begins to shift upwards!"

break;

}

That's it for mag\_areas(). Remember, that function is only for the general message and to evaluate the list of characters in the room. The calculation of damage and inflicting it is being handled by mag\_damage(). There's a heading for case statements for area spells. Let's put the damage formula under that heading:

/\* Area spells \*/

case SPELL\_EARTHQUAKE:

dam = dice(2, 8) + level;

break;

case SPELL\_REVERSE\_GRAVITY: ç Case statement for our spell

dam = dice(3, 8) + GET\_WEIGHT(victim)/2;

/\* Is the spell cast underwater? If so, quarter the damage \*/

if (SECT(IN\_ROOM(ch)) == SECT\_UNDERWATER)

dam /= 4;

break;

And again, we just let those 2 functions evaluate the list of characters in the room for valid targets and inflict the damage to each of them. Next, define the characteristic of the spell, create some battle messages about how they were thrown high up in the air and come crashing down, and specify the class and level that can use this spell.

spello(SPELL\_REVERSE\_GRAVITY, "reverse gravity", 120, 100, 5,

POS\_FIGHTING, TAR\_IGNORE, TRUE, MAG\_AREAS);

Not everyone will be thrown up in the air by the reversal of the gravity. A tree monster that has strong roots deep inside the earth will definitely survive the ordeal. [Part 14](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part14) is designed to explain on how to give some NPCs some form of immunity on certain spells (or skills).

**Part 5: Affect Spells - Affect Them for Better or Worse…**

We mention earlier on that affect spells can attach an affect flag or apply an affect on someone, or do both on that person. Let's do 2 spells here -- one to apply an affect, another to attach a flag. First, let's add in "Holy Purpose," a level 15 clerical affect application spell, which will increase the victim's hitroll and damroll temporarily. To make this as a "holy" spell, the caster must be good aligned, and if the victim is also good aligned then there's an added bonus. The duration of the spell is up to (caster's level) / 2 ticks. This spell is benign and can be cast on anybody, including the caster, but not during a fight.

Functions or files involved in this spell:

* MAG\_AFFECTS() in MAGIC.C, where the affect of the spell (whether to change, set an affect flag, or both) is defined and checked for validity. AFFECT\_JOIN() from HANDLER.C is called from within to place the affect on the victim.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Victim - pointer to the victim of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.

Again, a switch/case statement is used to identify the spell cast inside mag\_affects(). However, there are other variables that we need to look at first. Once we are familiar with these variables, only then we can begin.

* The elements in *affected\_type structure*. This structure can be found inside STRUCTS.H with a clear explanation what each element is for. We can apply or attach a maximum of 5 affects through a single spell.
* *accum\_duraction* and *accum\_affect*, which let the function knows whether the affect of the spell and its duration will accumulate or increase with every cast of the same spell.

case SPELL\_INFRAVISION:

af[0].duration = 12 + level;

af[0].bitvector = AFF\_INFRAVISION;

accum\_duration = TRUE;

to\_vict = "Your eyes glow red.";

to\_room = "$n's eyes glow red.";

break;

case SPELL\_HOLY\_PURPOSE: ç Case statement for our spell

/\* First check for the caster's alignment \*/

if (!IS\_GOOD(ch)) {

send\_to\_char("Your soul is not pure enough!\r\n", ch);

return;

}

/\* first set of affect to apply \*/

af[0].duration = level/2; /\* Duration of affect \*/

af[0].location = APPLY\_HITROLL; /\* Where it applies to \*/

af[0].modifier = 2 + (IS\_GOOD(victim)); /\* Amount being applied \*/

af[0].bitvector = 0; /\* Affect flag to use \*/

/\* second set of affect to apply \*/

af[1].duration = level/2; /\* If any of the values \*/

af[2].location = APPLY\_DAMROLL; /\* are 0, it is not \*/

af[3].modifier = 2 + (IS\_GOOD(victim)); /\* necessary to include \*/

af[4].bitvector = 0; /\* the field here \*/

accum\_duration = FALSE; /\* Duration and affect doesn't accumulate \*/

accum\_affect = FALSE; /\* Not necessary to add if value is FALSE \*/

to\_vict = "You feel a deep sense of purpose in your heart.";

to\_room = "$n glows slightly and $s face is set with determination.";

break;

For our second spell, let's do a Mage spell that uses an affect flag called "Silence." When cast, the victim will temporarily lose his voice. Therefore, he can no longer say something, shout or yell for everyone to hear, as well as do anything that requires the use of a voice, like casting spells and reciting scrolls! Offensive? Definitely… it's a spellcaster's nightmare! However, the affect is only temporary (duration: (caster's level) / 2), can't be cast while fighting and the victim has a chance to make his saves for not being affected.

Functions or files involved in this spell:

* MAG\_AFFECTS() in MAGIC.C, where the affect of the spell is defined and placed on the victim.
* MAG\_SAVINGTHROW() needs to be called to check if a save is made by victim.
* STRUCTS.H and CONSTANTS.C for adding new affect flags.
* Various function where characters with certain affect flags are allowed or disallowed to do certain things. This is applicable when we're doing the second spell.

But before anything else, we need to create the affect flag for this spell first. First, we need to define the affect flag in STRUCTS.H under the heading Affect Bits. Place the new flag at the end of the list and increase the number by 1. The number is another form of a slot, which ranges from 0 to 31. Therefore we can only have a limited number of affect flags:

/\* Affect bits: used in char\_data.char\_specials.saved.affected\_by \*/

/\* WARNING: In the world files, NEVER set the bits marked "R" ("Reserved") \*/

#define AFF\_BLIND (1 << 0) /\* (R) Char is blind \*/

#define AFF\_INVISIBLE (1 << 1) /\* Char is invisible \*/

#define AFF\_DETECT\_ALIGN (1 << 2) /\* Char is sensitive to align \*/

#define AFF\_DETECT\_INVIS (1 << 3) /\* Char can see invis chars \*/

... [ snipped to make it shorter ]

#define AFF\_UNUSED20 (1 << 20) /\* Room for future expansion \*/

#define AFF\_CHARM (1 << 21) /\* Char is charmed \*/

#define AFF\_SILENCE (1 << 22) ç Our flag is defined as #22

And like the spell name, we also need to add in a name for our flag, but this time, it's inside CONSTANTS.C under affected\_bits[]. Like before, we need to add the name at the end of the list, but before the termination ('\n') symbol:

/\* AFF\_x \*/

const char \*affected\_bits[] =

{

"BLIND", /\* 0 \*/

"INVIS", /\* 1 \*/

"DET-ALIGN",

"DET-INVIS",

... [ again, snipped to make it shorter ]

"UNUSED", /\* 20 \*/

"CHARM",

"SILENCE", ç The name of our affect flag

"\n"

};

Only after this will we be able to make use of the affect flag. Now, let's add in another case statement inside mag\_affects() for our new spell:

case SPELL\_SILENCE: ç Case statement for our spell

/\* Check if the victim managed to make his save \*/

if (mag\_savingthrow(victim, savetype, 0)) {

send\_to\_char("You failed in your attempt!\r\n", ch);

return;

}

/\* Set up the affect flag \*/

af[0].duration = level/2; /\* Duration of affect \*/

af[0].location = 0; /\* Where it applies to \*/

af[0].modifier = 0; /\* Amount being applied \*/

af[0].bitvector = AFF\_SILENCE; /\* Affect flag to use \*/

accum\_duration = FALSE; /\* Duration and affect doesn't accumulate \*/

accum\_affect = FALSE; /\* Not necessary to add if value is FALSE \*/

to\_vict = "An ethereal hand forces itself down your throat and steals your voice!";

to\_room = "An ethereal hand forces itself down $n's throat, leaving $m speechless!";

break;

Now comes the most difficult part. We want those who are affected by this spell to lose his verbal ability. To do this, we have to go through all verbal communication functions, and do a check to see if that person is affected or not. If he is, then stop him from continuing. We also want to stop the affected person from casting spells and reciting scrolls, since both of them also require verbal ability. I will only include an example of a check in DO\_SAY().

ACMD(do\_say)

{

/\* Check if the person is affected by Silence spell \*/

if (AFF\_FLAGGED(ch, AFF\_SILENCE)) { ç Our check for the flag

send\_to\_char("But you have lost your voice!\r\n", ch);

return;

}

skip\_spaces(&;argument);

if (!\*argument)

send\_to\_char("Yes, but WHAT do you want to say?\r\n", ch);

else {

sprintf(buf, "$n says, '%s'", argument);

act(buf, FALSE, ch, 0, 0, TO\_ROOM);

if (!IS\_NPC(ch) && PRF\_FLAGGED(ch, PRF\_NOREPEAT))

send\_to\_char(OK, ch);

else {

sprintf(buf, "You say, '%s'\r\n", argument);

send\_to\_char(buf, ch);

}

}

}

That's it. The function will apply the affects on the victim when the spells are cast. Now, define the characteristic of the spells, and instead of battle messages, create some fancy wear off messages like how the player regrets that the feeling of determination is now gone, or how good it feels to get his voice back again. Don't forget to set the class and level of the spells.

spello(SPELL\_HOLY\_PURPOSE, "holy purpose", 80, 50, 5, POS\_STANDING,

TAR\_CHAR\_ROOM, FALSE, MAG\_AFFECTS);

spello(SPELL\_SILENCE, "silence", 80, 50, 5, POS\_STANDING,

TAR\_CHAR\_ROOM | TAR\_NOT\_SELF, TRUE, MAG\_AFFECTS);

Note that even though an affect spell (like Silence spell) might be offensive, unlike damage spells the caster and victim will not automatically fight each other. They will only fight if the victim is an NPC and the caster loses his concentration while casting. Don't forget that some NPCs might be immune to this spell…. Check out [Part 14](https://www.circlemud.org/cdp/wtfaq/spellinfo.html" \l "Part14) on how to do this.

**Part 6: Unaffect Spells - "First You Have It, Now You Don't!"**

If someone is affected by Silence spell, it will be a while before he can get his voice back. Let's code in something that will help him get his voice back faster. A different type of spell called "Vocalize" which will remove the Silence affect flag from someone. What's the different? This spell is only available in potions. There is a slight different when defining a non-castable spell.

First, let's define the spell. When defining a non-castable spell in SPELLS.H, we have to look further down list. There's a section for it:

/\* NON-PLAYER AND OBJECT SPELLS AND SKILLS \*/

#define SPELL\_IDENTIFY 201

#define SPELL\_FIRE\_BREATH 202

#define SPELL\_GAS\_BREATH 203

#define SPELL\_FROST\_BREATH 204

#define SPELL\_ACID\_BREATH 205

#define SPELL\_LIGHTNING\_BREATH 206

#define SPELL\_VOCALIZE 207 ç Our spell is defined as #207

Now, let's create a potion that will use this new non-castable spell. I'm going to set the prototype of the potion as object number #3056, after the metal staff (#3055). After that, we can just set it so that the local Magic Shop in Midgaard or other places to sell it. Better yet, give it to an NPC somewhere so players will have to kill the NPC first before getting their hands on it!

#3056

potion blue glowing~

a blue, glowing potion~

A small vial filled with blue, glowing liquid has been left here.~

~

10 g 16385

12 207 -1 -1 ç This potion will use our spell, set at level 12

1 4000 100

E

potion blue glowing~

The vial is filled with some thick bluish liquid, and the liquid

seems to glow! Somehow, while examining the vial you feel a

strange urge to speak and speak and speak....

~

Functions or files involved:

* MAG\_UNAFFECTS() in MAGIC.C, where the opposite affect spell is identified and if the victim is affected by that spell. AFFECT\_FROM\_CHAR() is called to that remove that affect spell from the victim.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Victim - pointer to the victim of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.

Now, let's code the spell in. Through the use of switch/case statement, it's easy to identify what spell has been cast:

case SPELL\_REMOVE\_POISON:

spell = SPELL\_POISON;

to\_vict = "A warm feeling runs through your body!";

to\_room = "$n looks better.";

break;

case SPELL\_VOCALIZE: ç Case statement for out spell

spell = SPELL\_SILENCE; ç This is the opposite affect spell

to\_vict = "Your throat feels cool and tingling suddenly...";

to\_room = "A look of surprise, then relieve appears on $n's face";

break;

The function will check if the victim is affected by the opposite affect spell and will remove it. When defining the characteristic of the spell, the mana requirements and minimum position can be set as 0 since it's a non-castable spell. Besides that, since it's on a potion, neither target nor setting of the class and level for the spell is needed. This is not a violent spell and no fancy messages are needed.

/\* NON-castable spells should appear below here. \*/

spello(SPELL\_IDENTIFY, "identify", 0, 0, 0, 0,

TAR\_CHAR\_ROOM | TAR\_OBJ\_INV | TAR\_OBJ\_ROOM, FALSE, MAG\_MANUAL);

spello(SPELL\_VOCALIZE, "vocalize", 0, 0, 0, 0, ç Our spell

TAR\_IGNORE, FALSE, MAG\_UNAFFECTS);

**Part 7: Alter Object Spells - Bend It to Your Will.**

We have done an affect and unaffect spells that can attach and remove an affect flag on someone. Now, let's do the same for object. This time, we'll do a spell called "Holy Weapon" that will make the weapon do more damage to evil characters. However, since it's supposed to be a holy and magical weapon, the weapon must not be magical in the first place. For the limitation of this spell, let's make it take only good aligned characters can use it, and it will only do quarter damage if used against good aligned characters. Let's make it a level 26 Cleric spell and can only be cast on a weapon in the caster's inventory.

Functions or files involved:

* MAG\_ALTER\_OBJS() in MAGIC.C which will check to see if the correct object is cast on and will directly attach the object flag onto it.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Obj - pointer to the object of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.
* STRUCTS.H and CONSTANTS.C for adding new object flags.
* Various functions where a check needs to be made before a character is allowed to or stopped from manipulating an object.

Like, affect flags in affect spells, we need to create the flag first. The place to add is in STRUCTS.H under the heading Extra Object Flags. Place the new flag at the end of the list and increase the number by 1. The number is another form of a slot, which ranges from 0 to 31. Therefore we can only have a limited number of object flags:

/\* Extra object flags: used by obj\_data.obj\_flags.extra\_flags \*/

#define ITEM\_GLOW (1 << 0) /\* Item is glowing \*/

#define ITEM\_HUM (1 << 1) /\* Item is humming \*/

#define ITEM\_NORENT (1 << 2) /\* Item cannot be rented \*/

#define ITEM\_NODONATE (1 << 3) /\* Item cannot be donated \*/

... [ snipped to make it shorter ]

#define ITEM\_ANTI\_WARRIOR (1 << 15) /\* Not usable by warriors \*/

#define ITEM\_NOSELL (1 << 16) /\* Shopkeepers won't touch it \*/

#define ITEM\_HOLY\_WEAPON (1 << 17) ç Our flag is defined as #17

And like the spell name, we also need to add in a name for our flag, but this time, it's inside CONSTANTS.C under extra\_bits[]. Like before, we need to add the name at the end of the list, but before the termination ('\n') symbol:

/\* ITEM\_x (extra bits) \*/

const char \*extra\_bits[] = {

"GLOW", /\* 0 \*/

"HUM", /\* 1 \*/

"!RENT",

"!DONATE",

... [ snipped to make it shorter ]

"!WARRIOR", /\* 15 \*/

"!SELL",

"HOLYWPN", ç The name of our object flag

"\n"

};

Now that that's done, let's start with the coding of the spell itself. To make it so that only good aligned characters can use the weapon, we'll use 3 ready-made flags -- ITEM\_ANTI\_EVIL and ITEM\_ANTI\_NEUTRAL, to make it into good aligned only, and ITEM\_MAGIC to change it into a magical weapon. Let's add in our case statement inside mag\_alter\_objs():

case SPELL\_REMOVE\_POISON:

if (((GET\_OBJ\_TYPE(obj) == ITEM\_DRINKCON) || (GET\_OBJ\_TYPE(obj) == ITEM\_FOUNTAIN) ||

(GET\_OBJ\_TYPE(obj) == ITEM\_FOOD)) && GET\_OBJ\_VAL(obj, 3)) {

GET\_OBJ\_VAL(obj, 3) = 0;

to\_char = "$p steams briefly.";

}

break;

case SPELL\_HOLY\_WEAPON: ç Case statement for out spell

/\* Let's make sure that the object is a weapon, and not magical \*/

if ((GET\_OBJ\_TYPE(obj) == ITEM\_WEAPON) && (!IS\_OBJ\_STAT(obj, ITEM\_MAGIC))) {

/\* Let's make it so that it's a good-only weapon \*/

if (IS\_OBJ\_STAT(obj, ITEM\_ANTI\_GOOD))

REMOVE\_BIT(GET\_OBJ\_EXTRA(obj), ITEM\_ANTI\_GOOD);

if (!IS\_OBJ\_STAT(obj, ITEM\_ANTI\_NEUTRAL))

SET\_BIT(GET\_OBJ\_EXTRA(obj), ITEM\_ANTI\_NEUTRAL);

if (!IS\_OBJ\_STAT(obj, ITEM\_ANTI\_EVIL))

SET\_BIT(GET\_OBJ\_EXTRA(obj), ITEM\_ANTI\_EVIL);

/\* Now, set the weapon to become a holy and magical one \*/

SET\_BIT(GET\_OBJ\_EXTRA(obj), ITEM\_HOLY\_WEAPON);

SET\_BIT(GET\_OBJ\_EXTRA(obj), ITEM\_MAGIC)

to\_char = "$p starts to glow with a soft, white aura."

}

break;

We don't have to worry about doing a check for a character's alignment when he tries to wield it. With the setting of ITEM\_ANTI\_EVIL and ITEM\_ANTI\_NEUTRAL flags, the function equip\_char() will do the check for us. Now, what we need to think of is how to do the check when the weapon is used for fighting. For that we need to take a look inside hit() which will calculate the damage being done by a weapon while fighting:

... [ snipped to make it shorter ]

/\* okay, we know the guy has been hit. now calculate damage. \*/

/\* Start with the damage bonuses: the damroll and strength apply \*/

dam = str\_app[STRENGTH\_APPLY\_INDEX(ch)].todam;

dam += GET\_DAMROLL(ch);

/\* Maybe holding arrow? \*/

if (wielded && GET\_OBJ\_TYPE(wielded) == ITEM\_WEAPON) {

/\* Add weapon-based damage if a weapon is being wielded \*/

dam += dice(GET\_OBJ\_VAL(wielded, 1), GET\_OBJ\_VAL(wielded, 2));

/\* Does it have Holy Weapon flag? \*/ ç Our check for the flag

if (IS\_OBJ\_STAT(wielded, ITEM\_HOLY\_WEAPON)) {

if (IS\_GOOD(victim)) dam /= 4;

if (IS\_EVIL(victim)) dam \*= 1.5;

}

} else {

/\* If no weapon, add bare hand damage instead \*/

... [ snipped to make it shorter ]

We're almost done. The only things left to do are to define the characteristic of the spell and set the class and level for the spell. This spell doesn't have any fancy wear off or battle messages.

spello(SPELL\_HOLY\_WEAPON, "holy weapon", 120, 100, 5, POS\_STANDING,

TAR\_OBJ\_INV, FALSE, MAG\_ALTER\_OBJS);

Some people might wonder: if Enchant Weapon spell is supposed to alter an object, why does it fall under manual spell instead of this category? True, Enchant Weapon does alter an object. However, alter object spells are supposed to attach or remove object flags. Enchant Weapon not only attaches ITEM\_MAGIC flag on a weapon, but also applies 2 affects (APPLY\_HITROLL and APPLY\_DAMROLL) to it. And we can't make it into an affect spell as well since affect spells are only for characters, and not object.

**Part 8: Points Spells - Measuring the Beat of Our Life…**

Points spells are those that will modify a person's hit, mana and movement points. So far, the implemented points spells are benign and curative. Let's make another one that will "cure" a person's movement point, instead of his hitpoint, and call it "Refresh." Since this is a not a violent spell, it can be cast on anyone in the room, including the caster, can't be cast while fighting and let's give this spell to a Cleric level 12. For the amount to be added in: (caster's level) x 3.

Functions or files involved:

* MAG\_POINTS() in MAGIC.C, which identifies which points to increase and calculate the amount to increase which is added directly to it.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Victim - pointer to the victim of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.

There are 2 important variables inside mag\_points() that we need to get familiar with: *hit* for the amount to add to the victim's hitpoint, and *move* for the victim's movement point. Again, we just add in a case statement in the switch/case for our spell.

case SPELL\_HEAL:

hit = 100 + dice(3, 8);

send\_to\_char("A warm feeling floods your body.\r\n", victim);

break;

case SPELL\_REFRESH: ç Case statement for our spell

hit = 0 /\* Add to hitpoint. Not necessary if 0 \*/

move = level\*3; /\* Add to movement point. \*/

send\_to\_char("A refresh feeling flows through you.\r\n", victim);

break;

Again, the function will take care of everything and add the calculated amount to the victim's points. Now define the characteristics of the spell and set the class and level of spell before recompiling. You don't have to think up any fancy messages for this spell.

spello(SPELL\_REFRESH, "refresh", 60, 30, 5, POS\_STANDING,

TAR\_CHAR\_ROOM, FALSE, MAG\_POINTS);

**Part 9: Group-based Spells - Defend Thy Neighbors!**

This is also one of the easiest spell to create. Since the targets of these spells are the caster's own group members, they are usually benign and beneficial in nature. We can actually make use of spells from other categories and call them from within. An example will be a Mage level 10 spell called "Group Invisibility." Anyone in the room who is grouped with the caster will become invisible. This spell doesn't have any target, and can't be cast while fighting.

Functions or files involved:

* MAG\_GROUPS() in MAGIC.C which will determine whether the caster is in a group or not, and will search for the group members who are in the same room as the caster. For each member found, PERFORM\_MAG\_GROUPS() is called to actually cast the spell on them.
* PERFORM\_MAG\_GROUPS() in MAGIC.C which will call the appropriate function based on the category of the spell for each group member found.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Tch - pointer to one of the caster's group member who is in the same room.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.
* Various spell-type functions which will cast the spell on each group member.

Since we are going to use an already implemented Invisibility spell, we don't need to code that in. However, if you want to do something new, then you will need to add in the appropriate spell. Another interesting to note is that we don't actually need to make any changes to mag\_groups() since its function is to find the caster's group member who are in the same room. So, the only one we'll be changing is perform\_mag\_groups(), and maybe creating a new spell if you're not using a ready-made one. Just add in a case statement for our spell, and call the appropriate function:

case SPELL\_GROUP\_HEAL:

mag\_points(level, ch, tch, SPELL\_HEAL, savetype);

break;

case SPELL\_GROUP\_INVISIBILITY: ç Case statement for our spell

/\* Invisibility spell is an affect spell \*/

mag\_affects(level, ch, tch, SPELL\_INVISIBLE, savetype);

break;

As simple as that. Mag\_groups() will search for the group members, perform\_mag\_groups() will determine the spell being cast, and mag\_affects() will place the affect of the spell on them. As before, no fancy battle or wear off messages is needed. Define the characteristic of the spell, set the class and level to use this spell, and you're done.

spello(SPELL\_GROUP\_INVISIBILITY, "group invisibility", 100, 80, 4,

POS\_STANDING, TAR\_IGNORE, FALSE, MAG\_GROUPS);

**Part 10: Masses Spells - Let Them See Your Way…**

The word "mass" always reminds me of a preacher in a Sunday church. When the preacher starts preaching, everyone will listen (well, at least they were supposed to anyway). I guess that's why the description states that they are supposed to affect everyone the **but** the caster.

So far no spell has been implemented under this category. I guess it's a bit hard to figure out what is a suitable spell for this. If the spell should be offensive (damaging) then area spells is more appropriate since numerous checks are done. If the spell is beneficial, then why should a caster cast it since he won't benefit from it. Anyway, I have seen several MUDs that changed this so that it will **include** the caster as well, instead of excluding him.

Based on the description of the preacher above, let's make a spell called "Mass Bless" which let's a Cleric level 20 bless everyone in the room (except for himself that is…). Like the example for the group-based spell above, I'm going to use a ready-made Bless spell for this. No target is needed and it can only be cast while standing.

Functions or files involved:

* MAG\_MASSES() in MAGIC.C which will evaluate the list of characters in the room at that time. For each character that is not the caster, it will call the appropriate function based on the category of the spell being cast.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make.
* Various spell-type functions which will cast the spell on each character in the room.

Like the perform\_mag\_groups() above, just add in a case statement inside the switch/case in mag\_masses(), and call the appropriate function that will cast the spell. The function will go through the list of characters in the room and will cause the spell to affect each of them.

switch (spellnum) {

case SPELL\_MASS\_BLESS: ç Case statement for our spell

/\* Bless spell is an affect spell \*/

mag\_affects(level, ch, tch, SPELL\_BLESS, savetype);

}

Again, no fancy battle and wear off messages are needed. All we need to do now is define the characteristics of the spell, and set the class and level for it.

spello(SPELL\_MASS\_BLESS, "mass bless", 80, 50, 4, POS\_STANDING,

TAR\_IGNORE, FALSE, MAG\_MASSES);

**Part 11: Summon Spells - "Come and Follow Me!"**

So far we've been doing spells for Mages and Clerics. Let's do different one for this category. Let's base this one on nature, which a Druid or Ranger (even a Cleric actually) can make use off. This spell is called "Sticks to Snakes," which changes sticks in the room to become snakes, as the name said. Now, let's make it so that the number of snakes that will appear will depend on the sector type. The more nature-like the sector is, the more snakes will appear. This spell is not violent (at least I don't think so), but can be cast while fighting. There's no target for this spell, since the rooms aren't really littered with sticks to begin with. The snakes will be bound to the caster until one of them dies. By the way, this spell can't be cast on air, on watery surface and underwater.

Functions or files involved:

* MAG\_SUMMONS() in MAGIC.C, which will determine the prototype to use, load the NPC into the room, and set it to follow the caster. If a corpse is needed by the spell, it will also transfer all objects in the corpse to the NPC and removes the corpse afterwards.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Obj - pointer to the object of the spell.
  + Spellnum - the unique number that identifies the spell cast.
  + Savetype - the type of save the victim should make in order to take half damage.
* ASSIGN\_MOBILES() in SPEC\_ASSIGN.C to assign a special procedure to the NPC.

First, we have to get a prototype of a snake. I'm going to make one up and set it to use mobile number #3096, after the cryogenicist (#3095). The reason why I'm not using a ready-made prototype in the database is because I want the description to say that they are different:

#3096

snake brown stick~

the stick-like snake~

A brown stick-like snake is at your feet, hissing at you.

~

Unlike other snakes, this one is thin like a stick. Its coat is not

even glistening like the rest, but just dark brown. Even though the

snake puzzled you, looking at the poisoned saliva dripping from its

fangs does stop you from examining it even closer.

~

afnopr dks -200 S ç 'a' indicates the use of special procedure

11 7 4 5d10+50 3d6+0

0 12000

8 8 0

Now, to make this more interesting, let's make the snake poisonous as well. The prototype is set to have a special procedure. To attach the procedure itself, we have to do that inside SPEC\_ASSIGN.C file. Inside ASSIGN\_MOBILES(), there is an entry for Midgaard. Just attach the snake procedure, which allows snakes to have a chance at poisoning the person who they are fighting with, using ASSIGNMOB() to our mobile prototype number:

/\* Midgaard \*/

ASSIGNMOB(3005, receptionist);

ASSIGNMOB(3010, postmaster);

ASSIGNMOB(3020, guild);

ASSIGNMOB(3021, guild);

... [ snipped to make it shorter ]

ASSIGNMOB(3095, cryogenicist);

ASSIGNMOB(3105, mayor);

ASSIGNMOB(3096, snake); ç Our prototype now has 'snake' procedure

Okay, now that the NPC that we want to use is ready, let's look inside mag\_summons(). There are a few variables and structures that we need to get familiar first:

* *mag\_summons\_msg[]* is sent to the room upon the success of the spell.
* *mag\_summons\_fail\_msg[]* is sent to the caster upon the failure of the spell.
* *fmsg* and *msg* signify which message number should be used for failure and success.
* *mob\_num*is the mobile prototype number that will be loaded.
* *num* is the number of mobiles that will be loaded or summoned into the room.
* *pfail* is to indicate the percentage of failing the spell.
* *handle\_corpse* is to indicate that a corpse is used and should be handled by the function.

case SPELL\_CLONE:

msg = 10;

fmsg = number(2, 6); /\* Random fail message. \*/

mob\_num = MOB\_CLONE; /\* MOB\_CLONE has been defined as 10 \*/

pfail = 50; /\* 50% failure, should be based on something later. \*/

break;

case SPELL\_STICKS\_TO\_SNAKES: ç Case statement for our spell

/\* Check where the caster is currently \*/

if (SECT(IN\_ROOM(ch)) >= SECT\_WATER\_SWIM) {

send\_to\_char("You can't cast the spell here!\r\n", ch);

return;

}

/\* Set the possibility and amount to load base on sector type \*/

mob\_num = 3096;

num = number(1, 1 + SECT(IN\_ROOM(ch)));

pfail = 100 - 10\*SECT(IN\_ROOM(ch));

/\* Set the appropriate values for the others \*/

msg = 9;

fmsg = number(2, 6);

handle\_corpse = FALSE;

break;

Like normal, the function will take care of the loading of the NPC and set them to follow the caster. And like normal also, we need to set the characteristic, as well as the class and level of the spell. No fancy battle and wear off messages are needed for this spell.

spello(SPELL\_STICKS\_TO\_SNAKES, "sticks to snakes", 80, 50, 4,

POS\_FIGHTING, TAR\_IGNORE, FALSE, MAG\_SUMMONS);

**Part 12: Creation Spells - Magical Sleight of Hand.**

Sleight of hand, a common trick that makes things seemed to appear from out of the thin air. However, with a creation spell like "Knowledge," we can make an identify scroll to appear in our hand for use. It is also simple to do actually, but first we need to know the prototype object number of the scroll which we'll be using (#3050). Only then we can continue. Let's make it a level 14 Mage spell, which can't be cast while fighting.

Functions or files involved:

* MAG\_CREATIONS() in MAGIC.C which will determine the prototype to use, load the object and place it in the caster's inventory.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Spellnum - the unique number that identifies the spell cast.

Inside mag\_creations(), we just add in a case statement for our spell in the switch/case combination:

case SPELL\_CREATE\_FOOD:

z = 10;

break;

case SPELL\_KNOWLEDGE: ç Case statement for our spell

z = 3050; ç The prototype number that will be used

break;

If you want to create a different scroll or potion, you can either create a new one and place it inside the database or use one that is already in there. From here, we just need to define the characteristic of the spell, set the level and class of the spell and we are done since there is no need for fancy messages.

spello(SPELL\_KNOWLEDGE, "knowledge", 60, 40, 4, POS\_STANDING,

TAR\_IGNORE, FALSE, MAG\_CREATIONS);

**Part 13: Manual Spells - Making It Special…**

Spells that can't fall into the other types will fall here. And since we don't have a ready-made function that will do everything for us, we have to code everything manually. For this, let's do a spell that will let Mages see into the rooms around him. A level 18 spell called "Farsight" which will show to the caster a list of who are in the rooms adjacent to the one he is in. This is not a violent spell, and can only be cast while standing.

Functions or files involved:

* ASPELL(spell\_xxx) in SPELLS.C that is manually coded, where xxx is the spell name.
* Argument passed:
  + Level - the level of the caster or level of spell on an object.
  + Ch - pointer to the caster of the spell.
  + Victim - pointer to the victim of the spell
  + Obj - pointer to the object used in the spell
* SPELLS.H for declaring the ASPELL(spell\_xxx) function.
* CALL\_MAGIC() in SPELL\_PARSER.C for an additional case statement for our spell.
* Various functions that are needed to make this spell works.

There is a slight different when defining the spell. After defining the unique spell number, we have to declare the function for the spell itself. This is done close to the end of SPELLS.H, together with the rest of manual spells function declaration:

ASPELL(spell\_charm);

ASPELL(spell\_identify);

ASPELL(spell\_enchant\_weapon);

ASPELL(spell\_detect\_poison);

ASPELL(spell\_farsight); ç Declaration of our spell function

The codes for the function is usually kept inside SPELLS.C:

ASPELL(spell\_farsight)

{

int door, found;

struct char\_data \*person;

/\* Check for every exit, from north to west \*/

for (door = NORTH; door <= WEST; door++)

if (CAN\_GO(ch, door)) {

found = FALSE;

sprintf(buf, "You look to the %s:", dir[door]);

/\* Is the room dark and can caster see in the dark? \*/

if (IS\_DARK(EXIT(ch, door)->to\_room) && !CAN\_SEE\_IN\_DARK(ch)) {

strcat(buf, " Too dark to see anyone...");

found = TRUE;

} else {

/\* If there is someone in the room, can caster see him? \*/

for (person = world[EXIT(ch, door)->to\_room].people; person;

person = person->next\_in\_room)

if (CAN\_SEE(ch, person)) {

sprintf(buf + strlen(buf)," %s", GET\_NAME(person));

found = TRUE;

}

}

/\* Tell the caster what he can see \*/

if (found) send\_to\_char(strcat(buf, "\r\n"), ch);

}

}

Okay, the last thing before we can finish this off like the rest, is that we need to add in a case statement for our new spell in SPELL\_PARSER.C under call\_magic() with the rest of the manual spells. Notice the difference in the way a manual spell function is called compared to the way other categories call theirs?:

if (IS\_SET(SINFO.routines, MAG\_MANUAL))

switch (spellnum) {

case SPELL\_CHARM: MANUAL\_SPELL(spell\_charm); break;

case SPELL\_CREATE\_WATER: MANUAL\_SPELL(spell\_create\_water); break;

case SPELL\_DETECT\_POISON: MANUAL\_SPELL(spell\_detect\_poison); break;

case SPELL\_ENCHANT\_WEAPON: MANUAL\_SPELL(spell\_enchant\_weapon); break;

case SPELL\_IDENTIFY: MANUAL\_SPELL(spell\_identify); break;

case SPELL\_LOCATE\_OBJECT: MANUAL\_SPELL(spell\_locate\_object); break;

case SPELL\_SUMMON: MANUAL\_SPELL(spell\_summon); break;

case SPELL\_WORD\_OF\_RECALL: MANUAL\_SPELL(spell\_recall); break;

case SPELL\_TELEPORT: MANUAL\_SPELL(spell\_teleport); break;

case SPELL\_FARSIGHT: MANUAL\_SPELL(spell\_farsight); break; ç

}

Like the rest, define the characteristic of the spell, and set the level and class for it. As to whether it needs a battle and/or wear off message, it depends on the nature of the spell. If you are placing an affect spell on someone, like Charm spell, then you'll require a wear off message. As for battle messages… I'm not sure. If the spell is too complex for damage and area type, then have a single set of battle messages might not be enough. Then again, I could be dead wrong!

spello(SPELL\_FARSIGHT, "farsight", 100, 75, 5, POS\_STANDING,

TAR\_IGNORE, FALSE, MAG\_MANUAL);

I know some people wondered why Control Weather spell couldn't be cast. If you take a look at the arguments that are passed into ASPELL(), you'll notice that there is no way the word "worse" or "better" can be passed in. There are a few ways to go around this, and someone wrote to the mailing list suggesting a method he used sometime in 1998. Hopefully a copy of the mail is still in the archive.

**Part 14: Giving NPCs Immunity.**

Not everyone is susceptible to the spells and skills that we are using. For example, a tree is extremely hard to be bashed over with a small sword (even though it's a **huge** sword to the wielder). And a tree will definitely be immune to the Reverse Gravity spell that we just made, since they tend to have roots so deep that they can't be thrown around easily. Therefore, a check can be made in the coding for the Bash skill or the Reverse Gravity spell for these situations.

The question is, how are we to know? If you have implemented race in your MUD, which covers the NPCs as well, it's easy to check if the victim is of RACE\_TREE for example. But until you add that in, you will have to make do with mobile flags for NPCs.

The existing flags are located in STRUCTS.H under Mobile Flags. Like affect and object flags, these flags are also limited, ranging from 0 to 31:

/\* Mobile flags: used by char\_data.char\_specials.act \*/

#define MOB\_SPEC (1 << 0) /\* Mob has a callable spec-proc \*/

#define MOB\_SENTINEL (1 << 1) /\* Mob should not move \*/

#define MOB\_SCAVENGER (1 << 2) /\* Mob picks up stuff on the ground\*/

#define MOB\_ISNPC (1 << 3) /\* (R)Automatically set on all Mobs\*/

... [ snipped to make it shorter ]

#define MOB\_NOSLEEP (1 << 15)/\* Mob can't be slept \*/

#define MOB\_NOBASH (1 << 16)/\* Mob can't be bashed (e.g. trees)\*/

#define MOB\_NOBLIND (1 << 17)/\* Mob can't be blinded \*/

There is already a flag called MOB\_NOBASH, to let NPCs who has it to be immune to bashing. However, not being able to bash someone only means that that person is either too heavy or too large to be bashed. So, let's make a flag that will protect NPCs from Reverse Gravity or other spells/skills that can throw them around. Let's call this flag MOB\_NOTHROW. Again, like the affect and object flags, we put our new flag at the end of the list, incrementing it by 1:

#define MOB\_NOTHROW (1 << 18)/\* Mob can't be thrown around \*/

Next, we have to put the name of the flag, again inside CONSTANTS.C, but this time in under another heading. Just add the name at the end of the list, but before the termination ('\n') symbol:

/\* MOB\_x \*/

const char \*action\_bits[] = {

"SPEC", /\* 0 \*/

"SENTINEL", /\* 1 \*/

"SCAVENGER",

"ISNPC",

... [ snipped to make it shorter ]

"!SLEEP", /\* 15 \*/

"!BASH",

"!BLIND",

"!THROW", ç The name of our mobile flag

"\n"

};

With that, we can now make use of our new flag. For an example of how to use this flag, let's go back and look at our coding for Reverse Gravity spell again. We'll take a look at the one inside mag\_damage() since that is where the damage is being calculated and inflicted from:

/\* Area spells \*/

case SPELL\_EARTHQUAKE:

dam = dice(2, 8) + level;

break;

case SPELL\_REVERSE\_GRAVITY: ç Case statement for our spell

/\* Can the victim be thrown around? \*/

if (MOB\_FLAGGED(victim, MOB\_NOTHROW) { ç Our check for the flag

send\_to\_char("You brace yourself, and survive the ordeal.\r\n", victim);

act("$n braces $mself and survives the ordeal!", TRUE, victim, 0, 0, TO\_ROOM);

return (0);

}

dam = dice(3, 8) + GET\_WEIGHT(victim)/2;

/\* Is the spell cast underwater? If so, quarter the damage \*/

if (SECT(IN\_ROOM(ch)) == SECT\_UNDERWATER)

dam /= 4;

break;

Wait a minute! The code checks if the NPC has the MOB\_NOTHROW flag. But what if the victim is a player and not an NPC? Simple…. If you follow the definition for MOB\_FLAGGED(), you'll see that it actually checks to see if the victim is an NPC or not. If it's a player, the condition will automatically be false.

Checking a victim or an NPC for specific flags can be done anywhere in the codes, not just for spells and skills. The only think that you have to remember though is that affect flags (AFF\_xxx) can be attached to both players and NPCs and uses AFF\_FLAGGED() for checking, which mobile flags are only for NPCs and uses MOB\_FLAGGED() for checking.