NOTE: I'm going to say this here and now. BACKUP your mud on a regular basis. Notice I don't say daily, hourly or monthly. How often is up to you, but if you break something and have to do a restore, you don't want to have to spend 30 hours reworking your code.

NOTE: Always have a stock copy of your codebase for reference. It also makes pfile wipes much easier.

I. Changing Levels -- Decide how many levels you want for mortal and immortals. Levels will always be concurrent, so the highest mortal level is LVL_IMMORT - 1. And the highest immortal level will always be LVL_IMPL Those are the only two levels that cannot be changed. For the purposes of this document, we will say there are going to be 50 mortal levels and 10 immortal levels, so LVL_IMMORT is 51 and LVL_IMPL is 60. We'll set LVL_GRGOD as 59 and LVL_GOD as 58. Now to identify what levels 52 thru 57 will be called. To make coding easier, we'll call them LVL_IMMORT2 thru LVL_IMMORT7.

A. Structs.h

1. Around line 625 in structs.h the listing of immortal levels can be found. Search for LVL_IMPL and you should see some #define statements similar to those below.

Change the defines so they reflect your levels. (NOTE: LVL_IMMORT must remain the lowest and LVL_IMPL must remain the highest). If you are following this document then the immortal levels are now listed like this:

```
#define LVL_IMPL      60
#define LVL_GRGOD     59
#define LVL_GOD       58
#define LVL_IMMORT7   57
#define LVL_IMMORT6   56
#define LVL_IMMORT5   55
#define LVL_IMMORT4   54
#define LVL_IMMORT3   53
#define LVL_IMMORT2   52
#define LVL_IMMORT    51
```

2. Right below those defines are where LVL_BUILDER and LVL_FREEZE are defined.
a. set LVL_BUILDER to the lowest immortal level that will have access to building commands (OLC, dg_scripts, etc).
b. set LVL_FREEZE to the lowest immortal level that will be allowed to freeze and/or thaw mortals characters.

3. Close and save structs.h

B. Class.c

1. At line 300 will start the listing of saving throws for the various classes. Search for saving_throws until you see something like the function below. For the sake of convenience there are formulas that will automatically figure out the saving throws and keep them balanced. These formulas match up with the time tested code. Remove the entire function (everything between the first { and the last } and replace it with the following code. (NOTE: the first line in the code and the first line in what you are replacing here are the EXACT same)

```
byte saving_throws(int class_num, int type, int level)
{
  switch (class_num) {
  case CLASS_MAGIC_USER:
    switch (type) {
    case SAVING_PARA:   /* Paralyzation */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (70 - ((70 * level) / (LVL_IMMORT - 1)));
      }

    case SAVING_ROD:    /* Rods */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (55 - ((55 * level) / (LVL_IMMORT - 1)));
      }

    case SAVING_PETRI:  /* Petrification */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (65 - ((65 * level) / (LVL_IMMORT - 1)));
      }

    case SAVING_BREATH: /* Breath weapons */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (75 - ((75 * level) / (LVL_IMMORT - 1)));
      }
    case SAVING_SPELL:  /* Generic spells */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (60 - ((60 * level) / (LVL_IMMORT - 1)));
      }
    default:
      log("SYSERR: Invalid saving throw type.");
      break;
    }
    break;
  case CLASS_CLERIC:
    switch (type) {
    case SAVING_PARA:   /* Paralyzation */
```

```
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (60 - ((60 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_ROD:    /* Rods */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (70 - ((70 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_PETRI:  /* Petrification */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (65 - ((65 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_BREATH: /* Breath weapons */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (80 - ((80 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_SPELL:  /* Generic spells */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (70 - ((70 * level) / (LVL_IMMORT - 1)));
    }

  default:
    log("SYSERR: Invalid saving throw type.");
    break;
  }
  break;
case CLASS_THIEF:
  switch (type) {
  case SAVING_PARA:   /* Paralyzation */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
```

```
      return (65 - ((65 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_ROD:    /* Rods */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (70 - ((70 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_PETRI:  /* Petrification */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (60 - ((60 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_BREATH: /* Breath weapons */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (80 - ((80 * level) / (LVL_IMMORT - 1)));
    }
  case SAVING_SPELL:  /* Generic spells */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (75 - ((75 * level) / (LVL_IMMORT - 1)));
    }

  default:
    log("SYSERR: Invalid saving throw type.");
    break;
  }
  break;
case CLASS_WARRIOR:
  switch (type) {
  case SAVING_PARA:   /* Paralyzation */
    if (level == 0)
      return 100;
    if (level >= LVL_IMMORT) {
      return 0;
    } else {
      return (70 - ((70 * level) / (LVL_IMMORT - 1)));
    }

  case SAVING_ROD:    /* Rods */
    if (level == 0)
      return 100;
```

```
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (80 - ((80 * level) / (LVL_IMMORT - 1)));
      }

    case SAVING_PETRI:  /* Petrification */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (75 - ((75 * level) / (LVL_IMMORT - 1)));
      }

    case SAVING_BREATH: /* Breath weapons */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (85 - ((85 * level) / (LVL_IMMORT - 1)));
      }

    case SAVING_SPELL:  /* Generic spells */
      if (level == 0)
        return 100;
      if (level >= LVL_IMMORT) {
        return 0;
      } else {
        return (85 - ((85 * level) / (LVL_IMMORT - 1)));
      }

    default:
      log("SYSERR: Invalid saving throw type.");
      break;
    }
  default:
    log("SYSERR: Invalid class saving throw.");
    break;
  }

  /* Should not get here unless something is wrong. */
  return 100;
}
```

2. THAC0 means To Hit Armor Class 0. It is based on older editions of Dungeons & Dragons system. Again a formula has been figured to simplify this process. Again, you will either cut or remark out all of the code necessary, this time anything having to do with THAC0 calculations. (NOTE: the first line in the code and the first line in what you are replacing here are the EXACT same)

```
int thaco(int class_num, int level)
{
  switch (class_num) {
  case CLASS_MAGIC_USER:
```

```
        if (level == 0)
                return 100;
        if (level >= LVL_IMMORT)
        {
                return 0;
        } else {
                return (20 - ((20 * level) / (LVL_IMMORT - 1)));
        }

  case CLASS_CLERIC:
        if (level == 0)
                return 100;
        if (level >= LVL_IMMORT)
        {
                return 0;
        } else {
                return (19 - ((19 * level) / (LVL_IMMORT - 1)));
        }

  case CLASS_THIEF:
        if (level == 0)
                return 100;
        if (level >= LVL_IMMORT)
        {
                return 0;
        } else {
                return (18 - ((18 * level) / (LVL_IMMORT - 1)));
        }

  case CLASS_WARRIOR:
        if (level == 0)
                return 100;
        if (level >= LVL_IMMORT)
        {
                return 0;
        } else {
                return (17 - ((17 * level) / (LVL_IMMORT - 1)));
        }

  default:
    log("SYSERR: Unknown class in thac0 chart.");
  }

  /* Will not get there unless something is wrong. */
  return 100;
}
```

3. Now to increase the maximum experience. This is done by adjusting the maximum experience that it takes to reach LVL_IMPL. Don't be mislead, there is no amount of experience that an immortal character can earn to automatically be increased in level. Due to coding constraints, though, each level has a set amount of experience to reach. By setting the level for the implementor the mud now has an upper limit. This is an arbitrary number, but it must be higher than any other level. If you have made the changes, then you will find this around line 1000 or you can just search for #define EXP_MAX. Change the value to 1500000000 just to be safe.

4. Around line 893 you will see the beginning of a listing of spells and skills for each class. This is in the function init_spell_levels. This an example of what you are looking at

spell_level(SKILL_WP_STAFF, CLASS_MAGIC_USER, 1);

All skills and spells assigned to any class are defined here. This is important to know in case the need arises to add a new skill or spell later. The important thing to know here is what the number near the end of the line means. In this case the number is 1. So magic users will learn the weapon skill of staff at level 1. Or SPELL_MAGIC_MISSILE in CWG-Buddha:

```
spell_level(SPELL_MAGIC_MISSILE, CLASS_MAGIC_USER, 1);
```

When your mud had 30 levels the skills and spells listed here where fairly balanced. Now that you have changed things up a bit, you will also have to go back and rebalance your skills and spells. Simply decide at what level a player of that particular race can learn them and change the number in that line to that level. If you leave them as is your players will run out of things to learn much quicker since you've increased your number of levels.

5. To change the experience needed for each mortal level is similar to doing both THAC0 and saving throws since, again, someone took the time to make a formula for you. Search for the function level_exp. Again, you will have to remove the code that deals with experience and replace it with the code that follows. (NOTE: the first line in the code and the first line in what you are replacing here are the EXACT same) (NOTE: ensure you also include the remarked out paragraph following the code for informational purposes.)

```
  int mod = 1;        /* always declare variables */
  switch (chclass) {
    case CLASS_MAGIC_USER:
                  mod = 45;
                  return ((level * mod)*(level * mod));
    break;
    case CLASS_CLERIC:
                  mod = 44;
                  return ((level * mod)*(level * mod));
    break;
    case CLASS_THIEF:
                  mod = 43;
                  return ((level * mod)*(level * mod));
    break;
    case CLASS_WARRIOR:
                  mod = 42;
                  return ((level * mod)*(level * mod));
    break;
  }
/*
   * The higher the mod value you use, the more exp is needed per level.
   *
   * When you change your mod values (modifiers) you want to make absolutely
   * sure they don't exceed the maximum level for your mud which you defined
   * earlier as being #define EXP_MAX 1500000000
   *
   * To check this, take the highest Immortal level (The Implementor level)
   * and multiply it by your mod. As an example, if you have 100 total levels
   * on your mod, then your highest mod could be 50. Use the following formula
```

```
   * to check yourself:  (level * mod) * (level * mod)
*/

  log("SYSERR: XP table error in class.c!");
  return 1234567;
}
```

6. Changing level names for your mortals title is a simple task compared to what you have accomplished to get this far. You will need to update the title_male and title_female functions. An example follows:

```
      case 18: return "the Wizard";
```

In this example, cut from CLASS_MAGIC_USER, it shows that characters that reach level 18 in this class are called "the Wizard". I recommend that you find case 1 and change it to read "the Magic User", "the Thief", "the Cleric", or "the Warrior" depending on the class, then change default to read the same thing. This keeps you from having to come up with something to put in a players title, since most players change their title anyways.

7. Close and save class.c

C. act.wizard.c -- the do_cheat function allows player number 1 (the first character, and only the first character created) to automatically advance/demote himself to the highest immortal level after changing the number of levels in the game. It can also be used to reinstate the imp should he "set level" to a lower level for testing purposes

1. First, near the top of the file where local functions are defined find ACMD(do_saveall); and immediately below it enter ACMD(do_cheat);

2. Now to add the do_cheat function to your code. At the very end of the file cut and paste this function to your code:

```
ACMD(do_cheat)
{
  if (GET_IDNUM(ch) != 1) {
    send_to_char(ch, "Huh?!?\r\n");
  } else {
    GET_LEVEL(ch) = LVL_IMPL;
    send_to_char(ch, "Advanced.\r\n");
  }
}
```

3. close and save act.wizard.c

D. interpreter.c

1. Find the section near the top with the prototypes for all do-x functions and locate do_cast in that list. Add ACMD(do_cheat); immediately below AMCD(do_cast);

2. Find do_cast again later in the code in another listing of commands. This command list is where the minlevel for commands is set. Immediately after cedit (so the commands stay in alphabetical order) add this line

```
  { "cheat"     , "cheat"     , POS_DEAD , do_cheat     , 34, 0 },
```

3. The final 0 in each of those lines is for subcommands, but you are interested in 1 before it. As that lines sits, anyone with a player level of 1 can try to use that command. If you change the level that this command works at then you may not be able to use it for your imp character after you change levels. You will also need to scroll through this list one item at a time and find and reassign each of your commands. Notice that some are set to LVL_FREEZE and some to LVL_BUILDER. Those where both defined earlier in structs.h. Keep all of your immortal commands defined as LVL_WHATEVER and not level 51, level 59 etc.

4. close and save interpreter.c

E. autowiz.c

1. On line 79 in the function control_rec, there begins a listing of the immortal levels. Add in the missing levels as needed. If you are using the system outlined here, then it may look something like this:

```
/* Change these if you have different Immortal level names */
struct control_rec level_params[] =
{
  {LVL_IMMORT, "Immortals"},
  {LVL_IMMORT2, "Imm Twos"},
  {LVL_IMMORT3, "Imm Threes"},
  {LVL_IMMORT4, "Imm Fours"},
  {LVL_IMMORT5, "Imm Fives"},
  {LVL_IMMORT6, "Imm Sixes"},
  {LVL_IMMORT7, "Imm Sevens"},
  {LVL_GOD, "Gods"},
  {LVL_GRGOD, "Greater Gods"},
  {LVL_IMPL, "Implementors"},
  {0, ""}
};
```

2. close and save autowiz.c

F. act.informative.c

1. do_who requires no code changes to operate with the new levels

2. In ACMD(do_whois) immortals are listed by their rank, so again, the new rank must be put in. Around line 2211 a listing of the immortal ranks is again seen. Replace the ones that are there with this:

```
  "[Immortal]",        /* highest mortal level +1 */
  "[Imm Two]",         /* highest mortal level +2 */
  "[Imm Three]",       /* highest mortal level +3 */
  "[Imm Four]",        /* highest mortal level +4 */
  "[Imm Five]",        /* highest mortal level +5 */
  "[Imm Six]",         /* highest mortal level +6 */
  "[Imm Seven]",       /* highest mortal level +7 */
  "[God]",             /* highest mortal level +8 */
  "[Greater God]",     /* highest mortal level +9 */
  "[Implementor]",     /* highest mortal level +10 */
```

3. close and save act.informative.c

G. Compile and fix errors

1. in the src/ directory make clean
2. in the src/ directory make
3. there shouldn't be any errors, but accidents happen... fix and try again

H. start up your mud and log on with your implementor character. You will notice right away that if you increased the levels of your mud that you are no longer the imp. Type cheat, and you should be advanced to the highest level, if not then a player file wipe may be in order.
Now enter cedit (configuration editor) and choose option A. From that menu you will have to change the minlevel of your immortals. In this instance of this document it will be 51, so change option B to 51. Quit and save out of the editor as necessary and type wizupdate. Your name should once again be up in lights.

II. The dreaded player file wipe -- pfile wipes should be few and far between, if ever. A pfile wipe entails removing every character, their settings, objects and levels. You will be starting over with no characters, not even an implementor.

A. The hard way.

1. enter the lib/pfiles/ directory and open the file called plr_index. Remove all the entries from that file but make sure you leave the ~ in place on the last line. In some type of UNIX/Linux/CYGWIN environment you could use the command: “> plr_index” instead. Minus the “ “ of course. This effectively turns plr_index in to 0 byte file.

2. enter each of the lettered subdirectories and remove any files that exist. From the same lib/pfiles/ directory you can remove any of the player files in some type of UNIX/Linux/CYGWIN environment as follows: “find . –type f –exec rm {} \;” instead. Minus the “ “ of course.

3. boot up the mud and log in, the first person to do so is the new implementor

B. The easy way.

1. from a clean back-up copy the pfiles folder and all of the subdirectories in to the lib/ directory and overwrite anything that already exists there.
2. boot up the mud and congratulate yourself on taking 20 seconds to do a 10 minute job