The unofficial DM2 format description

Uwe Girlich

uwe@half-empty.de

v1.0.4, 1/23/2000

This document describes the DM2 file format. This file format is the result of "recording" a game in Quake II. The extended DM2 version used in the Quake II Relay modification is also part of this documentation.

Table of Contents

1. Introduction

1.1. Recording and Playback

To create a recording of your Quake II play start any map and then use the console command *record name*. This records the game play from your point of view into the file `demos/name.dm2`. `demos` is a sub-directory of the current game directory. To stop this recording use *stop* or even quit the whole game (*quit*). To play it back, use the command *demomap name.dm2* or simply *map name.dm2*. This kind of recording is called "client side recording".

To create a "server side recording" invoke at the server during a running game the console command *serverrecord name*. This records all information of all entities in the whole level but no player information into the file `demos/name.dm2`. Quake II can't play back these server side recordings.

The third variant of DM2 files should be better called RLA files. They are created by the Quake II Relay server modification. This is a user-made modification to create a special kind of server side demos but they can be read in again (at the server side) and replayed by any client, who connects to this modified server. Read more about this interesting modification at http://www.planetquake.com/relay/.

1.2. Versions

In this document I'll discuss the DM2 format used by Quake II. There are significant differences between the Quake II test release (engine version 3.00) and the CD retail version (3.05). This difference consists especially in the reordered message code bytes, so that the "normal codes" have the values 1 to 5 in 3.05. There are some more differences in the *spawnbaseline* message. Due to the existence of a sole DM2 file in the 3.00 format (packed in the PAK file of the test release) and considering the fact, that the test release can't even record new ones, I will confine myself to the "new" format, introduced with the CD retail version 3.05 (protocol version 26).

With 3.15 (protocol version 32) a new kind of DM2 file was introduced: server side recordings. These are for DM2 editors only, since they contain all information from all entities but no player information and Quake II isn't even able to play them back.

In 3.17 (protocol version 33) the server side recordings became easier to parse because *serverdata* can now be used to distinguish between client side and server side recordings.

Table 1. Covered Quake II versions.

version	platform	note
3.05 x86 Nov 30 1997 RELEASE	Win32	CD retail
3.06 x86 Dec 9 1997 RELEASE	Win32	first patch
3.07 x86 Dec 27 1997 RELEASE	Win32	interim release
3.08 x86 Dec 28 1997 RELEASE	Win32	another interim release
3.09 x86 Dec 29 1997 RELEASE	Win32	with patch program
3.10 x86 Jan 4 1998 RELEASE	Win32	last before point release
3.10 NON-WIN32 Jan 5 1998 NON-WIN32	Linux	first Linux release
3.12 x86 Feb 16 1998 RELEASE	Win32	point release
3.13 x86 Feb 23 1998 RELEASE	Win32	point fix release
3.13 i386 Feb 24 1998 RELEASE	Linux	point fix release
3.14 i386 Mar 2 1998 RELEASE	Linux	another fix release
3.14 x86 Mar 2 1998 RELEASE	Win32	another fix release
3.14 i386 Mar 3 1998 RELEASE	Linux	bug fix release 3.14a
3.15 x86 May 27 1998 Win32 RELEASE	Win32	Mission Pack 1 release
3.15 i386 May 29 1998 Linux	Linux	bug fix release 3.15a
3.17 i386 Jun 21 1998 Linux	Linux	stable release
3.17 x86 Jun 22 1998 Win32 RELEASE	Win32	stable release
3.18 x86 Aug 19 1998 Win32 RELEASE	Win32	Mission Pack 2 release
3.19 x86 Sep 1 1998 Win32 RELEASE	Win32	bug fix release

3.19 x86 Sep 10 1998 Linux	Linux	bug fix release
3.20 x86 Oct 9 1998 Linux	Linux	final release(equal to 3.20 beta)
3.20 x86 Oct 16 1998 Win32 RELEASE	Win32	final release(equal to 3.20 beta)

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 DM2 file (3.05 ... 3.20) to a simple text file and
- "compile" such a (modified) text file back to a binary DM2 file.

With LMPC it is very easy to analyse a DM2 file but you may change it as well and so create a DM2 file of a Quake II game you never played. The current version of LMPC can be found at my Demo Specs page (http://demospecs.half-empty.de).

2. File structure

2.1. General remarks

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 Quake II binary but almost all from the many published source code snippets.

All multi-byte structures in DM2 files are "little endian" (VAX or Intel ordered, lowest byte first),

Descriptive sentences in *italics* are copied from the published game source. There are several different source packages available from id Software:

- the oldest from december 1997 with the source of the game library and some tools (unpacked in the directory `q2source_12_11`):
  - general version: ftp://ftp.idsoftware.com/pub/quake2/source/old/q2source_12_11.zip
- the newer one from march 1998 with the source of the game library only:
  - Linux version: ftp://ftp.idsoftware.com/pub/quake2/source/old/q2-3.14-source.shar.Z

  - Windows version: ftp://ftp.idsoftware.com/pub/quake2/source/old/q2-3.14-source.exe
- The newest release from the end of november 1998 with both Mission Packs:
  - Plain game (unpacked in the directory `q2src320/game/`)
    - Linux version: ftp://ftp.idsoftware.com/pub/quake2/source/q2src320.shar.Z
    - Windows version: ftp://ftp.idsoftware.com/pub/quake2/source/q2src320.exe
  - Xatrix Quake II Mission Pack 1: The Reckoning (unpacked in the directory `xatrixsrc320/game/`)
    - Linux version: ftp://ftp.idsoftware.com/pub/quake2/source/xatrixsrc320.shar.Z
    - Windows version: ftp://ftp.idsoftware.com/pub/quake2/source/xatrixsrc320.exe
  - Rouge Quake II Mission Pack 2: Ground Zero (unpacked in the directory `roguesrc320/game`)
    - Linux version: ftp://ftp.idsoftware.com/pub/quake2/source/roguesrc320.shar.Z
    - Windows version: ftp://ftp.idsoftware.com/pub/quake2/source/roguesrc320.exe

I refer to the files with the full path name (e.g. `q2src320/game/q_shared.h`).

And now some QuakeEd compliant coordinate `typedef`'s:

```
typedef float vec_t;

typedef vec_t vec3_t[3];
```

2.2. Block of messages

A DM2 file is a set of "blocks" of "messages". A block consists of a 4 byte length entry and the actual messages:

```
typedef struct {
                  unsigned long size;
                  unsigned char messages[size];
                } block_t;
```

```
#define MAX_MSGLEN 1400
```

is the maximum message size (real game data in a network packet). The value must be less than or equal to 1472 to avoid fragmentation of the UDP packets on an Ethernet. The MTU in PPP is much smaller and QuakeWorld used up to 2.10 also much bigger network packets but the reduced packet size is for LAN play definitely a great improvement. In server side recordings there is not such a restriction.

```
if (serverdata.isdemo != RECORD_SERVER && block.size>MAX_MSGLEN)
```

```
      error("Demo message > MAX_MSGLEN");
```

Beginning with version 3.05 (protocol version 26) Quake II uses an empty block with a block size of -1 (`0xffffffff`) as the last block. This remindes (at least) me of the DOOM LMP end byte (`0x80`).

Between 2 levels of a multi-level recoring there is an empty block with the block size 0. Quake II can't replay multi-level recordings.

2.3. 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` and `get_next_long`, `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`.

In the following I often use a count variable

```
int i;
```

without declaration. I hope this does not confuses you.

```
int ReadChar
{
  return (int) get_next_signed_char;
}



int ReadByte
{
  return (int) get_next_unsigned_char;
}



int ReadShort
{
  return (int) get_next_short;
}



int ReadLong
{
  return (int)get_next_long;
}



float ReadFloat
```

```
{
  return get_next_float;
}
```

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);
}
```

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

```
ReadPosition(vec_t* pos)
{
  for (i=0 ; i<3 ; i++) pos[i] = ReadCoord;
}
```

A direction for temporary entities is stored in a single byte. The 162 possible orientations are precalculated and stored in a template list. Each direction is represented by a normalised vector. The template list can be found in the source, `q2source_12_11/utils3/qdata/anorms.h`

I tried to reproduce (for `WriteDir`) the template list but I didn't find the algorithm behind it. Take it as it is: even Quake II uses the list and calculates with dot products the best fitting direction.

```
ReadDir(vec_t* pos)
{
  #define NUMVERTEXNORMALS 162
  int code;
  float avertexnormals[NUMVERTEXNORMALS][3] = {
    #include "q2source_12_11/utils3/qdata/anorms.h"
  }
  code = ReadByte;
  if (code >= NUMVERTEXNORMALS) error("MSF_ReadDir: out of range");
```

```
  pos[0] = avertexnormals[code][0];
  pos[1] = avertexnormals[code][1];
  pos[2] = avertexnormals[code][2];
}
```

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

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

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

2.4. Message

This is the message structure:

```
typedef struct {
                  unsigned char ID;
                  unsigned char unicast_client // RLA only
                  char          messagecontents[???];
                } message_t;
```

The length of a message depends on its type (or `ID`) but it can't be bigger than `MAX_MSGLEN`, the size of a full block.

2.5. Unicast

RLA files (DM2 files with Relay extension) can have special uni-cast messages. This is done to distinguish between ordinary messages, which go to every client (multi-cast) and messages, which go only to selected clients. A good example are the *playerinfo* messages. Every client should get a different one and in RLA files all these messages are marked with the destination client number:

```
ID = ReadByte;
if (ID & 0x80) {              // the special sign for uni-cast
  ID &= ~0x80;                // remove it
  unicast.use = 1;            // but memorize it
  unicast.client = ReadByte;  // get the uni-cast destination client
}
```

```
else {
  unicast.use = 0;            // multi-cast message
}
```

3. List of all message types

The easiest way to explain a message 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. Future version will be based on the exact same code base to simplify my task.

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

3.1. bad

`ID`

`0x00`

purpose

Something is bad. This message should never appear.

parse routine

```
error("CL_ParseServerMessage: Illegible server message\n");
```

3.2. muzzleflash

`ID`

`0x01`

purpose

muzzle flashes / player effects.

variables

`long entity;`

is the player entity with the effect.

`long value;`

is the muzzle flash / player effect. The `value` should be one of the `MZ_` constants (source, `q2src320/game/q_shared.h`, 602 - 638). It may be "OR"ed with

`#define MZ_SILENCED 128`
if the player uses the Silencer.

parse routine

```
entity = ReadShort;
value = ReadByte;
```

3.3. muzzlflash2

`ID`

`0x02`

purpose

monster muzzle flashes.

variables

`long entity;`

is the monster entity with the effect.

`long value;`

is the monster muzzle flash. The `value` should be one of the `MZ2_` constants (source, `q2src320/game/q_shared.h`, 640 - 871).

parse routine

```
entity = ReadShort;
value = ReadByte;
```

3.4. temp_entity

`ID`

`0x03`

purpose

Spawns a temporary entity. *Temp entity events are for things that happen at a location seperate from any existing entity. Temporary entity messages are explicitly constructed and broadcast.*

variables

Due to the special parse method I use in my LMPC program, I have to use the variables over all different temporary entities in the same order. This was no real restriction but with game version 3.18 (protocol version 34) came many new types of temporary entities, which reordered their arguments almost random. So I had to rename some variables and to insert some totally senseless new ones (like `flash_entity` instead of `entity` or `type` in addition to `style`). This simply comes, because these values come now at the other side of `origin`. The old order was `entity`, `origin`, `style` but now a `style` comes sometimes before the `origin` or an `entity` comes after the `origin`. After I finish the text parse rewrite of LMPC, I will surely remove the additional senseless new variables.

`long entitytype;`

is the type of the temporary entity. `entitytype` should be one of the `TE_` constants (source, `q2src320/game/q_shared.h`, 876 - 942). They are defined with a `typedef enum` and not as `#define` but who cares?

There are many different kinds of temporary entities in Quake II:

point entity

is a point like entity like explosions or teleport fog. It needs a position.

`#define TE_EXPLOSION1 5`

boss, barrels etc. explosion

`#define TE_EXPLOSION2 6`

barrels explosion

`#define TE_ROCKET_EXPLOSION 7`

rocket explosion

`#define TE_GRENADE_EXPLOSION 8`

grenade explosion

`#define TE_ROCKET_EXPLOSION_WATER 17`

rocket explosion under water

`#define TE_GRENADE_EXPLOSION_WATER 18`

grenade explosion under water

`#define TE_BFG_EXPLOSION 20`

BFG explosion

`#define TE_BFG_BIGEXPLOSION 21`

big BFG explosion

`#define TE_BOSSTPORT 22`

boss teleports away

`#define TE_PLASMA_EXPLOSION 28`

(new in 3.15, MP1) plasma explosion on touch

`#define TE_PLAIN_EXPLOSION 35`

(new in 3.18, MP2) flame burst, turret death

`#define TE_CHAINFIST_SMOKE 45`

(new in 3.18, MP2) *this spits out some smoke from the motor. it's a two-stroke, you know.*

`#define TE_TRACKER_EXPLOSION 47`

(new in 3.18, MP2) tracker explosion

`#define TE_TELEPORT_EFFECT 48`

(new in 3.18, MP2) teleport away effect

`#define TE_DBALL_GOAL 49`

(new in 3.18, MP2) ball disappears after a goal

`#define TE_NUKEBLAST 51`

(new in 3.18, MP2) nuke explosion

`#define TE_WIDOWSPLASH 52`

(new in 3.18, MP2) last part of the death of the widow

`#define TE_EXPLOSION1_BIG 53`

(new in 3.18, MP2) big explosion

`#define TE_EXPLOSION1_NP 54`

(new in 3.18, MP2) widow explosion

impact entity

is a small point like entity, spawned on an impact of a bullet etc. It needs a position (often a trace end) and a direction from there.

`#define TE_GUNSHOT 0`

hit with Machine Gun and Chain Gun

`#define TE_BLOOD 1`

clients and monsters bleed when hit

`#define TE_BLASTER 2`

hit with Blaster

`#define TE_SHOTGUN 4`

hit with Shotgun

`#define TE_SPARKS 9`

hit someone

`#define TE_SCREEN_SPARKS 12`

hit on an Energy Armor

`#define TE_SHIELD_SPARKS 13`

hit on a different Energy Armor

`#define TE_BULLET_SPARKS 14`

hit someone with bullets

`#define TE_GREENBLOOD_new 26`

(new in 3.15, MP1) a “gekk” bleeds green. Note: the value 26 stood from game version 3.12 to 3.14 (protocol version 31) for `TE_PLASMATRAIL` and was a line entity.

`#define TE_GREENBLOOD_old 27`

(from 3.12 to 3.14) not used. Note: the value 27 stands from game version 3.15 (protocol version >= 32) on for `TT_BLUEHYPERBLASTER` and is a line entity.

`#define TE_BLASTER2 30`

(new in 3.18, MP2) hit by green monster blaster shot

`#define TE_MOREBLOOD 42`

(new in 3.18, MP2) more blood with chainfist

`#define TE_HEATBEAM_SPARKS 43`

(new in 3.18, MP2) heat beam water impact

`#define TE_HEATBEAM_STEAM 44`

(new in 3.18, MP2) heat beam impact

`#define TE_FLECHETTE 55`

(new in 3.18, MP2) touch a flechette

line entity

is a 2 dimensional entity. It needs an origin and an end position.

`#define TE_RAILTRAIL 3`

trail of the Rail Gun (blue)

`#define TE_BUBBLETRAIL 11`

shot with a bullet through water

`#define TE_BFG_LASER 23`

(new in 3.12) BFG laser

`#define TE_PLASMATRAIL 26`

(from 3.12 to 3.14) not used. Note: the value 26 stands from game version 3.15 (protocol version >= 32) on for `TE_GREENBLOOD_new` and is an impact entity.

`#define TE_BLUEHYPERBLASTER 27`

(new in 3.15, MP1) not used. Note: the value 27 stood from game version 3.12 to 3.14 (protocol version 31) for `TE_GREENBLOOD_old` and was an impact entity.

`#define TE_DEBUGTRAIL 34`

(new in 3.18, MP2) debug use, trail drawing

`#define TE_BUBBLETRAIL2 41`

(new in 3.18, MP2) bubbles above heat beam

`#define TE_ELECTRIC_SPARKS 46`

(new in 3.18, MP2) hit on something mechanical

special entity

is a entity which doesn't fit into the other categories.

`#define TE_SPLASH 10`

creates a particle splash effect

`#define TE_LASER_SPARKS 15`

laser hits a wall

`#define TE_PARASITE_ATTACK 16`

parasite attack

`#define TE_MEDIC_CABLE_ATTACK 19`

medic cable attac

`#define TE_GRAPPLE_CABLE 24`

(new in 3.12) used in CFT

`#define TE_WELDING_SPARKS 25`

(new in 3.12) ionripper fire, fixbot melder fire

`#define TE_TUNNEL_SPARKS 29`

(new in 3.15, MP1) trigger effect

`#define TE_FLAME 32`

(new in 3.18, MP2) flame thrower, Quake II can't parse this

`#define TE_LIGHTNING 33`

(new in 3.18, MP2) tesla weapon

`#define TE_FLASHLIGHT 36`

(new in 3.18, MP2) player with torch

`#define TE_FORCEWALL 37`

(new in 3.18, MP2) force wall

`#define TE_HEATBEAM 38`

(new in 3.18, MP2) heat beam

`#define TE_MONSTER_HEATBEAM 39`

(new in 3.18, MP2) monster fires a heat beam

`#define TE_STEAM 40`

(new in 3.18, MP2) *Creates a steam effect (particles w/ velocity in a line).*

`#define TE_WIDOWBEAMOUT 50`

(new in 3.18, MP2) part of the death of the widow

bad entity

is a entity which is not used anymore.

`#define TE_RAILTRAIL2 31`

(new in 13.18, MP2) not used, Quake II can't parse this

`long dest_entity;`

is the entity, where an effect ends.

`long entity;`

is the entity, which spawned the effect.

`long nextid;`

is a special value to chain multiple `TE_STEAM` effects. The last one has here the value `-1`.

`long count;`

is a number for multi-particle temporary entities.

`vec3_t start;`

is the start of the unparsable temporary `TE_FLAME` entity.

`vec3_t dest_origin;`

is the origin of an entity, where the temporary entity ends.

`long type;`

is an additional type value for the temporary `TE_WIDOWBEAMOUT` entity. number for multi-particle temporary. It may be 20001 or 20002 (source, `roguesrc320/game/m_widow.c`, 831 - 842).

`vec3_t origin;`

is the origin of the temporary entity.

`long flash_entity;`

is the entity, which spawned the temporary `TE_FLASHLIGHT` entity.

`vec3_t trace_endpos;`

is the end position of the line like temporary entity.

`vec3_t pos1, pos2, pos3, pos4;`

are additional positions for bigger enitities.

`vec3_t movedir;`

is the moving direction for the temporary entity.

`long style;`

is an additional style value. In the special `TE_SPLASH` temporary entity it should be one of the `SPLASH_` constants (source, `q2src320/game/q_shared.h`, 944 - 950).

`long plat2flags;`

are used for the `TE_STEAM` temporary entity.

`long wait;`

is a timeout value (in milliseconds) for the `TE_STEAM` temporary entity.

parse routine

```
entitytype = ReadByte;
switch (entitytype) {
  // version problems
  // case TE_PLASMATRAIL:
  case TE_GREENBLOOD_new:
    if (serverversion >= 32) // game version >= 3.15
      goto impact_entity;
    else
      goto line_entity;
  break;
  // case TE_GREENBLOOD_old:
  case TE_BLUEHYPERBLASTER:
    if (serverversion >= 32) // game version >= 3.15
      goto line_entity;
    else
      goto impact_entity;
  break;
  // point entity
  case TE_EXPLOSION1:
  case TE_EXPLOSION2:
  case TE_ROCKET_EXPLOSION:
  case TE_GRENADE_EXPLOSION:
  case TE_ROCKET_EXPLOSION_WATER:
  case TE_GRENADE_EXPLOSION_WATER:
  case TE_BFG_EXPLOSION:
  case TE_BFG_BIGEXPLOSION:
  case TE_BOSSTPORT:
  case TE_PLASMA_EXPLOSION:
  case TE_PLAIN_EXPLOSION:
  case TE_CHAINFIST_SMOKE:
  case TE_TRACKER_EXPLOSION:
  case TE_TELEPORT_EFFECT:
  case TE_DBALL_GOAL:
  case TE_NUKEBLAST:
  case TE_WIDOWSPLASH:
  case TE_EXPLOSION1_BIG:
  case TE_EXPLOSION1_NP:
    ReadPosition(origin);
  break;
  // impact entity
  case TE_GUNSHOT:
  case TE_BLOOD:
  case TE_BLASTER:
  case TE_SHOTGUN:
  case TE_SPARKS:
```

```
case TE_SCREEN_SPARKS:
case TE_SHIELD_SPARKS:
case TE_BULLET_SPARKS:
// case TE_GREENBLOOD_new:
// case TE_GREENBLOOD_old:
case TE_BLASTER2:
case TE_MOREBLOOD:
case TE_HEATBEAM_SPARKS:
case TE_HEATBEAM_STEAM:
case TE_ELECTRIC_SPARKS:
case TE_FLECHETTE:
impact_entity:
  ReadPosition(origin);
  ReadDir(movedir);
break;
// line entity
case TE_RAILTRAIL:
case TE_BUBBLETRAIL:
case TE_BFG_LASER:
// case TE_PLASMATRAIL:
// case TE_BLUEHYPERBLASTER:
case TE_DEBUGTRAIL:
case TE_BUBBLETRAIL2:
line_entity:
  ReadPosition(origin);
  ReadPosition(trace_endpos);
break;
// special entity
case TE_SPLASH:
case TE_LASER_SPARKS:
case TE_WELDING_SPARKS:
case TE_TUNNEL_SPARKS:
  count = ReadByte;
  ReadPosition(origin);
  ReadDir(movedir);
  style = ReadByte;
break;
case TE_PARASITE_ATTACK:
case TE_MEDIC_CABLE_ATTACK:
case TE_HEATBEAM:
case TE_MONSTER_HEATBEAM:
  entity = ReadShort;
  ReadPosition(origin);
  ReadPosition(trace_endpos);
break;
case TE_GRAPPLE_CABLE:
  entity = ReadShort;
  ReadPosition(origin);
  ReadPosition(trace_endpos);
  ReadPosition(pos1);
break;
case TE_FLAME: // Quake2 can't parse this!
  entity = ReadShort;
```

```
      count = ReadShort;
      ReadPosition(start);
      ReadPosition(origin);
      ReadPosition(pos1);
      ReadPosition(pos2);
      ReadPosition(pos3);
      ReadPosition(pos4);
    break;
    case TE_LIGHTNING:
      dest_entity = ReadShort;
      entity = ReadShort;
      ReadPosition(dest_origin);
      ReadPosition(origin);
    break;
    case TE_FLASHLIGHT:
      ReadPosition(origin);
      flash_entity = ReadShort;
    break;
    case TE_FORCEWALL:
      ReadPosition(origin);
      ReadPosition(trace_endpos);
      style = ReadShort;
    break;
    case TE_STEAM:
      nextid = ReadShort;
      count = ReadByte;
      ReadPosition(origin);
      ReadDir(movedir);
      style = ReadByte;
      plat2flags = ReadShort;
      if (nextid != -1)
        wait = ReadLong;
    break;
    case TE_WIDOWBEAMOUT:
      type = ReadShort;
      ReadPosition(origin);
    break;
    case TE_RAILTRAIL2: // senseless, I know
    default:
      error("CL_ParseTEnt: bad type");
    break;
  }
```

3.5. layout

```
ID
    0x04
```

purpose

This message displays the Field Computer ("*cmd help*", bound to F1). It contains the summary screen (Deathmatch Scoreboard or single player secrets, goals etc.). It stores the message only on the client siede. The actual display will be triggered by the *playerinfo* message with a special `stats` command.

variable

```
char text[MAX_MSGLEN];
```

is the summary screen text with some kind of control characters. The control language is very simple. Read some examples in the source, `q2src320/game/p_hud.c`, functions `DeathmatchScoreboardMessage` and `HelpComputer`.

parse routine

```
text = ReadString;
```

3.6. inventory

`ID`

`0x05`

purpose

Tells the clients its inventory.

variable

```
int inventory[MAX_ITEMS];
```

is the player's inventory array.

`#define MAX_ITEMS 256`
is the number different items in the inventory.

The inventory concept of Quake II is very open and expandable. In an global array (source, `q2src320/game/g_items.c`, 1115 - 2097)

`gitem_t itemlist[];`
is the list of all possible things a player can pickup and carry around. This array will be filled at compile time. In the `inventory` array are stored how many of each thing of the `itemlist` a players carries. The server tells the meaning of each index with *configstring* messages. I give

the original table for `itemlist`. Every modification can change it totally and it changed indeed from 3.05 to 3.12.

Table 2. Standard inventory entries.

value	purpose
0	not used
1	Body Armor
2	Combat Armor
3	Jacket Armor
4	Armor Shard
5	Power Screen
6	Power Shield
7	Blaster
8	Shotgun
9	Super Shotgun
10	Machinegun
11	Chaingun
12	Grenades
13	Grenade Launcher
14	Rocket Launcher
15	HyperBlaster
16	Railgun
17	BFG10K
18	Shells
19	Bullets
20	Cells
21	Rockets
22	Slugs
23	Quad Damage
24	Invulnerability
25	Silencer
26	Rebreather
27	Environment Suit
28	Ancient Head
29	Adrenaline
30	Bandolier
31	Ammo Pack
32	Data CD
33	Power Cube
34	Pyramid Key

35	Data Spinner
36	Security Pass
37	Blue Key
38	Red Key
39	Commander's Head
40	Airstrike Marker
41	Health

parse routine

```
for (i=0 ; i<MAX_ITEMS ; i++) inventory[i] = ReadShort;
```

3.7. nop

`Quake II ID`

```
0x06
```

purpose

Do nothing.

parse routine

none

3.8. disconnect

`Quake II ID`

```
0x07
```

purpose

Diconnect from the server.

parse routine

```
print("Server disconnected\n");
```

3.9. reconnect

```
Quake II ID
    0x08
```

purpose

Reconnect to the server.

parse routine

```
    print("Server disconnected, reconnecting\n");
```

3.10. sound

```
Quake II ID
    0x09
```

purpose

Plays a sound.

variables

`long mask;`

is a bit mak to reduce the network traffic.

`long soundnum;`

is the index in the precache sound table.

`float vol;`

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

`float attenuation;`

is the attenuation of the sound. `attenuation` should have one of the `ATTN_` constants (source, `q2src320/game/q_shared.h`, 966 - 970).

`#define ATTN_NONE 0`

full volume the entire level

`#define ATTN_NORM 1`

the normal attenuation

`#define ATTN_IDLE 2`

for idle monsters

`#define ATTN_STATIC 3`

diminish very rapidly with distance

`float timeofs;`

is the offset in seconds between the frame start and the sound start. It is 0 in the entire source.

`long channel;`

is the sound channel. There are 8 possible sound channels for each entity in Quake II (0-7) but it uses 5 only. `channel` should be one of the `CHAN_` constants (source, `q2src320/game/q_shared.h`, 953 - 960).

`#define CHAN_AUTO 0`

selects a channel automatically

`#define CHAN_WEAPON 1`

weapon use sounds

`#define CHAN_VOICE 2`

pain calls

`#define CHAN_ITEM 3`

item get sounds

`#define CHAN_BODY 4`

jump and fall sounds

`long entity;`

is the entity which caused the sound. The maximum value of `entity` is

`#define MAX_EDICTS 1024`

.

`vec3_t origin;`

is the origin of the sound.

parse routine

```
long entity_channel; // combined variable

mask = ReadByte;
soundnum = ReadByte;
vol = (mask & 0x01) ? ((float)ReadByte / 255.0) : (1.0);
attenuation = (mask & 0x02) ? ((float)ReadByte / 64.0) : (1.0);
timeofs = (mask & 0x10) ? ((float)ReadByte * 0.001) : (0.0);
if (mask & 0x08) {
  entity_channel = ReadShort;
  entity = (entity_channel >> 3);
  channel = entity_channel & 0x07;
  if (entity > MAX_EDICTS) {
    error("CL_ParseStartSoundPacket: ent = %i", entity);
  }
} else {
  channel = 0;
  entity = 0;
}
if (mask & 0x04) {
  ReadPosition(origin);
}
```

3.11. print

`Quake II ID`

`0x0A`

purpose

Prints a text at the top of the screen.

variables

`long level;`

is the priority level. `level` should be one of the following:

`#define PRINT_LOW 0`

pickup messages

```
#define PRINT_MEDIUM 1
```

death messages

```
#define PRINT_HIGH 2
```

critical messages

```
#define PRINT_CHAT 3
```

chat messages

```
char string[MAX_MESSAGE_SIZE];
```

is the the text to be displayed.

parse routine

```
level = ReadByte;
if (level == PRINT_CHAT) sound("misc/talk.wav");
string = ReadString;
```

3.12. stufftext

```
Quake II ID
```

```
0x0B
```

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;
```

3.13. serverdata

`Quake II ID`

`0x0C`

purpose

Set some global info.

variables

`long serverversion;`

is the protocol version coming from the server.

Table 3. Quake II values for `PROTOCOL_VERSION`.

game	protocol
3.00	25
3.05	26
3.06	26
3.07	27
3.08	27
3.09	28
3.10	30
3.12	31
3.13	31
3.14	31
3.15	32
3.17	33
3.18	34
3.19	34
3.20	34

`long key;`

some kind of key. Will be used again later in a *stufftext* message for the login hand-shake.

`long isdemo;`

indicates the demo type. Possible values are:

`#define RECORD_NETWORK 0x00`

data actually over the wire (proxy)

`#define RECORD_CLIENT 0x01`

recorded on the client side. Containes information in the direct neighborhood of the recording player.

`#define RECORD_SERVER 0x02`

recorded on the server side. Containes all information on all entities in the level. These files tend to become very large. They can't be played back by Quake II directly. This value appears in 3.17 but server side recordings work from 3.15 on. This variable is sometimes called `attractloop` for whatever reason.

`#define RECORD_RELAY 0x80`

recorded with the Relay modificiation at the server side. These files can't be played back only with the Relay modification.

`char* game;`

is the game directory (may be empty, which means `baseq2`).

`long client;`

is the client id.

`char* mapname;`

is the name of the map.

`int compatible;`

compatibility with the CD retail version 3.05 (protocol 26). How to set this? ??FIXME??

parse routine

```
log("Serverdata packet received.\n");
serverversion = ReadLong;
if (!compatible) {
  if (serverversion != PROTOCOL_VERSION)
    error("Server returned version %i, not %i", serverversion, PROTOCOL_VERSION);
  }
}
key = ReadLong;
isdemo = ReadByte;
game = ReadString;
client = ReadShort;
mapname = ReadString;
```

3.14. configstring

`Quake II ID`

`0x0D`

purpose

config strings are a general means of communication from the server to all connected clients. Each config string can be at most `MAX_QPATH` *characters.*

variables

`int index;`

is the number of the config string. The following constants (source, `q2src320/game/q_shared.h`, 1069 - 1092) determine where to find something in the full array of config strings.

`#define CS_NAME 0`

is the name of the level.

`#define CS_CDTRACK 1`

is the audio CD track for this level.

`#define CS_SKY 2`

is the sky texture.

`#define CS_SKYAXIS 3`

is the sky axis in the *%f %f %f format*.

`#define CS_SKYROTATE 4`

is the rotation speed in the format %f.

`#define CS_STATUSBAR 5`

is the start of the list of *display program string*s for the statusbar.

`#define CS_MAXCLIENTS 30`

is the current maximum number of clients on a server.

`#define CS_MAPCHECKSUM 31`

is the map checksum *for catching cheater maps*.

`#define CS_MODELS 32`

is the start of the precache model table.

`#define MAX_MODELS 256`

is the maximum number of the precache model table.

`#define CS_SOUNDS (CS_MODELS+MAX_MODELS)`

(288) is the start of the precache sound table.

`#define MAX_SOUNDS 256`

is the maximum number of the precache sound table.

`#define CS_IMAGES (CS_SOUNDS+MAX_SOUNDS)`

(544) is the start of the image list.

`#define MAX_IMAGES 256`

is the maximum numer of the images.

`#define CS_LIGHTS (CS_IMAGES+MAX_IMAGES)`

(800) is the start of the light styles list.

`#define MAX_LIGHTSTYLES 256`

is the maximum number of the light styles.

`#define CS_ITEMS (CS_LIGHTS+MAX_LIGHTSTYLES)`

(1056) is the start of the items list.

`#define MAX_ITEMS 256`

is the maximum number of items in the inventory list.

`#define CS_PLAYERSKINS (CS_ITEMS+MAX_ITEMS)`

(1312) is the start of the player skin list.

`#define MAX_CLIENTS 256`

is the maximum number of players.

`#define CS_GENERAL (CS_PLAYERSKINS+MAX_CLIENTS)`

(1568) is the start of the general config strings list.

`#define MAX_GENRAL (MAX_CLIENTS*2)`

(512) is the maximum number of *general config strings*.

`#define MAX_CONFIGSTRINGS (CS_GENERAL+MAX_GENERAL)`

(2080) is the maximum number of config strings.

`char string[MAX_QPATH];`

is the corresponding config string.

`#define MAX_QPATH 64`

is the maximum length of a config string.

parse routine

```
index = ReadShort;
if (index > MAX_CONFIGSTRINGS) error("configstring > MAX_CONFIGSTRINGS");
string = ReadString;
```

3.15. spawnbaseline

`ID`

`0x0E`

purpose

Spawns a new entity.

variables

`long mask;`

is a bit-mask to reduce the network traffic.

`long entity;`

is the number of the entity.

`vec3_t origin;`

is the origin.

`vec3_t angles;`

is the orientation.

`vec3_t old_origin;`

is the old origin *for lerping*. It is used for client-side prediction and interpolation calculations. To compress a DM2 file, just leave it out, if the last frame had a `origin` entry for the entity in question. It is used with *cl_nodelta 1* only. From 3.17 on `old_origin` is used for player entities only.

`long modelindex;`

is the model index.

`long modelindex2;`

is *weapons, CTF, flags, etc*.

`long modelindex3;`

is *weapons, CTF, flags, etc*.

`long modelindex4;`

is *weapons, CTF, flags, etc*.

`long frame;`

is the frame of the model.

`long skin;`

is the number of the skin for the model.

`long vwep;`

is the number of visual weapon skin for the model.

`long effects;`

Effects are things handled on the client side (lights, particles, frame animations) that happen constantly on the given entity. An entity that has effects will be sent to the client even if it has a zero index model. The bit-mask `effects` holds the `EF_` constants (source, `q2src320/game/q_shared.h`, 530 - 569).)

`long renderfx;`

are some special render flags. The bit-mask `renderfx` holds the `RF_` constants (source, `q2src320/game/q_shared.h`, 571 - 591).

`long solid;`

for client side prediction, 8(bits 0-4) is x/y radius 8*(bits 5-9) is z down distance, 8(bits10-15) is z up*

`long sound;`

for looping sounds, to guarantee shutoff

`long event;`

impulse events -- muzzle flashes, footsteps, etc events only go out for a single frame, they are automatically cleared each frame `event` should have one of the following values (source, `q2src320/game/q_shared.h`, 1098 - 1112):

```
typedef enum
{
        EV_NONE,            // 0
        EV_ITEM_RESPAWN,    // 1
        EV_FOOTSTEP,        // 2
        EV_FALLSHORT,       // 3
        EV_FALL,            // 4
```

```
            EV_FALLFAR,          // 5
            EV_PLAYER_TELEPORT   // 6
    } entity_event_t;
```

parse routine

```
mask = ReadByte;
if (mask & 0x00000080) mask |= (ReadByte <<  8);
if (mask & 0x00008000) mask |= (ReadByte << 16);
if (mask & 0x00800000) mask |= (ReadByte << 24);
entity = (mask & 0x00000100) ? ReadShort : ReadByte;
if (mask & 0x00000800) modelindex = ReadByte;
if (mask & 0x00100000) modelindex2 = ReadByte;
if (mask & 0x00200000) modelindex3 = ReadByte;
if (mask & 0x00400000) modelindex4 = ReadByte;
if (mask & 0x00000010) frame = ReadByte;
if (mask & 0x00020000) frame = ReadShort;
if (mask & 0x00010000) {
  if (mask & 0x02000000) skin = ReadLong;
  else skin = ReadByte;
}
else {
  if (mask & 0x02000000) skin = ReadShort;
}
vwep = skin >> 8;
skin &= 0xFF;
if (mask & 0x00004000) {
  if (mask & 0x00080000) effects = ReadLong;
  else effects = ReadByte;
}
else {
  if (mask & 0x00080000) effects = ReadShort;
}
if (mask & 0x00001000) {
  if (mask & 0x00040000) renderfx = ReadLong;
  else renderfx = ReadByte;
}
else {
  if (mask & 0x00040000) renderfx = ReadShort;
}
if (mask & 0x00000001) origin[0] = ReadCoord;
if (mask & 0x00000002) origin[1] = ReadCoord;
if (mask & 0x00000200) origin[2] = ReadCoord;
if (mask & 0x00000400) angles[0] = ReadAngle;
if (mask & 0x00000004) angles[1] = ReadAngle;
if (mask & 0x00000008) angles[2] = ReadAngle;
if (mask & 0x01000000) ReadPosition(old_origin);
if (mask & 0x04000000) sound = ReadByte;
event = (mask & 0x00000020) ? ReadByte : 0;
if (mask & 0x08000000) solid = ReadShort;
```

3.16. centerprint

`ID`

`0x0F`

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.

variables

```
char* text;
```

is the text to be displayed.

parse routine

```
text = ReadString;
```

3.17. download

`ID`

`0x10`

purpose

Download a file (sound, model etc.) from the server. It needs at least version 3.15 to transfer any data. There is no position information in the packet, so the client can't rearrange packets, which arrive in the wrong order. Therefore all download network packets are reliable packets and they need the usual acknowledgement. The client itself starts the whole thing, if something is missing and asks the server for it. I'm not sure, if such blocks may actually do something useful in a DM2 file.

variables

```
long size;
```

is the number of bytes transferred in the current packet.

```
long percent;
```

is the total amount sended in percent.

`char* downloadbuffer;`

is a buffer for downloaded files.

`char* filename;`

is the name for the downloaded file.

`FILE* fp;`

is the file pointer for the downloaded file.

parse routine

```
size = ReadShort;
percent=ReadByte;
if (size == -1) {
  error("File not found.\n");
}
if (serverdata.serverversion >= 32) { // game version >= 3.15
  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);
  }
}
```

3.18. playerinfo

`ID`

`0x11`

purpose

Player info. *Have to* come directly after a *frame* message in client recording. There is no such message in server side recordings since there is no special player who does the recording.

variables

`long mask;`

is a bit mask to reduce the network traffic.

`long mask2;`

is a bit mask to reduce the network traffic.

`long pm_type;`

is important for client side prediction and should be one of the `PM_` constants (source, `q2src320/game/q_shared.h`, 440 - 451).

`vec3_t origin;`

is the origin.

`vec3_t velocity;`

is the velocity.

`byte pm_flags;`

ducked, jump_held, etc and should be one of the `PMF_` constants (source, `q2src320/game/q_shared.h`, 453 - 460).

`byte pm_time;`

is unknown (*each unit = 8 ms*).

`short gravity;`

is the gravity (800, cvar `sv_gravity`).

`vec3_t delta_angles;`

add to command angles to get view direction, changed by spawns, rotating objects, and teleporters

`vec3_t viewangles;`

for fixed views

`vec3_t viewoffset;`

add to `pmovestate->origin`

`vec3_t kick_angles;`

add to view direction to get render angles set by weapon kicks, pain effects, etc

`vec3_t gunangles;`

direction of weapon

`vec3_t gunoffset;`

offset of weapon

`int gunindex;`

model index of weapon

`int gunframe;`

frame of weapon

`float blend[4];`

rgba full screen effect

`long fov;`

horizontal field of view. Since the field of view is no client side variable (as it is in Quake), it is much easier to create zoom effects in movies.

`int rdflags;`

refdef flags, should be one of the `RDF_` constants (source, `q2src320/game/q_shared.h`, 593 - 600).

`short stats[MAX_STATS];`

fast status bar updates Each entry in this array stays for something on the statusbar. The value of an icon entry is the image index defined in a *configstring* message. A value of 0 on an icon entry switches the icon off. (source, `q2src320/game/q_shared.h`, 973 - 993)

`#define MAX_STATS 32`

maximum number of things in the status bar

`#define STAT_HEALTH_ICON 0`

health icon image

`#define STAT_HEALTH 1`

health value

`#define STAT_AMMO_ICON 2`

ammo icon image

`#define STAT_AMMO 3`

ammo value

`#define STAT_ARMOR_ICON 4`

armour icon

`#define STAT_ARMOR 5`

armour value

`#define STAT_SELECTED_ICON 6`

icon image of the selected weapon

`#define STAT_PICKUP_ICON 7`

icon image of a recently gotten thing

`#define STAT_PICKUP_STRING 8`

configstring index to describe the recently gotten thing, 0 = string off

`#define STAT_TIMER_ICON 9`

icon image of timed object (Quad Damage, Invulnerability etc.)

`#define STAT_TIMER 10`

timer value for a timed object (in seconds)

`#define STAT_HELPICON 11`

help icon: 0 off, 1 on

`#define STAT_SELECTED_ITEM 12`

item number of the selected weapon

`#define STAT_LAYOUTS 13`

layout activator: 0 off, 1 display help/summary screen, 2 display inventory

`#define STAT_FRAGS 14`

client score value

`#define STAT_FLASHES 15`

flash the backgrounds behind the status numbers, cleared each frame, 1 = health, 2 = armor, 0 = off

`#define STAT_CHASE 16`

chase target skin number (number in *configstring* list)

`#define STAT_SPECTATOR 17`

0 = normal play, 1 = spectator

parse routine

```
mask = ReadShort;
if (mask & 0x0001) pm_type = ReadByte;
if (mask & 0x0002)
  ReadPostion(origin);
if (mask & 0x0004)
  ReadPosition(velocity);
```

```
if (mask & 0x0008) teleport_time = ReadByte;
if (mask & 0x0010) pm_flags = ReadByte;
if (mask & 0x0020) gravity = ReadShort;
if (mask & 0x0040) {
  delta_angles[0] = ReadAngle16;
  delta_angles[1] = ReadAngle16;
  delta_angles[2] = ReadAngle16;
}
if (mask & 0x0080) {
  viewoffset[0] = ReadChar / 4.0;
  viewoffset[1] = ReadChar / 4.0;
  viewoffset[2] = ReadChar / 4.0;
}
if (mask & 0x0100) {
  viewangles[0] = ReadAngle16;
  viewangles[1] = ReadAngle16;
  viewangles[2] = ReadAngle16;
}
if (mask & 0x0200) {
  kick_angles[0] = ReadChar / 4.0;
  kick_angles[1] = ReadChar / 4.0;
  kick_angles[2] = ReadChar / 4.0;
}
if (mask & 0x1000) gunindex = ReadByte;
if (mask & 0x2000) {
  gunframe = ReadByte;
  gunoffset[0] = ReadChar / 4.0;
  gunoffset[1] = ReadChar / 4.0;
  gunoffset[2] = ReadChar / 4.0;
  gunangles[0] = ReadChar / 4.0;
  gunangles[1] = ReadChar / 4.0;
  gunangles[2] = ReadChar / 4.0;
}
if (mask & 0x0400) {
  blend[0] = ReadByte / 255.0;
  blend[1] = ReadByte / 255.0;
  blend[2] = ReadByte / 255.0;
  blend[3] = ReadByte / 255.0;
}
if (mask & 0x0800) fov = ReadByte;
if (mask & 0x4000) rdflags = ReadByte;
mask2 = ReadLong;
for (i=0;i<32;i++) if (mask2 & (0x00000001 << i)) stats[i] = ReadShort;
```

3.19. packetentities

ID

0x12

purpose

Entity updates. *Have to* come in a client side recording directly after a *playerinfo* message and in a server side recording directly after a *frame* message. This message is in fact a list of *spawnbaseline* messages. Look there for a longer variable description. The list ends with `entity==0`.

variables

`long mask;`

is used to reduce the network traffic.

`long entity;`

is the number of the entity.

`long remove;`

indicates a disappearing entity.

`vec3_t origin;`

is the origin.

`vec3_t angles;`

is the orientation.

`vec3_t old_origin;`

old origin (for client side prediction).

`long modelindex;`

is the model index.

`long modelindex2;`

is *weapons, CTF, flags, etc*.

`long modelindex3;`

is *weapons, CTF, flags, etc*.

`long modelindex4;`

is *weapons, CTF, flags, etc*.

`long frame;`

is the frame of the model.

`long skin;`

is the number of the skin for the model.

`long vwep;`

is the number of visual weapon skin for the model.

`long effects;`

is the entity effect.

`long renderfx;`

are some special render flags.

`long solid;`

for client side prediction, 8(bits 0-4) is x/y radius 8*(bits 5-9) is z down distance, 8(bits10-15) is z up*

`long sound;`

for looping sounds, to guarantee shutoff

`long event;`

impulse events -- muzzle flashes, footsteps, etc events only go out for a single frame, they are automatically cleared each frame

parse routine

```
for (;;) {
  mask = ReadByte;
  if (mask & 0x00000080) mask |= (ReadByte <<  8);
  if (mask & 0x00008000) mask |= (ReadByte << 16);
  if (mask & 0x00800000) mask |= (ReadByte << 24);
  entity = (mask & 0x00000100) ? ReadShort : ReadByte;
  if (entity >= MAX_EDICTS) error("CL_ParsePacketEntities: bad number:%i",entity);
  if (entity == 0) break;
  remove = (mask & 0x00000040) ? 1 : 0;
  if (mask & 0x00000800) modelindex = ReadByte;
  if (mask & 0x00100000) modelindex2 = ReadByte;
  if (mask & 0x00200000) modelindex3 = ReadByte;
  if (mask & 0x00400000) modelindex4 = ReadByte;
  if (mask & 0x00000010) frame = ReadByte;
  if (mask & 0x00020000) frame = ReadShort;
  if (mask & 0x00010000) {
    if (mask & 0x02000000) skin = ReadLong;
    else skin = ReadByte;
  }
  else {
    if (mask & 0x02000000) skin = ReadShort;
  }
  vwep = skin >> 8;
  skin &= 0xFF;
  if (mask & 0x00004000) {
    if (mask & 0x00080000) effects = ReadLong;
```

```
      else effects = ReadByte;
    }
    else {
      if (mask & 0x00080000) effects = ReadShort;
    }
    if (mask & 0x00001000) {
      if (mask & 0x00040000) renderfx = ReadLong;
      else renderfx = ReadByte;
    }
    else {
      if (mask & 0x00040000) renderfx = ReadShort;
    }
    if (mask & 0x00000001) origin[0] = ReadCoord;
    if (mask & 0x00000002) origin[1] = ReadCoord;
    if (mask & 0x00000200) origin[2] = ReadCoord;
    if (mask & 0x00000400) angles[0] = ReadAngle;
    if (mask & 0x00000004) angles[1] = ReadAngle;
    if (mask & 0x00000008) angles[2] = ReadAngle;
    if (mask & 0x01000000) ReadPosition(old_origin);
    if (mask & 0x04000000) sound = ReadByte;
    event = (mask & 0x00000020) ? ReadByte : 0;
    if (mask & 0x08000000) solid = ReadShort;
  }
```

3.20. deltapacketentities

ID

```
0x13
```

purpose

Entity updates. May come after a *packetentities* block. It isn't really necessary since *packetentities* handles itself all the delta encoding.

parse routine

```
unknown ??FIXME??
```

3.21. frame

ID

```
0x14
```

purpose

Sequence numbers to cope with UDP packet loss, delta encoding and portal border crossings. Start of the *playerinfo* and *packetentities* messages.

In server side recordings this message contains only the current frame number.

In Relay recordings, this message looks like the client side *frame* message but it contains also the list of connected clients.

variables

`long seq1;`

is the sequence number of the current packet or frame. Since the Quake II server uses a fixed time gap of 100ms (10Hz) between game state changes `seq1 / 10` is the time in seconds since the server started.

`long seq2;`

is the sequence number of the delta reference frame. The reference holds for both following *playerinfo* and *packetentities* messages. The Quake II server has a table with some old entity states and get from each client the sequence number of frames, which arrived correctly at the client side. So the server can decide which most currently sended old frame should be used now as the delta encoding reference frame. `seq2` is -1 to reference to the *spawnbaseline* values.

`long count;`

is the number of bytes in the `areas` array.

`#define MAX_MAP_AREAS 256`

is the maximum number of areas in a map.

`unsigned char areas[MAX_MAP_AREAS / 8];`

is the array which defines the areas to be rendered. Each bit in the `areas` array stays for one area in the map.

`long frame;`

is the frame number in server side recordings and similar to `seq1`. I may rename it even to `seq1` in a future revision.

`long connected_count`

is the number of connected clients in the `connected` array. It is used in Relay recordings only.

`unsigned char connected[MAX_CLIENTS]`

is the array, which contains the numbers of the connected clients. It is used in Relay recordings only.

parse routine

```
long uk_b1;

if (serverdata.isdemo == RECORD_CLIENT ||
    serverdata.isdemo == RECORD_NETWORK ||
    serverdata.isdemo == RECORD_RELAY) {
  seq1 = ReadLong;
  seq2 = ReadLong;
  if (serverdata.serverversion != 26) uk_b1 = ReadByte;
  count = ReadByte;
  for (i=0;i<count;i++) areas[i] = ReadByte;
  if (serverdata.isdemo == RECORD_RELAY) {
    connected_count = ReadByte;
    for (i=0;i<connected_count;i++) connected[i] = ReadByte;
  }
}
if (serverdata.isdemo == RECORD_SERVER) {
  frame = ReadLong;
}
```

4. Version History and Acknowledgements

0.0.1, 13 December, 1997

- First version (working paper) completed.
- Only based on the published game source.
- Never ever published.

0.0.2, 23 December, 1997

- Old 3.00 ID codes included.
- Version info included.
- All simple ID codes ok.
- packetentities, deltapacketentities, playerinfo missing.
- spawnbaseline is very difficult.
- Never published.

0.0.3, 27 December, 1997

- Old 3.00 ID codes removed. Nobody wants to know them.
- spawnbaseline is ok now.
- playerinfo ok.
- packetentities should be ok.
- ReadDir for temp_entity ok.
- deltapacketentities missing.
- Never published.

0.0.4, 28 December, 1997

- SGML-Tools 1.0.2 used for formatting.
- Some minor tweaks.

0.0.5, 1 January, 1998

- SGML-Tools 1.0.2 formatting problems solved.
- Long #define tables removed.
- More references to the source.

0.0.6, 12 March, 1998

- Some details better. Many thanks to Kekoa Proundfoot (kekoa@graphics.stanford.edu (mailto:kekoa@graphics.stanford.edu)).
- PlanetQuake is the new home.
- Compatible with Quake II up to version 3.14.
- SGML-Tools 1.0.5 used.

1.0.0, 17 June, 1998

- Thanks to Ben Swartzlander (swartz@rice.edu (mailto:swartz@rice.edu)) for the config string index 30.
- Changed some command names (sound volume, server protocol version, config string, print text).
- Changed the coordinates in playerinfo to standard coordinates.
- Most changes in variable types and names were necessary to reduce the total number of tokens in the Quake and Quake II text parser of LMPC.

- Compatible with Quake II up to 3.15a.

1.0.1, 15 July, 1998

- server side recording information included.
- Compatible with Quake II up to 3.17.
- Some hints on `old_origin` updates and sequence numbering.
- SGML-Tools 1.0.7 used.

1.0.2, 16 August, 1998

- More block size hints.
- Visual Weapon (VWep) support.
- Small cosmetic corrections.
- Some hints to multi-level recordings.

1.0.3, 8 January, 1999

- Missing auxiliary function `ReadCoord` included.
- More versions analysed.
- Compatible with Quake II up to 3.20 with Mission Packs 1 and 2.
- New published source code incorporated.
- SGML-Tools 1.0.9 used.
- General clean-up.

1.0.4, 23 January, 2000

- New DM2 format variant for Quake II Relay modification included.
- Thanks to Conor Davis (cedavis@planetquake.com (mailto:cedavis@planetquake.com)) for the information about the Quake II Relay project.