For anyone working on adding elements to the HUD in Quake 2, here is a list of my findings / a quick tutorial. I hope this helps someone.

For myself I’ve added HUD elements for:

* A count that increments when the player picks up a particular item.
  + It displays above the normal item-pickup message ( above health )
  + The name of the item is displayed for 3 seconds next to my number.
  + The background of the number flashes when the item is picked up.
  + An icon next to the number
  + The whole element ( icon, number, and string ) disappear at 0 items
* Two different popup displays (similar to the F1-displayed Help Computer )
  + One for the number and type of items picked up when F8 is pressed
  + One for the current abilities of the player when F7 is pressed

The following are all of the .dll-side things I needed to use. Depending on what you want to create, you may not need all of these, and you certainly don’t need to change something in all of them. However, I found the background information helpful, maybe you will too.

First, it’s important to understand that all things drawn to the screen have to be communicated to the server. That is done with network messaging function-pointers as seen here:

**game.h**

// network messaging

void (\*multicast) (vec3\_t origin, multicast\_t to);

void (\*unicast) (edict\_t \*ent, qboolean reliable);

void (\*WriteChar) (int c);

void (\*WriteByte) (int c);

void (\*WriteShort) (int c);

void (\*WriteLong) (int c);

void (\*WriteFloat) (float f);

void (\*WriteString) (char \*s);

void (\*WritePosition) (vec3\_t pos); // some fractional bits

void (\*WriteDir) (vec3\_t pos); // single byte encoded, very coarse

void (\*WriteAngle) (float f);

… …

So, you’ll see at various points chucks of code like:

gi.WriteByte (svc\_temp\_entity);

gi.WriteByte (TE\_EXPLOSION2);

gi.WritePosition (self->s.origin);

gi.multicast (self->s.origin, MULTICAST\_PVS); // let everyone see it who can

OR

gi.WriteByte (svc\_layout);

gi.WriteString (string);

gi.unicast (ent, true); // let the specific entity see it alone

The first part ( with svc\_ ) tells the server the type of message.

The last part tells it where to put what it makes of the message content.

The stuff in-between is the message content.

There are several protocol bytes that describe the messages. They are found here:

**g\_local.h**

***specifically the server commands around lines 33-40:***

// protocol bytes that can be directly added to messages

#define svc\_muzzleflash 1

#define svc\_muzzleflash2 2

#define svc\_temp\_entity 3

#define svc\_layout 4

#define svc\_inventory 5

#define svc\_stufftext 11

… …

Note the general *svc\_layout* and the more specific *svc\_inventory*. The Help Computer ( seen in-game when pressing F1 ) is a *svc\_layout*, while the Inventory ( seen in-game when pressing TAB ) is a *svc\_inventory*.

You’ll also notice that there is no svc\_hud. That’s because the hud is being continuously updated each frame. The server isn’t waiting for a specific message to update it. However, the states of the HUD’s various variables are changed and the .exe-side code does monitor that via a configstring as seen here:

**g\_spawn.c**

***specifically the hud layout string around lines 648-704***

char \*single\_statusbar =

"yb -24 "

// health

"xv 0 "

"hnum "

"xv 50 "

"pic 0 "

// ammo

"if 2 "

" xv 100 "

" anum "

" xv 150 "

" pic 2 "

"endif "

… …

***specifically the deathmatch hud layout string around lines 705-797***

char \*dm\_statusbar =

"yb -24 "

// health

"xv 0 "

"hnum "

"xv 50 "

"pic 0 "

… …

void SP\_worldspawn (edict\_t \*ent)

***specifically the lines:***

// set configstrings for items

SetItemNames ();

***and the line:***

gi.configstring (CS\_STATUSBAR, single\_statusbar);

The single\_statusbar is the HUD. It is a massive configstring INDEXED by the server with the line gi.configstring (CS\_STATUSBAR, single\_statusbar); as seen in SP\_worldspawn.

You may be asking what things like "if 2 " or "xv 50 " or "hnum " or "pic 0 " mean.

"if" means what you think it means, it’s a condition check. The "2" next to it is the INDEX of the player\_state->stats[] array continuously available to the server. So "if 2 " means:

if the second index of the stats[] array is non-zero, resolve the rest of the ‘code’ UNTIL the "endif" then keep reading the configstring, otherwise don’t resolve any ‘code’ until the next "endif" then keep reading the configstring.

"xv 50 "? The "xv" means go to the point on the screen where the top-left corner of the Help Computer is ( about halfway horizontally ). The same goes for "yv". Other designations after x and y just mean things like “start at the bottom” ( yb ), “start at the top” ( yt ), or “start at the left” ( xl )

"hnum "? This is a specific type of number that specifically checks the players health value in the STAT\_HEALTH index of the stats[] array. The same applies to “rnum” ( armor number ) and “anum” ( ammo number ).

"pic 0 "? You’ve probably guess by now that the server is going to see that the configstring is telling it to draw a picture ( or icon ) found via the 0-th index of the stats[] array. "pic 0 " is the same as: “pic player\_state->stats[STAT\_HEALTH\_ICON]”

**This** single\_statusbar **is the FIRST thing you’d add to if you wanted to add another ever-present HUD element.** Just tack some similar code on the end of the string, move the x-y coordinates and you’re halfway there.

The following is where the indexes of the stats[] array can be found.

It is important to note that the stats[] array is several HUNDRED indexes long, but the game only uses about 20 or so of them.

That gives you plenty of extra room for lots of other HUD elements.

**q\_shared.h**

***specifically these STAT\_ variables around lines 990-1015***

// player\_state->stats[] indexes

#define STAT\_HEALTH\_ICON 0

#define STAT\_HEALTH 1

#define STAT\_AMMO\_ICON 2

#define STAT\_AMMO 3

#define STAT\_ARMOR\_ICON 4

#define STAT\_ARMOR 5

#define STAT\_SELECTED\_ICON 6

#define STAT\_PICKUP\_ICON 7

#define STAT\_PICKUP\_STRING 8

… …

***and the CS\_ variables around lines 1090-1115***

//

// 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.

//

#define CS\_NAME 0

#define CS\_CDTRACK 1

#define CS\_SKY 2

#define CS\_SKYAXIS 3 // %f %f %f format

#define CS\_SKYROTATE 4

#define CS\_STATUSBAR 5 // display program string

The stats[] array only contains numbers. It does not contain strings. The way the game figures out what any particular number means is a system of caching configstrings ( one of which is the HUD’s single\_statusbar configstring.

Remember the line gi.configstring (CS\_STATUSBAR, single\_statusbar); from SP\_worldspawn above? That is where the game associated a specific number with a specific string. That applies not only to the hud, it applies to player names, lights, sounds, models, and so on.

Going back to game.h ( where all this started), around lines 110-130 are found the function-pointers for this caching system:

**game.h**

// config strings hold all the index strings, the lightstyles,

// and misc data like the sky definition and cdtrack.

// All of the current configstrings are sent to clients when

// they connect, and changes are sent to all connected clients.

void (\*configstring) (int num, char \*string);

void (\*error) (char \*fmt, ...);

// the \*index functions create configstrings and some internal server state

int (\*modelindex) (char \*name);

int (\*soundindex) (char \*name);

int (\*imageindex) (char \*name);

void (\*setmodel) (edict\_t \*ent, char \*name);

**The SECOND/LAST thing you’d add to if you wanted to add another ever-present HUD element is in p\_hud.c.**

**p\_hud.c**

void G\_SetStats (edict\_t \*ent)

void Cmd\_Help\_f (edict\_t \*ent)

void HelpComputer (edict\_t \*ent)

To change the values of what’s showing up on the HUD use G\_SetStats. Look through that function’s code to see that the indexes of the stats[] array are being set and cleared on various conditions. **That function is where you’d add a new condition and index to conditionally update.**

Going a bit further, the Cmd\_Help\_f function conditionally toggles the Help Computer to display when pressing F1, and the HelpComputer function actually constructs the configstring that’ll be passed to the server as a svc\_layout.

**If you want to add new keybindings you’d look in g\_cmds.c**

**g\_cmds.c**

void ClientCommand (edict\_t \*ent)

***specifically: "help" and "inven"***

void Cmd\_Inven\_f (edict\_t \*ent)

ClientCommand checks some of the non-movement based keybindings. This means it’ll check for “help” or “inven” in the command list and act accordingly. Key binding values are set using config.cfg in the baseq2 folder. Their behavior is set in ClientCommand on the .dll-side and keys.c on the .exe-side. I only ever added new movement and other bindings using the .dll-side code.

Cmd\_Inven\_f passes the contents of cl->pers.inventory[i] to the server via a svc\_inventory server message in WriteShort, instead of a WriteString like the Help Computer does. This is the first clue that the svc\_inventory server message handling is notably different and may require .exe editing if you want to change it. As I said at the top of this, I found that using svc\_layout ( like the Help Computer does ) is a lot more versatile.

Moving on, the way the game displays the name of the item the player just picked up is another cached/indexed string. All the item names are indexed at SP\_worldspawn. Check out the definition for SetItemNames in g\_items.c

When Touch\_Item is called, the index of the specific item name is passed to the stats[] array as seen below:

**g\_items.c**

void SetItemNames (void)

void Touch\_Item (edict\_t \*ent, edict\_t \*other, cplane\_t \*plane, csurface\_t \*surf)

***specifically the lines:***

// show icon and name on status bar

other->client->ps.stats[STAT\_PICKUP\_ICON] = gi.imageindex(ent->item->icon);

other->client->ps.stats[STAT\_PICKUP\_STRING] = CS\_ITEMS+ITEM\_INDEX(ent->item);

other->client->pickup\_msg\_time = level.time + 3.0;

If you want to make the background of your hud element flash ( at least like the health and armor flash when the player gets hurt ), then checkout P\_DamageFeedback.

Depending on which bit of the STAT\_FLASHES index of the stats[] array is set either the health or the armor number will flash. I made my number flash independent of the health or armor flashing. I did that by adding a few lines to the quake2 project code ( .exe-side edit ). There may be another way to accomplish that using the .dll-side code ( game project ), I’m not sure.

**p\_view.c**

void ClientEndServerFrame (edict\_t \*ent)

void P\_DamageFeedback (edict\_t \*player)

***specifically the lines:***

// flash the backgrounds behind the status numbers

client->ps.stats[STAT\_FLASHES] = 0;

if (client->damage\_blood)

client->ps.stats[STAT\_FLASHES] |= 1;

if (client->damage\_armor && !(player->flags & FL\_GODMODE)

&& (client->invincible\_framenum <= level.framenum))

client->ps.stats[STAT\_FLASHES] |= 2;

If you have the **quake2** project ( and not just the **game** project ) you could go into viewing/editing the .exe for the HUD stuff with:

**cl\_scrn.c**

void SCR\_UpdateScreen (void)

***specifically the lines:***

SCR\_DrawStats ();

if (cl.frame.playerstate.stats[STAT\_LAYOUTS] & 1)

SCR\_DrawLayout ();

if (cl.frame.playerstate.stats[STAT\_LAYOUTS] & 2)

CL\_DrawInventory ();

void SCR\_ExecuteLayoutString (char \*s)

In SCR\_UpdateScreen

* Note how the STAT\_LAYOUTS index of the stats array is checked. The first and second bits were set way back in G\_SetStats and this is where they matter.
* The call to SCR\_DrawStats draws the HUD ( health, ammo, etc ).
* SCR\_DawLayout draws whatever svc\_layout messages the server currently has ( the Help Computer for example ).
* CL\_DrawInventory is a specific type of layout written for the inventory alone. The end-result on-screen is the same but the code-side behavior is different because the inventory needs to know the keybindings for weapons, how much of each item is had and so on. I only ever used the general svc\_layout not the svc\_inventory. I found it more versatile, and I didn’t want to corrupt the inventory display.

SCR\_ExecuteLayoutString is invaluable to understanding the *layout string language* because it’s the function that parses all those types of server messages and tangibly draws to the screen.