QuakeC Reference Manual

By:
David “DarkGrue” Hesprich
darkgrue@iname.com

Last Revision: February 4, 1998

Table of Contents

1. Introduction

1.1. What is QuakeC?

QuakeC is a language similar to C. QuakeC source can be compiled with the QuakeC compiler to produce PROGS.DAT, a file that Quake can load at startup. In that file Quake search the engines for various things in the Quake World.

Monsters, player, buttons, weapons are the target of QuakeC, you cannot modify maps or graphics - those must be edited with an external editor. Nor can you edit the functions of the core Quake engine; that is proprietary code held by id Software.

To compile QuakeC programs you must have at least `qcc.tar.gz` from id Software; it contains `QCCDOS.EXE` (the DOS QuakeC compiler – source and executable are available for Win 32 and Linux) and all of the `.QC` files. Note that later versions of the QuakeC base source (such as `PROGS106.ZIP`) and third-party compiliers (such as Lee Smith's ProQCC) exist and make QuakeC development considerably easier.

1.1. Contributions

A major part of this document is taken from the QuakeC compilier source, by id Software. Some material has been updated from Olivier Montanuy's QuakeC Manual 1.0. Original `.TXT` format of the QuakeC Manual 1.0 is by Ferrara Francesco.

2. Lexical Elements

The text of a program consists of the texts of one or more compilations. The text of each compilation is a sequence of separate lexical elements. Each lexical element is formed from a sequence of characters, and is either a *delimiter*, an *identifier*, a *reserved_word*, a *numeric_literal*, a *character_literal*, a *string_literal*, or a *comment*. The meaning of a program depends only on the particular sequences of lexical elements that form its compilations, excluding comments.

The text of a compilation is divided into lines. A semicolon terminates a line.

In some cases an explicit separator is required to separate adjacent lexical elements. A separator is any of a space character, a format effector, or the end of a line, as follows:

- A space character is a separator except within a comment, a string literal, or a character literal.
- The end of a line is always a separator.

One or more separators are allowed between any two adjacent lexical elements, before the first of each compilation, or after the last. At least one separator is required between an identifier, a reserved word, or a numeric literal and an adjacent identifier, reserved word, or numeric literal.

2.1. Delimiters

A *delimiter* is either one of the following special characters:

```
&  '  (  )  *  +  ,  -  .  /  :  ;  <  =  >  |  [  ]
```

or one of the following compound delimiters each composed of two adjacent special characters:

```
==  !=  >=  <=  &&  ||
```

Each of the special characters listed for single character delimiters is a single delimiter except if this character is used as a character of a compound delimiter, or as a character of a comment, string literal, character literal, or numeric literal.

2.2. Whitespace

Whitespace characters are spaces, newlines, tabs, and page breaks. Whitespace is used to improve the readability of your programs and to separate tokens from each other, when necessary. (A token is an indivisible lexical unit such as an identifier or number). Whitespace is otherwise insignificant. Whitespace may occur between any two tokens, but not within a token. Whitespace may also occur within a string, when it is significant.

All whitespace characters are *delimiters.*

2.3. Identifiers

An *identifier* is a sequence of one or more non-delimiter characters. Identifiers are used in several ways in Quake C programs.

- Certain identifiers are reserved for use as syntactic keywords; they should not be used as variables.
- Any identifier that is not a syntactic keyword can be used as an identifier.

A potential identifier is a sequence of non-delimiter characters with a maximum of 64 characters, beginning with "A-Z", "a-z", or "_", and that can continue with those characters in addition to "0-9". Quake C is case-sensitive.

The names of functions, variables and fields must be unique. For instance, you cannot define a variable with the same name as a field.

2.4. Literals

A literal represents a value literally, that is, by means of notation suited to its kind. A literal is either a *numeric_literal*, or a *string_literal*.

2.4.1. Numeric Literals

A *real_literal* is a *numeric_literal* that includes a point; an *integer_literal* is a *numeric_literal* without a point. All numeric values in QuakeC are floating point values.

```
5

0.2

-25.0
```

A *vector_literal* is a set of three *real_literals* enclosed by single quotes (').

```
'0 0 0'

'10 –12.5 0.0001'
```

2.4.2. String Literals

A *string_literal* is formed by a sequence of graphic characters (possibly none) enclosed between two quotation marks (") used as string brackets.

```
"This is a string"

"This is a string followed by a newline\n"
```

2.4.3. Built-in Functions

A built-in function immediate is a pound sign ("#"), followed by an integer.

```
#1

#12
```

2.5. Comments

Comments are the same as in C++ (and many C languages). The beginning of a comment is indicated with a double forward slash (`//`). Quake C ignores everything on a line starting with the forward double slash until the end of the line.

An alternative form of comment (called an *extended comment*) begins with the characters "`/*`" and ends with the characters "`*/`". As with ordinary comments, all of the characters of an extended comment, including the leading "`*/`" and trailing "`*/`", are treated as whitespace. Comments of this form may extend over multiple lines.

```
// followed by comments, until the next line.

/* enclose an extended comment */
```

2.6. Model Pragma

Here are a few definitions that are commonly found in the QuakeC code defining the behavior of animated models (monsters, players, etc.). The QuakeC compiler does not interpret most of this information, but it's useful for the program *modelgen* that generates the models.

2.6.1. Animation Flags

```
$flags rotation
```

Rotation characteristic of the object. QuakeC does not interpret this field, but it's useful for the program *modelgen* that generates the models. Possible values for `$flags`: 8 (the object keeps rotating, like armors, etc.). Other values are not known yet.

2.6.2. Base

```
$base object
```

QuakeC does not interpret this field, but it's useful for the program *modelgen* that generates the models. The parameter `object` is the name of a model file that will be used as a kind of starting position for animation.

2.6.3. Directory

```
$cd <dir>
```

Specify the directory where your model file (.MDL) is located.

2.6.4. Frame Definitions

```
$frame [<frame> [<frame2> ...]]
```

This defines several animation frames of the object. For every animation frame defined, you must also define a QuakeC frame function that will be called during this animation frame.

2.6.5. Model Name

```
$modelname name
```

The parameter `name` is the name of the model file defining the object.

2.6.6. Origin

```
$origin vector
```

This field is not interpreted by QuakeC, but it's useful for the program *modelgen* that generates the models. The parameter `vector` is the location of the object within the bounding box, in the quake editor.

2.6.7. Scale Factor

```
$scale number
```

This field is not interpreted by QuakeC, but it's useful for the program *modelgen* that generates the models. The parameter `number` comes from the *texmake* number that is generated. You can use different values if you want.

2.6.8. Skin File

```
$skin skinfile
```

This field is not interpreted by QuakeC, but it's useful for the program *modelgen* that generates the models. The parameter `skinfile` is the name (without extension) of the .LBM file that defines the skin of the object, as generated by the program *texmake*.

3. Declarations and Types

3.1. Types

A set of values, and a set of primitive operations that implement the fundamental aspects of its semantics characterize a *type*. An object of a given type is a run-time entity that contains (has) a value of the type.

You cannot define new types from the existing ones. In particular, you cannot define new structures or objects. These restrictions make QuakeC compare unfavorably even to BASIC.

You can add only fields to the most important type in QuakeC: *entity*

3.2. Simple Types

3.2.1.1. Void Types

```
void_declaration ⇒ void
```

An empty result, mostly used for definition of procedures (i.e. functions that return no result at all).

3.2.1.2. Floating Point Types

```
floating_point_declaration ⇒ float name
```

A floating point value. Floats are also used to store Boolean (TRUE, FALSE) values; integer values, like counters; or bit flags.

A parsing ambiguity is present with negative constants: "a-5" (five subtracted from a) will be parsed as "a", then "-5", causing an error. Separate the "-" from the digits with a space ("a - 5") to get the proper behavior.

3.2.1.3. Vector Types

```
vector_declaration ⇒ vector name
```

A vector is made up of three float coordinates. Used to represent positions or directions in 3D space. Note that single quotes (') surround a vector. Do not use double quotes, they are reserved for strings.

If you declare a vector `foobar`, then you can access it's x, y and z fields with the float types: `foobar_x`, `foobar_y`, and `foobar_z`.

3.2.1.4. String Types

```
string_declaration ⇒ string name
```

Represents a character string. Used to indicate file names, or messages to be broadcast to players. Use "\n " for a newline. The \" escape can be used to include a quotation mark (") in the string.

3.2.1.5. Entity Types

```
entity_declaration ⇒ entity name
```

An entity represents objects in the game, like things, players, and monsters. For instance, this is the type of the entities self and other. The entity type is a structured type, made of fields.

3.2.2. Field Types

Contrary to the other types, the entity type is a reference to an instance of a structured object that contains information of many different kinds, stored as fields of the entity object. Each field is given a name and a type.

Some of the fields do not store values; instead, they store the function to be executed under certain conditions, called *methods*.

If QuakeC was an object oriented programming language, method functions would be distinguished from the other fields and you would be able to create new object types, with their own fields.

As QuakeC stands currently, all the field definitions are definitions of entity fields. So, anywhere in your code you could add definition of new fields, and the compiler would interpret them as an extension of the entity definition.

Here are all the possible definitions of entity fields, with their types:

```
.float name

.string name

.vector name

.entity name
```

3.2.3. Reserved Field Types

In the first file read by the QuakeC compiler, `DEFS.QC`, there must be a definition for the entity fields and world fields. This definition is hard coded, and cannot be changed without requiring a recompilation of the core Quake engine.

In `DEFS.QC`, globals are defined before the special definition "`void end_sys_globals;`", while the entity fields are defined before the special definition "`void end_sys_fields;`". It's not important to understand the tags, just don't modify `DEFS.QC` before those two tags, and you won't be in trouble. Better yet, avoid modifying `DEFS.QC` altogether and use header files to define entity fields, globals, constants, and prototypes.

3.3. Definition of Variables

```
variable_declaration ⇒ [local] simple_type name [ = immediate][, name][ =
<immediate>] …
```

There are two levels of *scoping*. By default all variables are global: they can be accessed by any functions, and they are shared by all the functions (and all the clients of a given network server, of course). Note that variables cannot be givien an initial default value as part of their declaration.

Using the keyword local just before the declaration of a variable, makes the variable(s) visible only to the function itself (i.e. it will be allocated on the stack).

Note that parameters of functions are treated like local variables, they are only visible to the function, but they can be modified.

3.4. Definition of Constants (Immediates)

```
constant_declaration ⇒ simple_type name = value
```

Any global variable that is initialized by setting a value to it is actually assumed to be a constant. Since a constant is in fact represented by immediate values, you should NEVER attempt to modify a constant by giving it another value. Also, do not use another constant as the value for a constant. literals should only be used as the values for constants.

4. Names and Expressions

4.1. Names

Names can denote declared entities, whether declared explicitly or implicitly (see 3.1). Names can also denote objects or subprograms designated by access values; the results of *function_calls*; protected subprograms, single entries, entry families, and entries in families of entries. Finally, names can denote attributes of any of the foregoing.

4.2. Expressions

An *expression* is a formula that defines the computation or retrieval of a value.

4.3. Operators and Expression Evaluation

The language defines the following five categories of operators (given in order of increasing precedence):

4.3.1. Logical Operators

```
&&    // logical AND

||    // logical OR

!     // logical NOT
```

Take care that in `if ()` conditional expressions containing two or more logical clauses, all the clauses will be evaluated before the condition test (like in BASIC, and unlike C). That means that if one part of your condition is not always valid or defined, you had better decompose your `if()` into two successive `if()` statements, which should also make it faster.

4.3.2. Relational Operators

```
<=    // less than or equal to

<     // less than

>=    // greater than or equal to

>     // greater than

==    // equal, like in C.

!=    // not equal, like in C.
```

4.3.3. Binary Adding Operators

```
+     // addition

-     // subtraction

&     // bitwise AND

|     // bitwise OR
```

4.3.4. Unary Adding Operators

```
+      // identity

-      // negation
```

4.3.5. Multiplying Operators

```
*      // multiplication

/      // division
```

5. Statements

A *statement* is either simple or compound. A *simple*_statement encloses no other statement. A *compound_statement* can enclose *simple_statements* and other *compound_statements*.

5.1. Assignment Statements

```
assignment_statement ⇒ variable_name = expression;
```

An *assignment_statement* replaces the current value of a variable with the result of evaluating an expression.

5.2. If Statements

```
if_statement ⇒ if (condition) simple_statement [else simple_statement]

if_statement ⇒ if (condition) { compound_statement [else
compound_statement] }

      condition ⇒ boolean_expression
```

An *if_statement* selects for execution at most one of the enclosed *simple_statement* or *compound_statements*, depending on the (truth) value of one or more corresponding conditions.

Note that a conditional statement returning a nonzero value always evaluates as TRUE, while a value of zero evaluates as FALSE.

5.3. Loop Statements

```
while (condition) sequence_of_statements

do sequence_of_statments while (condition)
```

A *loop_statement* includes a *sequence_of_statements* that is to be executed repeatedly zero or more times.

6. Subprograms

A *subprogram_declaration* declares a *procedure* or *function*.

6.1. Subprogram Declaration

```
function_declaration ⇒ type (formal_parameter[, formal_parameter …]) name
```

A subprogram must be declared before it is used.

Procedures are declared by specifying a type of `void`.

6.2. Subprogram Specification

```
procedure_specification ⇒ type (formal_parameter[, formal_parameter …])
name = { sequence_of_statements return(expression) }

function_specification ⇒ type (formal_parameter[, formal_parameter …])
name = { sequence_of_statements return(expression) }
```

There is a maximum of eight (8) formal parameters.

```
frame_function_specification ⇒ void() framename = [$framenum, nextthink] {
sequence_of_statements }
```

Frame functions (also called *states*) are special functions made for convenience. They are meant to facilitate the definition of animation frames, by making them more readable.

It is strictly equivalent to:

```
void() framename = {

self.frame= $framenum;          // the model frame to displayed

self.nextthink = time + 0.1;    // next frame happens in 1/10 of second

self.think = nextthink;         // the function to call at the next frame

sequence_of_statements }
```

6.3. Subprogram Calls

```
subprogram_call ⇒ name (actual_parameter[, actual_parameter …])
```

There is a maximum of eight (8) parameters.

7. Predefined Constants

7.1. Entities

7.1.1. Temporary Entities

```
// point entity is a small point-like entity.
 0 TE_SPIKE                    // unknown
 1 TE_SUPERSPIKE               // superspike hits (spike traps)
 2 TE_GUNSHOT                  // hit on the wall (Axe, Shotgun)
 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
// large entity is a 2 dimensional entity.
 5 TE_LIGHTNING1               // flash of the Shambler
 6 TE_LIGHTNING2               // flash of the Thunderbolt
 9 TE_LIGHTNING3               // flash in e1m7 to kill Chthon
```

7.2. Items

Values used for the entity field `.items`.

```
IT_SHOTGUN = 1;

IT_SUPER_SHOTGUN = 2;

IT_NAILGUN = 4;

IT_SUPER_NAILGUN = 8;

IT_GRENADE_LAUNCHER = 16;

IT_ROCKET_LAUNCHER = 32;

IT_LIGHTNING = 64;

IT_EXTRA_WEAPON = 128;

IT_SHELLS = 256;

IT_NAILS = 512;

IT_ROCKETS = 1024;

IT_CELLS = 2048;

IT_AXE = 4096;

IT_ARMOR1 = 8192;

IT_ARMOR2 = 16384;

IT_ARMOR3 = 32768;

IT_SUPERHEALTH = 65536;

IT_KEY1 = 131072;

IT_KEY2 = 262144;

IT_INVISIBILITY = 524288;

IT_INVULNERABILITY = 1048576;

IT_SUIT = 2097152;

IT_QUAD = 4194304;
```

7.2.1. Behavior of Solid Objects

Values used with the entity field `.solid`.

```
SOLID_NOT = 0;                  // no interaction with other objects:
                                // inactive triggers
SOLID_TRIGGER = 1;              // touch on edge, but not blocking active
                                // triggers: pickable items (.MDL models,
                                // like armors)
SOLID_BBOX = 2;                 // touch on edge, block: pickable items
                                // (.BSP models, like ammo box, grenades,
                                // missiles
SOLID_SLIDEBOX = 3;             // touch on edge, but not an onground: most
                                // monsters
SOLID_BSP = 4;                  // BSP clip, touch on edge, block: buttons,
                                // platforms, doors, missiles
```

7.2.2. Movement Types

Values used for entity field `.movetype`.

```
MOVETYPE_NONE = 0;              // never moves
//float MOVETYPE_ANGLENOCLIP = 1;
//float MOVETYPE_ANGLECLIP = 2;
MOVETYPE_WALK = 3;              // walking players only
MOVETYPE_STEP = 4;              // walking monster
MOVETYPE_FLY = 5;               // hovering flight: meant for flying
                                // monsters (and players)
MOVETYPE_TOSS = 6;              // ballistic flight: meant for gibs and the
                                // like
MOVETYPE_PUSH = 7;              // not blocked by the world, push and
                                // crush meant for doors, spikes and
                                // crushing platforms
MOVETYPE_NOCLIP = 8;            // not blocked by the world
MOVETYPE_FLYMISSILE = 9;        // like fly, but size enlarged against
                                // monsters: meant for rockets
MOVETYPE_BOUNCE = 10;           // bounce off walls
MOVETYPE_BOUNCEMISSILE = 11     // bounce off walls, but size enlarged
                                // against monsters: meant for grenades
```

7.2.3. Entity Damage Types

Values used for entity field `.takedamage`. Most damageable entities have `DAMAGE_AIM`, so that when they chew on a grenade, it explodes. If you make an entity `DAMAGE_YES`, the grenades will bounce off it.

```
DAMAGE_NO = 0;                  // Can't be damaged
DAMAGE_YES = 1;                 // Grenades don't explode when touching
                                // entity
DAMAGE_AIM = 2;                 // Grenades explode when touching entity
```

7.2.4. Entity Dead Flag

Values used for the entity field `.deadflag`.

```
DEAD_NO = 0;                    // still living

DEAD_DYING = 1;                 // dying (helpless)

DEAD_DEAD = 2;                  // really dead

DEAD_RESPAWNABLE = 3;           // dead, but can respawn
```

7.2.5. Spawnflags

The entity field `.spawnflags` is a bit field, whose interpretation depend on the concerned entity. There is quite a bit of a hack here that could cause unexpected problems with QuakeC code that does not pay particular attention to the context in which the value of this field is evaluated.

```
DOOR_START_OPEN = 1;            // allow entity to be lighted in closed
                                // position
SPAWN_CRUCIFIED= 1;             // for zombie
PLAT_LOW_TRIGGER = 1;           // for func_plat()
SPAWNFLAG_NOTOUCH= 1;
SPAWNFLAG_NOMESSAGE= 1;
PLAYER_ONLY = 1;
SPAWNFLAG_SUPERSPIKE = 1;       // for spike shooter
SECRET_OPEN_ONCE = 1;           // secret door, stays open
PUSH_ONCE = 1;
WEAPON_SHOTGUN = 1;             // weapon, shotgun
H_ROTTEN = 1;                   // health, rotten (5-10 points)
WEAPON_BIG2 = 1;                // items
START_OFF = 1;                  // light, is off at start.
SILENT = 2;
SPAWNFLAG_LASER = 2;            // for spike shooter
SECRET_1ST_LEFT = 2;            // secret door, 1st move is left of arrow
WEAPON_ROCKET = 2;              // weapon, rocket
H_MEGA = 2;                     // health, mega (100 points)
DOOR_DONT_LINK = 4;
SECRET_1ST_DOWN = 4;            // secret door, 1st move is down from arrow
WEAPON_SPIKES = 4;              // weapon, nailgun
DOOR_GOLD_KEY = 8;
SECRET_NO_SHOOT = 8;            // secret door, only opened by trigger
WEAPON_BIG = 8;                 // weapon, super model
DOOR_SILVER_KEY = 16;
```

```
SECRET_YES_SHOOT = 16;          // secret door, shootable even if targeted

DOOR_TOGGLE = 32;
```

7.3. Light Effects

Values used by the entity field `.effects`.

```
EF_BRIGHTFIELD = 1;             // glowing field of dots

EF_MUZZLEFLASH = 2;

EF_BRIGHTLIGHT = 4;

EF_DIMLIGHT = 8;
```

7.4. Point Contents

```
CONTENT_EMPTY = -1;             // empty area

CONTENT_SOLID = -2;             // totally solid area (rock)

CONTENT_WATER = -3;             // pool of water

CONTENT_SLIME = -4;             // pool of slime

CONTENT_LAVA = -5;              // lava

CONTENT_SKY = -6;               // sky
```

7.5. Protocol Messages

7.5.1. Message Routing

Values used by the `Write*()` functions.

```
MSG_BROADCAST = 0;              // unreliable message, sent to all

MSG_ONE = 1;                    // reliable message, sent to msg_entity()

MSG_ALL = 2;                    // reliable message, sent to all

MSG_INIT = 3;                   // write to the init string
```

Use unreliable (but fast) messages, when it's of no importance if a client misses the message (examples: sound, explosions, monster deaths, taunts, etc.). Use reliable messages when it's very important that every client sees the message, lest a game inconsistency occur (examples: shots, player deaths, door moves, game ends, and CD track changes).

7.5.2. Message Types

These are some of message types defined in the Quake network protocol. Values are used by the `WriteByte()` function.

```
SVC_SETVIEWPORT = 5;

SVC_SETANGLES = 10;

SVC_TEMPENTITY = 23;

SVC_KILLEDMONSTER = 27;

SVC_FOUNDSECRET = 28;

SVC_INTERMISSION = 30;

SVC_FINALE = 31;

SVC_CDTRACK = 32;

SVC_SELLSCREEN = 33;

SVC_UPDATE = 128;
```

7.6. Sound

7.6.1. Attenuation

Those values are meant to be used with the functions `sound()` and `ambientsound()` as values for the parameter `attenuation`.

```
ATTN_NONE = 0;                  // full volume everywhere in the level

ATTN_NORM = 1;                  // normal

ATTN_IDLE = 2;                  // [FIXME]

ATTN_STATIC = 3;                // [FIXME]
```

7.6.2. Channel

These values are meant to be used with the function `sound()`as values for the parameter `channel`.

```
CHAN_AUTO = 0;                  // create a new sound

CHAN_WEAPON = 1;                // replace entity's weapon noise

CHAN_VOICE = 2;                 // replace entity's voice

CHAN_ITEM = 3;                  // replace entity's item noise

CHAN_BODY = 4;                  // replace entity's body noise
```

8. Predefined Global Variables

These variables are accessible in every function. QuakeC functions are not intended to modify them directly.

8.1. coop

```
float coop;                    // a Boolean value, 0 or 1
```

TRUE if playing cooperative.

8.2. deathmatch

```
float deathmatch;              // a Boolean value, 0 or 1
```

TRUE if playing deathmatch.

8.3. force_retouch

```
float force_retouch;           // counter
```

Force all entities to touch triggers next frame. This is needed because non-moving things don't normally scan for triggers, and when a trigger is created (like a teleport trigger), it needs to catch everything. It is decremented each frame, so it is usually set to 2 to guarantee everything is touched.

8.4. found_secrets

```
float found_secrets;           // counter
```

Number of secrets found.

8.5. frametime

```
float frametime;               // in seconds
```

No idea what this can be. Used only when jumping in water.

8.6. killed_monsters

```
float killed_monsters;         // counter
```

Store the total number of monsters killed.

8.7. mapname

```
string mapname;
```

Name of the level map currently being played, like "start".

8.8. msg_entity

```
entity msg_entity;
```

If you want to send a message to just one entity e, then set `msg_entity = e` and send the message with flag `MSG_ONE`, instead of `MSG_ALL`.

8.9. other

```
entity other;
```

The object concerned by an impact; not used for thinks.

8.10. parm1...parm16

8.11. self

```
entity self;
```

The entity that is subject to the current function.

8.12. serverflags

```
float serverflags;              // bit fields
```

Propagated from level to level, and used to keep track of the completed episodes. If `serverflag & ( 1 << e)` is TRUE, then episode e was already completed. Generally equal to `player.spawnflags & 15`.

8.13. teamplay

```
float teamplay;                 // a Boolean value, 0 or 1
```

TRUE if playing by teams.

8.14. time

```
float time;                     // in seconds
```

The current game time in seconds. Note that because the entities in the world are simulated sequentially, time is NOT strictly increasing. An impact late in one entity's time slice may set time higher than the think function of the next entity. The difference is limited to 0.1 seconds.

8.15. total_monsters

```
float total_monsters;           // counter
```

Total number of monsters that were spawned, since the beginning of the level.

8.16. total_secrets

```
float total_secrets;            // counter
```

Number of secrets found by the players. Affected only by trigger_secret.

8.17. world

```
entity world;
```

The server's world object, which holds all global state for the server, like the deathmatch flags and the body queues.

```
float parm1;                    // items bit flag (IT_SHOTGUN | IT_AXE )

float parm2;                    // health

float parm3;                    // armorvalue

float parm4;                    // ammo

float parm5;                    // ammo

float parm6;                    // ammo

float parm7;                    // ammo

float parm8;                    // weapon

float parm9;                    // armortype * 100

float parm10, parm11, parm12, parm13, parm14, parm15, parm16;
```

These parameters seem to be a bit of hack. They are used when a client connects. Spawnparms are used to encode information about clients across server level changes.

9. Entities

In Quake, monsters, players, items, and the level itself are all entities.

9.1. Types of Entities

There are three kinds of entities, and you will encounter all of them in the QuakeC code.

9.1.1. Static entities

A static entity doesn't interact with the rest of the game. These are flames (`progs/flame.mdl`), lights, illusionary objects, and the like. It is never be necessary to reference such an entity, so they don't get an entity reference number.

A static entity will be created by the function:

```
makestatic()
```

which causes a spawnstatic message to be sent to every client. A static entity cannot be removed once created. The maximum number of static entities is 127.

9.1.2. Temporary entities

A temporary entity is a short life time entity. For instance, Quake uses these entities for hits on the wall (point-like entities) or for the Thunderbolt flash (line-like entities), gun shots, and anything that is not supposed to last more than one frame.

Sending a valid temporary entity message will create a temporary entity. A temporary entity need not be removed it disappears by itself.

9.1.3. Dynamic entities

A dynamic entity is anything that changes its behavior or its appearance. These are ammunition boxes, spinning armors, player models and the like.

A dynamic entity will be created by the sequence:

```
entity = spawn();

setmodel(entity, "progs/entity.mdl");

setsize(entity, vector_min, vector_max);

setorigin(entity, position);
```

It will have to be removed by the function:

```
remove(entity);
```

The maximum number of dynamic entities is 449.

9.2. Predefined Entity Fields

These are the fields that are available in the entity objects (like self, other). Beware that this is not true object oriented programming: there is no protection when accessing those fields, and no guarantee on the validity of values. So if you put garbage there you will probably crash the game.

9.2.1. Fields Shared between QUAKE.EXE and QuakeC

These fields describe the most common entity fields. They are shared between the C code of QUAKE.EXE, and the QuakeC code of PROGS.DAT.

Some of the fields are managed by the C code: you can read their value, but YOU SHOULD NEVER MODIFY THEIR VALUE DIRECTLY (there are special built-in functions for that).

9.2.1.1. Technical

```
entity chain;                   // next entity, in a chain list of entities
```

John Cash advises programmers to not use the chain field. The chain field gets written by some of the built-in functions (like `findradius()`). If you need to make a linked list, make your own link field.

```
float ltime;                    // local time for entity

float teleport_time;            // to avoid backing up

float spawnflags;
```

9.2.1.2. Entity Appearance

```
float   modelindex;             // index of model, in the precached list

string  classname;              // spawn function

string  model;                  // the name of the file that contains the

                                // entity model


float frame;
```

This is the index of the currently displayed model frame. Frames must be defined by a `$frame` construct in the model file, and manipulated in the code as `$xxx` (where xxx is the name of the frame).

```
float skin;
```

This is the index of the model skin currently displayed. If your model has more than one skin defined, then this value indicates the skin in use. You can change it freely, as long as it remains in a valid range. For instance, it's used by the armor model to show the yellow, red or green skin.

```
float effects;
```

This is a flag that defines the special light effects that the entity is subject to. This can supposedly be used to make an entity glow, or to create a glowing field of dots around it.

9.2.1.3. Position in 3D

Quirks: setting the angles on a player entity doesn't work.

```
vector origin;                  // position of model
                                // origin_x, origin_y, origin_z
vector mins;                    // bounding box extents reletive to origin
                                // mins_x, mins_y, mins_z
vector maxs;                    // bounding box extents reletive to origin
                                // maxs_x, maxs_y, maxs_z
vector size;                    // the x, y and z lengths of the entity's
                                // bounding box; obtained by subtracting
                                // absmin from absmax
                                // size_x,size_y,size_z
vector absmin;                  // lower left-hand corner of the
                                // entity's bounding box relative to the
                                // entity's origin
                                // absmin_x absmin_y absmin_z
vector absmax;                  // origin + mins and maxs
                                // absmax_x absmax_y absmax_z
vector oldorigin;               // old position
vector angles;                  // = 'pitch_angle yaw_angle flip_angle'
```

9.2.1.4. Situation of the Entity

Since `groundentity` is used nowhere in PROGS.DAT, it's meaning is just a wild guess from a similar field in messages.

```
float waterlevel;               // 0 = not in water, 1 = feet, 2 = waist,
                                // 3 = eyes
float watertype;                // a content value
```

```
entity groundentity;          // indicates that the entity moves on the
                              // ground
```

9.2.1.5. Movement in 3D

```
vector  velocity;             // = 'speed_x      speed_y    speed_z'
vector  avelocity;            // = 'pitch_speed yaw_speed 0', angle
                              // velocity
vector  punchangle;           // temp angle adjust from damage or recoil
float   movetype;             // type of movement
float   yaw_speed;            // rotation speed
float   solid;                // specifies if entity can block movement
```

9.2.1.6. Monster Behavior

```
entity goalentity;            // monster's movetarget or enemy
float ideal_yaw;              // monster's ideal direction, on paths
float yaw_speed;              // monster's yaw speed
string target;                // target of a monster
string targetname;            // name of the target
```

9.2.1.7. Automatic Behavior

When you want an entity to do something specific after a certain delay (exploding, disappearing, or the like…), you set `.nextthink` to that delay (in seconds), and set `.think` to the function to execute.

```
float   nextthink;            // next time when entity must act
void()  think;                // function invoked when entity must act
void()  touch;                // function invoked if entity is touched
void()  use;                  // function invoked if entity is used
void()  blocked;              // function for doors or plats, called when
                              // can't push other
vector  movedir;              // mostly for doors, but also used for
                              // waterjump
```

```
string  message;                // trigger messages
float   sounds;                 // either a CD track number or sound number
string  noise;                  // sound played on entity noise channel 1
string  noise1;
string  noise2;
string  noise3;
```

9.2.1.8. Player/Monster Statistics and Damage

```
float deadflag;                 // tells if an entity is dead
float health;                   // health level
float max_health;               // players maximum health is stored here
float takedamage;               // indicates if entity can be damaged

float dmg_take;
float dmg_save;
```

Damage is accumulated through a frame and sent as one single message, so the super shotgun doesn't generate huge messages.

```
entity dmg_inflictor;           // entity that inflicted the damage
                                // (player, monster, missile, door)
```

9.2.1.9. Player Inventory

```
float items;                    // bit flags
float armortype;                // fraction of damage absorbed by armor
float armorvalue;               // armor level
float weapon;                   // one of the IT_SHOTGUN, etc flags
string weaponmodel;             // entity model for weapon
float weaponframe;              // frame for weapon model
float currentammo;              // ammo for current weapon
float ammo_shells;              // remaining shells
float ammo_nails;               // remaining nails
float ammo_rockets;             // remaining rockets and grenades
float ammo_cells;               // remaining lightning bolts

float impulse;                  // weapon changes
```

When set to 0, the player's weapon doesn't change. When different from zero, this field is interpreted by the QuakeC impulse command as a request to change weapon (see `ImpulseCommand( )`).

9.2.1.10. Player Combat

```
entity owner;                   // entity that owns this one
                                // (missiles, bubbles are owned by the
                                // player)
entity enemy;                   // personal enemy (only for monster
                                // entities)
float button0;                  // fire
float button1;                  // use
float button2;                  // jump
vector view_ofs;                // position of player eye, relative to
                                // origin
float fixangle;                 // set to 1 if you want angles to change
                                // now
```

```
vector v_angle;                 // view or targeting angle for players

float idealpitch;               // calculated pitch angle for lookup up

                                // slopes

entity aiment;                  // aimed entity?
```

9.2.1.11. Deathmatch

```
float frags;                    // number of frags

string netname;                 // name, in network play

float colormap;                 // colors of shirt and pants

float team;                     // team number

float flags;
```

9.2.2. Fields used only by QuakeC

These entity fields are used only by QuakeC programs, and are never referenced by the C code of QUAKE.EXE. So you can do whatever you want with the values, so long as it's compatible with what other QuakeC modules do.

If the fields defined here are not suitable for you, you can define new fields, by adding them at the end of the definition of fields. As a matter of fact, the number of fields in an entity (hence the size of all the instances of entity objects) is determined by QuakeC: in the PROGS.DAT header, a value named `entityfields` indicates to QUAKE.EXE the size of the entity object. Beware however that the more fields you add, the more each entity will suck memory. Add just one float (4 bytes) and it will take, in memory, 4 bytes times the number of entities.

The best way is to share fields between distinct classes of entities, by reusing the same position for another kind of field. If the QuakeC Compiler was a real object-oriented compiler, that would be done very safely by single-inheritance (multiple-inheritance would be a deadly mistake). You will also notice that id Software has made quite a lousy usage of most of the fields, defining much more than were actually needed, since they are only used by a few entities.

9.2.2.1. World Fields

```
string wad;                     // name of WAD file with misc graphics

string map;                     // name of the map being played

float worldtype;                // see below
```

9.2.2.2. QuakeEd

```
string killtarget;

float light_lev;                // not used by game, but parsed by light

                                // utility

float style;
```

9.2.2.3. Monster Behavior

These functions are called when these specific events happen:

```
void() th_stand;                // when stands idle

void() th_walk;                 // when is walking

void() th_run;                  // when is running

void() th_missile;              // when a missile comes

void() th_melee;                // when fighting in melee

void() th_die;                  // when dies


void(entity attacker, float damage) th_pain;
```

Executed when the monster takes a certain amount of damage from an attacker (a player, or another monster). Usually causes the monster to turn against the attacker.

9.2.2.4. Monsters

```
entity oldenemy;                // mad at this player before taking damage

float  speed;

float  lefty;

float  search_time;

float  attack_state;

float   pausetime;

entity  movetarget;
```

9.2.2.5. Player

```
float walkframe;

float attack_finished;

float pain_finished;            // time when pain sound is finished

float invincible_finished;

float invisible_finished;

float super_damage_finished;

float radsuit_finished;

float invincible_time;          // time when player ceases to be invincible

float invincible_sound;

float invisible_time;           // time when player ceases to be invisible

float invisible_sound;

float super_time;               // time when quad shot expires?

float super_sound;

float rad_time;

float fly_sound;

float axhitme;                  // TRUE if hit by axe

float show_hostile;             // set to time + 0.2 whenever a client
                                // fires a weapon or takes damage

float jump_flag;                // player jump flag

float swim_flag;                // player swimming sound flag

float air_finished;             // when time > air_finished, start drowning

float bubble_count;             // keeps track of the number of bubbles

string deathtype;               // keeps track of how the player died
```

9.2.2.6. Objects

```
string mdl;                    // model name?
vector mangle;                 // angle at start. 'pitch roll yaw'
vector oldorigin;              // only used by secret door
float t_length;
float t_width;
```

9.2.2.7. Doors

```
vector dest;
vector dest1;
vector dest2;
float wait;                    // time from firing to restarting
float delay;                   // time from activation to firing
entity trigger_field;          // door's trigger entity
string noise4;
float aflag;
float dmg;                     // damage done by door when hit
```

9.2.2.8. Miscellaneous

```
float cnt;                    // counter
void() think1;
vector finaldest;
vector finalangle;
// triggers
float count;                  // for counting triggers
// plats/doors/buttons
float lip;
float state;
vector pos1;
vector pos2;                  // top and bottom positions
float height;
// sounds
float waitmin;
float waitmax;
float distance;
float volume;
```

10. Built-in Functions

These are the built-in functions of QuakeC. Since they are hard-coded in C, they cannot be redefined, but they are very fast.

10.1. Basic Math Functions

10.1.1. anglemod

```
float anglemod(float angle)
```

Returns angle in degree, modulo 360.

10.1.2. ceil

```
float ceil(float val)
```

Returns `val`, rounded up to the integer above (like the equivalent function in C).

10.1.3. fabs

```
float fabs(float val)
```

Returns absolute value of `val` (like the equivalent function in C).

10.1.4. floor

```
float floor(float val)
```

Returns `val`, rounded up to the integer below (like the equivalent function in C).

10.1.5. ftos

```
string ftos(float value)
```

Float to string: converts value to string. Note: may return a string with leading spaces when the converted value is non-integer and also cuts it off after the first decimal (e.g. " 0.9" for 0.93). This is confirmed behavior for DOSQuake 1.08 and WinQuake 1.0.

10.1.6. random

```
float random()
```

Returns a random floating point number between 0.0 and 1.0.

10.1.7. rint

```
float rint(float val)
```

Returns `val`, rounded up to the closest integer value.

10.2. Basic Vector Math Functions

10.2.1. makevectors

```
void makevectors(vector angles)
```

Constructs an angle = 'pitch yaw 0'. Calculates the unit vectors (a length of 1 "unit") pointing forward, to the right, and up (positive x, y and z directions, respectively) from the given angle. Result is returned in the global variables:

```
vector  v_forward;              // points forward

vector  v_up;                   // points up

vector  v_right;                // points toward the right
```

10.2.2. normalize

```
vector normalize(vector v)
```

Returns a vector of length 1. Gives the vector collinear to v, but of length 1. This can be useful for calculation of distance along an axis.

10.2.3. vlen

```
float vlen(vector v)
```

Returns the length of vector v (never < 0).

10.2.4. vectoangles

```
vector vectoangles(vector v)
```

Returns vector = 'pitch yaw 0 '. Vector to angles: calculates the pitch angle (aiming) and yaw angle (bearing) corresponding to a given 3D direction v.

10.2.5. vectoyaw

```
float vectoyaw(vector v)
```

Returns an angle in degrees. Vector to yaw: calculates the yaw angle (bearing) corresponding to a given 3D direction v.

10.2.6. vtos

```
string vtos(vector v)
```

Vector to string: convert a vector into a string.

10.3. Collision Checking Functions

10.3.1. checkbottom

```
float checkbottom(entity e)
```

Returns TRUE or FALSE. Returns TRUE if on the ground. Used only for jumping monsters that need to jump randomly not to get hung up (or whatever it actually means).

10.3.2. checkpos

```
scalar checkpos (entity e, vector position)
```

CURRENTLY DISABLED. DO NOT USE. Returns TRUE if the given entity can move to the given position from its current position by walking or rolling.

10.3.3. pointcontents

```
float pointcontents(vector pos)
```

Returns the contents of the area situated at position pos. Used to know if an area is in water, in slime or in lava. Makes use of the BSP tree, and is supposed to be very fast.

10.3.4. traceline

```
traceline(vector v1, vector v2, float nomonsters, entity forent)
```

Where: v1= start of line; v2= end of line; nomonster= if TRUE, then see through other monsters, else FALSE; forent= ignore this entity, it's owner, and it's owned entities; if forent = world, then ignore no entity.

Trace a line of sight, possibly ignoring monsters, and possibly ignoring the entity `forent` (usually, forent = self). This function is used very often, tracing and shot targeting. Bounding boxes and exact BSP entities blocks traces. Returns the results in the global variables:

```
float trace_allsolid;          // never used

float trace_startsolid;        // never used

float trace_fraction;          // fraction (percent) of the line that was

                               // traced, before an obstacle was hit;

                               // equal to 1 if no obstacle were found

vector trace_endpos;           // point where line ended or met an

                               // obstacle

vector trace_plane_normal;     // direction vector of trace (?)

float  trace_plane_dist;       // distance to impact along direction

                               // vector (?)
```

```
entity trace_ent;               // entity hit by the line created by
                                // traceline(); valid only if
                                // trace_fraction != 1.0
float  trace_inopen;            // Boolean, true if line went through non-
                                // water area.
float  trace_inwater;           // Boolean, true if line went through
                                // water area.
```

10.4. Combat Functions

10.4.1. aim

```
vector aim(entity e, float missilespeed)
```

Returns a vector along which the entity e can shoot. Usually, e is a player, and the vector returned is calculated by auto aiming to the closest enemy entity.

10.4.2. checkclient

```
entity checkclient()
```

Returns client (or object that has a client enemy) that would be a valid target. If there are more than one valid options, they are cycled each frame. If (`self.origin + self.viewofs`) is not in the PVS of the target, 0 (FALSE) is returned.

10.4.3. particle

```
void particle(vector origin, vector dir, float color, float count)
```

Where: origin = initial position; dir = initial direction; color = color index (color = 0 for chunk, color = 75 for yellow, color = 73 for blood red, color = 225 for entity damage); count = time to live, in seconds.

Create a particle effect (small dot that flies away).

10.5. Console Functions

10.5.1. cvar

```
float cvar(string console_variable)
```

Returns the value of a console variable. Note that this reads the console variables of the server only and cannot read the console variable of network server clients. Note also that some console variables cannot be read by `cvar( )` at all, such as “maxplayers”.

10.5.2. cvar_set

```
float cvar_set(string console_variable, string value)
```

Sets the value of a console variable. Note: may set the cvar to zero when the string value contains leading spaces. This is confirmed behavior for DOSQuake 1.08 and WinQuake 1.0.

10.5.3. dprint

```
void dprint(string text)
```

Prints a message to the server console. This function only creates output if the “developer” console variable is set to 1. It was a bug in earlier versions of Quake that it worked even when “developer” was set to 0.

10.5.4. localcmd

```
void localcmd(string text)
```

Execute a command on the server, as if it had been typed on the server's console. Don't forget the “\n“ (newline) at the end in order to complete the command.

Examples:

```
localcmd("restart\n");          // restart the level

localcmd("teamplay 1\n");       // set deathmatch mode to teamplay

localcmd("killserver\n");       // poor server...
```

10.6. Debug Functions

10.6.1. break

```
void break()
```

Exit the programs.

10.6.2. coredump

```
void coredump()
```

Print all entities.

10.6.3. eprint

```
void eprint(entity e)
```

Print details about a given entity (for debug purposes).

10.6.4. error

```
void error(string text)
```

Print an error message.

10.6.5. objerror

```
void objerror(string text)
```

Print an error message related to object self.

10.6.6. traceoff

```
void traceoff()
```

End traces started by `traceon()`.

10.6.7. traceon

```
void traceon()
```

Start tracing functions, end them with `traceoff()`.

10.7. Entity Management Functions

10.7.1. find

```
entity find (entity start, .string field, string match)
```

Where: start = beginning of list to search (world, for the beginning of list); field = entity field that must be examined (ex: targetname); match = value that must be matched (ex: other.target).

Returns the entity found, or world if no entity was found. Searches the server entity list beginning at start, looking for an entity that has entity.field = match.

Example: find the first player entity

```
e = find( world, classname, "player");
```

Take care that field is a name of an entity field, without dot, and without quotes.

10.7.2. findradius

```
entity findradius(vector origin, float radius)
```

Where: origin = origin of sphere; radius = radius of sphere.

Returns a chain of entities that have their origins within a spherical area. The entity returned is `e`, and the next in the chain is `e.chain`, until e == FALSE. Typical usage: find and harm the victims of an explosion.

```
e = findradius( origin, radius)

while(e) { T_Damage(e, ... ); e = e.chain }
```

10.7.3. lightstyle

```
void lightstyle(float style, string value)
```

Where: style = index of the light style, from 0 to 63; value = (ex: "abcdefghijklmlkjihgfedcb").

Modifies a given light style. The light style is used to create cyclic lighting effects, like torches or teleporter lighting. There are 64 light styles, from 0 to 63. If style is not strictly comprised in these values, the game may crash. Styles 32-62 are assigned by the light program for switchable lights. Value is a set of characters, whose ASCII value indicates a light level, from "a" (0) to "z" (30).

10.7.4. makestatic

```
void makestatic (entity e)
```

Make an entity static to the world, by sending a broadcast message to the network. The entity is then removed from the list of dynamic entities in the world, and it cannot be deleted (until the level ends).

10.7.5. nextent

```
entity nextent(entity e)
```

Returns entity that is just after e in the entity list. Useful to browse the list of entities, because it skips the undefined ones.

10.7.6. remove

```
void remove(entity e)
```

Removes an entity from the world.

10.7.7. setmodel

```
void setmodel(entity e, string model)
```

Where: e = entity whose model is to be set; model = name of the model (ex: "progs/soldier.mdl").

Changes the model associated to an entity. This model should also be declared by `precache_model`. Please set `e.movetype` and `e.solid` first.

10.7.8. spawn

```
entity spawn()
```

Creates a new entity, totally empty. You can manually set every field, or just set the origin and call one of the existing entity setup functions.

10.8. Movement Functions

10.8.1. ChangeYaw

```
void ChangeYaw()
```

Change the horizontal orientation of self. Turns towards `self.ideal_yaw` at `self.yaw_speed`, and sets the global variable `current_yaw`. Called every 0.1 sec by monsters.

10.8.2. droptofloor

```
float droptofloor()
```

Returns TRUE or FALSE.

Drops self to the floor, if the floor is less than -256 coordinates below. Returns TRUE if landed on floor. Mainly used to spawn items or walking monsters on the floor.

10.8.3. movetogoal

```
void movetogoal(float step)
```

Move self toward its goal. Used for monsters.

10.8.4. setorigin

```
void setorigin(entity e, vector position)
```

Where: e = entity to be moved, position = new position for the entity.

Move an entity to a given location. That function is to be used when spawning an entity or when teleporting it. This is the only valid way to move an object without using the physics of the world (setting velocity and waiting). DO NOT change directly e.origin, otherwise internal links and entity clipping will be invalidated.

10.8.5. setsize

```
void setsize(entity e, vector min, vector max)
```

Where: e = entity whose bounding box is to be set; min = minimum, for bounding box (ex: VEC_HULL2_MIN); max = maximum, for bounding box (ex: VEC_HULL2_MAX).

Set the size of the entity bounding box, relative to the entity origin. The size box is rotated by the current angle.

10.8.6. walkmove

```
float walkmove(float yaw, float dist)
```

Returns TRUE or FALSE. Moves self in the given direction. Returns FALSE if could not move (used to detect blocked monsters).

10.9. Message Functions

10.9.1. bprint

```
void bprint(string text)
```

Broadcast a message to all players on the current server.

10.9.2. centerprint

```
void centerprint(entity client, string text)
```

Sends a message to a specific player, and print it centered.

10.9.3. sprint

```
void sprint(entity client, string text)
```

Sends a message to a player.

10.10. Network Messages

QuakeC is not supposed to handle a lot of network messages, since most are already handled in C. However, built-in functions have not been built for every kind of message in the Quake protocol, so you might end-up composing protocol messages in QuakeC. It is recommended that you build a single function to handle a given message type, because the structure of those messages might change, and then all your code would have to be rewritten to compensate.

Beware: when generating messages, you had better respect the format of the existing messages, otherwise the game clients might not be able to interpret them (and will likely crash). The functions below all write to clients (players connected via the network or the local player).

For some reason, sending messages via `MSG_ONE` to a player that's just connected has no effect, as do messages sent via `MSG_INIT`. However, messages sent via `MSG_ALL` do get sent. Note that this also applies to `sprint()` (or any other function which uses a directed network message).

10.10.1. WriteAngle

```
void WriteAngle(float to, float value)
```

This function writes a single byte, that represents 256 * (angle / 380).

10.10.2. WriteByte

```
void WriteByte(float to, float value)
```

10.10.3. WriteChar

```
void WriteChar(float to, float value)
```

10.10.4. WriteCoord

```
void WriteCoord(float to, float value)
```

10.10.5. WriteEntity

```
void WriteEntity(float to, entity value)
```

This function writes an entity reference, taking two bytes.

10.10.6. WriteLong

```
void WriteLong(float to, float value)
```

10.10.7. WriteShort

```
void WriteShort(float to, float value)
```

10.10.8. WriteString

```
void WriteString(float to, string value)
```

This function writes a string, terminated by "`\0`" (the null character in C).

10.11. Precaching Functions

These functions are used to declare models, sounds and stuff, before the PAK file is built. Just follow this rule: whenever one of your functions makes use of a file that's not defined in Quake, precache this file in a function that will be called by `worldspawn()`. Then the QCC compiler can automatically include in the PAK file all the files that you really need to run your programs.

Once the level starts running, these precache orders will be executed, so as to attribute a fixed table index to all those files. DO NOT USE those functions in code that will be called after `worldspawn()` was called. As a matter of fact, that could bomb Quake (restarting the level, without crashing the game).

Files can only be precached in spawn functions.

10.11.1. precache_file

```
void precache_file(string file)
```

Where: file = name of the file to include in PAK file.

Does nothing during game play. Use `precache_file2()` for registered Quake.

10.11.2. precache_model

```
void precache_model(string file)
```

Where: file = name of the MDL or BSP file to include in PAK file.

Does nothing during game play. Must be used in a model's spawn function, to declare the model file. Use `precache_model2()` for registered Quake.

10.11.3. precache_sound

```
void precache_sound(string file)
```

Where: file = name of the WAV file to include in PAK file.

Does nothing during game play. Must be used in a model's spawn function, to declare the sound files. Use `precache_sound2()` for registered Quake.

10.12. Server-related Functions

10.12.1. changelevel

```
void changelevel(string mapname)
```

Warp to the game map named mapname. Actually executes the console command "changelevel" + mapname, so if you want to alias it...

10.12.2. setspawnparms

```
void setspawnparms(entity client)
```

Restore the original spawn parameters of a client entity. Doesn't work if client is not a player.

10.12.3. stuffcmd

```
stuffcmd(entity client, string text)
```

Send a command to a given player, as if it had been typed on the player's console. Mostly used to send the command ”bf“, that creates a flash of light on the client's screen. Don't forget the “\n “ (newline) at the end, otherwise your command will not be executed, and will stand still on the console window.

Examples:

```
stuffcmd(self, "bf\n");

stuffcmd(self, "name Buddy\n");
```

10.13. Sound Functions

10.13.1. ambientsound

```
void ambientsound(vector position, string sample, float volume, float
attenuation)
```

Where: position = position, in 3D space, inside the level; sample = name of the sample WAV file (ex: "ogre/ogdrag.wav"); volume = 0.0 for low volume, 1.0 for maximum volume; attenuation = attenuation of sound.

An ambient sound is emitted, from the given position.

10.13.2. sound

```
void sound(entity source, float channel, string sample, float volume, float
attenuation)
```

Where: source = entity emitting the sound (ex: self); channel = channel to use for sound; sample = name of the sample WAV file (ex: "ogre/ogdrag.wav"); volume = 0.0 for low volume, 1.0 for maximum volume; attenuation= attenuation of sound.

The entity emits a sound, on one of its 8 channels.

11. Functions That Are Mandatory in QuakeC

These functions must be defined in QuakeC, since they are invoked by Quake under certain conditions.

11.1. Behavior of Players

11.1.1. PlayerPostThink

```
void PlayerPostThink();
```

Called with self = player, for every frame, after physics are run.

11.1.2. PlayerPreThink

```
void PlayerPreThink();
```

Called with self = player, for every frame, before physics are run.

11.2. Management of Network Game Clients

11.2.1. ClientConnect

```
void ClientConnect();
```

Called when a player connects to a server, but also, for every player, when a new level starts. It is used to announce the new player to every other player.

11.2.2. ClientDisconnect

```
void ClientDisconnect();
```

Called when a player disconnects from a server. Announce that the player has left the game.

11.2.3. ClientKill

```
void ClientKill();
```

Called when a player suicides.

11.2.4. PutClientInServer

```
void PutClientInServer();
```

Called after setting `parm1... parm16`.

11.2.5. SetChangeParms

```
void SetChangeParms();
```

Call to set parms for self so they can be restored.

11.2.6. SetNewParms

```
void SetNewParms();
```

Called when a client first connects to a server. Sets `parm1...parm16` so that they can be saved off for restarts.

11.3. Miscellaneous

11.3.1. main

```
void main();
```

Only used for testing progs.

11.3.2. StartFrame

```
void StartFrame();
```

Called at the start of each frame.

12. Network Protocol

12.1. Message Structures

Here are some of the messages defined in the Quake network protocol. Beware, the structure of those messages might change in future version (Satan forbid!).

12.1.1. Set View Position

```
msg_entity = player

WriteByte(MSG_ONE, SVC_SETVIEWPORT);

WriteEntity( MSG_ONE, camera);
```

This message is meant for a single client player. It sets the view position to the position of the entity camera.

12.1.2. Set View Angles

```
msg_entity = player

WriteByte(MSG_ONE, SVC_SETVIEWANGLES);

WriteAngle( MSG_ONE, camera.angles_x);

WriteAngle( MSG_ONE, camera.angles_y);

WriteAngle( MSG_ONE, camera.angles_z);
```

This message is meant for a single client player. It sets the orientation of its view to the same orientation as the entity camera.

12.1.3. Temporary Entity

```
WriteByte(MSG_BROADCAST, SVC_TEMPENTITY);

WriteByte(MSG_BROADCAST, entityname);

WriteCoord(MSG_BROADCAST, origin_x);

WriteCoord(MSG_BROADCAST, origin_y);

WriteCoord(MSG_BROADCAST, origin_z);
```

12.1.4. Set CD Track

```
WriteByte(MSG_ALL, SVC_CDTRACK);

WriteByte(MSG_ALL, val1);      // CD start track

WriteByte(MSG_ALL, val2);      // CD end track
```

12.1.5. Final Message

```
WriteByte(MSG_ALL, SVC_FINALE);

WriteString(MSG_ALL, "any text you like\n");
```

12.1.6. Sell Screen

```
WriteByte(MSG_ALL, SVC_SELLSCREEN);
```

Shows the infamous sell screen (like you needed it to understand).

12.1.7. Intermission

```
WriteByte(MSG_ALL, SVC_INTERMISSION);
```

Shows the intermission camera view.

12.1.8. Killed Monster

```
WriteByte(MSG_ALL, SVC_KILLEDMONSTER);
```

Increase by one the count of killed monsters, as available to the client.

12.1.9. Found Secret

```
WriteByte(MSG_ALL, SVC_FOUNDSECRET);
```

Increase by one the count of secrets founds.

13. Execution

Code execution is initiated by C code in quake from two main places: the timed think routines for periodic control and the touch function when two objects impact each other. Execution is also caused by a few uncommon events, like the addition of a new client to an existing server.

The interpretation is fairly efficient, but it is still over an order of magnitude slower than compiled C code. All time-consuming operations should be made into built-in functions.

13.1. Preset Globals

There are three global variables that are set before beginning code execution:

```
entity world;           // the server's world object, which holds all
                        // global state for the server, like the
                        // deathmatch flags and the body queues.
entity self;            // the entity the function is executing for
entity other;           // the other object in an impact, not used for
                        // thinks
float time;             // the current game time.  Note that because the
                        // entities in the world are simulated
                        // sequentially, time is NOT strictly increasing.
                        // An impact late in one entity's time slice may
                        // set time higher than the think function of the
                        // next entity. The difference is limited to 0.1
                        // seconds.
```

It is acceptable to change the system set global variables. This is usually done to pose as another entity by changing self and calling a function.

13.2. Runaways

There is a runaway counter that stops a program if 100000 statements are executed, assuming it is in an infinite loop.

13.3. Profiling

A profile counter is kept for each function, and incremented for each interpreted instruction inside that function. The "profile" console command in Quake will dump out the top 10 functions, then clear all the counters. The "profile all" command will dump sorted statistics for every function that has been executed.

13.4. Composition of Functions

Composition of functions is not supported since all the functions use a single parameter marshaling area and a single global variable to store their return result. You should NEVER try to call a function within another function call. For example:

```
afunc(4, bfunc(1,2,3));
```

will fail because there is a shared parameter marshaling area, which will cause the 1 from `bfunc` to overwrite the 4 already placed in `parm0`. When a function is called, it copies the parameters from the globals into its privately scoped variables, so there is no collision when calling another function.

```
total = factorial(3) + factorial(4);
```

will fail because the return value from functions is held in a single global area

However, the following will work:

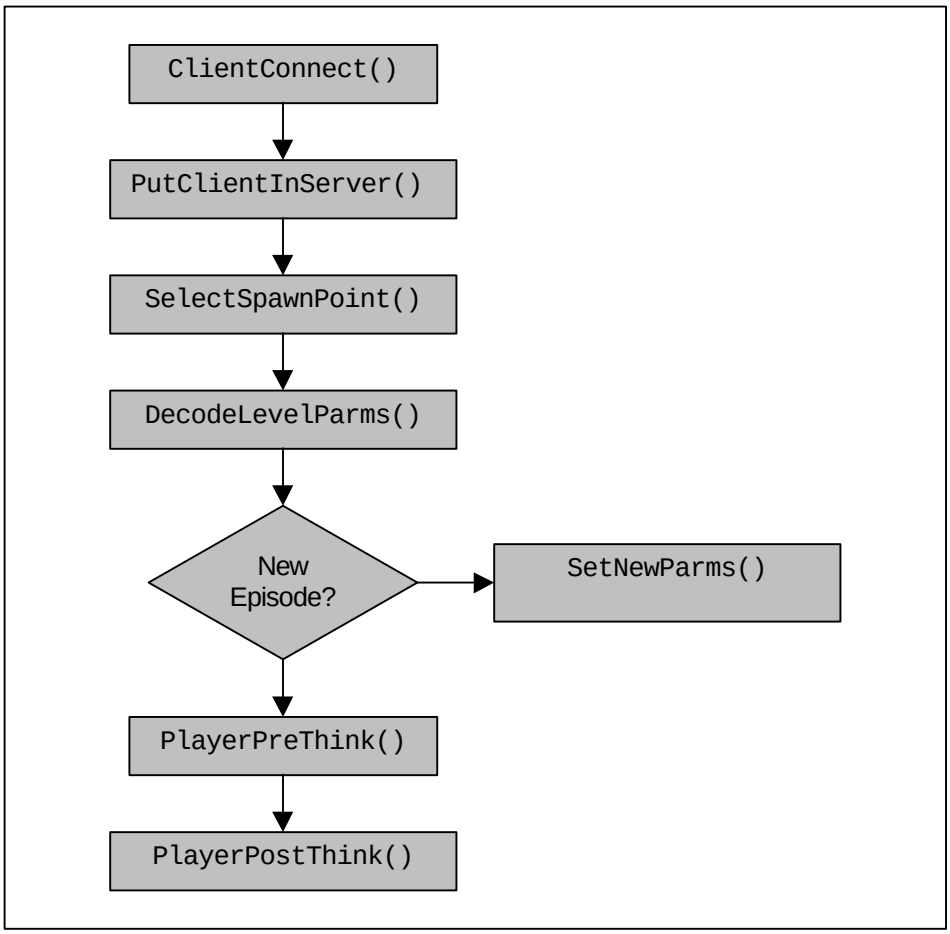
```
dprint(ftos(self.impulse));
```

This appears to be only composite function allowed.

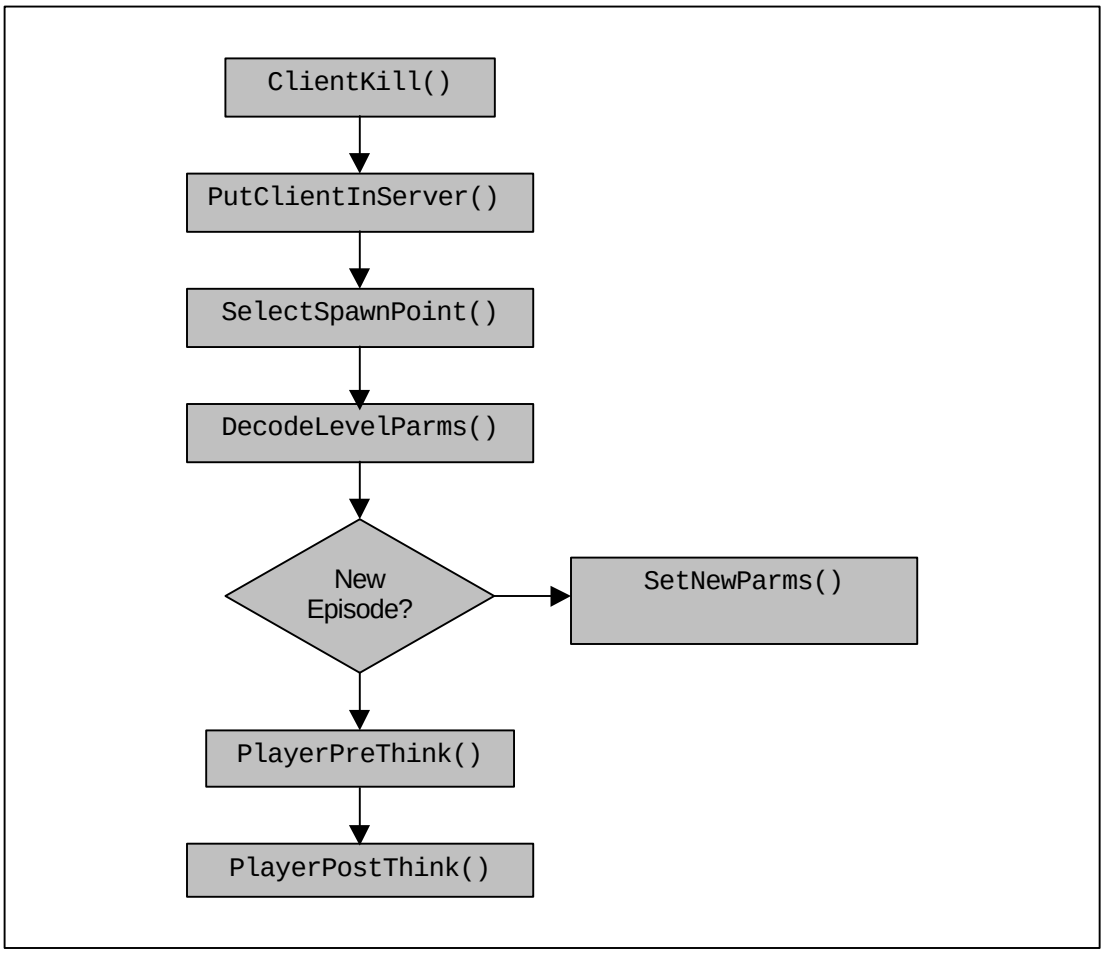
13.5. Program Flow

The following diagrams show the more common program flows during particular client events and states. Of particular interest is the handling of new clients, which is an event distinguishable from a level change only in the state of the client entity variables, the execution flows are identical (and as a result, care should be taken in storing/recalling client state across respawns and levels).

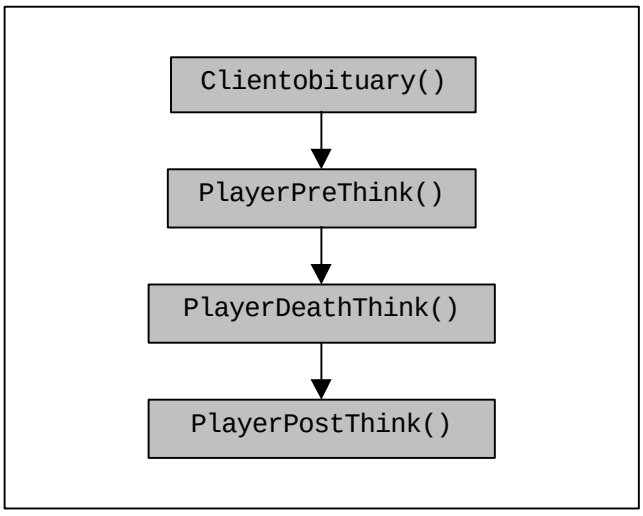
13.5.1. Client Connection/Changelevel



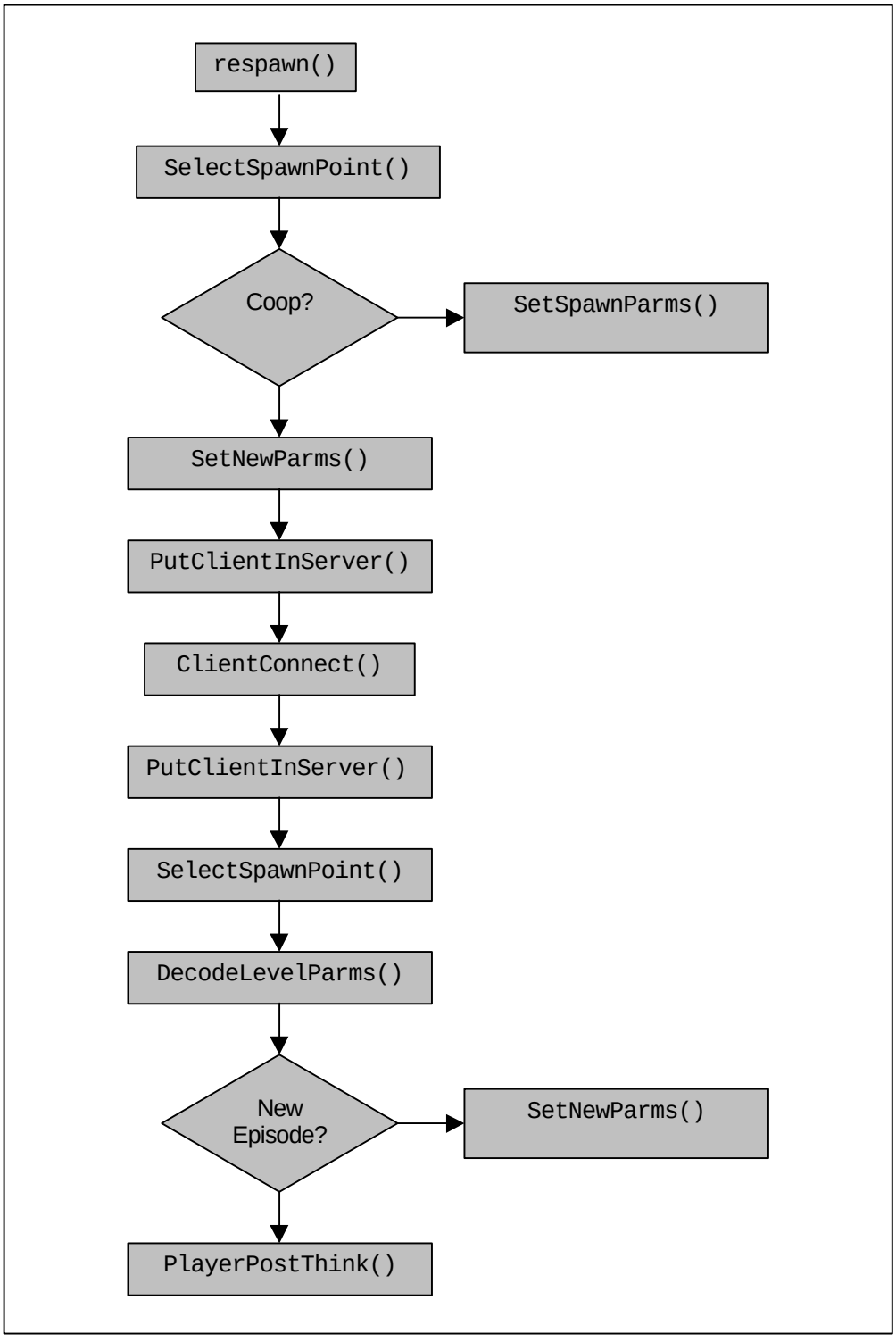
13.5.2. Suicide



13.5.3. Death



13.5.4. Respawn



14. Tips & Tricks

14.1. QuakeC

You cannot initialize a variable with default values

If you give a default value to a QuakeC variable, this variable will be considered as a constant. The value of such a constant may be changed, but it will be set to its initial value on level changes and restarts. Only real global variables before "`end_sys_globals`" in `DEFS.QC` keep their values on level changes and restarts (like `parm1`…`parm16`).

Constants cannot be given the value of a constant

Constants must be given the value of a literal only. Constants set equal to the value of another constant will return unpredictable results.

Coordinates are relative to the world

All the geometry (coordinate positions, directions, angles) are relative to the world. They are never relative to a given object. To know the direction an object is facing, you have to require calculation of the `v_front` vector (respectively `v_right` and `v_up` for the right top).

14.2. Compilation of QuakeC

The language is strongly typed and there are no casts.

Source files are processed sequentially without dumping any state, so if a file is the first one processed, the definitions in that file will be available to all subsequent files. Nothing may be references without first defining it.. Names can be defined multiple times until they are defined with an initialization, allowing functions to be prototyped before their definition.

Error recovery during compilation is minimal. It will skip to the next global definition, so you will never see more than one error at a time in a given function. All compilation aborts after ten error messages.

Beware of the QuakeC compiler!

14.3. Frequently Asked Questions about QuakeC

14.3.1. How do I combine QuakeC patches?

Despite what some people may tell you, QuakeC patches may not be combined by using multiple "`-game`" parameters on the command line, nor by using multiple `PROGS.DAT` files. The only way to combine patches is to get the QuakeC source for both patches, and combine the code by hand. This typically requires some expertise with QuakeC, and is generally not an easy task, given that much QuakeC code is uncommented, messy, and poorly documented.

14.3.2. When I compile valid code, the QuakeC compiler crashes, or I get error messages I know are false, what's wrong?

The QuakeC compiler has certain values hard-coded into it, such as the maximum number of globals. Chances are, you code has exceeded this limit. According to Dave "Zoid" Kirsch, the crash-and-burn behavior of the compiler doesn't mean you're out of luck:

```
"Compiler ran out of globals. You need to edit MAX_GLOBALS in one of the
QCC.H file for qcc and build a new qcc. Quake will load up the progs fine.
It's a qcc thing, not a quake thing."
```

Lee Smith, the author of ProQCC, has modified the latest version (v1.52) with larger table sizes in order to work around the problem.

14.3.3. When I start a game with "-dedicated", I get some messages, then nothing. What's wrong?

Nothing is wrong. What is happening is that when the "`-dedicated`" parameter is specified, the game starts in console mode only. The messages you are seeing are the diagnostic and precache commands that are executed when the server starts. If you want some information on what the server is doing at this point, you can type "`status`" at the console.

For somewhat similar reasons, you can ignore the warnings about "`viewsize`", "`joystick`", etc. commands. They do not have any meaning in the "`-dedicated`" context, and the Quake engine rejects them out of hand, generating the warnings.

14.3.4. How do I change the viewpoint?

You would like that a given player sees through the eyes of another entity. This commonly happens at the end of the level (all players see through a camera), or when the player head is severed (gibbed), or when a player is invisible (he only exists as his eyes).

But the example above works by changing the player entity, and what you want is probably just to see through a camera (Duke3D) or a missile (Descent).

This operation is known in the Quake network protocol as a setview message. But nowhere is it defined in QuakeC, and there's no function to change the view port. So the solution is to encode a set view port message, followed by a set view angles message (to take the orientation of the camera). This works fine, except that if for some reason the entity you are using as a camera was not previously declared to the client, then the view port will be set to '0 0 0', which is usually somewhere in the void.

14.3.5. How do I teleport a player into another server?

Not really, although you can use a few tricks to make something like it.

First, create an entity, such as a slipgate, and set its touch field to the following function:

```
void() changeserver = {

  // set the entity touch function to point here

  // other = entity that touched

  if(other.classname == "player") stuffcmd(other, "connect server.address\n");

};
```

When the slipgate is touched, the entity jumps to another server by virtue of the `stuffcmd()`. However, the player stats and weapons will not be preserved, and the player would be dumped to the console if the other server were full or not available, which makes it of little use.

14.3.6. Can QuakeC bots be listed in the player rankings, or have proper shirt and pants colors?

Yes, but it requires a hack of the Quake network protocol. The bot must be assigned a client number from the pool available (there are `maxplayers` client numbers). The obvious downside to this is that is requires the bot to take a slot that ordinarily would be available for a human player.

Once the bot has it's own client number, it can be assigned it's own colormap and broadcast it's name and frags to the clients. Note that since the bot is not really a client, every time the bot's colors, name, or frag count changes, the change must be forced by sending out another broadcast message.

Alan Kivlin (alan.kivlin@cybersurf.co.uk) has an example of how this might be accomplished in his QCBot. Quaker Server Modifications http://gue-tech.asee.org/quake/qkrmods/ uses this same trick to place CTF Bots in the player rankings screen.

14.3.7. How do I manipulate strings in QuakeC?

Well, you can have any kind of strings as long as they cannot be changed, since, `PR_COMP.C`, defines only operations "=", "==", and "`!=`" on strings.

14.3.8. How do I assemble a piecewise centerprint() from multiple strings?

In the DEFS.QC file, after the lines:

```
// sprint, but in middle

void(entity client, string s) centerprint = #73;
```

Insert the following function definitions, all pointing to the same built-in function immediate:

```
void(entity client, string s1, string s2) centerprint2 = #73;

void(entity client, string s1, string s2, string s3) centerprint3 = #73;

void(entity client, string s1, string s2, string s3, string s4) centerprint4 = #73;

void(entity client, string s1, string s2, string s3, string s4, string s5) centerprint5 = #73;

void(entity client, string s1, string s2, string s3, string s4, string s5, string s6) centerprint6 = #73;

void(entity client, string s1, string s2, string s3, string s4, string s5, string s6, string s7) centerprint7 = #73;
```

You are limited to assembling up to seven strings, since the maximum number of parameters in a QuakeC function is eight (one is consumed by the pointer to the target entity).

14.3.9. How do I move an entity in QuakeC?

You have better not touch it's position fields, else some stuff in the C code might not be valid anymore. Use the `setposition()` built-in function to move an entity.

14.3.10. How to change the velocity of an entity (make it bounce off walls)?

Information by Greg Lewis.

It seems that an entity's velocity can't be changed in the Touch function of the entity. Making the calculations there will be of no use. So just set `entity.movetype` to `MOVETYPE_BOUNCE`, `entity.nextthink` to 0.1 (to let it bounce off), and set `entity.think` to the name of a function that, when called 0.1 second later, will set `entity.velocity` to the right direction.

14.3.11. How to calculate the direction a player is facing?

Assuming the player is `self`, the entity field `self.angles` contains the orientation angles of the player (as set by moving the mouse).

Then the function `makeverctors(self.angles)` will calculate three vectors, that point in the direction the player is facing, but also to the right of the player (strafing direction) and to the direction the player is standing.

Note that those vectors are normalized to 1, so if you want to know what lays 100 units in front of the player, use `self.origin + 100 * facing`.

14.3.12. How to send a message to a client when he logs in?

It has been noticed that using a `sprint()` in function `ClientConnect()` just plain doesn't send any message at all. Maybe the client is not ready to receive messages at this point.

However, Doug Keenan (doug.keegan@tamu.edu) has reported he could send such a text message by putting the `sprint()` close to the beginning of the `ClientConnect()` function. It doesn't work at the end apparently.

14.4. Writing QuakeC Code

Here are some suggestions that you should really consider when writing QuakeC code that would make life simpler for others when they read your code (and also for you to maintain it).

You want to develop code that others can re-use, or that can be mixed seamlessly with code written by others. If you are reinventing the whole world all by yourself, you hardly need any help or counsel (by the way, the first command is "`+light`").

1) Please put comments in your code. Of course, the real gurus don't need comments. They understand raw QuakeC, even compiled. They can even imagine all the parts of your code before they read them. Even before you write them. But actually, they seldom read your code. Only normal people do.

2) Please tag the beginning and end of your modifications if you are fixing a code from someone else. Also put a date, and put a reason for the fix.

3) Each time you create a new function, a new variable or a new field, please give it a name that will not conflict with a function or variable defined by others. A rather sure way to do this is to prefix every name with some abbreviated module name, or your initials, or whatever rare combination of three or four letters.

4) Each time you implement some set of related functions, you should create a new QuakeC module, and give it a name different from the existing ones. Please do not use one of the module names used by id Software, this would be confusing. Try to be original, else we might end-up with two hundred modules called `IMPULSE.QC`.

5) When you want to distribute some modified QuakeC programs: include a `FILE_ID.DIZ` file explaining in about five lines what your patch does, and where it should be stored in the archives (this file is to be read by system administrators), a `README.TXT` file explaining in detail what your patch does, how it does it, and what common files you had to modify. Include the `.QC` modules that you created from scratch.

6) Consider distributing your patches as diff-style patch files.

7) You should compile and distribute a version of your code, as a single `PROGS.DAT` file, to be used by those who just wanna have fun. Don't forget them, they largely outnumber the people who directly deal with QuakeC.

8) Upload your QuakeC patches to the primary quake ftp site at ftp.cdrom.com. Be sure to follow the submission guidelines posted there.

#

A

B

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

V

W