DoomSpecs

COLLABORATORS			
	TITLE : DoomSpecs		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		November 28, 2024	

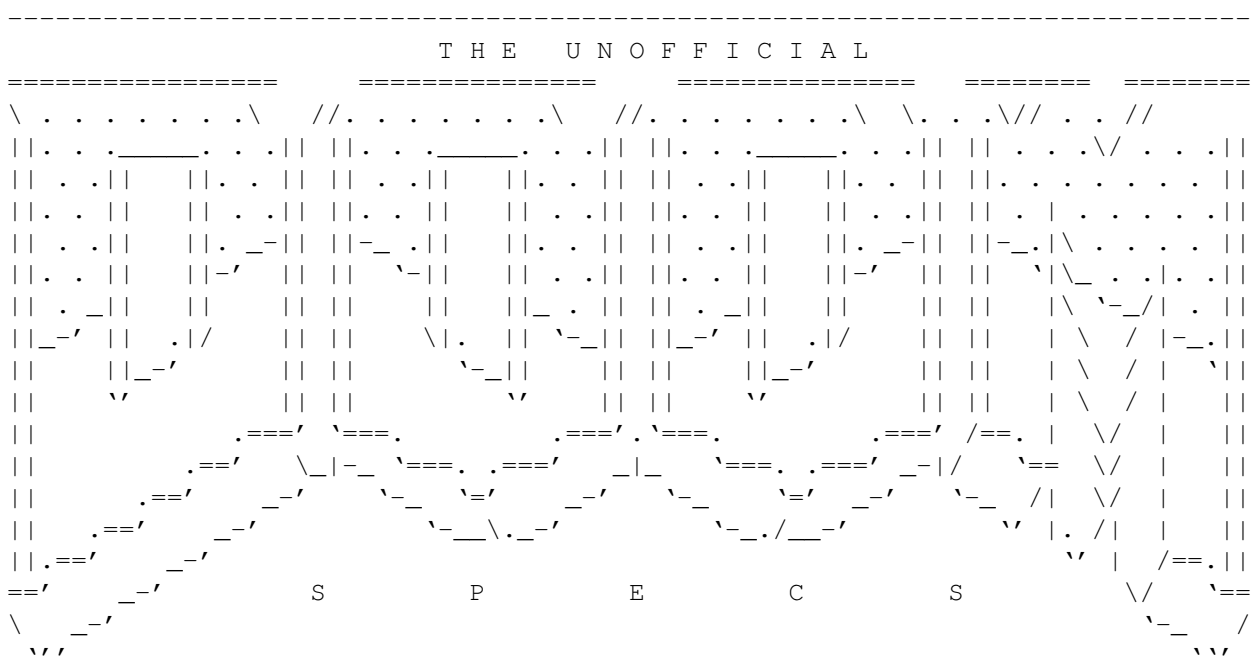
REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

DoomSpecs

1.1 main



Release v1.3 - April 13, 1994 EST
Written by: Matt Fell (matt.burnett@acebbs.com)
Distributed by: Hank Leukart (ap641@cleveland.freenet.edu)
"DOOM: Where hackers gnaw the bones left from the banquet
of data prepared by the mighty wizards of id."
"The poets talk about love, but what I talk about is DOOM,
because in the end, DOOM is all that counts."
- Alex Machine/George Stark/Stephen King, _The Dark Half_

```
----------
DISCLAIMER
----------
```

These specs are to aid in informing the public about the game DOOM, by id Software. In no way should this promote your killing yourself, killing others, or killing in any other fashion. Additionally, neither Hank Leukart

nor Matt Fell claim ANY responsibility regarding ANY illegal activity concerning this file, or indirectly related to this file. The information contained in this file only reflects id Software indirectly, and questioning id Software regarding any information in this file is not recommended.

COPYRIGHT NOTICE

INTRODUCTION FROM HANK LEUKART

Here are the long awaited unofficial specs for DOOM. These specs should be used for creating add-on software for the game. I would like to request that these specs be used in making utilities that ONLY work on the registered version of DOOM.

I did not write these specs. I am handling the distribution so Matt Fell is not bombarded with E-mail with requests for the specs, etc. If you would like a copy of the specs, E-mail Hank Leukart at "ap641@cleveland.freenet.edu" on the Internet. If you would like to ask technical questions or give technical suggestions about the specs, please write to Matt Fell at "matt.burnett@acebbs.com".

```
        Literature also written/distributed by Hank Leukart:

        - The "Official" DOOM FAQ:  A comprehensive guide to DOOM
        - DOOM iNsAnItY: A humorous look at DOOM and its players

--------
CONTENTS
--------

@{ "[1] Author's Notes" Link Chapter1 }
        [1-1] id Software's Copyright
        [1-2] What's New in the 1.3 Specs
        [1-3] Acknowledgments
@{ "[2] Basics" Link Chapter2 }
@{ "[3] Directory Overview" Link Chapter3 }
@{ "[4] The Maps, The Levels" Link Chapter4 }
        [4-1] ExMy
        [4-2] THINGS
                [4-2-1] Thing Types
                [4-2-2] Thing Attributes
        [4-3] LINEDEFS
                [4-3-1] Linedef Attributes
                [4-3-2] Linedef Types
        [4-4] SIDEDEFS
        [4-5] VERTEXES
        [4-6] SEGS
        [4-7] SSECTORS
        [4-8] NODES
        [4-9] SECTORS
                [4-9-1] Special Sector Types
        [4-10] REJECT
        [4-11] BLOCKMAP
                [4-11-1] Automatically Generating the BLOCKMAP
@{ "[5] Pictures" Link Chapter5 }
        [5-1] Headers
        [5-2] Pointers
        [5-3] Pixel Data
@{ "[6] Floor and Ceiling Textures" Link Chapter6 }
        [6-1] Animated floors, see [8-4-1]
@{ "[7] Songs and Sounds" Link Chapter7 }
        [7-1] Songs
        [7-2] Sounds
@{ "[8] Some Important Non-Picture Resources" Link Chapter8 }
        [8-1] PLAYPAL
        [8-2] COLORMAP
        [8-3] DEMOs
        [8-4] TEXTURE1 and TEXTURE2
                [8-4-1] Animated Walls
        [8-5] PNAMES
```

1.2 chapter1

```
--------------------------
CHAPTER [1]: Author's Notes
--------------------------
```

[1-1]: id Software's Copyright and the Shareware Version

The LICENSE.DOC says:

> 'You may not: rent, lease, modify, translate, disassemble, decompile, reverse engineer, or create derivative works based upon the Software. Notwithstanding the foregoing, you may create a map editor, modify maps and make your own maps (collectively referenced as the "Permitted Derivative Works") for the Software. You may not sell or distribute any Permitted Derivative Works but you may exchange the Permitted Derivative Works at no charge amongst other end-users.'
>
> '(except for backup purposes) You may not otherwise reproduce, copy or disclose to others, in whole or in any part, the Software.'

I think it is clear that you may not distribute a wad file that contains any of the original data resources from DOOM.WAD. A level that only has new things should be distributed as a pwad with only two entries in its directory (explained below, in chapter [2]) - e.g. E3M1 and THINGS. And the THINGS resource in the pwad should be substantially different from the original one in DOOM.WAD. You should not distribute any pwad files that contain episode one maps. Here's an excerpt from README.EXE:

> 'id Software respectfully requests that you do not modify the levels for the shareware version of DOOM. We feel that the distribution of new levels that work with the shareware version of DOOM will lessen a potential user's incentive to purchase the registered version.
>
> 'If you would like to work with modified levels of DOOM, we encourage you to purchase the registered version of the game.'

Recently, Jay Wilbur of id Software announced the formulation of a policy on third-party additions to the game. You can find the announcement on alt.games.doom, and probably lots of other places too. Or you can send me mail asking for a copy of the announcement. Basically, they are preparing a document, and if it was done, then I could tell you more, but it isn't finished at the time I'm writing this.

If you're making add-ons, plan on them not working on the shareware game, and plan on including statements about the trademarks and copyrights that id Software owns, as well as disclaimers that they won't support your add-on product, nor will they support DOOM after it has been modified.

[1-2]: What's New in the 1.3 Specs

The main reason for this release of the specs, 1.3, is of course the explanation of the NODES structure. I've been delaying a little bit, because I wanted to see if it would be feasible to include a good algorithm herein. Also, I wanted to wait and see if someone could actually implement "node theory" in a level editor, thereby verifying it.

Now the theory HAS been verified. However, the actual implementation is still being worked on (debugged) as I'm writing this. Also, I don't want to steal anyone's hard work outright. This means that there is NOT a node creation algorithm here, but I do outline how one can be done. I have tried to come up with one on my own, but it is too difficult for me, especially

with all the other things I'm simultaneously doing.

Where you WILL find pseudo-code is in the BLOCKMAP section. I borrowed an excellent idea from a contributor, and code based on the algorithm given here should be very fast. Even huge levels should recalculate in seconds.

Another new section completely explains the REJECT resource.

This entire document has been re-formatted, and there have been several other additions, and hopefully the last of the typos has been rooted out. I consider these specs to be at least 95% complete. There are only minor gaps in the information now. If the promised "official specifications" were released today, I expect this would compare favorably with them (although I know exactly what parts of it I would look to first).

I've been notified of something very disappointing, and after a couple weeks of trying there seems to be no way around it. The pictures that are used for sprites (things like barrels, demons, and the player's pistol) all have to be listed together in one .WAD file. This means that they don't work from pwad files. The same thing goes for the floor pictures. Luckily, the walls are done in a more flexible way, so they work in pwads. All this is explained in chapter [5].

[1-3]: Acknowledgments

I have received much assistance from the following people. They either brought mistakes to my attention, or provided additional information that I've incorporated into these specs:

Ted Vessenes (tedv@geom.umn.ed)
I had the THING angles wrong in the original specs.
Matt Tagliaferri (matt.tagliaferri@pcohio.com)
The author of the DOOMVB40 editor (aka DOOMCAD). I forgot to describe the TEXTURE1/2 pointer table in the 1.1 specs. Also, helped with linedef types, and provided a good BLOCKMAP algorithm.
Raphael Quinet (quinet@montefiore.ulg.ac.be)
The author of the NEWDEU editor, now DEU 5, the first editor that can actually do the nodes. Go get it. Gave me lots of rigorous contributions on linedef types and special sectors.
Robert Fenske (rfenske@swri.edu)
Part of the team that created the VERDA editor. Gave me a great list of the linedef attributes; also helped with linedef types, a blockmap list, special sectors, and general tips and suggestions.
John A. Matzen (jamatzen@cs.twsu.edu)
Instrument names in GENMIDI.
Jeff Bird (jeff@wench.ece.jcu.edu.au)
Good ideas and suggestions about the NODES, and a blockmap algorithm.
Alistair Brown (A.D.Brown@bradford.ac.uk)
Helped me understand the NODES; and told me how REJECT works.
Robert D. Potter (potter@bronze.lcs.mit.edu)
Good theory about what BLOCKMAP is for and how the engine uses it.
Joel Lucsy (jjlucsy@mtu.edu)
Info on COLORMAP and PLAYPAL.
Tom Nettleship (mastn@midge.bath.ac.uk)
I learned about BSP trees from his comp.graphics.algorithms messages.
Colin Reed (dyl@cix.compulink.co.uk)
I had the x upper and lower bounds for node bounding boxes backwards.
Frans P. de Vries (fpdevries@hgl.signaal.nl)
Thanks for the cool ASCII DOOM logo used for the header.

Thanks for all the help! Sorry if I left anyone out. If you have any comments or questions, have spotted any errors, or have any possible additions, please send me e-mail.

1.3 chapter2

```
------------------
CHAPTER [2]: Basics
------------------
```

There are two types of "wad" files. The original DOOM.WAD file is an "IWAD", or "Internal wad", meaning it contains all of the data necessary to play. The other type is the "PWAD" file, "Patch wad", an external file which has the same structure, but with far fewer entries in the directory (explained below). The data in a pwad is substituted for the original data in the DOOM.WAD, thus allowing for much easier distribution of new levels. Only those resources listed in the pwad's directory are changed, everything else stays the same.

A typical pwad might contain new data for a single level, in which case in would contain the 11 entries necessary to define the level. Pwad files need to have the extension .WAD, and the filename needs to be at least four characters: X.WAD won't work, but rename it XMEN.WAD, and it will work. Most of the levels available now are called something like E2L4BOB.WAD, meaning episode 2, level 4, by "Bob". I think a better scheme is the one just starting to be used now, two digits for the episode and level, then up to six letters for the level's name, or its creator's name. For example, if Neil Peart were to create a new level 6 for episode 3, he might call it 36NEILP.WAD.

To use this level instead of the original e3m6 level, a player would type DOOM -FILE 36NEILP.WAD at the command line, along with any other parameters. More than one external file can be added at the same time, thus in general:

```
DOOM -FILE [pwad_filename] [pwad_filename] [pwad_filename] ...
```

When the game loads, a "modified game" message will appear if there are any pwads involved, reminding the player that id Software will not give technical support or answer questions regarding modified levels.

A pwad file may contain more than one level or parts of levels, in fact there is apparently no limit to how many entries may be in a pwad. The original doom levels are pretty complicated, and they are from 50-200 kilobytes in size, uncompressed. Simple prototype levels are just a few k.

The first twelve bytes of a wad file are as follows:

```
Bytes 0 to 3    must contain the ASCII letters "IWAD" or "PWAD"
Bytes 4 to 7    contain a long integer which is the number of entries in the
                "directory"
Bytes 8 to 11   contain a pointer to the first byte of the "directory"
```

Bytes 12 to the start of the directory contain resource data. The directory referred to is a list, located at the end of the wad file, which contains the pointers, lengths, and names of all the resources in the wad file. The resources in the wad include item pictures, monster's pictures for animation, wall patches, floor and ceiling textures, songs, sound effects,

map data, and many others.
        As an example, the first 12 bytes of the DOOM.WAD file might be "49 57 41 44 d4 05 00 00 c9 fd 6c 00" (in hexadecimal). This is "IWAD", then 5d4 hex (=1492 decimal) for the number of directory entries, then 6cfdc9 hex (=7142857 decimal) for the first byte of the directory.
        Each directory entry is 16 bytes long, arranged this way:

```
First four bytes:       pointer to start of resource (a long integer)
Next four bytes:        length of resource (another long integer)
Last eight bytes:       name of resource, in ASCII letters, ending with
                        00s if less than eight bytes.
```

1.4 chapter3

```
------------------------------
CHAPTER [3]: Directory Overview
------------------------------
```

        This is a list of most of the directory entries. It would take 2000 lines to list every single entry, and that would be silly. All the ST entries are for status bar pictures, so why list every one? And the naming convention for the 700 sprites is easy (see chapter [5]), so there's no need to list them all individually.

```
PLAYPAL   contains fourteen 256 color palettes, used while playing Doom.
COLORMAP  maps colors in the palette down to darker ones, for areas of less
            than maximum brightness (quite a few of these places, huh?).
ENDOOM    is the text message displayed when you exit to DOS.
DEMOx     x=1-3, are the demos which will play if you just sit and watch.
E1M1      etc, to E3M9, along with its 10 subsequent entries, defines the
          map data for a single level or mission.
TEXTURE1  is a list of wall type names used in the SIDEDEF portion of each
            level , and their composition data, i.e. what wall patches make
            up each texture.
TEXTURE2  contains the walls that are only in the registered version.
PNAMES    is the list of wall patches, which are referenced by number in the
            TEXTURE1/2 resources.
GENMIDI   has the names of every General Midi standard instrument in order
            from 0-127. Anyone know more...?
DMXGUS    obviously has to do with Gravis Ultra Sound. It's a text file, easy
            to read. Just extract it (WadTool works nicely).
D_ExMy    is the music for episode x level y.
D_INTER   is the music played on the summary screen between levels.
D_INTRO   is the 4 second music played when the game starts.
D_INTROA  is also introductory music.
D_VICTOR  is the music played on the victory text-screen after an episode.
D_BUNNY   is music for while a certain rabbit has his story told...
DP_xxxxx  DP and DS come in pairs and are the sound effects. DP_ are the PC
DS_xxxxx    speaker sounds, DS_ are the sound card sounds.
```

        All the remaining entries in the directory, except the floor textures at the end, and the "separators" like S_START, refer to resources which are pictures, in the doom/wad picture format described in chapter [5]. The floor textures are also pictures, but in a raw format described in chapter [6].
        The next seven are full screen (320 by 200 pixel) pictures:

```
HELP1     The ad-screen that says Register!, with some screen shots.
HELP2     The actual help, all the controls explained.
TITLEPIC  Maybe this is the title screen? Gee, I dunno...
CREDIT    The credits, the people at id Software who created this great game.
VICTORY2  The screen shown after a victorious end to episode 2.
PFUB1     A nice little rabbit minding his own peas and queues...
PFUB2     ...maybe a hint of what's waiting in Doom Commercial version.
ENDx      x=0-6, "THE END" text, with (x) bullet holes.
AMMNUMx   x=0-9, are the gray digits used in the status bar for ammo count.
STxxxxxx  are small pictures and text used on the status bar.
M_xxxxxx  are text messages (yes, in picture format) used in the menus.
BRDR_xxx  are tiny two pixel wide pictures use to frame the viewing window
            when it is not full screen.
WIxxxxxx  are pictures and messages used on the summary screen after
            the completion of a level.
WIMAPx    x=0-2, are the summary-screen maps used by each episode.
S_START   has 0 length and is right before the item/monster "sprite" section.
            See chapter [5] for the naming convention used here.
S_END     is immediately after the last sprite.
P_START   marks the beginning of the wall patches.
P1_START    before the first of the shareware wall patches.
P1_END      after the last of the shareware wall patches.
P2_START    before the first of the registered wall patches.
P2_END      before the first of the registered wall patches.
P_END     marks the end of the wall patches.
F_START   marks the beginning of the floors.
F1_START    before the first shareware floor texture.
F1_END      after the last shareware floor texture.
F2_START    before the first registered floor texture.
F2_END      after the last registered floor texture.
F_END     marks the end of the floors.
```

And that's the end of the directory.

It is possible to include other entries and resources in a wad file, e.g. an entry called CLOWNS could point to a resource that includes the level creator's name, date of completion, or a million other things. None of these non-standard entries will be used by DOOM, nor will they cause it problems. Some of the map editors currently out add extra entries. There is a debate going on right now as to the merits of these extras. Since they are all non-standard, and potentially confusing, for now I'm in favor of not using any extra entries, and instead passing along a text file with a pwad. However, I can see some possible advantages, and I might change my mind...

1.5 chapter4

```
---------------------------------
CHAPTER [4]: The Maps, The Levels
---------------------------------
```

Each level needs eleven resources/directory entries: E[x]M[y], THINGS, LINEDEFS, SIDEDEFS, VERTEXES, SEGS, SSECTORS, NODES, SECTORS, REJECT, and BLOCKMAP.
In the DOOM.WAD file, all of these entries are present for every

level. In a pwad external file, they don't all need to be present. Whichever entries are in a pwad will be substituted for the originals. For example, a pwad with just two entries, E3M1 and THINGS, would use all the walls and such from the original E3M1, but could have a completely different set of THINGS.

A note on the coordinates: the coordinate system used for the vertices and the heights of the sectors corresponds to pixels, for purposes of texture- mapping. So a sector that's 128 high, or a multiple of 128, is pretty typical, since many wall textures are 128 pixels high.

[4-1]: ExMy

x is a single digit 1-3 for the episode number and y is 1-9 for the mission/level number.

This is just the name resource for a (single) level, and has zero length. It marks any map-data resources that follow it in the directory list as being components of that level. The assignment of resources to this level stops with either the next ExMy entry, or with a non-map entry like TEXTURE1.

[4-2]: THINGS

Each thing is ten bytes, consisting of five (integer) fields:

(1) X coordinate of thing
(2) Y coordinate of thing
(3) Angle the thing faces. On the automap, 0 is east, 90 is north, 180 is west, 270 is south. This value is only used for monsters, player starts, deathmatch random starts, and teleporter incoming spots. Others look the same from all directions. Values are rounded to the nearest 45 degree angle, so if the value is 80, it will actually face 90 - north.
(4) Type of thing, see next subsection, [4-2-1]
(5) Attributes of thing, see [4-2-2]

[4-2-1]: Thing Types

Bytes 6-7 of each thing are an integer which specifies its kind:

Dec/Hex The thing's number
Sprite The sprite name associated with this thing. This is the first four letters of the directory entries that are pictures of this thing.
seq. The sequence of frames displayed. "-" means it displays nothing. Unanimated things will have just an "a" here, e.g. a backpack's only sprite can be found in the wad under BPAKA0. Animated things will show the order that their frames are displayed (they cycle back after the last one). So the blue key uses BKEYA0 then BKEYB0, etc. The soulsphere uses SOULA0-SOULB0-C0-D0-C0-B0 then repeats.

Thing 15, a dead player, is PLAYN0.
+ Monsters and players and barrels. They can be hurt, and they have a more complicated sprite arrangement. See chapter [5].
CAPITAL Monsters, counts toward the KILL ratio at the end of a level.
* An obstacle, players and monsters can't move through it.
^ Hangs from the ceiling, or floats (if a monster).
$ A regular item that players may get.
! An artifact item; counts toward the ITEM ratio at level's end.

```
Dec. Hex Sprite seq.    Thing is:

0    0000 ---- -        (nothing)
1    0001 PLAY +        Player 1 start (Player 1 start is needed even on)
2    0002 PLAY +        Player 2 start (levels intended for deathmatch only.)
3    0003 PLAY +        Player 3 start (Player starts 2-4 are only needed
for)
4    0004 PLAY +        Player 4 start (cooperative mode multiplayer games.)
5    0005 BKEY ab     $ Blue keycard
6    0006 YKEY ab     $ Yellow keycard
7    0007 SPID +      * SPIDER DEMON: giant walking brain boss
8    0008 BPAK a      $ Backpack
9    0009 SPOS +      * FORMER HUMAN SERGEANT: black armor shotgunners
10   000a PLAY w        Bloody mess (an exploded player)
11   000b ---- -        Deathmatch start positions. Must be at least 4/level.
12   000c PLAY w        Bloody mess, this thing is exactly the same as 10
13   000d RKEY ab     $ Red Keycard
14   000e ---- -        Marks the spot where a player (or monster) lands when
                        they teleport to the SECTOR that contains this thing.
15   000f PLAY n        Dead player
16   0010 CYBR +      * CYBER-DEMON: robo-boss, rocket launcher
17   0011 CELP a      $ Cell charge pack
18   0012 POSS a        Dead former human
19   0013 SPOS a        Dead former sergeant
20   0014 TROO a        Dead imp
21   0015 SARG a        Dead demon
22   0016 HEAD a        Dead cacodemon
23   0017 SKUL a        Dead lost soul, invisible (they blow up when killed)
24   0018 POL5 a        Pool of blood
25   0019 POL1 a      * Impaled human
26   001a POL6 ab     * Twitching impaled human
27   001b POL4 a      * Skull on a pole
28   001c POL2 a      * 5 skulls shish kebob
29   001d POL3 ab     * Pile of skulls and candles
30   001e COL1 a      * Tall green pillar
31   001f COL2 a      * Short green pillar
32   0020 COL3 a      * Tall red pillar
33   0021 COL4 a      * Short red pillar
34   0022 CAND a        Candle
35   0023 CBRA a      * Candelabra
36   0024 COL5 ab     * Short green pillar with beating heart
37   0025 COL6 a      * Short red pillar with skull
38   0026 RSKU ab     $ Red skullkey
39   0027 YSKU ab     $ Yellow skullkey
40   0028 BSKU ab     $ Blue skullkey
41   0029 CEYE abcb   * Eye in symbol
42   002a FSKU abc    * Flaming skull-rock
43   002b TRE1 a      * Gray tree
44   002c TBLU abcd   * Tall blue firestick
45   002d TGRE abcd   * Tall green firestick
46   002e TRED abcd   * Tall red firestick
47   002f SMIT a      * Small brown scrub
48   0030 ELEC a      * Tall, techno column
49   0031 GOR1 abcb   *^Hanging victim, swaying, legs gone
50   0032 GOR2 a      *^Hanging victim, arms out
51   0033 GOR3 a      *^Hanging victim, 1-legged
```

```
52   0034 GOR4 a      *^Hanging victim, upside-down, upper body gone
53   0035 GOR5 a      *^Hanging severed leg
54   0036 TRE2 a      * Large brown tree
55   0037 SMBT abcd   * Short blue firestick
56   0038 SMGT abcd   * Short green firestick
57   0039 SMRT abcd   * Short red firestick
58   003a SARG +      * SPECTRE: invisible version of the DEMON
59   003b GOR2 a       ^Hanging victim, arms out
60   003c GOR4 a       ^Hanging victim, upside-down, upper body gone
61   003d GOR3 a       ^Hanging victim, 1-legged
62   003e GOR5 a       ^Hanging severed leg
63   003f GOR1 abcb    ^Hanging victim, swaying, legs gone
2001 07d1 SHOT a      $ Shotgun
2002 07d2 MGUN a      $ Chaingun, gatling gun, mini-gun, whatever
2003 07d3 LAUN a      $ Rocket launcher
2004 07d4 PLAS a      $ Plasma gun
2005 07d5 CSAW a      $ Chainsaw
2006 07d6 BFUG a      $ BFG9000
2007 07d7 CLIP a      $ Ammo clip
2008 07d8 SHEL a      $ 4 shotgun shells
2010 07da ROCK a      $ 1 rocket
2011 07db STIM a      $ Stimpak
2012 07dc MEDI a      $ Medikit
2013 07dd SOUL abcdcb ! Soulsphere, Supercharge, +100% health
2014 07de BON1 abcdcb ! Health bonus
2015 07df BON2 abcdcb ! Armor bonus
2018 07e2 ARM1 ab     $ Green armor 100%
2019 07e3 ARM2 ab     $ Blue armor 200%
2022 07e6 PINV abcd   ! Invulnerability
2023 07e7 PSTR a      ! Berserk Strength
2024 07e8 PINS abcd   ! Invisibility
2025 07e9 SUIT a      ! Radiation suit
2026 07ea PMAP abcdcb ! Computer map
2028 07ec COLU a      * Floor lamp
2035 07f3 BAR1 ab+    * Barrel; blown up (BEXP sprite) no longer an obstacle.
2045 07fd PVIS ab     ! Lite goggles
2046 07fe BROK a      $ Box of Rockets
2047 07ff CELL a      $ Cell charge
2048 0800 AMMO a      $ Box of Ammo
2049 0801 SBOX a      $ Box of Shells
3001 0bb9 TROO +      * IMP: brown fireball hurlers
3002 0bba SARG +      * DEMON: pink bull-like chewers
3003 0bbb BOSS +      * BARON OF HELL: cloven hooved minotaur boss
3004 0bbc POSS +      * FORMER HUMAN: regular pistol shooting slimy human
3005 0bbd HEAD +      *^CACODEMON: red one-eyed floating heads. Behold...
3006 0bbe SKUL +      *^LOST SOUL: flying flaming skulls, they really bite
```

I couldn’t figure out a way to squeeze the following information into the above table. RAD is the thing’s radius, they’re all circular for collision purposes. HT is its height, for purposes of crushing ceilings and testing if monsters or players are too tall to enter a sector. SPD is a monster’s speed. So now you know that a player is 56 units tall. Even though this table implies that they’re 16*2 wide, players can’t enter a corridor that’s 32 wide. It must be at least 34 units wide (48 is the lowest width divisible by 16). Although obstacles and monsters have heights, they are infinitely tall for purposes of a player trying to go through them.

Dec.	Hex	RAD	HT	SPD	Thing or class of things:
-	-	16	56	-	Player
7	0007	128	100	12	Spider-demon
9	0009	20	56	8	Former sergeant
16	0010	40	110	16	Cyber-demon
58	003a	30	56	8	Spectre
3001	0bb9	20	56	8	Imp
3002	0bba	30	56	8	Demon
3003	0bbb	24	64	8	Baron of Hell
3004	0bbc	20	56	8	Former human
3005	0bbd	31	56	8	Cacodemon
3006	0bbe	16	56	8	Lost soul
2035	07f3	10	42		barrel
		20	16		all gettable things
		16	16		most obstacles
54	0036	32	16		large brown tree

[4-2-2]: Thing attributes

The last two bytes of a THING control a few attributes, according to which bits are set:

bit 0 the THING is present at skill 1 and 2
bit 1 the THING is present at skill 3 (hurt me plenty)
bit 2 the THING is present at skill 4 and 5 (ultra-violence, nightmare)
bit 3 indicates a deaf guard.
bit 4 means the THING only appears in multiplayer mode.
bits 5-15 have no effect.

The skill settings are most used with the monsters, of course...the most common skill level settings are hex 07/0f (on all skills), 06/0e (on skill 3-4-5), and 04/0c (only on skill 4-5).

"deaf guard" only has meaning for monsters, who will not attack until they see a player if they are deaf. Otherwise, they will activate when they hear gunshots, etc (including the punch!). Sound does not travel through solid walls (walls that are solid at the time of the noise). Also, lines can be set so that sound does not pass through them (see [4-3-1] bit 6). This attribute is also known as the "ambush" attribute.

[4-3]: LINEDEFS
===

Each linedef represents a line from one of the VERTEXES to another, and each is 14 bytes, containing 7 (integer) fields:

(1) from the VERTEX with this number (the first vertex is 0).
(2) to the VERTEX with this number (31 is the 32nd vertex).
(3) attributes, see [4-3-1] below.
(4) types, see [4-3-2] below.
(5) is a "trigger" or "tag" number which ties this line's effect type to all SECTORS that have the same trigger number in their last field.
(6) "right" SIDEDEF, indexed number.
(7) "left" SIDEDEF, if this line adjoins 2 SECTORS. Otherwise, it is equal to -1 (FFFF hex).

"right" and "left" are based on the direction of the linedef as indicated by the "from" and "to" VERTEXES. This attempt at a sketch should make it clear what I mean:

```
            left side              right side
    from ------------------> to <------------------ from
            right side             left side
```

IMPORTANT: All lines must have a right side. If it is a one-sided line, then it must go the proper direction, so its single side is facing the sector it is part of.

[4-3-1]: Linedef Attributes

The third field of each linedef is an integer which controls that line's attributes with bits, as follows:

bit # condition if it is set (=1)

bit 0 Impassable. Players and monsters cannot cross this line, and projectiles explode or end if they hit this line. Note, however, that if there is no sector on the other side, things can't go through this line anyway.

bit 1 Monster-blocker. Monsters cannot cross this line.

bit 2 Two-sided. If this bit is set, then the linedef's two sidedefs can have "-" as a texture, which means "transparent". If this bit is not set, the sidedefs can't be transparent: if "-" is viewed, it will result in the hall of mirrors effect. However, the linedef CAN have two non-transparent sidedefs, even if this bit is not set, if it is between two sectors.
Another side effect of this bit is that if it is set, then projectiles and gunfire (pistol, etc.) can go through it if there is a sector on the other side, even if bit 0 is set.
Also, monsters see through these lines, regardless of the line's other attribute settings and its textures ("-" or not doesn't matter).

bit 3 Upper texture is "unpegged". This is usually done at windows. "Pegged" textures move up and down when the neighbor sector moves up or down. For example, if a ceiling comes down, then a pegged texture on its side will move down with it so that it looks right. If the side isn't pegged, it just sits there, the new material is spontaneously created. The best way to get an idea of this is to change a linedef on an elevator or door, which are always pegged, and observe the result.

bit 4 Lower texture is unpegged.

bit 5 Secret. The automap will draw this line like a normal solid line that doesn't have anything on the other side, thus protecting the secret until it is opened. This bit is NOT what determines the SECRET ratio at the end of a level, that is done by special sectors (#9).

bit 6 Blocks sound. Sound cannot cross a line that has this bit set. Sound normally travels from sector to sector, so long as the floor and ceiling heights allow it (e.g. sound wouldn't go from a sector with 0/64 floor/ceiling height to one with 72/128, but sound WOULD go from a sector with 0/64 to one with 56/128).

bit 7 Not on map. The line is not shown on the regular automap, not even if the computer all-map power up is gained.

bit 8 Already on map. When the level is begun, this line is already on the

automap, even though it hasn't been seen (in the display) yet.

bits 9-15 are unused, EXCEPT for a large section of e2m7, where every wall on the border of the section has bits 9-15 set, so they have values like 1111111000000000 (-511) and 1111111000010000 (-495). This area of e2m7 is also the only place in all 27 levels where there is a linedef 4 value of -1. But the linedef isn't a switch. These unique values either do nothing, or something that is as yet unknown. The currently prevailing opinion is that they do nothing.

Another rare value used in some of the linedef's attribute fields is ZERO. It occurs only on one-sided walls, where it makes no difference whether or not the impassibility bit (bit 0) is set. Still, it seems to indicate a minor glitch in the DOOM-CAD editor (on the NExT), I suppose.

[4-3-2]: Linedef Types

The integers in field 4 of a linedef control various special effects, mostly to do with what happens to a triggered SECTOR when the line is crossed or activated by a player.

Except for the ones marked DOOR, the end-level switches, and the special type 48 (hex 30), all these effects need trigger/tag numbers. Then any and all sectors whose last field contains the same trigger number are affected when the linedef's function is activated.

All functions are only performed from the RIGHT side of a linedef. Thus switches and doors can only be activated from the right side, and teleporter lines only work when crossed from the right side.

What the letters in the ACT column mean:

W means the function happens when a player WALKS across the linedef.
S means a player must push SPACEBAR near the linedef to activate it (doors and switches).
G means a player or monster must shoot the linedef with a pistol or shotgun.
1 means it works once only.
R means it is a repeatable function.

Some functions that appear to work only once can actually be made to work again if the conditions are reset. E.g. a sector ceiling rises, opening a little room with baddies in it. Usually, that's it. But perhaps if some other linedef triggered that sector ceiling to come down again, then the original trigger could make it go up, etc...

Doors make a different noise when activated than the platform type (floor lowers and rises), the ones that exhibit the door-noise are so called in the EFFECT column. But only the ones marked DOOR in capitals don't need trigger numbers.

Dec/Hex	ACT	EFFECT
-1 ffff	? ?	None? Only once in whole game, on e2m7, (960,768)-(928,768)
0 00	- -	nothing
1 01	S R	DOOR. Sector on the other side of this line has its ceiling rise to 8 below the first neighbor ceiling on the way up, then comes back down after 6 seconds.
2 02	W 1	Open door (stays open)
3 03	W 1	Close door
5 05	W 1	Floor rises to match highest neighbor's floor height.
7 07	S 1	Staircase rises up from floor in appropriate sectors.

```
8    08   W 1   Stairs
```

Note: The stairs function requires special handling. An even number of steps will be raised, starting with the first sector that has the same trigger number as this linedef. Then the step sector's trigger number alternates between 0 and any other value. The original maps use either 99 or 999, and this is wise. The steps don't all have to start at the same altitude. All the steps rise up 8, then all but the first rise another 8, etc. If a step hits a ceiling, it stops. Some interesting effects are possible with steps that aren't the same size or shape as previous steps, but in general, the most reliable stairways will be just like the originals, congruent rectangles.

```
9    09   S 1   Floor lowers; neighbor sector's floor rises and changes
                TEXTURE and sector type to match surrounding neighbor.
10   0a   W 1   Floor lowers for 3 seconds, then rises
11   0b   S -   End level. Go to next level.
13   0d   W 1   Brightness goes to 255
14   0e   S 1   Floor rises to 64 above neighbor sector's floor
16   10   W 1   Close door for 30 seconds, then opens.
18   12   S 1   Floor rises to equal first neighbor floor
19   13   W 1   Floor lowers to equal neighboring sector's floor
20   14   S 1   Floor rises, texture and sector type change also.
21   15   S 1   Floor lowers to equal neighbor for 3 seconds, then rises back
                up to stop 8 below neighbor's ceiling height
22   16   W 1   Floor rises, texture and sector type change also
23   17   S 1   Floor lowers to match lowest neighbor sector
26   1a   S R   DOOR. Need blue key to open. Closes after 6 seconds.
27   1b   S R   DOOR. Yellow key.
28   1c   S R   DOOR. Red key.
29   1d   S 1   Open door, closes after 6 seconds
30   1e   W 1   Floor rises to 128 above neighbor's floor
31   1f   S 1   DOOR. Stays open.
32   20   S 1   DOOR. Blue key. Stays open.
33   21   S 1   DOOR. Yellow key. Stays open.
34   22   S 1   DOOR. Red key. Stays open.
35   23   W 1   Brightness goes to 0.
36   24   W 1   Floor lowers to 8 above next lowest neighbor
37   25   W 1   Floor lowers, change floor texture and sector type
38   26   W 1   Floor lowers to match neighbor
39   27   W 1   Teleport to sector. Only ONE sector can have the same tag #
40   28   W 1   Ceiling rises to match neighbor ceiling
41   29   S 1   Ceiling lowers to floor
42   2a   S R   Closes door
44   2c   W 1   Ceiling lowers to 8 above floor
46   2e   G 1   Opens door (stays open)
48   30   - -   Animated, horizontally scrolling wall.
51   33   S -   End level. Go to secret level 9.
52   34   W -   End level. Go to next level
56   38   W 1   Crushing floor rises to 8 below neighbor ceiling
58   3a   W 1   Floor rises 32
59   3b   W 1   Floor rises 8, texture and sector type change also
61   3d   S R   Opens door
62   3e   S R   Floor lowers for 3 seconds, then rises
63   3f   S R   Open door, closes after 6 seconds
70   46   S R   Sector floor drops quickly to 8 above neighbor
73   49   W R   Start crushing ceiling, slow crush but fast damage
74   4a   W R   Stops crushing ceiling
```

```
75   4b   W R   Close door
76   4c   W R   Close door for 30 seconds
77   4d   W R   Start crushing ceiling, fast crush but slow damage
80   50   W R   Brightness to maximum neighbor light level
82   52   W R   Floor lowers to equal neighbor
86   56   W R   Open door (stays open)
87   57   W R   Start moving floor (up/down every 5 seconds)
88   58   W R   Floor lowers quickly for 3 seconds, then rises
89   59   W R   Stops the up/down syndrome started by #87
90   5a   W R   Open door, closes after 6 seconds
91   5b   W R   Floor rises to 8 below neighbor ceiling
97   61   W R   Teleport to sector. Only ONE sector can have the same tag #
98   62   W R   Floor lowers to 8 above neighbor
102  66   S 1   Floor lowers to equal neighbor

103  67   S 1   Opens door (stays open)
104  68   W 1   Light drops to lowest light level amongst neighbor sectors
```

[4-4]: SIDEDEFS

A sidedef is a definition of what wall texture to draw along a LINEDEF, and a group of sidedefs define a SECTOR.

There will be one sidedef for a line that borders only one sector, since it is not necessary to define what the doom player would see from the other side of that line because the doom player can't go there. The doom player can only go where there is a sector.

Each sidedef is 30 bytes, comprising 2 (integer) fields, then 3 (8-byte string) fields, then a final (integer) field:

(1) X offset for pasting the appropriate wall texture onto the wall's "space": positive offset moves into the texture, so the left portion gets cut off (# of columns off left side = offset). Negative offset moves texture farther right, in the wall's space
(2) Y offset: analogous to the X, for vertical.
(3) "upper" texture name: the part above the juncture with a lower ceiling of an adjacent sector.
(4) "lower" texture name: the part below a juncture with a higher floored adjacent sector.
(5) "full" texture name: the regular part of the wall
(6) SECTOR that this sidedef faces or helps to surround

The texture names are from the TEXTURE1/2 resources. 00s fill the space after a wall name that is less than 8 characters. The names of wall patches in the directory are not directly used, they are referenced through the PNAMES.

Simple sidedefs have no upper or lower texture, and so they will have "-" instead of a texture name. Also, two-sided lines can be transparent, in which case "-" means transparent (no texture).

If the wall is wider than the texture to be pasted onto it, then the texture is tiled horizontally. A 64-wide texture will be pasted at 0, 64, 128, etc. If the wall is taller than the texture, than the same thing is done, it is vertically tiled, with one very important difference: it starts new ones ONLY at 128, 256, 384, etc. So if the texture is less than 128 high, there will be junk filling the undefined areas, and it looks ugly.

[4-5]: VERTEXES

==========

These are the beginnings and ends for LINEDEFS and SEGS, each is 4 bytes, 2 (integer) fields:

(1) X coordinate
(2) Y coordinate

On the automap within the game, with the grid on (press 'G'), the lines are hex 80 (decimal 128) apart, two lines = hex 100, dec 256.

[4-6]: SEGS
==========

The SEGS are in a sequential order determined by the SSECTORS, which are part of the NODES recursive tree. Each seg is 12 bytes, having 6 (integer)
fields:

(1) from VERTEX with this number
(2) to VERTEX
(3) angle: 0= east, 16384=north, -16384=south, -32768=west.
In hex, it's 0000=east, 4000=north, 8000=west, c000=south.
This is also know as BAMS for Binary Angle Measurement.
(4) LINEDEF that this seg goes along
(5) 0 = this seg is on the right SIDEDEF of the linedef.
1 = this seg is on the left SIDEDEF.
This could also be thought of as direction: 0 if the seg goes the same direction as the linedef it's on, 1 if it goes the opposite direction.
(6) Offset distance along the linedef to the start of this seg (the vertex in field 1). The offset is in the same direction as the seg. If field 5 is 0, then the distance is from the "from" vertex of the linedef to the "from" vertex of the seg. If feild 5 is 1, it is from the "to" vertex of the linedef to the "from" vertex of the seg. So if the seg begins at one of the two endpoints of the linedef, this will be 0.

For diagonal segs, the offset distance can be obtained from the formula DISTANCE = SQR((x2 - x1)^2 + (y2 - y1)^2). The angle can be calculated from the inverse tangent of the dx and dy in the vertices, multiplied to convert PI/2 radians (90 degrees) to 16384. And since most arctan functions return a value between -(pi/2) and (pi/2), you'll have to do some tweaking based on the sign of (x2-x1), to account for segs that go "west".

[4-7]: SSECTORS
==========

SSECTOR stands for sub-sector. These divide up all the SECTORS into convex polygons. They are then referenced through the NODES resources. There will be (number of nodes) + 1 ssectors. Each ssector is 4 bytes, having 2 (integer) fields:

(1) This many SEGS are in this SSECTOR...
(2) ...starting with this SEG number

[4-8]: NODES
==========

The full explanation of the nodes follows this list of a node's structure in the wad file. Each node is 28 bytes, composed of 14 (integer) fields:

```
(1)  X coordinate of partition line's start
(2)  Y coordinate of partition line's start
(3)  DX, change in X to end of partition line
(4)  DY, change in Y to end of partition line
       64, 128, -64, -64 would be a partition line from (64,128) to (0,64)
(5)  Y upper bound for right bounding-box.\
(6)  Y lower bound                         All SEGS in right child of node
(7)  X lower bound                         must be within this box.
(8)  X upper bound                        /
(9)  Y upper bound for left bounding box. \
(10) Y lower bound                         All SEGS in left child of node
(11) X lower bound                         must be within this box.
(12) X upper bound                        /
(13) a NODE or SSECTOR number for the right child. If bit 15 is set, then the
       rest of the number represents the child SSECTOR. If not, the child is
       a recursed node.
(14) a NODE or SSECTOR number for the left child.
```

The NODES resource is by far the most difficult to understand of all the data structures in DOOM. A new level won't display right without a valid set of precalculated nodes, ssectors, and segs. This is why so much time has passed without a fully functional map-editor, even though many people are working on them.

Here I will explain what the nodes are for, and how they can be generated automatically from the set of linedefs, sidedefs, and vertices. I do NOT have a pseudo-code algorithm. There are many reasons for this. I'm not actually programming a level editor myself, so I have no way of testing and debugging a fully elaborated algorithm. Also, it is a very complicated process, and heavily commented code would be very long. I'm not going to put any language-specific code in here either, since it would be of limited value. Finally, there are some implementations of automatic node generation out there, but they are still being worked on, i.e. they still exhibit some minot bugs.

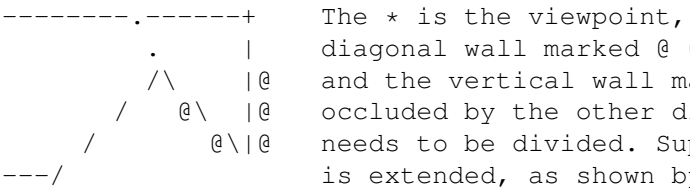
The NODES are branches in a binary space partition (BSP) that divides up the level and is used to determine which walls are in front of others, a process know as hidden-surface removal. The SSECTORS (sub-sectors) and SEGS (segments) resources are necessary parts of the structure.

A BSP tree is normally used in 3d space, but DOOM uses a simplified 2d version of the scheme. Basically, the idea is to keep dividing the map into smaller spaces until each of the smallest spaces contains only wall segments which cannot possibly occlude (block from view) other walls in its own space. The smallest, undivided spaces will become SSECTORS. Each wall segment is part or all of a linedef (and thus a straight line), and becomes a SEG. All of the divisions are kept track of in a binary tree structure, which is used to greatly speed the rendering process (drawing what is seen).

Only the SECTORS need to be divided. The parts of the levels that are "outside" sectors are ignored. Also, only the walls need to be kept track of. The sides of any created ssectors which are not parts of linedefs do not become segs.

Some sectors do not require any dividing. Consider a square sector. All the walls are orthogonal to the floor (the walls are all straight up and down), so from any viewpoint inside the square, none of its four walls can possibly block the view of any of the others. Now imagine a sector shaped

like this drawing:



now each of the two resulting sub-sectors are sufficient, because while in either one, no wall that is part of that sub-sector blocks any other.

In general, being a convex polygon is the goal of a ssector. Convex means a line connecting any two points that are inside the polygon will be completely contained in the polygon. All triangles and rectangles are convex, but not all quadrilaterals. In doom's simple Euclidean space, convex also means that all the interior angles of the polygon are <= 180 degrees.

Now, an additional complication arises because of the two-sided linedef. Its two sides are in different sectors, so they will end up in different ssectors too. Thus every two-sided linedef becomes two segs (or more), or you could say that every sidedef becomes a seg. Creating segs from sidedefs is a good idea, because the seg can then be associated with a sector. Two segs that aren't part of the same sector cannot possibly be in the same ssector, so further division is required of any set of segs that aren't all from the same sector.

Whenever a division needs to be made, a SEG is picked, somewhat arbitrarily, which along with its imaginary extensions, forms a "knife" that divides the remaining space in two (thus binary). This seg is the partition line of a node, and the remaining spaces on either side of the partition line become the right and left CHILDREN of the node. All partition lines have a direction, and the space on the "right" side of the partition is the right child of the node; the space on the "left" is the left child (there's a cute sketch in [4-3]: LINEDEFS that shows how right and left relate to the start and end of a line). Note that if there does not exist a seg in the remaining space which can serve as a partition line, then there is no need for a further partition, i.e. it's a ssector and a "leaf" on the node tree.

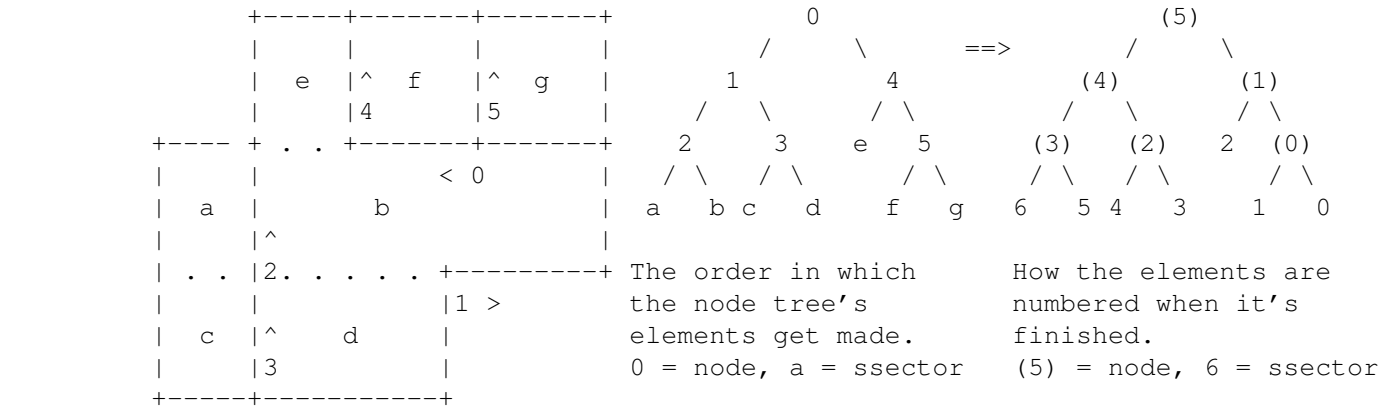
If the "knife" passes through any lines/segs (but not at vertices), they are split at the intersection, with one part going to each child. A two sided linedef, which is two segs, when split results in four segs. A two sider that lies along an extension of the partition line has each of its two segs go to opposite sides of the partition line. This is the eventual fate of ALL segs on two-sided linedefs.

As the partition lines are picked and the nodes created, a strict ordering must be maintained. The node tree is created from the "top" down. After the first division is made, then the left child is divided, then its left child, and so on, until a node's child is a ssector. Then you move back up the tree one branch, and divide the right child, then its left, etc. ALWAYS left first, on the way down.

Since there will be splits along the way, there is no way to know ahead of time how many nodes and ssectors there will be at the end. And the top of the tree, the node that is created first, is given the highest number. So as nodes and ssectors are created, they are simply numbered in order from 0 on up, and when it's all done, nothing's left but ssectors, then ALL the numbers, for nodes and ssectors, are reversed. If there's 485 nodes, then 485 becomes 0 and 0 becomes 485.

Here is another fabulous drawing which will explain everything. + is a vertex, - and | indicate linedefs, the . . indicates an extension of a partition line. The <, >, and ^ symbols indicate the directions of partition

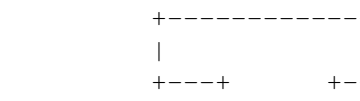
lines. All the space within the drawing is actual level space, i.e. sectors.



1. Make segs from all the linedefs. There are 5 two-sided lines here.
2. Pick the vertex at 0 and go west (left). This is the first partition line. Note the . . extension line.
3. Pick the vertex at 1, going east. The backwards extension . . cuts the line 3>2>, and the unlabeled left edge line. The left edge was one seg, it becomes two. The 3>2> line was two segs, it becomes four. New vertices are created at the intersection points to do this.
4. Pick the (newly created) vertex at 2. Now the REMAINING spaces on both sides of the partition line are suitable for ssectors. The left one is first, it becomes a, the right b. Note that ssector a has 3 segs, and ssector b has 5 segs. The . . imaginary lines are NOT segs.
5. Back up the tree, and take 1's right branch. Pick 3. Once again, we can make 2 ssectors, c and d, 3 segs each. Back up the tree to 0.
6. Pick 4. Now the left side is a ssector, it becomes e. But the right side is not, it needs one more node. Pick 5, make f and g.
7. All done, so reverse all the ordination of the nodes and the ssectors. Ssector 0's segs become segs 0-2, ssector 1's segs become segs 3-7, etc. The segs are written sequentially according to the ssector numbering.

If we want to create an algorithm to do the nodes automatically, it needs to be able to pick partition lines automatically. From studying the original maps, it appears that they usually choose a linedef which divides the child's space roughly in "half". This is restricted by the availability of a seg in a good location, with a good angle. Also, the "half" refers to the total number of ssectors in any particular child, which we have no way of knowing when we start! Optimization methods are probably used, or maybe brute force, trying every candidate seg until the "best" one is found.

What is the best possible choice for a partition line? Well, there are apparently two goals when creating a BSP tree, which are partially exclusive. One is to have a balanced tree, i.e. for any node, there are about the same total number of sub-nodes on either side of it. The other goal is to minimize the number of "splits" necessary, in this case, the number of seg splits needed, along with the accompanying new vertexes and extra segs. Only a very primitive and specially constructed set of linedefs could avoid having any splits, so they are inevitable. It's just that with some choices of partition lines, there end up being fewer splits. For example,



If a and b are chosen as partition lines, there will be four extra vertices needed, and this shape becomes five ssectors and 16 segs. If A and B are chosen,

```
    |^    ^|           however, there are no extra vertices, and only three
+---+a    b+---+ < B   ssectors and 12 segs.
|              |
+--------------+
```

I've read that for a "small" number of polygons (less than 1000?), which is what we're dealing with in a doom level, one should definitely try to minimize splits, and not worry about balancing the tree. I can't say for sure, but it does appear that the original levels strive for this. Their trees are not totally unbalanced, but there are some parts where many successive nodes each have a node and a ssector as children (this is unbalanced). And there are a lot of examples to prove that the number of extra segs and vertices they create is very low compared to what it could be. I think the algorithm that id Software used tried to optimize both, but with fewer splits being more important.

[4-9]: SECTORS

A SECTOR is a horizontal (east-west and north-south) area of the map where a floor height and ceiling height is defined. It can have any shape. Any change in floor or ceiling height or texture requires a new sector (and therefore separating linedefs and sidedefs).

Each is 26 bytes, comprising 2 (integer) fields, then 2 (8-byte string) fields, then 3 (integer) fields:

(1) Floor is at this height for this sector
(2) Ceiling height
    A difference of 24 between the floor heights of two adjacent sectors is passable (upwards), but a difference of 25 is "too high". The player may fall any amount.
(3) name of floor texture, from the directory.
(4) name of ceiling texture, from directory.
    All the pictures in the directory between F_START and F_END work as either floors or ceilings. F_SKY1 is used as a ceiling to indicate that it is transparent to the sky above/behind.
(5) brightness of this sector: 0 = total dark, 255 (ff) = maximum light
(6) special sector: see [4-9-1] immediately below.
(7) is a "trigger" number corresponding to a certain LINEDEF with the same trigger number. When that linedef is crossed, something happens to this sector - it goes up or down, etc...

[4-9-1]: Special Sector Types

These numbers control the way the lighting changes, and whether or not a player gets hurt while standing in the sector. -10/20% means that the player takes 20% damage at the end of every second that they are in the sector, except at skill 1, they take 10% damage. If the player has armor, then the damage is split between health and armor.

For all the lighting effects, the brightness levels alternates between the value given for this sector, and the lowest value amongst all the sector's neighbors. Neighbor means a linedef has a side in each sector. If no neighbor sector has a lower light value, then there is no lighting effect. "blink off" means the light goes to the lower value for just a moment. "blink on" means the light is usually at the neighbor value, then jumps up to the normal value for a moment. "oscillate" means that the light level goes

smoothly from one value to the other; it takes about 2 seconds to go from maximum to minimum and back (255 to 0 to 255).

Dec	Hex	Condition or effect
0	00	is normal, no special characteristic.
1	01	light blinks off at random times.
2	02	light blinks on every 0.5 second
3	03	light blinks on every 1.0 second
4	04	-10/20% health AND light blinks on every 0.5 second
5	05	-5/10% health
7	07	-2/5% health, this is the usual NUKAGE acid-floor.
8	08	light oscillates
9	09	SECRET: player must move into this sector to get credit for finding this secret. Counts toward the ratio at the end of the level.
10	0a	?, ceiling comes down 30 seconds after level is begun
11	0b	-10/20% health. When player's HEALTH <= 10%, then the level ends. If it is a level 8, then the episode ends.
12	0c	light blinks on every 1.0 second
13	0d	light blinks on every 0.5 second
14	0e	??? Seems to do nothing
16	10	-10/20% health

All other values cause an error and exit to DOS.

[4-10]: REJECT

The REJECT resource is optional. Its purpose in the original maps is to help speed the process of calculating when monsters detect the player(s). It can also be used for some special effects.

The size of a REJECT in bytes is (number of SECTORS ^ 2) / 8, rounded up. It is an array of bits, with each bit controlling whether monsters in a given sector can detect players in another sector.

Make a table of sectors vs. sectors, like this:

		sector that the player is in				
		0	1	2	3	4
sector that the monster is in	0	0	1	0	0	0
	1	1	0	1	1	0
	2	0	1	0	1	0
	3	0	1	1	1	0
	4	0	0	1	0	0

A 1 means the monster cannot become activated by seeing a player, nor can it attack the player. A 0 means there is no restriction. All non-deaf monsters still become activated by weapon sounds that they hear (including the bare fist!). And activated monsters will still pursue the player, but they will not attack if their current sector vs. sector bit is "1". So a REJECT that's set to all 1s gives a bunch of pacifist monsters who will follow the player around and look menacing, but never actually attack.

How the table turns into the REJECT resource:

Reading left-to-right, then top-to-bottom, like a page, the first bit in the table becomes bit 0 of byte 0, the 2nd bit is bit 1 of byte 0, the 9th bit is bit 0 of byte 1, etc. So if the above table represented a level with only 5 sectors, its REJECT would be 4 bytes:

10100010 00101001 01000111 xxxxxxx0 (hex A2 29 47 00, decimal 162 41 71 0)

In other words, the REJECT is a long string of bits which are read from least significant bit to most significant bit, according to the multi-byte storage scheme used in a certain "x86" family of CPUs.

Usually, if a monster in sector A can't detect a player in sector B, then the reverse is true too, thus if 0/1 is set, 1/0 will be set also. Same sector prohibitions, e.g. 0/0, 3/3, etc. are very rarely set, only in tiny sectors that monsters can't get to anyway. If a large sector with monsters in it has this assignment, they'll exhibit the pacifist syndrome.

I think the array was designed to help speed the monster-detection process. If a sector pair is prohibited, the game engine doesn't even bother checking line-of-sight feasibility for the monster to "see" the player and thus become active. I suppose it can save some calculations if there are lots of monsters.

It is possible to automatically generate some reject pairs, but to calculate whether or not one sector can "see" another can be very complicated. You can't judge line-of-sight just by the two dimensions of the map, you also have to account for the floor and ceiling heights. And to verify that every point in the 3d volume of one sector is out of sight of every point in the other sector...whew! The easy way is to just leave them all 0, or have a user interface which allows them to use their brains to determine which reject pairs should be set.

[4-11]: BLOCKMAP

The BLOCKMAP is a pre-calculated structure that the game engine uses to simplify collision-detection between moving things and walls. Below I'll give a pseudo-code algorithm that will automatically construct a blockmap from the set of LINEDEFS and their vertices.

If a level doesn't have a blockmap, it can display fine, but everybody walks through walls, and no one can hurt anyone else.

The whole level is cut into "blocks", each is a 128 (hex 80) wide square (the grid lines in the automap correspond to these blocks).

All of the blockmap's fields are integers.

The first two integers are XORIGIN and YORIGIN, which specify the coordinates of the bottom-left corner of the bottom-left (southwest) block. These two numbers are usually equal to 8 less than the minimum values of x and y achieved in any vertex on the level.

Then come COLUMNS and ROWS, which specify how many "blocks" there are in the X and Y directions. COLUMNS is the number of blocks in the x direction.

For a normal level, the number of blocks must be large enough to contain every linedef on the level. If there are linedefs outside the blockmap, they will not be able to prevent monsters or players from crossing them, which can cause problems, including the hall of mirrors effect.

Then come (ROWS * COLUMNS) integers which are pointers to the offset within the blockmap resource for that "block". These "offsets" refer to the INTEGER number, NOT the byte number, where to find the block's list. The blocks go right (east) and then up (north). The first block is at row 0, column 0; the next at row 0, column 1; if there are 34 columns, the 35th block is row 1, column 0, etc.

After all the pointers, come the block lists. Each blocklist describes the numbers of all the LINEDEFS which are partially or wholly "in" that block. Note that lines and points which seem to be on the "border"

between two blocks are actually only in one. For example, if the origin of the blockmap is at (0,0), the first column is from 0 to 127 inclusive, the second column is from 128 to 255 inclusive, etc. So a vertical line with x coordinate 128 which might seem to be on the border is actually in the easternmost/rightmost column only. Likewise for the rows - the north/upper rows contain the border lines.

An "empty" block's blocklist consists of two integers: 0 and then -1. A non-empty block will go something like: 0 330 331 333 -1. This means that linedefs 330, 331, and 333 are "in" that block. Part of each of those line segments lies within the (hex 80 by 80) boundaries of that block. What about the block that has linedef 0? It goes: 0 0 ... etc ... -1.

Here's another way of describing blockmap as a list of the integers, in order:

```
Coordinate of block-grid X origin
Coordinate of block-grid Y origin
# of columns (blocks in X direction)
# of rows (blocks in Y direction)
Block 0 offset from start of BLOCKMAP, in integers
  .
  .
Block N-1 offset (N = number of columns * number of rows)
Block 0 list: 0, numbers of every LINEDEF in block 0, -1 (ffff)
  .
  .
Block N-1 list: 0, numbers of every LINEDEF in block N-1, -1 (ffff)
```

[4-11-1]: How to automatically generate the BLOCKMAP

Here is an algorithm that can create a blockmap from the set of linedefs and their vertices' coordinates. For reasons of space and different programming tastes, I won't include every little detail here, nor is the algorithm in any particular language. The pseudocode below is like BASIC or PASCAL, sort of. I'm not being very formal about variable declarations and such, since that's such a pain.

There are basically two ways that the blockmap can be automatically generated. The slow way is to do every block in order, and check every linedef to see if part of the linedef is in the block. This method is slow because it has to perform (number of blocks) * (number of linedefs) iterations, and in most iterations it will have to do at least one fairly complicated formula do determine an intersection. With the number of blocks at 500-2500 for a typical level, and linedefs at 500-1500, this can really bog down on a big level.

The better way is to do the linedefs in order, keeping a dynamic list for every block, and adding the linedef number to the end of the blocklist for every block it passes through. We won't have to test every block to see if the line passes through it; in fact, we won't be testing ANY blocks, we'll be calculating exactly which blocks it goes through based on its coordinates and slope. This method will have to go through one cycle for each linedef, with very few calculations needed for most cycles, since most linedefs are in only one or two blocks.

```
' Pseudo-code algorithm to generate a BLOCKMAP. The goal is speed. If you
' can top this approach, I'd be surprised.
' Most variables are of type integer, except slope and its pals, see below.
' Some of the ideas here are borrowed from Matt Tagliaferri.
```

```
' x_minimum is the minimum x value in the set of vertices, etc.
' the -8 is to make the blockmaps just like the original ones.

  x_origin = -8 + x_minimum
  y_origin = -8 + y_minimum
  Columns = ((x_maximum - x_origin) DIV 128) + 1
  Rows = ((y_maximum - y_origin) DIV 128) + 1

' DIV is whatever function performs integer division, e.g. 15 DIV 4 is 3.

  number_of_blocks = Rows * Columns
  INITIALIZE Block_string(number_of_blocks - 1)
  FOR count = 0 to number_of_blocks DO
    Block_string(count) = STRING(0)
  NEXT count

' STRING is whatever function or typecast will change the integer "int"
' to its two-byte string format. Here we set up an array to hold all the
' blocklists. All blocklists start with the integer 0, and end with -1;
' we'll add the -1s at the end.
' A string array is best, because we need to haphazardly add to the
' blocklists. line 0 might be in blocks 34, 155, and 276, for instance.
' And string's lengths are easily determined, which we'll need at the end.
' To save on memory requirements, the size of each array element can be
' limited to c. 200 bytes = 100 integers, since what is the maximum number
' of linedefs which will be in a single block? Certainly less than 100.

  FOR line = 0 TO Number_Of_Linedefs DO
    x0 = (x coordinate of that linedef's vertex 0) - x_origin
    y0 = (y coordinate of vertex 0) - y_origin
    x1 = (x coordinate of vertex 1) - x_origin
    y1 = (y coordinate of vertex 1) - y_origin

' subtracting the origins shifts the axes and makes calculations simpler.

    blocknum = (y0 DIV 128) * COLUMNS + (x0 DIV 128)
    Block_string(blocknum) = Block_string(blocknum) + STRING(line)

    boolean_column = ((x0 DIV 128)=(x1 DIV 128))
    boolean_row = ((y0 DIV 128)=(y1 DIV 128))

' This is meant to assign boolean values for whether or not the linedef's
' two vertices are in the same column and/or row. I'm assuming that the
' expressions will be evaluated as 1 if "true" and 0 if "false".
' So if both vertices are in the same block, both of these booleans will be
' true and we can go to the next linedef, because it's only in one block.
' If a line is horizontal or vertical, it is easy to calculate exactly which
' blocks it occupies. Since many, if not most, lines are orthogonal and
' short, that is where this algorithm gets most of its speed.

    CASE (boolean_column * 2 + boolean_row):
      CASE 3: NEXT line
      CASE 2: block_step = SIGN(y1-y0) * Columns
              FOR count = 1 TO ABS((y1 DIV 128) - (y0 DIV 128)) DO
                blocknum = blocknum + block_step
                Block_string(blocknum) = Block_string(blocknum) +
STRING(line)
```

```
              NEXT count
              NEXT line
      CASE 1: block_step = SIGN(x1-x0)
              count = 1 TO ABS((x1 DIV 128) - (x0 DIV 128)) DO
                blocknum = blocknum + block_step
                Block_string(blocknum) = Block_string(blocknum) +
STRING(line)
              NEXT count
              NEXT line
    END CASE

' now to take care of the longer, diagonal lines...

    y_sign = SIGN(y1-y0)
    x_sign = SIGN(x1-x0)

' Important: the variables "slope", "x_jump", "next_x" and "this_x" need to
' be of type REAL, not integer, to maintain the accuracy. "slope" will not
' be 0 or undefined, these situations were weeded out by CASE 1 and 2 above.
' An alternative was pointed out to me, but I haven't implemented it in this
' algorithm. If you scale these numbers by 1000, then 32 bit integer
' arithmetic will be precise enough, you won't need sloppy and slow real #s.

    slope = (y1-y0)/(x1-x0)
    x_jump = (128/slope) * y_sign
    CASE (y_sign):
      CASE -1: next_x = x0 + ((y0 DIV 128) * 128 - 1 - y0)/slope
      CASE 1: next_x = x0 + ((y0 DIV 128) * 128 + 128 - y0)/slope
    END CASE

' Suppose the linedef heads northeast from its start to its end. We'll
' first calculate all the blocks in the start row, which will all be
' successively to the right of the first block (blocknum). Then we'll move
' up to the next row, set the block, and go right, then the next row, etc.
' until we've passed the second/end vertex. (the three other directions
' NW SE SW are taken care of too, all by proper use of sign)

' x_jump is how far x goes right or left when y goes up/down 128.
' next_x will be the x coordinate of the next intercept with a "critical"
' y value. When the line goes up, the critical values are equal to 128, 256,
' etc, the first y-values in a new block. If the line goes down, then the
' intercepts occur when y equals 255, 127, etc. Remember, all this is in the
' "shifted" coord system.

' INT is whatever function will discard the decimal part of a real number,
' converting it to an integer. It doesn't matter which way it rounds
' negatives, since next_x and this_x are always positive.

    last_block = INT(next_x/128) - (x0 DIV 128) + blocknum
    IF last_block > blocknum THEN
      FOR count = (blocknum + x_sign) TO last_block STEP x_sign DO
        Block_string(count) = Block_string(count) + STRING(line)
      NEXT count

    REPEAT
      this_x = next_x
```

```
      next_x = this_x + x_jump
      IF (x_sign * next_x) > (x_sign * x1) THEN next_x = x1
      first_block = last_block + y_sign * Columns
      last_block = first_block + INT(next_x/128) - INT(this_x/128)
      FOR count = first_block TO last_block STEP x_sign DO
        Block_string(count) = Block_string(count) + STRING(line)
      NEXT count
    UNTIL INT(next_x) = x1


  NEXT line

' That's it. Now all we have to do is write the BLOCKMAP to wherever.

  WRITE Blockmap, AT OFFSET 0, x_origin
  WRITE Blockmap, AT OFFSET 2, y_origin
  WRITE Blockmap, AT OFFSET 4, Columns
  WRITE Blockmap, AT OFFSET 6, Rows

  pointer_offset = 8
  blocklist_offset = 8 + 2 * number_of_blocks
  FOR count = 0 TO number_of_blocks - 1 DO
     WRITE Blockmap, AT OFFSET pointer_offset, blocklist_offset / 2
     WRITE Blockmap, AT OFFSET blocklist_offset, Block_string(count)
     blocklist_offset = blocklist_offset + LENGTH(Block_string(count)) + 2
     WRITE Blockmap, AT OFFSET (blocklist_offset - 2), STRING(-1)
     pointer_offset = pointer_offset + 2
  NEXT count

' Done! blocklist_offset will equal the total size of the BLOCKMAP, when
' this last loop is finished
```

1.6 chapter5

```
-----------------------------
CHAPTER [5]: Pictures' Format
-----------------------------
```

The great majority of the entries if the directory reference resources that are in a special picture format. The same format is used for the sprites (monsters, items), the wall patches, and various miscellaneous pictures for the status bar, menu text, inter-level map, etc. The floor and ceiling textures are NOT in this format, they are raw data; see chapter [6].

After much experimenting, it seems that sprites and floors cannot be added or replaced via pwad files. However, wall patches can (whew!). This is apparently because all the sprites' entries must be in one "lump", in the IWAD file, between the S_START and S_END entries. And all the floors have to be listed between F_START and F_END. If you use those entries in pwads, either nothing will happen, or an error will occur. There are also P_START and P_END entries in the directory, which flank the wall patch names, so how come they work in pwads? I think it is somehow because of the PNAMES resource, which lists all the wall patch names that are to be used. Too bad there aren't SNAMES and FNAMES resources!

It is still possible to change and manipulate the sprites and floors, its just more difficult to do, and very difficult to figure out a scheme for

potential distribution of changes. The DOOM.WAD file must be changed, and that is a pain.

All the sprites follow a naming scheme. The first four letters are the sprite's name, or and abbreviation. TROO is for imps, BKEY is for the blue key, etc. See [4-2-1] for a list of them.

For most things, the unanimated ones, the next two characters of the sprite's name are A0, like SUITA0, the radiation suit. For simple animated things, there will be a few more sprites, e.g. PINVA0, PINVB0, PINVC0, and PINVD0 are the four sprites for the Invulnerability power-up. Monsters are the most complicated. They have several different sequences, for walking, firing, dying, etc, and they have different sprites for different angles. PLAYA1, PLAYA2A8, PLAYA3A7, PLAYA4A6, and PLAYA5 are all for the first frame of the sequence used to display a walking (or running) player. 1 is the view from the front, 2 and 8 mean from front-right and front-left (the same sprite is used, and mirrored appropriately), 3 and 7 the side, 5 the back.

Each picture has three sections, basically. First, a four-integer header. Then a number of long-integer pointers. Then the picture pixel color data.

[5-1]: Header

The header has four fields:

(1) Width. The number of columns of picture data.
(2) Height. The number of rows.
(3) Left offset. The number of pixels to the left of the center; where the first column gets drawn.
(4) Top offset. The number of pixels above the origin; where the top row is.

The width and height define a rectangular space or limits for drawing a picture within. To be "centered", (3) is usually about half of the total width. If the picture had 30 columns, and (3) was 10, then it would be off-center to the right, especially when the player is standing right in front of it, looking at it. If a picture has 30 rows, and (4) is 60, it will appear to "float" like a blue soul-sphere. If (4) equals the number of rows, it will appear to rest on the ground. If (4) is less than that for an object, the bottom part of the picture looks awkward.

With walls patches, (3) is always (columns/2)-1, and (4) is always (rows)-5. This is because the walls are drawn consistently within their own space (There are two integers in each SIDEDEF which can offset the beginning of a wall's texture).

Finally, if (3) and (4) are NEGATIVE integers, then they are the absolute coordinates from the top-left corner of the screen, to begin drawing the picture, assuming the VIEW is FULL-SCREEN (the full 320x200). This is only done with the picture of the doom player's current weapon - fist, chainsaw, bfg9000, etc. The game engine scales the picture down appropriately if the view is less than full-screen.

[5-2]: Pointers

After the header, there are N = (# of columns) long integers (4 bytes each). These are pointers to the data for each COLUMN. The value of the pointer represents the offset in bytes from the first byte of the picture resource.

```
[5-3]: Pixel Data
================
```

Each column is composed of some number of BYTES (NOT integers), arranged in "posts":

The first byte is the row to begin drawing this post at. 0 means whatever height the header (4) upwards-offset describes, larger numbers move correspondingly down.

The second byte is how many colored pixels (non-transparent) to draw, going downwards.

Then follow (# of pixels) + 2 bytes, which define what color each pixel is, using the game palette. The first and last bytes AREN'T drawn, and I don't know why they are there. Probably just leftovers from the creation process on the NExT machines. Only the middle (# of pixels in this post) are drawn, starting at the row specified in byte 1 of the post.

After the last byte of a post, either the column ends, or there is another post, which will start as stated above.

255 (hex FF) ends the column, so a column that starts this way is a null column, all "transparent". Goes to the next column.

Thus, transparent areas can be defined for either items or walls.

1.7 chapter6

```
---------------------------------------
CHAPTER [6]: Floor and Ceiling Textures
---------------------------------------
```

All the names for these textures are in the directory between the F_START and F_END entries. There is no look-up or meta-structure as with the walls. Each texture is 4096 raw bytes, making a square 64 by 64 pixels, which is pasted onto a floor or ceiling, with the same orientation as the automap would imply, i.e. the first byte is the color at the NW corner, etc. The blocks in the grid are 128 by 128, so four floor tiles will fit in each block.

The data in F_SKY1 isn't even used since the game engine interprets that special ceiling as see-through to the SKY texture beyond. So the F_SKY1 entry can have zero length.

As discussed in chapter [5], replacement and/or new-name floors don't work right from pwad files, only in the main IWAD.

You can change all the floors and ceilings you want by constructing a new DOOM.WAD, but you have to make sure no floor or ceiling uses an entry name which isn't in your F_ section. And you have to include these four entries, although you can change their contents (pictures): FLOOR4_8, SFLR6_1, MFLR8_4, and FLOOR7_2. The first three are needed as backgrounds for the episode end texts. The last is what is displayed "outside" the display window if the display is not full-screen.

```
[6-1]: Animated floors
----------------------
```

See Chapter [8-4-1] for a discussion of how the animated walls and floors work. Unfortunately, the fact that the floors all need to be lumped together in one wad file means that its not possible to change the animations via a pwad file, unless it contains ALL the floors, which amounts to several hundred k. Plus you can't distribute the original data, so if you want to

pass your modification around, it must either have all the floors all-new,
or you must create some sort of program which will construct the wad from
the original DOOM.WAD plus your additions.

1.8 chapter7

```
-----------------------------
CHAPTER [7]: Sounds and Songs
-----------------------------

[7-1]: D_[xxxxxx]
=================
```

Songs. What format are they? Apparently the MUS format, which I have absolutely no knowledge of. But it's obvious what each song is for, from their names.

```
[7-2]: DP[xxxxxx] and DS[xxxxxx]
================================
```

These are the sound effects. They come in pairs - DP for pc speaker sounds, DS for sound cards.

The DS sounds are in RAW format: they have a four integer header, then the sound samples (each is 1 byte since they are 8-bit samples).

The headers' four (unsigned) integers are: 3, then 11025 (the sample rate), then the number of samples, then 0. Since the maximum number of samples is 65535, that means a little less than 6 seconds is the longest possible sound effect.

1.9 chapter8

```
-------------------------------------------------
CHAPTER [8]: Some Important Non-picture Resources
-------------------------------------------------

[8-1]: PLAYPAL
==============
```

There are 14 palettes here, each is 768 bytes = 256 rgb triples. That is, the first three bytes of a palette are the red, green, and blue portions of color 0. And so on.

Note that standard VGA boards whose palettes only encompass 262,144 colors only accept values of 0-63 for each channel (rgb), so the values would need to be divided by 4.

Palette 0 is the one that is used for almost everything.

Palettes 10-12 are used (briefly) when an item is picked up, the more items that are picked up in quick succession, the brighter it gets, palette 12 being the brightest.

Palette 13 is used while wearing a radiation suit.

Palettes 3, 2, then 0 again are used after getting berserk strength.

If the player is hurt, then the palette shifts up to X, then comes "down" one every half second or so, to palette 2, then palette 0 (normal)

again. What X is depends on how badly the player got hurt: Over 100% damage (add health loss and armor loss), X=8. 93%, X=7. 81%, X=6. 55%, X=5. 35%, X=4. 16%, X=2.

[8-2]: COLORMAP
==========

This contains 34 sets of 256 bytes, which "map" the colors "down" in brightness. Brightness varies from sector to sector. At very low brightness, almost all the colors are mapped to black, the darkest gray, etc. At the highest brightness levels, most colors are mapped to their own values, i.e. they don't change.

In each set of 256 bytes, byte 0 will have the number of the palette color to which original color 0 gets mapped.

The colormaps are numbered 0-33. Colormaps 0-31 are for the different brightness levels, 0 being the brightest (light level 248-255), 31 being the darkest (light level 0-7).

Colormap 32 is used for every pixel in the display window (but not the status bar), regardless of sector brightness, when the player is under the effect of the "Invulnerability" power-up. This map is all whites/greys.

Colormap 33 is all black for some reason.

[8-3]: DEMO[1-3]
==========

These are the demos that will be shown if you start doom, and do nothing else. Demos can be created using the devparm parameter:

DOOM -devparm -record DEMONAME

The extension .LMP is automatically added to the DEMONAME. Other parameters may be used simultaneously, such as -skill [1-5], -warp [1-3] [1-9], -file [pwad_filename], etc. The demos in the WAD are in exactly the same format as these LMP files, so a LMP file may be simply pasted or assembled into a WAD, and if its length and pointer directory entries are correct, it will work.

This is assuming the same version of the game, however. For some illogical reason, demos made with 1.1 doom don't work in 1.2 doom, and vice versa. If I had a pressing need to convert an old demo, I might try to figure out why, but I don't.

The game only accesses DEMO1, DEMO2, and DEMO3, so having more than that in a pwad file is pointless.

[8-4]: TEXTURE1 and TEXTURE2
==========

These resources contains a list of the wall names used in the various SIDEDEFS sections of the level data. Each wall name actually references a meta-structure, defined in this list. TEXTURE2 has all the walls that are only in the registered version.

First is a table of pointers to the start of the entries. There is a long integer (say, N) which is the number of entries in the TEXTURE resource. Then follow N long integers which are the offsets in bytes from the beginning of the TEXTURE resource to the start of that texture's definition entry.

Then follow N texture entries, which each consist of a 8-byte name field and then a variable number of 2-byte integer fields:

(1) The name of the texture, used in SIDEDEFS, e.g. "STARTAN3".
(2) always 0.
(3) always 0.
(4) total width of texture
(5) total height of texture

The fourth and fifth fields define a "space" (usually 128 by 128 or 64 by 72 or etc...) in which individual wall patches are placed to form the overall picture. This is done because there are some wall patches that are used in several different walls, like computer screens, etc. Note that to tile properly in the vertical direction on a very tall wall, a texture has to have height 128, the maximum. The maximum width is 256. The sum of the sizes of all the wall patches used in a single texture must be <= 64k.

(6) always 0.
(7) always 0.
(8) Number of 5-field patch descriptors that follow. This is why each texture entry has variable length. Many entries have just 1 patch, one has 64!

1. x offset from top-left corner of texture space defined in field 4/5 to start placement of this patch
2. y offset
3. number, from 0 to whatever, of the entry in the PNAMES resource, which contains the name from the directory, of the wall patch to use...
4. always 1, is for something called "stepdir"...
5. always 0, is for "colormap"...

The texture's entry ends after the last of its patch descriptors.
Note that patches can have transparent parts, since they are in the same picture format as everything else. Thus there can be (and are) transparent wall textures. These should only be used on a border between two sectors, to avoid the "displaying nothing" problems.
Here is how one can add walls, while still retaining any of the original ones it came with: in a pwad, have replacement entries for PNAMES and TEXTURE2. These will be the same as the originals, but with more entries, for the wall patches and assembled textures that you're adding. Then have entries for every new name in PNAMES, as well as old names which you want to associate to new pictures. You don't need to use the P_START and P_END entries.

[8-4-1]: Animated walls

It is possible to change the walls and floors that are animated, like the green blocks with a sewer-like grate that's spewing green slime (SLADRIPx). The game engine sets up as many as 8 animation cycles for walls based on the entries in the TEXTURE resources, and up to 5 based on what's between F_START and F_END. The entries in FirstTexture and LastTexture, below, and all the entries between them (in the order that they occur in a TEXTURE list), are linked. If one of them is called by a sidedef, that sidedef will change texture to the next in the cycle about 5 times a second , going back to First after Last. Note that the entries between First and Last need not be the same in number as in the original, nor do they have to follow the same naming pattern, though that would probably be wise. E.g. one could set up ROCKRED1, ROCKREDA, ROCKREDB, ROCKREDC, ROCKREDD, ROCKREDE, ROCKRED3 for a 7-frame animated wall!

If First and Last aren't in either TEXTURE, no problem. Then that cycle isn't used. But if First is, and Last either isn't or is listed BEFORE First, then an error occurs.

FirstTexture	LastTexture	Normal # of frames
BLODGR1	BLODGR4	4
BLODRIP1	BLODRIP4	4
FIREBLU1	FIREBLU2	2
FIRELAV3	FIRELAVA	2
FIREMAG1	FIREMAG3	3
FIREWALA	FIREWALL	3
GSTFONT1	GSTFONT3	3
ROCKRED1	ROCKRED3	3
SLADRIP1	SLADRIP3	3

(floor/ceiling animations) -

NUKAGE1	NUKAGE3	3
FWATER1	FWATER3	3
SWATER1	SWATER4	4
LAVA1	LAVA4	4
BLOOD1	BLOOD3	3

Note that the SWATER entries aren't in the regular DOOM.WAD.

[8-5]: PNAMES

This is a lookup table for the numbers in TEXTURE[1 or 2] to reference to an actual entry in the directory which is a wall patch (in the picture format described in chapter [5]).

The first two bytes of the PNAMES resource is an integer P which is how many entries there are in the list.

Then come P 8-byte names, each of which duplicates an entry in the directory. If a patch name can't be found in the directory (including the external pwad's directories), an error will occur. This naming of resources is apparently not case-sensitive, lowercase letters will match uppercase.

The middle integer of each 5-integer "set" of a TEXTURE1 entry is something from 0 to whatever. Number 0 means the first entry in this PNAMES list, 1 is the second, etc...

Thanks for reading the "Official" DOOM Specs!