# The unofficial QWD format description


## Uwe Girlich

**uwe@half-empty.de**



**v1.0.5, 6/3/1999**



This document describes the QWD file format, which is the result of "recording" a game in QuakeWorld.


## Table of Contents

# 1. Introduction

## 1.1. Recording and Playback

To create a recording of your network play use up to QuakeWorld 2.10 the console command *record name server*. This connects you to the server `server` and records the game play from your point of view into the file `name.qwd`.

From version 2.20 on you have first to connect to the server and start the recording with the console command *record name* later.

The recording stops when you disconnect from the server or you use the console command *stop*. To play it back, use the commands *playdemo name* or *timedemo name*.

## 1.2. Versions

**Table 1. Covered QuakeWorld versions**

| `Exe:` | version | platform |
|---|---|---|
| 12:43:52 Jun 13 1997 | Linux QuakeWorld (0.94) 1.64 | Linux, SVGA |
| 19:51:52 Aug 7 1997 | Linux QuakeWorld 2.00 | Linux, SVGA |
| 23:46:07 Oct 13 1997 | Linux QuakeWorld (0.94) 2.10 | Linux, SVGA |
| 00:03:05 Oct 14 1997 | Linux QuakeWorld (0.94) 2.10 | Linux, X11, 8bpp |
| 23:46:22 Oct 23 1997 | QuakeWorld Server 2.10 | Linux |
| 18:32:34 Nov 2 1997 | Linux QuakeWorld (0.94) 2.10 | Linux, OpenGL |
| 14:47:43 May 16 1998 | QuakeWorld Server 2.20 | Linux |
| 14:48:28 May 16 1998 | Linux (0.96) QuakeWorld 2.20 | Linux, SVGA |
| 14:48:28 May 16 1998 | Linux (0.96) QuakeWorld 2.20 | Linux, X11 |
| 14:50:20 May 16 1998 | Linux (0.96) QuakeWorld 2.20 | Linux, OpenGL |
| 14:37:00 May 20 1998 | QuakeWorld Server 2.21 | Linux |
| 14:37:46 May 20 1998 | Linux (0.97) QuakeWorld 2.21 | Linux, SVGA, 8bpp |
| 14:39:38 May 20 1998 | Linux (0.97) QuakeWorld 2.21 | Linux, OpenGL |
| 17:27:37 Jul 13 1998 | QuakeWorld Server 2.29BETA | Linux |
| 17:27:56 Jul 13 1998 | Linux (0.98) QuakeWorld 2.29 | Linux, SVGA |
| 17:27:56 Jul 13 1998 | Linux (0.98) QuakeWorld 2.29 | Linux, X11 |
| 17:28:42 Jul 13 1998 | Linux (0.98) QuakeWorld 2.29 | Linux, OpenGL |

| | | |
|---|---|---|
| 17:05:37 Aug 26 1998 | QuakeWorld Server 2.30 | Linux |
| 17:06:04 Aug 26 1998 | Linux (0.98) QuakeWorld 2.30 | Linux, SVGA |
| 17:06:04 Aug 26 1998 | Linux (0.98) QuakeWorld 2.30 | Linux, X11 |
| 17:07:08 Aug 26 1998 | Linux (0.98) QuakeWorld 2.30 | Linux, OpenGL (Mesa) |
| 17:07:08 Aug 26 1998 | Linux (0.98) QuakeWorld 2.30 | Linux, OpenGL (any) |

I actually write and check my documentation with Linux `qwcl` and `qwsv` 2.30 on the same machine.

The older versions of QuakeWorld couldn't record a QWD file but playback seemed to work. I checked QuakeWorld 1.54c and 1.55 with a QWD file recorded with 1.64 but both old versions crashed on playback.

If you find that this documentation covers even more versions (other operating systems) please drop me a note.

## 1.3. Advertising

As the clever reader may know I'm the author of LMPC, the Little Movie Processing Centre. With this tool you may

- "decompile" an existing QWD file to a simple text file and
- "compile" such a (modified) text file back to a binary QWD file.

With LMPC it is very easy to analyse a QWD file but you may change it as well and so create a QWD file of a network game you never played. The current version of LMPC can be found at my Demo Specs page (http://demospecs.half-empty.de).

## 1.4. General remarks

The QWD format is very different from the original Quake DEM format. That applies as well to the internal message format as to the general game scene representation. Many things are now described totally different, which makes an accurate converter (particles, nails etc.) very difficult.

I know for sure that I'll never write a DEM <-> QWD converter.

# 2. Game font

A string may contain any 8 bit characters except '\377' and it ends with '\000'. The special characters '\n' and '\r' have their normal meaning.

The QuakeWorld font is an extended ASCII font (7 bit) which contains in the upper half a similar font but with a different colour.

I used a simple Quake DEM file (http://demospecs.half-empty.de/misc/qfont.html) to print all 252 ASCII characters.

# 3. File structure

To describe the file structure, which is very complicated, I use C like program fragments and `struct` definitions. This simplifies my task a lot.

I invented all used names (messages, variables etc.) for myself, took them from the QuakeWorld binary, QuakeEd but almost all from the QuakeC source.

All multi-byte structures in QWD files are “little endian” (VAX or Intel, lowest byte first).

At first some QuakeEd compliant coordinate `typedef`'s:

```
typedef float vec_t;

typedef vec_t vec3_t[3];
```

A QWD file is the recording of the network traffic between the client and the server in both directions. Each network packet and its time stamp will be stored in a ‘block’ of the QWD file.

Every block has the structure

```
typedef struct {
  float time;
  char  code;
  char  data[???];
} block_t;
```

`float time;`

is the time stamp of the block.

`char code;`

is the sign to indicate the block type. Each block type will be parsed totally different.

**Table 2. Block types**

| `code` | block type |
|---|---|
| `0x00` | client block |
| `0x01` | server block |
| `0x02` | frame block |

`char data[???];`

is actual data of the block.

## 3.1. Client block

A client block is 41 bytes long and contains an expanded version of a network packet from the client to the server. The actual network packet is packed.

The client block has the following structure:

```
typedef struct {
  float         time;
  char          code;       // == 0
  long          load;
  vec3_t        angles;
  short         speed[3];
  unsigned char flag;
  unsigned char impulse;
  vec3_t        uk_angles;
} client_block_t;
```

`long load;`

is connected somehow with the workload on the client. ?FIXME?

`vec3_t angles;`

point in the viewing direction (in degree) of the client.

`short speed[3];`

is the intended translation of the client (forward, right, up).

`flag;`

is a collection of flags and must be splitted:

```
fire = (flag & 0x01) ? 1 : 0;
jump = (flag & 0x02) ? 1 : 0;
```

Other bits may contain additional information.

`int fire;`

indicates an *attack* console command.

`int jump;`

indicates a *jump* console command.

`long impulse;`

is the value of a currently activated *impulse* console command.

## 3.2. Server block

A server block has a variable length and contains a copy of a network packet from the server to the client. It has the following structure:

```
typedef struct {
  float         time;
  char          code;        // == 1
  long          blocksize;
  unsigned long seq_rel_1;
  char          messages[blocksize-4];
} server_block_t;
```

`long blocksize;`

is the number of bytes in the block following the `blocksize` variable itself (but including `seq_rel_1`). The full server block has `blocksize+9` bytes. The maximim value for `blocksize` (`MAX_MSGLEN`) used to be 7500 but changed from protocol 25 to 26 (game version 2.10 to 2.20) to 1450. The new value avoids fragmentation of the UDP packets on an Ethernet. The MTU in PPP is much smaller but the reduced packet size is for LAN play definitely a great improvement.

`unsigned long seq_rel_1;`

is a sign to distinguish between a connectionless block (`==0xFFFFFFFF`) or a game block (`!=0xFFFFFFFF`).

### 3.2.1. Connectionless block

Each connectionless block or packet contains one server command to control the server-client network connection and has the structure

```
typedef struct {
  float          time;
  char           code;       // == 1
  long           blocksize;
  unsigned long  seq_rel_1;  // == 0xFFFFFFFF
  char           connless_id;
  char           connless_data[blocksize-5];
} connless_block_t;
```

`char connless_id;`

is a code to explain the following `connless_data`.

`char connless_data[blocksize-5];`

is the rest of the block and depends on the value of `connless_id`.

### 3.2.2. Game block

A game block has the structure

```
typedef struct {
  float          time;
  char           code;       // == 1
  long           blocksize;
  unsigned long  seq_rel_1;  // != 0xFFFFFFFF
  unsigned long  seq_rel_2;
  char           messages[blocksize-8];
} game_blocks_t;
```

`unsigned long seq_rel_1;`

is a compound variable and must be splitted:

```
seq1 = seq_rel_1 & 0x7FFFFFFF;
reliable1 = ( seq_rel_1 >> 31 ) & 0x01;
```

`unsigned long seq1;`

is the sequence code of the sent network packet (from the server to the client).

`int reliable1;`

Indicates, that this packet is a reliable one. In the actual network protocol, the server retransmit it until the client send the acknowledge.

`unsigned long seq_rel_2;`

is a compound variable and must be splitted:

```
seq2 = seq_rel_2 & 0x7FFFFFFF;
reliable2 = ( seq_rel_2 >> 31 ) & 0x01;
```

`unsigned long seq2;`

is the sequence code of the last received network packet (from the client to the server).

`int reliable2;`

Indicates, that this packet was a reliable one. In the actual network protocol, the server acknowledge so a reliable packet from the client.

```
char messages[blocksize-8];
```

contain several game messages with the main game data. The structure is similar to the messages in Quake DEM files.

## 3.3. Frame block

A frame block is 13 bytes long and contains 2 sequence numbers. It is a server to client block and the first sequence number goes up in the same sequence as in all the surrounding game blocks. The exact meaning of these sequence numbers is unknown to me. ?FIXME? A frame block appears first in protocol version 26 (game version 2.20).

The frame block has the following structure:

```
typedef struct {
  float         time;
  char          code;       // == 2
  unsigned long seq1;
  unsigned long seq2;
} frame_block_t;
```

```
unsigned long seq1;
```

is the current sequence number (from the server to the client).

```
unsigned long seq2;
```

is certainly some kind of an already received sequence number.

## 3.4. Auxiliary routines

Here comes the definition of some small auxiliary routines to simplify the main message description. `get_next_unsigned_char`, `get_next_signed_char`, `get_next_short`, `get_next_long` and `get_next_float` are basic functions and they do exactly what they are called. Please note: `byte`, `char` or `short` will be converted to `long`. Second note: Don't look at the variable types for size calculations. Look at the routine names.

In the following I often use a count variable

```
int i;
```

without declaration. I hope this does not confuses you.

```
long ReadByte
{
```

```
  return (long) get_next_unsigned_char;
}



long ReadChar
{
  return (long) get_next_signed_char;
}



long ReadShort
{
  return (long) get_next_short;
}



long ReadLong
{
  return get_next_long;
}
```

Note: A signed angle in a single byte. There are only 256 possible direction to look into.

```
vec_t ReadAngle
{
  return (vec_t) ReadChar / 256.0 * 360.0;
}
```

This angle can point in 65536 directions.

```
vec_t ReadAngle16
{
  return (vec_t) ReadShort / 65536.0 * 360.0;
}
```

A coordinate is stored in 16 bits: 1 sign bit, 12 integer bits and 3 fraction bits.

```
vec_t ReadCoord
{
  return (vec_t) ReadShort * 0.125;
}
```

The string reading stops at '\0' or after 0x7FF bytes. The internal buffer has only 0x800 bytes available.

```
char* ReadString
{
  char* string_pointer;
  char string_buffer[0x800];

  string_pointer=string_buffer;
  for (i=0 ; i<0x7FF ; i++, string_pointer++) {
    if (! (*string_pointer = ReadChar) ) break;
  }
  *string_pointer = '\0';
  return strdup(string_buffer);
}
```

```
long ReadFloat
{
  return get_next_float;
}
```

# 4. List of all message types in connectionless blocks

This is the general message structure:

```
typedef struct {
  char connless_id;
  char connless_data[???];
} connless_message_t;
```

The length of a message depends on its type.

The easiest way to explain a message in a connectionless block is to give a short C like program fragment to parse such a message. It is not really the same code base as in LMPC but it should be *very* similar. Each message can be described by its `connless_id` or its name.

## 4.1. disconnect

`connless_id`

`0x02`

purpose

Stop the playback. It is usually the last block of a QWD file.

variables

`char* text;`

is an unused and (by QuakeWorld) unparsed text. Its value is "EndOfDemo".

parse routine

```
text=ReadString;
```

## 4.2. client_command

`connless_id`

`0x42` = 'B'

purpose

The client transfers the text to the console and runs it. The command can only come from a local server.

variables

`char* text;`

is the console command, to be executed.

parse routine

```
text=ReadString;
```

## 4.3. challenge

`connless_id`

`0x63` = 'c'

purpose

This special message appears first with protocol version 26 (game version 2.20).

parse routine

```
challenge = ReadString;
```

## 4.4. connect

`connless_id`

`0x6A` = 'j'

purpose

The server tells the client to start the game initialisation procedure.

parse routine

none

## 4.5. ping

`connless_id`

`0x6B` = 'k'

purpose

The server tells the client that it is still alive.

parse routine

none

## 4.6. console

`connless_id`

`0x6E` = 'n'

purpose

The client transfers the text to the console and prints it.

variables

`char* text;`

is the text to be printed.

parse routine

```
text=ReadString;
```

# 5. List of all game message types

This is the general game message structure:

```
typedef struct {
                  unsigned char ID;
                  char          messagecontent[????];
                } game_message_t;
```

The length of a message depends on its type (or `ID`).

Each message can be described by its `ID` or its name.

## 5.1. bad

`ID`

`0x00`

purpose

Something is bad. This message should never appear.

parse routine

```
error("CL_ParseServerMessage: Bad server message");
```

## 5.2. nop

`ID`

`0x01`

purpose

No operation.

parse routine

none

## 5.3. disconnect

`ID`

`0x02`

purpose

Disconnect from the server. Stops the game.

parse routine

none

## 5.4. updatestat

`ID`

`0x03`

purpose

Updates directly a `byte` value in the player state array of `long` numbers. To update a `long` value, look in section Section 5.39.

variables

`long index;`

is the index in the `playerstate` array. Table '`updatestat` indices' lists all possible indices and their real name. The value of an `items` entry (`index`=15) is a bit difficult and will be

explained in the table ‘`items` bits’.

**Table 3. `updatestat` indices**

| index | variable |
|---|---|
| 0 | `health` |
| 1 | ??? (not used) |
| 2 | `weaponmodel` |
| 3 | `currentammo` |
| 4 | `armorvalue` |
| 5 | `weaponframe` |
| 6 | `ammo_shells` |
| 7 | `ammo_nails` |
| 8 | `ammo_rockets` |
| 9 | `ammo_cells` |
| 10 | `weapon` |
| 11 | `total_secrets` |
| 12 | `total_monsters` |
| 13 | `found_secrets` |
| 14 | `killed_monsters` |
| 15 | `items` |
| . | |
| . | |
| . | |
| 23 | ??? |
| . | |
| . | |
| . | |
| 31 | ??? |

**Table 4. `items` bits**

| bit | value | QuakeC | purpose |
|---|---|---|---|
| 0 | `0x00000001` | IT_SHOTGUN | Shotgun (should be always 1) |
| 1 | `0x00000002` | IT_SUPER_SHOTGUN | Double-barrelled Shotgun |
| 2 | `0x00000004` | IT_NAILGUN | Nailgun |
| 3 | `0x00000008` | IT_SUPER_NAILGUN | Perforator |

| | | | |
|---|---|---|---|
| 4 | `0x00000010` | IT_GRENADE_LAUNCHER | Grenade Launcher |
| 5 | `0x00000020` | IT_ROCKET_LAUNCHER | Rocket Launcher |
| 6 | `0x00000040` | IT_LIGHTNING | Thunderbolt |
| 7 | `0x00000080` | IT_EXTRA_WEAPON | extra weapon (there is no extra weapon) |
| 8 | `0x00000100` | IT_SHELLS | Shells are active |
| 9 | `0x00000200` | IT_NAILS | Nails are active |
| 10 | `0x00000400` | IT_ROCKETS | Grenades are active |
| 11 | `0x00000800` | IT_CELLS | Cells are active |
| 12 | `0x00001000` | IT_AXE | Axe (should be always 1) |
| 13 | `0x00002000` | IT_ARMOR1 | green Armor |
| 14 | `0x00004000` | IT_ARMOR2 | yellow Armor |
| 15 | `0x00008000` | IT_ARMOR3 | red Armor |
| 16 | `0x00010000` | IT_SUPERHEALTH | Megahealth |
| 17 | `0x00020000` | IT_KEY1 | silver keycard (or runekey or key) |
| 18 | `0x00040000` | IT_KEY2 | gold keycard (or runekey or key) |
| 19 | `0x00080000` | IT_INVISIBILITY | Ring of Shadows |
| 20 | `0x00100000` | IT_INVULNERABILITY | Pentagram of Protection |
| 21 | `0x00200000` | IT_SUIT | Biosuit |
| 22 | `0x00400000` | IT_QUAD | Quad Damage |
| 23 | `0x00800000` | unknown | unknown (is 0) |
| 24 | `0x01000000` | unknown | unknown (is 0) |
| 25 | `0x02000000` | unknown | unknown (is 0) |
| 26 | `0x04000000` | unknown | unknown (is 0) |
| 27 | `0x08000000` | unknown | unknown (is 0) |
| 28 | `0x10000000` | unknown | Rune 1 |
| 29 | `0x20000000` | unknown | Rune 2 |
| 30 | `0x40000000` | unknown | Rune 3 |
| 31 | `0x80000000` | unknown | Rune 4 |

`long value;`

is the new (`byte`) value.

```
long playerstate[32];
```

is the array to describe the player state.

parse routine

```
index = ReadByte;
if (index > 31)
  error("CL_SetStat: %i is invalid", index);
value = ReadByte;
playerstate[index] = value;
```

## 5.5. version

`ID`

```
0x04
```

purpose

Not used any more. Calls the *bad*-routine.

## 5.6. setview

`ID`

```
0x05
```

purpose

Not used any more. Calls the *bad*-routine.

## 5.7. sound

`ID`

```
0x06
```

purpose

This message starts the play of a sound at a specific point.

variables

`float vol;`

is the volume of the sound (0.0 off, 1.0 max).

`float attenuation;`

is the attenuation of the sound.

**Table 5. Sound attenuations**

| value | QuakeC | purpose |
|---|---|---|
| 0 | ATTN_NONE | i. e. player's death sound doesn't get an attenuation |
| 1 | ATTN_NORM | the normal attenuation |
| 2 | ATTN_IDLE | for idle monsters |
| 3 | ATTN_STATIC | for spawnstaticsound messages |

`long channel;`

is the sound channel. There are 8 possible sound channels for each entity in QuakeWorld but it uses 5 only.

**Table 6. Sound channels**

| value | QuakeC | purpose |
|---|---|---|
| 0 | CHAN_AUTO | selects a channel automatically |
| 1 | CHAN_WEAPON | weapon use sounds |
| 2 | CHAN_VOICE | pain calls |
| 3 | CHAN_ITEM | item get sounds |
| 4 | CHAN_BODY | jump and fall sounds |

Channel 0 never willingly overrides. Other channels (1-4) always override a playing sound on that channel.

`long entity;`

is the entity which caused the sound.

`long soundnum;`

is the index in the precache sound table.

`vec3_t origin;`

is the origin of the sound.

parse routine

```
long entity_channel; // combined variable

entity_channel = ReadShort;
vol = entity_channel & 0x8000 ? (float) ReadByte / 255.0 : 1.0;
attenuation = entity_channel & 0x4000 ? (float) ReadByte / 64.0 : 1.0;
channel = entity_channel & 0x07;
entity = (entity_channel >> 3) & 0x03FF;
if (entity >= 0x0300)
  error("CL_ParseStartSoundPacket: ent = %i", entity);
soundnum = ReadByte;
for (i=0 ; i<3 ; i++) origin[i] = ReadCoord;
```

## 5.8. time

`ID`

`0x07`

purpose

Not used any more. Calls the *bad*-routine.

## 5.9. print

`ID`

`0x08`

purpose

The client prints the text in the top left corner of the screen. There is space for 4 lines. They scroll up and the text disappears. The text will be printed on the console as well.

variables

`long level;`

is the priority level of the text.

**Table 7. Print priority levels**

| value | QuakeC | purpose |
|---|---|---|
| 0 | PRINT_LOW | pickup messages (white) |
| 1 | PRINT_MEDIUM | death messages |
| 2 | PRINT_HIGH | critical messages (red) |
| 3 | PRINT_CHAT | also goes to chat console |

`char* text;`

is the text to be displayed. All font specials are explained in section Section 2.

parse routine

```
level = ReadByte;
text = ReadString;
```

## 5.10. stufftext

`ID`

`0x09`

purpose

The client transfers the text to the console and runs it.

variables

`char* text;`

is the command, which the client has to execute.

parse routine

```
text = ReadString;
```

## 5.11. setangle

`ID`

`0x0A`

purpose

This message set the camera orientation.

variables

`vec3_t angles;`

is the new camera orientation.

parse routine

```
for (i=0 ; i<3 ; i++) angles[i] = ReadAngle;
```

## 5.12. serverdata

`ID`

`0x0B`

purpose

This message initialises a new level.

variables

`long serverversion;`

is the protocol version coming from the server.

**Table 8. QuakeWorld values for `PROTOCOL_VERSION`**

| game | protocol |
|---|---|
| 1.64 | 24 |
| 2.00 | 25 |
| 2.10 | 25 |
| 2.20 | 26 |

| | |
|---|---|
| 2.21 | 26 |
| 2.29BETA | 27 |
| 2.30 | 28 |

`long age;`

is the number of levels analysed since the existence of the server process. Starts with 1.

`char* game;`

is the QuakeWorld game directory. It has usually the value "qw";

`long client;`

is the client id.

`char* mapname;`

is the name of the level.

`float maxspeed;`

is the maximum running speed. It may be changed during the game with the *maxspeed* message.

`float entgravity;`

is the gravity in the level. It may be changed during the game with the *entgravity* message.

`float var0, var1, var3, var4, var4, var5, var6, var7, var8`

are other global definition variables. ?FIXME?

parse routine

```
serverversion = ReadLong;
if (serverversion != PROTOCOL_VERSION)
  error("Server returned version %i, not %i", version, PROTOCOL_VERSION);
age = ReadLong;
game = ReadString;
client = ReadByte;
mapname = ReadString;
if (serverversion >= 25) { // from 2.00 on
  var0 = ReadFloat;
  var1 = ReadFloat;
  maxspeed = ReadFloat;
  var3 = ReadFloat;
  var4 = ReadFloat;
  var5 = ReadFloat;
  var6 = ReadFloat;
  var7 = ReadFloat;
  var8 = ReadFloat;
```

```
  entgravity = ReadFloat;
}
```

## 5.13. lightstyle

`ID`

`0x0C`

purpose

This message defines a light animation style.

variables

`long style;`

is the light style number.

`char* string;`

is a string of letters ‘a’ .. ‘z’, where ‘a’ means black and ‘z’ white. All effects from nervous flashing (“az”) to slow dimming (“zyxwvu ... edcba”) can so be described.

`#define MAX_LIGHTSTYLES 63`

is the last number number of a light style.

parse routine

```
style = ReadByte;
if (style>MAX_LIGHTSTYLES)
  error("svc_lightstyle > MAX_LIGHTSTYLES");
string = ReadString;
```

## 5.14. updatename

`ID`

`0x0D`

purpose

Not used any more. Calls the *bad*-routine.

## 5.15. updatefrags

ID

```
0x0E
```

purpose

This message updates the frag count of a specific player.

variables

```
long player;
```

is the player number (0 .. MAX_SCOREBOARD).

```
long frags;
```

is the new frag count for this player.

```
#define MAX_SCOREBOARD 31
```

is the last possible number of a player.

parse routine

```
player = ReadByte;
if (player>MAX_SCOREBOARD)
  error("CL_ParseServerMessage: svc_updatefrags > MAX_SCOREBOARD");
frags = ReadShort;
```

## 5.16. clientdata

ID

```
0x0F
```

purpose

Not used any more. Calls the *bad*-routine.

## 5.17. stopsound

`ID`

`0x10`

purpose

Stops a sound. It looks for a sound started with a *sound* message with the same `channel` and `entity`.

variables

`long channel;`

is the sound channel.

`long entity;`

is the entity which caused the sound.

parse routine

```
long channel_entity; // combined variable

channel_entity = ReadShort;
channel = channel_entity & 0x07;
entity = (channel_entity >> 3) & 0x03FF;
```

## 5.18. updatecolors

`ID`

`0x11`

purpose

Not used any more. Calls the *bad*-routine.

## 5.19. particle

`ID`

`0x12`

purpose

Not used any more. Calls the *bad*-routine.

## 5.20. damage

`ID`

`0x13`

purpose

Tells how severe was a hit and from which point it came.

variables

`long save;`

will be subtracted from the current armor.

`long take;`

will be subtracted from the current health.

`vec3_t origin;`

is the origin of the hit. It points to the weapon (not the origin) of the attacking entity or it is (0,0,0) if the damage was caused by drowning or burning.

parse routine

```
save = ReadByte;
take = ReadByte;
for (i=0 ; i<3 ; i++) origin[i] = ReadCoord;
```

## 5.21. spawnstatic

`ID`

`0x14`

purpose

This message creates a static entity and sets the internal values.

variables

`long StaticEntityCount;`

is the number of already started static entities. The maximum number is 127.

`entity_t* staticentities;`

is the array filled up with the data of the static entities.

`long default_modelindex;`

is the model index in the precache model table for the entity.

`long default_frame;`

is the frame number of the model.

`long default_colormap;`

is the colormap number to display the model.

`long default_skin;`

is the skin number of the model.

`vec3_t default_origin;`

is the origin of the entity.

`vec3_t default_angles;`

is the orientation of the entity.

parse routine

```
if (StaticEntityCount > 127)
  error("Too many static entities");
staticentities[StaticEntityCount].default_modelindex = ReadByte;
staticentities[StaticEntityCount].default_frame = ReadByte;
staticentities[StaticEntityCount].default_colormap = ReadByte;
staticentities[StaticEntityCount].default_skin = ReadByte;
for (i=0 ; i<3 ; i++) {
```

```
  staticentities[StaticEntityCount].default_origin[i] = ReadCoord;
  staticentities[StaticEntityCount].default_angles[i] = ReadAngle;
}
StaticEntityCount++;
```

## 5.22. spawnbinary

`ID`

`0x15`

purpose

Not used any more. Calls the *bad*-routine.

## 5.23. spawnbaseline

`ID`

`0x16`

purpose

Creates a dynamic entity and sets the internal default values.

variables

`long entity;`

is the number of the entity. In Quake there was a test if this number is too big. There is no such test in QuakeWorld.

`entity_t* entities;`

is the array filled up with the data of the dynamic entities.

`long default_modelindex;`

is the model index in the precache model table for the entity.

`long default_frame;`

is the frame number of the model.

`long default_colormap;`

is the colormap number to display the model.

`long default_skin;`

is the skin number of the model.

`vec3_t default_origin;`

is the origin of the entity.

`vec3_t default_angles;`

is the orientation of the entity.

parse routine

```
entity = ReadShort;
entities[entity].default_modelindex = ReadByte;
entities[entity].default_frame = ReadByte;
entities[entity].default_colormap = ReadByte;
entities[entity].default_skin = ReadByte;
for (i=0 ; i<3 ; i++) {
  entities[entity].default_origin[i] = ReadCoord;
  entities[entity].default_angles[i] = ReadAngle;
}
```

## 5.24. temp_entity

`ID`

`0x17`

purpose

Creates a temporary entity.

variables

`long entitytype;`

is the type of the temporary entity. There are three kinds of temporary entities:

point entity

is a small point like entity.

**Table 9. point entities**

| value | QuakeC | purpose |
|---|---|---|
| 0 | TE_SPIKE | unknown |
| 1 | TE_SUPERSPIKE | superspike hits (spike traps) |
| 3 | TE_EXPLOSION | grenade/missile explosion |
| 4 | TE_TAREXPLOSION | explosion of a tarbaby |
| 7 | TE_WIZSPIKE | wizard's hit |
| 8 | TE_KNIGHTSPIKE | hell knight's shot hit |
| 10 | TE_LAVASPLASH | Chthon awakes and falls dead |
| 11 | TE_TELEPORT | teleport end |
| 13 | TE_LIGHTNINGBLOOD | hit by Thunderbolt |

line entity

is a two-dimensional entity.

**Table 10. line entities**

| value | QuakeC | purpose |
|---|---|---|
| 5 | TE_LIGHTNING1 | flash of the Shambler |
| 6 | TE_LIGHTNING2 | flash of the Thunderbolt |
| 9 | TE_LIGHTNING3 | flash in e1m7 to kill Chthon |

multi entity

is a cluster of point entities.

**Table 11. multi entities**

| value | QuakeC | purpose |
|---|---|---|
| 2 | TE_GUNSHOT | shot multiple pellets |
| 12 | TE_BLOOD | hit something that can bleed |

`long entity;`

is the entity which created the temporary entity.

`vec3_t origin;`

is the origin of the entity.

`vec3_t trace_endpos;`

is the destination of the line entity.

`long count;`

is the number of particles in a multi entity.

parse routine

```
entitytype = ReadByte;
switch (entitytype) {
  case 0,1,3,4,7,8,10,11,13:
       for (i=0 ; i<3 ; i++) origin[i] = ReadCoord;
  break;
  case 5,6,9:
       entity = ReadShort;
       for (i=0 ; i<3 ; i++) origin[i] = ReadCoord;
       for (i=0 ; i<3 ; i++) trace_endpos[i] = ReadCoord;
  break;
  case 2,12:
       count = ReadByte;
       for (i=0 ; i<3 ; i++) origin[i] = ReadCoord;
  break;
  default:
    error("CL_ParseTEnt: bad type");
  break;
}
```

## 5.25. setpause

`ID`

`0x18`

purpose

Set the pause state. This message was not implemented up to game version 2.10 (protocol version 25) but appears from game version 2.20 (protocol version 26) on.

variables

```
long pausestate;
```

is non-zero to start the pause and zero to stop it.

parse routine

```
pausestate = ReadByte;
if (pausestate) {
  // pause is on
}
else {
  // pause is off
}
```

## 5.26. signonum

ID

```
0x19
```

purpose

Not used any more. Calls the *bad*-routine.

## 5.27. centerprint

ID

```
0x1A
```

purpose

Prints the specified text at the centre of the screen. There is only one text line with a maximum of 40 characters. To print more than this one line, use '\n' in a single *centerprint* message for a new line. Every text line (the first 40 characters) will be centred horizontally.

All font specials are explained in section Section 2.

variables

```
char* text;
```

is the text to be displayed.

parse routine

```
text = ReadString;
```

## 5.28. killedmonster

`ID`

`0x1B`

purpose

Indicates the death of a monster.

variables

```
long killed_monsters;
```

is the number of killed monsters. It may be displayed with the console command *showscores*.

parse routine

```
killed_monsters++;
```

## 5.29. foundsecret

`ID`

`0x1C`

purpose

Indicates, that the player just entered a secret area. It comes usually with a `centerprint` message.

variables

```
long found_secrets;
```

is the number of found secrets. It may be displayed with the console command *showscores*.

parse routine

```
found_secrets++;
```

## 5.30. spawnstaticsound

ID

```
0x1D
```

purpose

This message starts a static (ambient) sound not connected to an entity but to a position.

variables

```
vec3_t origin;
```

is the origin of the sound.

```
long soundnum;
```

is the sound index in the precache sound table.

```
float vol;
```

is the volume (0.0 off, 1.0 max)

```
float attenuation;
```

is the attenuation of the sound. Possible attenuations can be found in the table ‘Sound attenuations’ of section Section 5.7.

parse routine

```
for (i=0 ; i<3 ; i++) origin[i] = ReadCoord;
soundnum = ReadByte;
vol = (float) ReadByte / 255.0;
attenuation = (float) ReadByte / 64.0;
```

## 5.31. intermission

`ID`

`0x1E`

purpose

Displays the level end screen.

variables

`vec3_t origin;`

is the origin of the intermission waiting place.

`vec3_t angles;`

is the viewing direction from the intermission waiting place.

parse routine

```
for ( i=0 ; i<3 ; i++) origin[i] = ReadCoord;
for ( i=0 ; i<3 ; i++) angles[i] = ReadAngle;
```

## 5.32. finale

`ID`

`0x1F`

purpose

Displays the episode end screen and some text. The text will be printed like `centerprint` but slower.

variables

`char* text;`

is the episode end text.

parse routine

```
text = ReadString;
```

## 5.33. cdtrack

`ID`

```
0x20
```

purpose

Selects the audio CD track number.

variables

```
long track;
```

is the audio CD track to play.

parse routine

```
track = ReadByte;
```

## 5.34. sellscreen

`ID`

```
0x21
```

purpose

Displays the help and sell screen.

parse routine

none

## 5.35. smallkick

`ID`

`0x22`

purpose

The recoil of the Shotgun, Nailgun, Perforator, Grenade Launcher, Rocket Launcher and Thunderbolt kicks the shooter soft.

parse routine

none

## 5.36. bigkick

`ID`

`0x23`

purpose

The recoil of the Double-barrelled Shotgun kicks the shooter hard.

parse routine

none

## 5.37. updateping

`ID`

`0x24`

purpose

Updates the ping time. ?FIXME?

variables

`long player;`

is the number of the player (0 .. MAX_SCOREBOARD).

`long ping;`

is the ping time in milliseconds.

parse routine

```
player = ReadByte;
if (player>MAX_SCOREBOARD)
  error("CL_ParseServerMessage: svc_updateping > MAX_SCOREBOARD");
ping = ReadShort;
```

## 5.38. updateentertime

`ID`

`0x25`

purpose

Defines the start time, when the client enters the server. ?FIXME?

variables

`long player;`

is the number of the player (0 .. MAX_SCOREBOARD).

`float entertime;`

is the time stamp, as the client enters a server.

parse routine

```
player = ReadByte;
if (player>MAX_SCOREBOARD)
  error("CL_ParseServerMessage: svc_updateentertime > MAX_SCOREBOARD");
entertime = ReadFloat;
```

## 5.39. updatestatlong

`ID`

`0x26`

purpose

Updates directly a `long` value in the player state array of `long` numbers. To update a `byte` value, look in section Section 5.4.

variables

`long index;`

is the index in the `playerstate` array. Look in table '`updatestat` indices' in section Section 5.4 for a list of possible indices.

`long value;`

is the new (`long`) value.

`long playerstate[32];`

is the array to describe the player state.

parse routine

```
index = ReadByte;
if (index > 31)
  error("CL_SetStat: %i is invalid", index);
value = ReadLong;
playerstate[index] = value;
```

## 5.40. muzzleflash

`ID`

`0x27`

purpose

The entity lights up a bit when it shoots.

variables

long entity;

is the entity number.

parse routine

```
entity = ReadShort;
```

## 5.41. updateuserinfo

`ID`

`0x28`

purpose

Updates some user information from the master server.

variables

long player;

is the player number (0 .. MAX_SCOREBOARD).

long user;

is 0, if there is no such player connected. Some kind of user identification. ?FIXME?

char* text

is a string with variable definitions, separated by '\'. These variables define the color, the name etc.

parse routine

```
player = ReadByte;
user = ReadLong;
if (player>MAX_SCOREBOARD)
  error("CL_ParseServerMessage: svc_updateuserinfo > MAX_SCOREBOARD");
text = ReadString;
```

## 5.42. download

`ID`

`0x29`

purpose

Download a file from the server.

variables

`long size;`

is the length of the data block. A typical value is 1024.

`long percent;`

is the transferred amount of data in percent.

`char* downloadbuffer;`

is the buffer for downloaded files.

parse routine

```
size = ReadShort;
percent = ReadByte;
if (size == -1)
  error("File not found.\n");
}
if (percent == 0)
  fp = fopen(filename, "wb");
}
for ( i=0 ; i<size ; i++ ) {
  downloadbuffer[i] = ReadByte;
}
fwrite(fp, size, 1, downloadbuffer);
if (percent != 100) {
  servercommand("nextdl"); /* ask for the next part */
}
else {
  fclose(fp);
}
```

## 5.43. playerinfo

`ID`

`0x2A`

purpose

Updates player information.

variables

`long player;`

is the number of the player (0 .. MAX_SCOREBOARD).

`long mask;`

is a bit mask to reduce the network traffic.

`vec3_t origin;`

is the origin of the player.

`long frame;`

is the frame number of the player model.

`float ping;`

has something to do with the ping time. ?FIXME?

`long mask2;`

is another bit mask to reduce the network traffic.

`unsigned char load;`

is connected somehow to the workload of the client. ?FIXME?

`vec3_t angles;`

point in the viewing direction (in degree) of the client.

`vec3_t speed;`

is the translation of the client (forward, right, up).

`long flag;`

is a collection of flags and must be splitted.

`int fire;`

indicates an *attack* console command.

`int jump;`

indicates a *jump* console command.

`long impulse;`

is the value of a currently activated *impulse* console command.

`vec3_t cspeed;`

is the current speed in the x,y, and z direction. Used to predict the future.

`long model;`

is the model index in the precache model table of the player.

`long playermodel;`

is the model index in the precache model table for the standard value `"progs/player.mdl"`.

`long uk_byte6;`

is an unknown byte. ?FIXME?

`long weapon;`

contains a bit mask for the current weapon.

**Table 12. `weapon` bits**

| bit | value | QuakeC | weapon |
|---|---|---|---|
| ? | `0x00` | not available | Axe |
| 0 | `0x01` | IT_SHOTGUN | Shotgun |
| 1 | `0x02` | IT_SUPER_SHOTGUN | Double-barrelled Shotgun |
| 2 | `0x04` | IT_NAILGUN | Nailgun |
| 3 | `0x08` | IT_SUPER_NAILGUN | Perforator |
| 4 | `0x10` | IT_GRENADE_LAUNCHER | Grenade Launcher |
| 5 | `0x20` | IT_ROCKET_LAUNCHER | Rocket Launcher |
| 6 | `0x40` | IT_LIGHTNING | Thunderbolt |
| 7 | `0x80` | IT_EXTRA_WEAPON | extra weapon (there is no extra weapon) |

`long weaponframe;`

is the frame of the current weapon model.

parse routine

```
player = ReadByte;
mask = ReadShort;
for (i=0;i<3;i++) origin[i] = ReadCoord;
frame = ReadByte;
if (mask & 0x0001) ping = ReadByte * 0.001;        // bit 0
if (mask & 0x0002) {                               // bit 1
  mask2 = ReadByte;
  if (serverdata.serverversion >= 27) { // from game version 2.29BETA on
    if (mask2 & 0x01) angles[0] = ReadAngle16;       // bit 0
    if (mask2 & 0x80) angles[1] = ReadAngle16;       // bit 7
    if (mask2 & 0x02) angles[2] = ReadAngle16;       // bit 1
    if (mask2 & 0x04) speed[0] = ReadShort;          // bit 2
    if (mask2 & 0x08) speed[1] = ReadShort;          // bit 3
    if (mask2 & 0x10) speed[2] = ReadShort;          // bit 4
    if (mask2 & 0x20) flag = ReadByte;               // bit 5
    fire = (flag & 0x01) ? 1 : 0;
    jump = (flag & 0x02) ? 1 : 0;
    if (mask2 & 0x40) impulse = ReadByte;            // bit 6
    load = ReadByte);
  }
  else { // serverdata.serverversion <= 26, game version up to 2.21
    if (mask2 & 0x01) angles[0] = ReadAngle16;       // bit 0
    angles[1] = ReadAngle16;
    if (mask2 & 0x02) angles[2] = ReadAngle16;       // bit 1
    if (mask2 & 0x04) speed[0] = ReadByte;           // bit 2
    if (mask2 & 0x08) speed[1] = ReadByte;           // bit 3
    if (mask2 & 0x10) speed[2] = ReadByte;           // bit 4
    if (mask2 & 0x20) flag = ReadByte;               // bit 5
    fire = (flag & 0x01) ? 1 : 0;
    jump = (flag & 0x02) ? 1 : 0;
    if (mask2 & 0x40) impulse = ReadByte;            // bit 6
    if (mask2 & 0x80) load = ReadByte;               // bit 7
  }
}
if (mask & 0x0004) cspeed[0] = ReadCoord;          // bit 2
if (mask & 0x0008) cspeed[1] = ReadCoord;          // bit 3
if (mask & 0x0010) cspeed[2] = ReadCoord;          // bit 4
model = (mask & 0x0020) ? ReadByte : playermodel;  // bit 5
uk_byte6 = (mask & 0x0040) ? ReadByte : 0;         // bit 6
if (mask & 0x0080) weapon = ReadByte;              // bit 7
if (mask & 0x0100) weaponframe = ReadByte;         // bit 8
```

## 5.44. nails

`ID`

`0x2B`

purpose

Describes the position and orientation of all currently flying nails.

types

```
struct {
  vec3_t origin;
  float angle_1;
  float angle_2;
} nail_t;
```

variables

`long nailcount;`

is the number of nails.

`nail_t* nails;`

is the internal array with all nail coordinates.

`vec3_t origin;`

is the origin of the nail.

`float angle_1;`

is the tilt angle of the nail.

`float angle_2;`

is the yaw angle of the nail.

parse routine

```
unsigned char b[5];
int j;
nail_t* n;

nailcount = ReadByte;
for (j=0,n=nails;j<nailcount;j++,n++) {
  for (i=0;i<5;i++) b[i] = ReadByte;
  // 3 12 bit values
  n->origin[0] = (b[0] & 0xFF) | ((b[1] & 0x0F) << 8);
```

```
  n->origin[1] = ((b[1] & 0xF0) >> 4) | (b[2] << 4);
  n->origin[2] = (b[3] & 0xFF) | ((b[4] & 0x0F) << 8);
  // shift and scale to standard (even) coordinates
  for (i=0;i<3;i++) n->origin[i] = (n->origin[i] - 2048) * 2;
  // signed value in 4 bits
  n->angle_1 = (b[4] & 0xF0) >> 4;
  // respect the sign
  if (n->angle_1>=8) n->angle_1 = n->angle_1 - 16;
  // scale it
  n->angle_1 *= 360.0 / 16.0;
  n->angle_2 = ReadAngle;
}
```

exact structure

The structure of the nail message is so strange, that I can't suppress some general remarks.

**Table 13. the 6 nails bytes**

| byte | bit 7 ... 4 | bit 3 ... 0 |
|---|---|---|
| 0 | origin[0] b7 ... b4 | origin[0] b3 ... b0 |
| 1 | origin[1] b3 ... b0 | origin[0] b11 ... b8 |
| 2 | origin[1] b11 ... b8 | origin[1] b7 ... b4 |
| 3 | origin[2] b7 ... b4 | origin[2] b3 ... b0 |
| 4 | angle_1 b3 ... b0 | origin[2] b11 ... b8 |
| 5 | angle_2 b7 ... b4 | angle_2 b3 ... b0 |

`origin[i]`

A standard coordinate is a signed 16 bit number with 1 sign bit, 12 integer bits and 3 fraction bits. Nail coordinates have only 12 bits in total. The lowest 4 bits of the 16 bit coordinate (the 3 fraction bits and the odd/even bit) are set to zero. Therefore a nail can only exist on an even integer coordinate. The sign transformation is not the usual one. The half nail coordinate is shifted in the positive range by 2048 before the network transmission.

`angle_1`

The tilt angle of a fast flying nail is not very important. There are only 4 bits to describe the tilt angle and only 3 of them are used at all. The 4 bits form a normal (but very short) signed integer. To compute an angle in degree back from this short value, it must be multiplied by 360/16=22.5. The range of values of a tilt angle of a nail is [-90,90] and not [-180,180]. Therefore bit 2 of `angle_1` is always equal to bit 3 (the sign bit). So the eight possible values for the tilt angle are -90.0, -67.5, -45.0, -22.5, 0.0, 22.5, 45.0, 67.5.

`angle_2`

The yaw angle is a standard 1 byte signed integer angle. It must be multiplied by the usual 360/256.

## 5.45. choke

ID

```
0x2C
```

purpose

There is an internal list with 64 entries. Each entry describes a full state (with all 32 players and such). This message defines how many entries in this list can be canceled, because they now contain irrelevant information.

variables

```
long choke;
```

is the number of entries to cancel.

parse routine

```
choke = ReadByte;
```

## 5.46. modellist

ID

```
0x2D
```

purpose

Reads (a part of) the precache model table.

variables

```
char* precache_models[256];
```

is the precache model table. It will be filled up with model file names. Many other messages contain an index in this array. The first used index is 1. Beginning with protocol version 26 (game version 2.20) QuakeWorld uses a new parse method and sends an unused index first.

`long nummodels;`

is the number of models in the precache model table.

`long first;`

is the index of the first model.

`long next;`

is the index for the next group of models. It is 0 to end the precache model table.

parse routine

```
if (serverdata.serverversion) >= 26) { // 2.20 and higher
  char *text;

  first = ReadByte(m);
  for ( i=first ; i<256 ; i++ ) {
    text = ReadString;
    if (strlen(text) == 0) break;
    precache_models[i+1] = strdup(text);  // store model one position later
  }
  next = ReadByte(m);
}
else { // up to 2.10
  nummodels = 0;
  do {
    if (++nummodels > 255)
      error("Server sent too many model_precache");
    precache_models[nummodels] = ReadString;
  } while (*precache_models[nummodels]);
}
```

## 5.47. soundlist

`ID`

`0x2E`

purpose

Reads (a part of) the precache sound table.

variables

`char* precache_sounds[256];`

is the precache sound table. It will be filled up with sound file names. Many other messages contain an index in this array. The first used index is 1. Beginning with protocol version 26 (game version 2.20) QuakeWorld uses a new parse method and sends an unused index first.

`long numsounds;`

is the number of sounds in the precache sound table.

`long first;`

is the index of the first sound.

`long next;`

is the index of the next group of sounds. It is 0 to end the precache sound table.

parse routine

```
if (serverdata.serverversion) >= 26) { // 2.20 and higher
  char *text;

  first = ReadByte(m);
  for ( i=index ; i<256 ; i++ ) {
    text = ReadString;
    if (strlen(text) == 0) break;
    precache_sounds[i+1] = strdup(text);  // store sound one position later
  }
  next = ReadByte(m);
}
else { // up to 2.10
  numsounds = 0;
  do {
    if (++numsounds > 255)
      error("Server sent too many sound_precache");
    precache_sounds[numsounds] = ReadString;
  } while (*precache_sounds[numsounds]);
}
```

## 5.48. packetentities

ID

0x2F

purpose

This message contains the entity numbers in sight and all the changed properties of these entities. The list ends with a `0x0000` for `mask`.

variables

`long mask;`

is a bit mask to reduce the network traffic.

`long entity;`

is the entity number.

`long remove;`

indicates a disappearing entity.

`entity_t* entities;`

is the array filled up with all dynamic entities.

`long modelindex;`

is the model index in the precache model table.

`long frame;`

is the frame number of the model.

`long colormap;`

is the colormap number to display the model.

`long skin;`

is the skin number of the model.

`long effects;`

contains a bit mask for special entity effects.

**Table 14. `effects` values**

| bit | value | QuakeC |
|---|---|---|
| 0 | `0x01` | EF_ROCKET |
| 1 | `0x02` | EF_GRENADE |
| 2 | `0x04` | EF_GIB |
| 3 | `0x08` | EF_ROTATE |
| 4 | `0x10` | EF_TRACER |
| 5 | `0x20` | EF_ZOMGIB |
| 6 | `0x40` | EF_TRACER2 |

| | | |
|---|---|---|
| 7 | 0x80 | EF_TRACER3 |

`vec3_t origin;`

is the origin of the entity.

`vec3_t angles;`

is the orientation of the entity.

parse routine

```
while (mask = ReadShort) {
  entity = mask & 0x01FF;
  mask &= 0xFE00;
  entities[entity].remove = (mask & 0x4000) ? 1 : 0;
  if (mask & 0x8000) mask |= ReadByte;
  if (mask & 0x0004) entities[entity].modelindex = ReadByte;
  if (mask & 0x2000) entities[entity].frame = ReadByte;
  if (mask & 0x0008) entities[entity].colormap = ReadByte;
  if (mask & 0x0010) entities[entity].skin = ReadByte;
  if (mask & 0x0020) entities[entity].effects = ReadByte;
  if (mask & 0x0200) entities[entity].origin[0] = ReadCoord;
  if (mask & 0x0001) entities[entity].angles[0] = ReadAngle;
  if (mask & 0x0400) entities[entity].origin[1] = ReadCoord;
  if (mask & 0x1000) entities[entity].angles[1] = ReadAngle;
  if (mask & 0x0800) entities[entity].origin[2] = ReadCoord;
  if (mask & 0x0002) entities[entity].angles[2] = ReadAngle;
}
```

## 5.49. deltapacketentities

`ID`

`0x30`

purpose

This message contains the changed properties of some entities in sight relative to a specified former frame number. The list ends with a `0x0000` `mask`.

variables

`long frame;`

is the lowest byte of the frame number with the full state, to which this message is the delta.

`long mask;`

is a bit mask to reduce the network traffic.

`entity_t* entities;`

is the array filled up with all dynamic entities.

`long entity;`

is the entity number.

`long remove;`

indicates a disappearing entity.

`long modelindex;`

is the model index in the precache model table.

`long frame;`

is the frame number of the model.

`long colormap;`

is the colormap number to display the model.

`long skin;`

is the skin number of the model.

`long effects;`

contains a bit mask for special entity effects.

`vec3_t origin;`

is the origin of the entity.

`vec3_t angles;`

is the orientation of the entity.

parse routine

```
frame = ReadByte;
while (mask = ReadShort) {
  entity = mask & 0x01FF;
  mask &= 0xFE00;
```

```
  entities[entity].remove = (mask & 0x4000) ? 1 : 0;
  if (mask & 0x8000) mask |= ReadByte;
  if (mask & 0x0004) entities[entity].modelindex = ReadByte;
  if (mask & 0x2000) entities[entity].frame = ReadByte;
  if (mask & 0x0008) entities[entity].colormap = ReadByte;
  if (mask & 0x0010) entities[entity].skin = ReadByte;
  if (mask & 0x0020) entities[entity].effects = ReadByte;
  if (mask & 0x0200) entities[entity].origin[0] = ReadCoord;
  if (mask & 0x0001) entities[entity].angles[0] = ReadAngle;
  if (mask & 0x0400) entities[entity].origin[1] = ReadCoord;
  if (mask & 0x1000) entities[entity].angles[1] = ReadAngle;
  if (mask & 0x0800) entities[entity].origin[2] = ReadCoord;
  if (mask & 0x0002) entities[entity].angles[2] = ReadAngle;
}
```

## 5.50. maxspeed

ID

```
0x31
```

purpose

This message contains the maximum players speed.

variables

```
float maxspeed;
```

is the maximum speed. It appears first in protocol version 25 (game version 2.00).

parse routine

```
maxspeed = ReadFloat;
```

## 5.51. entgravity

ID

```
0x32
```

purpose

This message defines the current gravity. It appears first in protocol version 25 (game version 2.00).

variables

`float gravity;`

is the gravity.

parse routine

```
gravity = ReadFloat;
```

## 5.52. setinfo

`ID`

`0x33`

purpose

This message set a variable for a specific player. It appears first in protocol version 26 (game version 2.20).

variables

`long player;`

is the player number.

`char* name;`

is the name of the variable.

`char* string;`

is the value of the variable.

parse routine

```
player = ReadByte;
name = ReadString;
string = ReadString;
```

## 5.53. serverinfo

`ID`

`0x34`

purpose

This message set a global variable. It appears first in protocol version 26 (game version 2.20).

variables

`char* name;`

is the name of the variable.

`char* string;`

is the value of the variable.

parse routine

```
name = ReadString;
string = ReadString;
```

## 5.54. updatepl

`ID`

`0x35`

purpose

This message updates the player specific packet loss information. It will be displayed in the score-board. It appears first in protocol version 28 (game version 2.30).

variables

`long player;`

is the player number (0 .. MAX_SCOREBOARD).

`long loss;`

is the packet loss in percent.

parse routine

```
player = ReadByte;
if (player>MAX_SCOREBOARD)
  error("CL_ParseServerMessage: svc_updatepl > MAX_SCOREBOARD");
loss = ReadByte;
```

# 6. Version History and Acknowledgements

0.0.1, 6 July, 1997

- First version (working paper) completed.
- Many thanks to Olivier Montanuy (Olivier.Montanuy@wanadoo.fr (mailto:Olivier.Montanuy@wanadoo.fr)) for his QuakeWorld Network Protocol Specs (http://www.ens.fr/~cridlig/bot/qwspec11.html).
- Many thanks to Steffen Winterfeldt (Steffen.Winterfeldt@itp.uni-leipzig.de (mailto:Steffen.Winterfeldt@itp.uni-leipzig.de)) for his reverse engineering work. He worked out many of the general structure information.

0.0.2, 12 July, 1997

- Many new structure information. *sound* and *nails* messages are now correct.

0.0.3, 28 July, 1997

- *packetentities* and *deltapacketentities* are better now.
- Table references.

0.0.4, 30 July, 1997

- General clean-up: some variables renamed.
- Table references don’t work: removed.

0.0.5, 16 August, 1997

- QuakeWorld 2.00 info included.

- Back again to SGML-Tools 0.99.0.
- General clean-up.

0.0.6, 12 March, 1998

- PlanetQuake is the new home.
- QuakeWorld version table restructured.
- SGML-Tools 1.0.5 used.

1.0.0, 14 July, 1998

- QuakeWorld version 2.20, 2.21 and 2.29BETA included.
- *stopsound* is OK now.
- SGML-Tools 1.0.7 used.

1.0.1, 16 August, 1998

- Corrected info on `blocksize`.
- Some small correction with respect to the new LMPC, which can compile QWD files beginning with version 3.1.9.
- Removed the senseless ReadEntity function.

1.0.2, 6 September, 1998

- QuakeWorld version 2.30 included.

1.0.3, 8 January, 1999

- QuakeWorld version 2.30 corrected.
- *updatepl* better.
- Many thanks to Tim Holliefield (holliefiel@bad-durkheim.netsurf.de (mailto:holliefiel@bad-durkheim.netsurf.de)) for some hints on the frame block.
- SGML-Tools 1.0.9 used.

1.0.4, 7 April, 1999

- *setpause* corrected. Many thanks to Christer Sandin (czsuch@ocag.ch (mailto:czsuch@ocag.ch)) for his bug report.

1.0.5, 3 June, 1999

- *modellist* and *soundlist* corrected. Many thanks to Hoffy (ripple@powerup.com.au (mailto:ripple@powerup.com.au)) for his bug report. The old token `last` is called `next` now, so better recreate all your QWD text files. I know, it is bad to change the text format but `last` is simply totally wrong.