FALLOUT 2 EDITOR INTRODUCTION

Here is the Fallout 2 editor, the scripts, and the script compiler. We were able to take some time and put together some documentation for it (and an installer), but a lot of it will require some trial and error. If you want to hammer away at it, well, here it is.

Some things to keep in mind:

This editor is not the holy grail. It was never meant to be released to the public. As a result, you may boot the editor up and realize that it doesn't match your expectations for a commercially-released RPG editor. You may suffer some retina burn. Perhaps a strange itching sensation. Constipation. So be prepared - you are about to experience a game editor, intended only for developers.

You use the Fallout 2 editors at your own risk. A poll was posted some time ago (http://feedback.blackisle.com/forums/showthread.php?s=&threadid=51095) asking if fans interested in the editors would prefer to have the editors sooner with no documentation, and the answer was an overwhelming yes, so here it is. We ended up including some documentation, anyway, so hopefully that should help the pains of easing into the editor.

You need to have Fallout 2 installed for these editors to work.

Be sure to check out the readme.txt in the Scripts folder that will be installed with the editor. It explains compiling the scripts, and some examples of how to build them using different C compilers for the preprocessor. There is also information on the Open Watcom compiler and a link for where to get it.

This is not the Fallout 2 source code. There are no plans to release the Fallout 2 source code at this time.

Be warned - various issues with the editor are listed in the "Known Problems" section of this document. Check it out before using the editors.

The Fallout 2 editors are not supported by Interplay. If you have a problem with them, do not contact customer support because they'll have no idea what you're talking about.

If you have any questions about usage of the editors, post them up on the Fallout 1 and 2 forum (www.interplay.com), and if we can answer them, we will. Keep in mind it has been many years since these editors have been used, and a lot of the know-how is either rusty or has been forgotten, but we'll see what we can do.

There are a number of people you should thank for the editors.

Josh Sawyer. Without Josh, you would not have the F2 editor. He brought up the idea in a meeting, and everybody nodded their head because it seemed like a good idea before we realized what a pain in the ass it would be.

Darren Monahan and Feargus Urquhart. Without Darren and Ferg, you would not have the F2 editor.

Chris Heidari. Chris Heidari took a good chunk of his time to test the editor and, as an added bonus, set up the installer to make it easier for you to use.

Scotty Everts, who put the lion's share of work in setting up the editor doc summary.

...and most of all, Chris Jones. Chris worked on getting the F2 editors together during his free time. Without Chris, you would not have the F2 editor.

Thanks and enjoy,

Chris Avellone

FALLOUT EDITOR MAP BUILDING NOTES

Introduction

The Fallout editor was originally written over six years ago. While it was updated and enhanced during Fallout 1 & 2 development, it's archaic by today's standards, and it will take a lot of patience to make maps. (Your first few will be a challenge, but it'll get a little easier with practice.) The main problem with map-building will be finding all the art pieces you want.

There are 6 different art categories; unfortunately, there are no sub-categories for each - the engine didn't support it; as new art was made, it just got added to the end of the list. So you might find a set of wall sections with pieces spread all over the art list. Fortunately there is a Bookmark command which will make it a little easier (explained below).

Also, this version includes art for Fallout 2 and reused art from Fallout 1. Any of the custom art from Fallout 1 is not included as it was originally deleted to save disc space. For those enterprising individuals that have extracted the original Fallout 1 art, you might be able to re-insert it with some work.

Initial Concepts

There are 6 basic art objects (categories) that make up a map.

- **Tiles** - These are the floor and roof tiles; the basic pieces you build all your maps on. Tiles get put down as floors when "Roof" is off. "Roof" on will place tiles at the roof elevation. So any tile can be used either way.

- **Walls** - Building wall sections. They come in N/S & E/W directions. All external & internal buildings are built from these.

- **Scenery** - The detail pieces such as chairs, tables, trash, cars, etc, used to decorate and flesh out an area. Doors are in the scenery list.

- **Items** - Inventory items for guns, stimpaks, etc. For placing on critters, containers, or ground. Containers are also listed under items.

- **Critters** - Your people and monsters. Place them and attach scripts to make them interact in your environment.

- **Misc** - All the odd stuff like projectile graphics and such. Exit Grid sections are part of the Misc list.

Place an object by Right-Clicking on art object bar (see screenshot below for reference in the following directions). Then Left-Click where you want it on the map.

To select a placed object, Left-Click on it. There is no highlight but the selected object's name will appear in the Lower right info screen. Sometimes you might have to click it a few times to get it to

select. You can also move the object around. Hold down the left mouse button while selecting and you can drag to a new location.

Menu Bar - The menu bar is hidden until you move the cursor to the top of the screen.

Bookmarks - You can set a bookmark in each art object category. The 1-9 keys can be assigned to any location on the art list, and each category can have its own set. To do this, move to a location on the art object bar. Select the Menu "Tools/Set Bookmark." Then just press the number key you want that location to be assigned to.

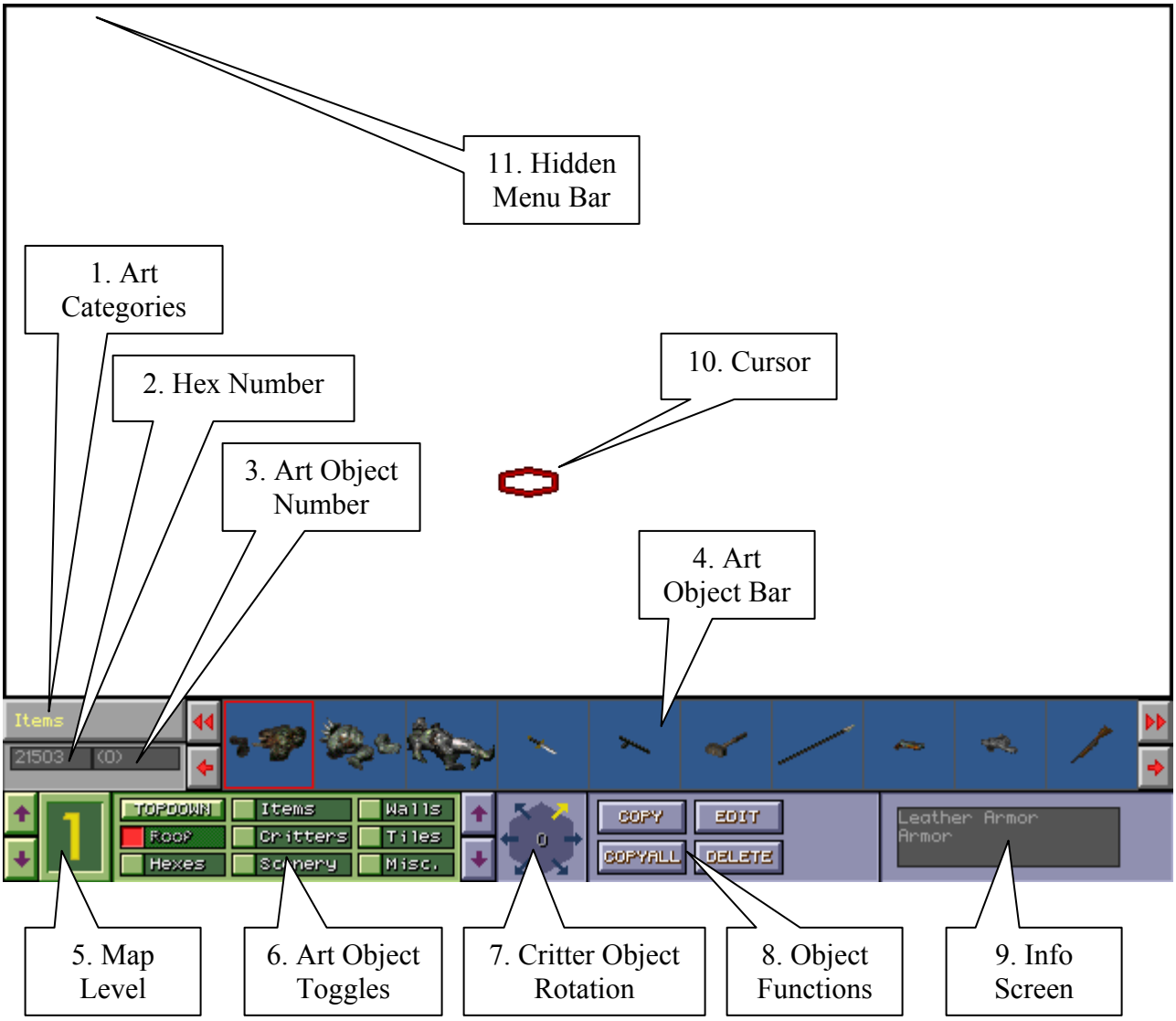
Each map can have 3 map levels. Use the Up/Down arrows on the lower left side of the edit screen to switch levels. This is used for similar levels like dungeons so you don't have to wait for a new map to load when switching levels.

The "**H**" key will toggle hexes on and off. A Hex filled in red is *blocked* and no PC or NPC can enter it. Most walls and scenery are set to Blocking on. Sometimes when laying out walls or scenery, there will be holes, which can result in embarrassing firefights in towns with enemies shooting you through these holes in the walls without you realizing it. Blocking hexes are explained in more detail below.

The "**F8**" key will switch the editor into "Game" mode. You can walk your character around the map and check holes in walls, or how scroll blocking is working, etc. This is the best key in the universe.

The "**F12**" key will take a screenshot. You do not have to be in game mode - you can take screenshots at any time while you are in the editor, compiling information on weapons, ammo, critter stats, and so on.

Basic Editor Functions



1. **Art Categories** - You can use F1-F6 to jump between them instead of using the menu.

2. **Hex Number** - The number of the map hex you are placing an art object. This is helpful for telling a scripter which of the 34+ hookers on a map you need a special script attached to, as well as laying boundaries for certain scripts.

3. **Art Object Number** - The number of the art object in position one of the Art Object Bar.

4. **Art Object Bar** - Shows the art objects of the selected category. Double Arrows scroll by page, single arrow one at a time. Right-Click to select one. Left-Click to place on map. “Plus” and “Minus” keys scroll through the list. “Shift” scrolls by page.

5. **Map Level** - You can have up to three maps in one file. Helps save on load times for dungeons or building interiors.

6. **Art Object Toggles** - Toggles the different categories of art objects on and off. Roofs are default to "off." To lay roof tiles you need toggle on. Remember to Toggle roof tiles off to lay

ground tiles again. “R” key is a quick key for doing this. “Top Down” button is non-functional - it was a feature that was never implemented.

7. **Critter Object Rotation** - To select which way a critter is facing when placed. If your super mutants are always getting placed down with their asses facing you, check the critter object rotation and change it until they're facing the right direction.

8. **Object Functions** -

    - “**Copy**” will copy selected elements of the art object group you are currently in. If you are in the Tiles category, only tiles will be copied, etc. Drag the box to select the copied elements. You can stamp the selected group anywhere you like. Right click to clear.

    - “**Copy All**” functions like Copy but will copy everything selected including tiles, critters, scenery, etc. Great fun at parties.

    - “**Edit**” is for changing attributes of the selected object.

    - “**Delete**” will permanently remove the selected object.

9. **Info Screen**- Lists art object names and other status info.

10. **Cursor** - Changes based on mode or function.

11. **Hidden Menu Bar** - Appears when cursor is moved to top of screen.

Menu Bar

File-

Standard Load, Save, Quit menu. You know the deal.

Tools-

- “**Create Pattern**” / “**Use Pattern**” - Allows you to set Tile patterns and place them. Making new patterns doesn’t seem to work. “Use Pattern” has a list of various pre-made patterns to make it easier to layout ground and floors. Select one off the list. You can now stamp that pattern down. The “Plus” & “Minus” keys change the size of the stamp. Right-Click to exit Pattern mode.

- “**Move Map**” - Allows you to shift the map. In case you run out of space on an edge, this allows you to move the whole map around to re-center it. The arrow keys should shift it but there was some trouble getting this function to work.

- “**Move Map Elevation**” - Moves the whole map to another map level.

- “**Copy Map Elevation**” - Copies the whole map to another map level. If you are building two similar levels, this will save time from having to recreate most of the previous level.

Note: There seems to be a problem when using the "Copy Map Elevation" function. Trying to copy a level to another map/elevation level, the editor will crash to the desktop with a generic "Program error" message. Hell, it may work for you, but it wasn't working for us.

- "**Toggle Block Object View**" - There are various invisible blocker objects used to block or restrict movement and screen scrolling. This menu option toggles them visible in the editor so you can edit and move them - they are never visible in-game, otherwise you would have one surreal Fallout game on your hands. Many times when placing art, there will be holes in walls or scenery. These blockers are used to fill in those holes so the PC or NPC's don't walk (or shoot) through areas of the map that they aren't supposed to. Check out some of the Fallout 2 maps for examples of where they are used, and you'll see what we mean.

  - **Wall Blocker** - Dark Green "W." Blocks movement. Shows up as Wall (Light Green) on in-game Map. Comes in two versions, but it appears only the one labeled "Wall S.T." is used. (Walls category, tile#621)

  - **Secret Blocking Hex** - Light Green "S." Blocks movement. Shows up as Scenery (Dark Green) on in-game Map. (Scenery category, tile#66)

  - **Block Hex Auto Inviso** - Yellow "SAI." Blocks movement. Invisible on in-game Map. Used to block out areas of a map that you don't want to show up on the in-game map. (Scenery category, tile#343)

  - **Light Source** - Green "Yellow Sun symbol." Does not block movement. Places light source in hex. Light can be adjusted by using "Edit." (Scenery category, tile#140)

  - **Exit Grid Map Marker** - Blue "EG." Special markers that display in Tan on the in-game map. Used to indicate where Exit Grids are. (Scenery category, tile#48)

  - **Scroll Blocker** - White "S." A special marker that restricts screen scrolling. When the center of the screen hits a row or column of these, the screen will no longer scroll any farther. Used to define the edge of a map. Please note the scroll blocking works both ways. If your character somehow gets outside the edge, you will not be able to scroll inside the map area. (Misc. category, tile#11)

  Many of these blockers were designed to work with the in-game map. The in-game map shows objects in the Wall category in bright green. Scenery objects in dark green. To keep the in-game map useful the different shades were used to make buildings stand out from the scenery. So when placing blocking hexes, use wall blockers for filling in holes in walls, and scenery blockers to fill in holes in scenery. The inviso blockers work well for blocking large areas like lakes, rad goo, etc. that you don't want to show up on the in-game map.

- **Exit Grid Options** - Exit Grids are the brown and green dithered areas that are used to warp the player to a new map. The art for the dithered pattern is in the Misc. category. Green is used for transitions between maps or map levels, and brown is used for transitions to the World Map. (There is also a black grid, but it's unused in the game.) You will notice that when placing grid pieces, there is a hotpoint for each that shows up light blue when Hexes are visible. That blue hex is important - for exit grids to work, the player must move across that hex. So when laying

down a series of Exit Grids, make sure the grid hotpoint is facing towards where the player is entering from. And make sure an uninterrupted line of them is placed so a character always has to cross one to reach the Exit Grid.

- ♦ **Set Exit Grid Data** - Sets the Exit grid data. After setting the data, you will mark the Exit Grid objects with the other two options. Entering -1 in "Exit Grid Dest Map" option will send the player to the World Map.

- ♦ **Mark Exit Grids** - This will put you into a mode where you can mark exit grids to what you entered in the above command. Clicking each Exit Grid piece will mark it. Press "ESC" to exit this mode. Please note there is no indication you are in this mode. Other functions may not work until you exit this mode.

- ♦ **Mark All Exit Grids** - This will mark ALL exit grids on the map to what was set in "Set Exit Grid Data". It will override whatever you previously set.

- **Clear Map Level** - Erases all objects from the map.

The remaining options are script related and are not covered in this document.

Known Problems with the Editor

The following problems exist with the editor:

- Mapper will crash if the user tries to show map script after having canceled out of setting a map script. To repeat, go to Scripts -> Set Map Script. Press ESC twice to cancel out of menu. Now try to show the map script by going to Scripts -> Show Map Script. At this point the mapper will crash to the desktop.

- You may encounter sporadic problems while going through and loading up all the maps - after awhile, maps may load in being completely engulfed in blackness. Moving the screen or cursor around would eventually make the blackness go away.

- Occasionally if you Alt-Tab out of the editor, the taskbar entry for the editor will disappear even though the Alt-Tab list still shows the program as running. As a warning, Alt-Tab seems to screw with the editor a bit.

- There is a problem when using the "Copy Map Elevation" function. Trying to copy a level to another map/elevation level, the editor will crash to the desktop with a generic "Program error" message.

Script Commands for the Editor

The following was a list of potential script commands that may prove useful for programmers and scripters out there in the fan community who are trying to decipher what does what. I do not know what the gray-shaded definition blocks mean, but it could be significant as you're tearing through the guts of these commands. Maybe gray-shaded means these commands are diseased. Or don't work. Or got cut. Explore and find out, or wait for Red! and some crazy Team X Russians to figure it out.

action_being_used *int Script*	Returns the current skill () being used on a script object.
add_obj_to_inven *void Inven* who (ObjectPtr) item (ObjectPtr)	Adds an object (item) to another object's (who's) inventory. Note that this only works with objects of type Item.
add_mult_objs_to_inven *void Inven* who (ObjectPtr) item (ObjectPtr) count (int)	Adds (count) instances of an object (item) to another object's (who's) inventory. Note that this only works with objects of type Item.
add_timer_event *void Meta(Time)* obj (ObjectPtr) time (int) info (int)	Adds a timed event call to the queue, at a given time offset, to call an object's (obj) script. Info is used to let scripts differentiate between timed event calls so that they can be hooked in multiple times. Info is read back by the script using the fixed_param operator. Note that time is in ticks (you can use game_ticks(seconds_num) to get the time in ticks from time in seconds).
anim *void Anim* who (ObjectPtr) anim (int) direction (int)	Sets up a single-frame animation (anim) for the object (who) that runs in the given direction.
anim_action_frame *int Anim* who (ObjectPtr) frame (int)	Returns the action frame of the given art frame on a given object (who). This can be used as the delay in an animation registration sequence.
anim_busy *int (boolean) Anim* who (ObjectPtr)	Returns True if object (who) is currently animating, otherwise False. This can be used to determine if a given object has completed an animation.
animate_move_obj_to_tile *void Anim* who (ObjectPtr) tile (int) speed (int)	Sets up an animation for a critter (who) to walk to a given tile (hex) at a given speed (speed). Speed (walk/run) can also have a flag attached (see define.h) to force the object (who) to stop it's current animation (for instance, if it was already walking somewhere) and then walk/run to the new location (tile).
animate_rotation *void Anim* direction (0-5)	Changes the orientation (facing) of the self-object to the given direction.
animate_run_to_tile *void Anim* tile (int)	Sets up an animation for the self-object to RUN to a given tile (hex).

animate_set_frame *void Anim* newFrame (int)	Changes the current animation frame of the self-object to the given frame # (newFrame). This can be used to make lights go to broken lights or to alarm/siren lights, for example. Should be used in place of animate_stand for 2-frame anims.
animate_stand *void Anim*	Sets up an animation for the currently focused object (self) to run it's stand animation. This can be used to open doors, open container items (Refridgerator, for example) or to run a critter's fidget animation.
animate_stand_obj *void Anim* obj (ObjectPtr)	Sets up an animation for an object (obj) to run it's stand animation. This can be used to open doors, open container items (Refridgerator, for example) or to run a critter's fidget animation.
animate_stand_reverse *void Anim*	Sets up an animation for the currently focused object (self) to run it's stand animation in reverse. This is used only for non-critters, to cause them to close (close doors, open containers, etc.)
animate_stand_reverse_obj *void Anim* obj (ObjectPtr)	Sets up an animation for an object (obj) to run it's stand animation in reverse. This is used only for non-critters, to cause them to close (close doors, open containers, etc.)
art_anim *void Anim* fid (int)	Returns the animation that this fid represents (ANIM_stand, ANIM_pickup, etc.).
attack *void Combat* who (ObjectPtr)	Causes the focused object (self) to attempt to attack an object (who). Note that this is a macro to attack_complex() below.
attack_complex *void Combat* who (ObjectPtr) called_shot (int) num_attacks (int) bonus (int) min_damage (int) max_damage (int) attacker_results (int) target_results (int)	Causes the current object (self – must be a critter) to attempt to attack a critter (who) with various parameters modifying the combat: called_shot – 0/1/specific means none/random/specific (head, torso, etc.) num_attacks – the # of extra attacks the self object gets before the target bonus – the bonus to hit the target on the first turn only min_damage – the minimum damage that will be done the first attack max_damage – the maximum damage that will be done the first attack attacker_results – what state the attacker ends in after the first attack target_results – what state the target ends in after the first attack
attack_setup *void Combat* who (ObjectPtr) victim (ObjectPtr)	Sets up an attack from who on victim, without expecting **this** script to be involved. Can be used to setup attacks on critters from the map script.
car_current_town *int Map*	Returns the current town area the car can be found at. Area #'s can be found in scripts\headers\maps.h
car_give_to_party *int Map*	Gives the car to the party, and takes them to the worldmap.
car_give_gas *int Map* amount (int)	Gives the car a given amount (amount) of gas.
combat_difficulty *int*	Returns the current **Combat** difficulty level of the game (defined in the options screen).
combat_is_initialized *int*	Returns True if the system is currently in combat mode, False otherwise.

create_object *int (?) Object* pid (int) tile_num (int) elev (0-2)	Creates a new object of prototype (pid), placing it at a given tile # and at a given elevation. If the prototype indicates a script should be attached, then it will be.
create_object_sid *int (?) Object* pid (int) tile_num (int) elev (0-2) sid (int)	Creates a new object of prototype (pid), placing it at a given tile # and at a given elevation. If sid is **not** -1, then it indicates that the default script should be overriden by this new script #.
critter_add_trait *int Critter* who (ObjectPtr) trait_type (int) trait (int) amount (int)	Adds a particular trait (trait) of a given type (trait_type) to a particular critter (who). Possible traits under the SPECIAL system are limited to: Perks Traits Object-instance information (such as team #'s, ai-packet #'s, etc.)
critter_attempt_placement *int Map* who (ObjectPtr) hex (int) elev (0-2)	Attempts to place a critter at a given destination hex & elevation, if it fails, then it tries to find a nearby hex that is that is as near as possible to the start hex. No LONGER checks to see if the hex is visible on-screen.
critter_damage *void Critter* who (ObjectPtr) dmg_amount (int)	Inflicts damage on a critter (who) of a given amount, killing it if necessary.
critter_heal *void Critter* who (ObjectPtr) amount (int)	Heals a critter for a given amount up to maximum.
critter_injure *int Critter* who (ObjectPtr) how (int)	Injures a given critter (who) by crippling given limbs/body parts (defined by DAM_CRIP_ARM_LEFT, DAM_BLIND, etc. in define.h)
critter_inven_obj *(ObjectPtr) Critter/Inven* who (ObjectPtr) where (int)	Returns a pointer to an object that is in a given spot (NULL if none). The appropriate values for where are: INVEN_TYPE_WORN, INVEN_TYPE_RIGHT_HAND, and INVEN_TYPE_LEFT_HAND.
critter_is_fleeing *int Critter* who (ObjectPtr)	Returns True if the critter object (who) has its FLEE flag set.
critter_mod_skill *int Critter* who (ObjectPtr) skill (int) amount (int)	Modifies a given skill in a given critter object (who) by a given amount. Note: this currently is only valid on the player (obj_dude) object.

critter_rm_trait *int Critter* who (ObjectPtr) trait_type (int) trait (int) amount (int)	Removes a particular trait (trait) of a given type (trait_type) from a particular critter (who). (See **critter_add_trait.**)
critter_set_flee_state *int Critter* who (ObjectPtr) flee_on (Boolean)	Sets the FLEE flag on or off. This controls whether the critter flees during combat.
critter_skill_level *int Critter* who (ObjectPtr) skillNum (int)	Returns the current skill level of a particular object's (who) skill (skillNum).
critter_state *int Critter* who (ObjectPtr)	Returns the state of a given critter object (from combat data), used to determine if a critter is dead, unconscious, etc..
critter_stop_attacking *int Critter* who (ObjectPtr)	Flags the critter object (who) as no longer wishing to be active in combat.
cur_map_index *int Map*	Returns the index # of the current map, to be matched with the define-constant in define.h.
cur_town *int Map*	Returns the index # of the current town, to be matched with the define-constant in define.h.
days_since_visited *int Map*	Returns the number of days since this map was last visited, or (-1) if it has never been visited before.
debug_msg *void Debug* text (string)	Prints a string to the debug monitor. Should be used exclusively for debug information, instead of display_msg()!
destroy_object *int Object* obj (ObjectPtr)	Destroys an object (obj), which will cause it's script to be called in the destroy_proc section if the object is *NOT* the calling object.
destroy_mult_objs *int Object* item (ObjectPtr) count (int)	Destroys count number of instances of an item object. This function will figure out which inventory this item is in (if it isn't on the ground). If it is on the ground, of course, there is only one instance of this object, so only one will be destroyed.
dialogue_reaction *void Dialog* mood (int)	Set up a reaction animation in the dialogue system.
dialogue_system_enter *void Dialog*	Tells the dialog system that this object is requesting the talk system. This is used when the **script** wants to start dialog instead of waiting for the player to initiate it. The script will be called back in its talk_proc section.
difficulty_level *int*	Returns the current **Game** difficulty level of the game (defined in the options screen).
display_msg *void* message (string)	Displays a string on the in-game PDA display (lower-left hand corner).

do_check *int (roll_result) Skill* who (ObjectPtr) check (int) modifier (int)	Do a check/test-roll versus one of the various basic traits (strength, perception, etc.).
drop_obj *void Inven* obj (ObjectPtr)	Causes the critter self-object to remove a given object (obj) from it's inventory and place it on the ground at its hex. This animates the self_obj.
drug_influence *int Critter* who (ObjectPtr)	Returns True if a given critter object (who) is currently under any drug influences, False otherwise.
dude_obj *(ObjectOtr)*	Returns a pointer to the dude object (the player).
elevation *int Map* obj (ObjectPtr)	Returns the current elevation being displayed.
end_dialogue *void Dialog*	Terminates the dialogue system.
endgame_movie *void Meta*	Plays the endgame movie.
endgame_slideshow *void Meta*	Plays the endgame slideshow. The slideshow will fade in to its palette, so it is proper to call gfade_out(1) and then expect this command to fix the palette for you.
explosion *int Anim* where (int) elevation (0-2) damage (int)	Sets up an explosion at a given tile number (where) on a given elevation, that will cause damage in a radius.
fixed_param *int*	Returns the value of the scripts fixed parameter. This is used with add_timer_event, for instance, to pass the info parameter back to the script.
float_msg *void* who (ObjectPtr) msg (str) type (int)	Attempts to create a floating-text message (str) attached to an object (who) using colors dictated by type. There are two special types, WARNING and SEQUENTIAL. WARNING is used to print a message centered on the screen (such as for end-of-quest notifications), and SEQUENTIAL will cycle through the colors, in an attempt to give critters different-colored messages to differentiate them.
game_ticks *int Time* seconds (int)	Returns the number of game ticks equal to a given # of seconds.
game_time *int Time*	Returns the current game time in ticks.
game_time_advance *void Time* amount (int)	Advances the current game time by (amount) ticks.
game_time_hour *int Time*	Returns the current hour of the day in a normal format, but without the colon. For example, the current starting game time is 721 (which is 7:21 am).

game_ui_disable *void Meta*	Disables game user-interface input from the player (to 'lock-out' the player). You *MUST* make sure to re-enable the UI at some point afterwords.
game_ui_enable *void Meta*	Re-enables game user-interface input from the player. This *MUST* be called relatively soon after disabling the UI or the player will be stuck, unable to do anything.
game_ui_is_disabled *int Meta*	Returns True if the game UI is currently disabled (the player is currently 'locked-out'), and False otherwise.
gdialog_barter *int Dialog*	Tells the dialog system to switch to the barter screen. (Sets the barter modifier to 0).
get_critter_stat *int Critter* who (ObjectPtr) stat (int)	Returns the value of a desired attribute/stat in a critter object (who).
get_day *int Time*	Returns the current day of the month.
gdialog_mod_barter *int Dialog* modifier (+/- percent)	Tells the dialog system to switch to the barter screen, using a given modifier.
get_month *int Time*	Returns the current month of the year.
get_pc_stat *int Critter* pcStat (int)	Returns the value of a desired pc-only stat of the obj_dude. These are found in define.h all starting with "PCSTAT_".
get_poison *Critter* who (ObjectPtr)	Returns the value of a given critters' (who) poison level.
gdialog_set_barter_mod *void Dialog* mod (int)	Sets the current modifier for barter to a given percentage (mod). Used to make barter easier/harder, even if the **player** initiates barter (as opposed to the script starting it.)
gfade_in *void Meta* time (int)	Does a palette fade to black. The Time parameter is currently not actually used.
gfade_out *void Meta* time (int)	Does a palette fade from black to the game palette. The Time parameter is currently not actually used.
giQ_Option *void Dialog* iq_test (int) msg_list (int) msg_num (int) target (procedure) reaction (int)	Sets up an option-choice for a reply block if the player's IQ statistic is equal to or greater than a given value (iq_test), getting the string from the message file (msg_list) and message number (msg_num), which will cause a given reaction (reaction), and if chosen will jump to the given (target) procedure.
give_exp_points *void* points (int)	Adds experience points (points) to the player's total. These points may then be used by the player to enhance skills, etc.

global_var *int Map* var_index (unsigned int)	Returns the value of a global variable # (var_index).
goto_xy *Map*	
gSay_End *void Dialog* var_index (unsigned int)	Ends a dialog sequence, which will bring up the sequence (actually display it).
gSay_Message *void Dialog* msg_list (int) msg_num (int) reaction (int)	Sets up a sayMessage, which is a reply with just a [Done] option. The msg_list determines which message file to look in, and the msg_num determines which line to use from the file.
gSay_Option *void Dialog* msg_list (int) msg_num (int) target (procedure) reaction (int)	Sets up an option-choice for a reply block, getting the string from the message file (msg_list) and message number (msg_num), which will cause a given reaction (reaction), and if chosen will jump to the given (target) procedure.
gSay_Reply *void Dialog* msg_list (int) msg_num (int)	Sets up a reply block (what the *CRITTER* says).
gSay_Start *void Dialog*	Starts a new dialog sequence.
has_skill *int Skill* who (ObjectPtr) skill (int)	Determines if a critter (who) 'knows' a particular skill. Actually, this currently returns the level of the skill, which will include defaults.
has_trait *int Critter* trait_type (int) who (ObjectPtr) trait (int)	Returns the value of a given critter object's (who) trait of a given Trait_type (see define.h). This can be used to determine if the player has a particular Perk, AI Packet, team num, current rotation, or Trait (finesse, bruiser, etc.).
how_much *int Skill* val (int)	Returns the value of a completed skill vs. skill contest (how much the rolls differed by). This requires that you first call one of the contest roll commands, such as roll_vs_skill, skill_contest, etc..
inven_count *int Critter* what (ObjectPtr)	Returns the count of how many inventory slots are filled on a given object (what).
inven_ptr *(ObjectPtr) Critter* what (ObjectPtr) slotNum (int)	Returns a pointer to the object in slot # (slotNum) in a given object (what).
inven_unwield *void Critter*	Attempts to cause a critter to unwield any wielded weapons/items. If animations are currently disabled, it will just instantly change the art.

is_critical *int Skill* val (int)	Returns True if a given contest roll result is a critical result, otherwise False.
is_loading_game *boolean Map*	Returns True if the game is currently loading, False otherwise. This is used so that bad things don't happen on game load because a script is doing map_enter setup stuff.
is_skill_tagged *int Skill* skillNum (int)	Returns True if a given skill is tagged.
is_success *int Skill* val (int)	Returns True if a given contest roll result value is a success, otherwise False.
item_caps_adjust *int Inven* obj (ObjectPtr) amount (int)	Modifies the current caps count in an object (obj) by a given amount (amount).
item_caps_total *int Inven* obj (ObjectPtr)	Returns the current caps total in a given object's (obj) inventory.
jam_lock *int Object* lockableObj (ObjectPtr)	Jams a lock, which prevents the player from picking the lock for approximately 24 hours. Meant to be used when a player critically fails to pick a lock.
kill_critter *void* obj (ObjectPtr) death_frame (int)	Kills a critter (obj) outright, placing it in the chosen death frame. Note: this does NOT animate the critter, and does NOT refresh the screen! It is meant to be used in scripts run when entering/exiting a map (map_init/map_exit).
kill_critter_type *void Map* pid (int)	Kills all critters of a given type (pid) outright. See kill_critter above.
language_filter_is_on *int (boolean) Meta*	Returns True if the language filter is currently filtering harsh language, False otherwise.
load_map *void Map* map_name (string) start_location (int)	Loads a new map (map_name), removing all scripts currently running and passing on the entrance location (start_location) to the new map's map_init script.
local_var *int Map* var_index (unsigned int)	Returns the value of a local variable of given index # (var_index).
map_first_run *int Map*	Returns True if the current map is being run for the first time (in other words, this map was not loaded from a save-game).
map_is_known *int Meta* mapNum (int)	Returns True if a given map index (mapNum) is known, False otherwise.
map_known *int Meta* mapNum (int)	Returns True if a given map # (mapNum) is known, False otherwise.

map_var *int Map* var_index (unsigned int)	Returns the value of a map-global variable of a given index # (var_index).
message_str *char ** list (int) msg_num (int)	Returns a string from the message module for a given list and a given # (msg_num).
move_to *int Map* obj (ObjectPtr) tile_num (int) elev (0-2)	Immediately moves an object (obj) to the given tile number and elevation on the current map.
move_obj_inven_to_obj *int Inven* srcObj (ObjectPtr) destObj (ObjectPtr)	Moves an object's (srcObj) inventory into another object's (destObj) inventory.
obj_art_fid *(ObjectPtr) Object* obj (ObjectPtr)	Returns the fid # (used to index art) of a given object (obj).
obj_being_used_with *(ObjectPtr) Object*	Returns a pointer to the object being used on another object.
obj_can_hear_obj *boolean Map* src_obj (ObjectPtr) dst_obj (ObjectPtr)	Returns True if the source object (src_obj) is capable of hearing the destination object (dst_obj). This includes distance factors, current activity (standing/walking/running), and skill use (stealth/etc.).
obj_can_see_obj *boolean Map* src_obj (ObjectPtr) dst_obj (ObjectPtr)	Returns True if the source object (src_obj) has line-of-sight (LOS) with the destination object (dst_obj). This also takes into account perception & stealth rolls of the objects are critters.
obj_carrying_pid_obj *(ObjectPtr) Object* who (ObjectPtr) pid (int)	Returns an Object pointer to an instance of an object of type pid if an object (who) is carrying an object of that type.
obj_close *void Object* what (ObjectPtr)	Attempts to close a given object (what) if it is of an openable type.
obj_drop_everything *void Inven* who (ObjectPtr)	Causes a critter object (who) to drop all objects in it's inventory and drop it on the ground at it's feet.
obj_is_carrying_obj_pid *boolean Object* obj (ObjectPtr) pid (int)	Returns the quantity of objects with matching prototype index #'s (pid) carried in the inventory of another object (obj).
obj_is_locked *int Object* what (ObjectPtr)	Returns True if a given object (what) is a locked object, False if it is unlocked or not a lockable object.

obj_is_visible_flag *int Object* who (ObjectPtr)	Returns True if a given object (who) is turned on (visible), False otherwise.
obj_is_open *int Object* what (ObjectPtr)	Returns True if a given object (what) is an open object, False if it is closed or not an openable object.
obj_item_subtype *int Object* obj (ObjectPtr)	Returns the subtype of an object of type 'item'. Examples would be food, armor, weapons, etc.
obj_lock *void Object* what (ObjectPtr)	Attempts to lock a given object (what) if it is of a lockable type.
obj_name *void Object* what (ObjectPtr)	Returns a string representing the name of the given object (what).
obj_on_screen *int Object* what (ObjectPtr)	Returns True if a given object (what) is currently being drawn on-screen, False if it is not.
obj_open *void Object* what (ObjectPtr)	Attempts to open a given object (what) if it is of an openable type.
obj_pid *int Object* obj (ObjectPtr)	Returns the prototype id # (pid) of an object (obj).
obj_set_light_level *void Object* obj (ObjectPtr) intensity (1-100) distance (0 - 8)	Set the light level for an object to a given intensity (percentage of possible maximum intensity), and distance of light in hexes.
obj_type *int Object* obj (ObjectPtr)	Returns the type of an object (obj). This would be ‘Item’, ‘Wall’, ‘Scenery’, etc.
obj_unlock *void Object* what (ObjectPtr)	Attempts to unlock a given object (what) if it is of a lockable type.
override_map_start *void Map* x (int) y (int) elev (0-2) rot (0-5)	Used when loading a new map, this forces the player (obj_dude) to start at a particular location and rotation when first coming up.
party_add *void Party* who (ObjectPtr)	Adds a given critter (who) into the list of party members. This will also setup those objects so that they will not be saved in maps, and certain other things.
party_member_obj *ObjectPtr Party* pid (int)	Returns an ObjectPtr to a party member that matches a given pid. If that critter isn’t currently a member of the party, then it will return NULL.
party_member_count *ObjectPtr Party* countHidden (int)	Returns the count of the currently in-party party members. (countHidden) determines whether or not to count the hidden members (hangers-on).

party_remove *void Party* who (ObjectPtr)	Removes a given critter (who) from the list of party members. This will also change those objects so that certain object- and map-level things will respond differently to them.
pickup_obj *void Inven* obj (ObjectPtr)	Causes the critter self-object to animate and attempt to pick up a given object (obj).
play_gmovie *Meta*	Plays one of the Fallout movies (full-screen, compressed, etc.).
play_sfx *Sound*	Starts a new sound effect to be played on the queue.
poison *Critter* who (ObjectPtr) amount (int)	Increases the a critters' poison level by a given amount.
proto_data *int OR string Object* pid (int) data_member (int)	Returns the value of a data-member of a given prototype (pid).
radiation_dec *Critter* who (ObjectPtr) amount (int)	Decrements a critter's radiation counter by a given amount. NOTE: This should only be done to the player (obj_dude) in Fallout due to design restrictions!
radiation_inc *Critter* who (ObjectPtr) amount (int)	Increments a critter's radiation counter by a given amount. NOTE: This should only be done to the player (obj_dude) in Fallout due to design restrictions!
random *int Script* min (int) max (int)	Returns a random value between (min) and (max), inclusive.
reg_anim_animate *void Anim* what (ObjectPtr) anim (int) delay (int)	Adds a single, in-place animation on an object (what) to an animation sequence-list, at a given delay from the previous animation (delay should always be -1)!
reg_anim_animate_forever *void Anim* what (ObjectPtr) anim (int) delay (int)	Adds a single, in-place animation on an object (what) to an animation sequence-list, at a given delay from the previous animation (delay should always be -1)! This animation will animate continuously until something in the system interrupts it. To be used *very* sparingly, for instance Gizmo's sign and the 'pray' signs in the children of the cathedral (which will have to be toned down).
reg_anim_animate_reverse *void Anim* what (ObjectPtr) anim (int) delay (int)	Adds a single, in-place reversed animation on an object (what) to an animation sequence-list, at a given delay from the previous animation (delay should always be -1)!
reg_anim_begin *void Anim*	Tells the system to start an animation sequence-list.

reg_anim_clear *void Anim* object (ObjectPtr)	Terminates all animations that are currently registered for a given object.
reg_anim_end *void Anim*	Activates the animation sequence-list. Without this call the animation will never occur. Note: All animation sequences must be registered at ONCE! In other words, you cannot let the script end and finish registering the animations later.
reg_anim_obj_move_to_obj *void Anim* who (ObjectPtr) dest_obj (ObjectPtr) delay (int)	Adds an animation to cause a critter object (who) to attempt to walk to another object (dest_obj) at a given delay from the previous animation (delay should always be -1)!
reg_anim_obj_run_to_obj *void Anim* who (ObjectPtr) dest_obj (ObjectPtr) delay (int)	Adds an animation to cause a critter object (who) to attempt to run to another object (dest_obj) at a given delay from the previous animation (delay should always be -1)!
reg_anim_obj_move_to_tile *void Anim* who (ObjectPtr) dest_tile (int) delay (int)	Adds an animation to cause a critter object (who) to attempt to walk to a given destination tile number (dest_tile) at a given delay from the previous animation (delay should always be -1)!
reg_anim_obj_run_to_tile *void Anim* who (ObjectPtr) dest_tile (int) delay (int)	Adds an animation to cause a critter object (who) to attempt to run to a given destination tile number (dest_tile) at a given delay from the previous animation (delay should always be -1)!
reg_anim_play_sfx *void Anim* who (ObjectPtr) sfx_name (string) delay (int)	Adds an animation to cause an object (who) to attempt to play a given sound effect (sfx_name) at a given delay from the previous animation!
rm_fixed_timer_event *void Meta(Time)* who (ObjectPtr) fixed_val (int)	Removes (clears) all timer events hooked to a given object's (obj) script that have a given fixed_value (fixed_val).
rm_obj_from_inven *void Inven* who (ObjectPtr) obj (ObjectPtr)	Removes an object (obj) from another object's (who's) inventory. Note: this leaves the removed object in at location (0,1) on the map! You must call move_to(…) to place it back on the map.

rm_mult_objs_from_inven *int Inven* who (ObjectPtr) obj (ObjectPtr) count (int)	Removes (count) instances of an object (obj) from another object's (who's) inventory. Note: this leaves the removed object in at location (0,1) on the map! You must call move_to(…) to place it back on the map. NOTE: This function returns the actual count that was removed (if you attempted to remove more instances than existed). You *MUST* store this value in a variable (though you don't have to actually do anything with it).
rm_timer_event *void Meta(Time)* obj (ObjectPtr)	Removes (clears) all timer events hooked to a given object's (obj) script.
roll_dice *Skill*	Returns the value of the completed dice roll. **UNIMPED!**
roll_vs_skill *int (roll_result) Skill* who (ObjectPtr) skill (int) modifier (int)	Returns the value of a completed skill roll made upon an object's (who's) skill level with a given skill, and modified by a given amount (may be zero). This value may then be passed to is_success and is_critical to determine the appropriate states, and the how_much call can be used to determine the difference succeeded or failed by.
rotation_to_tile *int (1…5) Map* srcTile (int) destTile (int)	Returns the rotation (0…5) to face a particular tile (destTile) **from** a particular tile (srcTile).
running_burning_guy *int*	Returns the setting for the running-burning-guy in the game (defined in the options screen).
scr_return *void Script*	Sets the return value for a scripts C-engine node, to be used by C code.
script_action *int Script*	Returns the action that has activated this script. Examples include requests for the description of an object (description_proc), notifications of a spatial script being activated by something hitting its boundary (spatial_proc), or a critter being given its heartbeat (critter_proc, in other words being told to move).
script_overrides *void Script*	Tells the C-engine that the script will override default behavior for the object. What this means is that the C-engine will not attempt to do things that it would normally do, in expectation that the script will handle those things itself. This is an **IMPORTANT** command! It is commonly used for the general player actions upon objects, such as looking at them (requesting a description), using them (opening doors, for example), or using items ON them (using a picklock or a key on a door lock).
self_obj *(ObjectPtr) Script*	Returns a pointer to the object connected to this script.
set_critter_stat *int Critter* who (ObjectPtr) stat (int) val (int)	Sets the value of a desired attribute/stat in a critter (who) to a given value (val).

set_exit_grids *void Map* markElev (elevation) mapID (int) elevation (int) tileNum (int) rotation (int)	Sets all exit grids on a given elevation (markElev) to point to a destination mapID (may be -1 which means stay on this map), elevation, tileNum, and rotation.
set_global_var *void Map* var_index (unsigned int) value (int)	Sets the value of a global variable (var_index) to a given (value).
set_light_level *void Map* level (int: 1-100)	Sets the ambient light level. The range is Full Darkness to Full Daylight.
set_local_var *void Map* var_index (unsigned int) value (int)	Sets the value of a local variable (var_index) to a given (value).
set_map_var *void Map* var_index (unsigned int) value (int)	Sets the value of a map-global variable (var_index) to a given (value).
set_map_start *void Map* x (int) y (int) elev (0-2) rot (0-5)	Sets the start location & rotation for the next time this map is entered (loaded & run).
set_obj_visibility *void Object* obj (ObjectPtr) visibility (boolean)	Sets the OBJ_OFF flag for an object (makes it not drawn).
signal_end_game *void*	Tells the system that a script is indicating the game should be ended. This will return the player to the main-menu.
skill_contest *Skill*	Returns the value of a completed skill vs skill contest (to run through is_success & is_critical).
source_obj *(ObjectPtr) Script*	Returns a pointer to the source object (activator) for this action. The source object for a pickup_proc (pickup an object script_action) would be the critter picking the object up, for instance.
start_dialogue *void Dialog* who (ObjectPtr) mood (int)	Start the dialogue system focusing on a critter (who) and starting in a given (mood). This call sets up the appropriate dialog windows, head art, etc. If this call is not made before the normal dialog calls (sayReply, sayMessage, sayOption, etc.) then the dialog windows will not come up, and only grey boxes will appear with the text.

start_gialog *void Dialog* msgFileNum (int) who (ObjectPtr) mood (int) headNum (int) backgroundIdx (int)	Start the dialogue system focusing on a critter (who) and starting in a given (mood). This call sets up the appropriate dialog windows, head art, etc. If this call is not made before the normal dialog calls (sayReply, sayMessage, sayOption, etc.) then the dialog windows will not come up, and only grey boxes will appear with the text.
target_obj *(ObjectPtr) Script*	Returns a pointer to the target object for this action. The target object is what is being acted upon.
terminate_combat *void Combat*	Tells the combat system to terminate prematurely. USE WITH CAUTION. This doesn't prevent another (or even the SAME) script from re-starting combat, so make sure you turn off any hostile flags, etc.
tile_contains_obj_pid *boolean Map* tile (int) elev (0-2) pid (int)	Returns True if a particular tile contains an object with a matching prototype index # (obj pid).
tile_contains_pid_obj *boolean Map* tile (int) elev (0-2) pid (int)	Returns a pointer to the first object that matches a particular pid # that is on a particular tile and elevation.
tile_distance *int Map* tile1 (int) tile2 (int)	Returns the tile distance between two tile #'s.
tile_distance_objs *int Map* obj1 (ObjectPtr) obj2 (ObjectPtr)	Returns the tile distance between two objects (between their tile #'s).
tile_is_visible *boolean Map* tile (int)	Returns True if a given hex (tile) is currently visible, i.e. an object on it could conceivably be displayed on-screen. This includes hexes that may technically have bases that are off-screen, but on whom objects could exist that would bound into the actual display area.
tile_num *int Map* obj (ObjectPtr)	Returns the tile number of object (obj).
tile_num_in_direction *int Map* start_tile (int) dir (0-5) distance (int)	Returns the tile number of a tile offset from a starting tile in a given direction (the next tile in that direction).
town_known *int Meta* townArea (int)	Returns True if a given town area (townArea) is known, False otherwise.
town_map *void*	Sends a request for the game engine to bring up the Town Map screen, for the player to go to different locations in an area (different areas in Vault13, for example).

use_obj *(ObjectPtr) Script* obj (ObjectPtr)	
use_obj_on_obj *(ObjectPtr) Script* item (ObjectPtr) targetObj (ObjectPtr)	Attempt to use an item object on a target object (targetObj). This could be used to have a critter use a Stimpack on the player, for instance, or to use a key on a door.
using_skill *boolean Skill* who (ObjectPtr) skill (int)	Returns True if an active skill is being used, False otherwise. Examples of active skills are Stealth and First Aid.
violence_level_setting *int (boolean) Meta*	Returns the current setting of the violence level. See define.h for values.
wield_obj *void Inven* obj (ObjectPtr)	Sets up an animation causing a critter (self_obj) to wield an object (obj) in that critters' inventory. This puts that object in the critter's hand.
wield_obj_critter *void Inven* who (ObjectPtr) obj (ObjectPtr)	Sets up an animation causing a critter (who) to wield an object (obj) in that critters' inventory. This puts that object in the critter's hand.
wm_area_set_pos *void* areaIdx (int) xPos (int) yPos (int)	Sets the World Map coordinates for a given area/town (areaIdx) to a given x and y position.
world_map *void*	Sends a request for the game engine to bring up the World Map screen, for the player to move around to different locations.
world_map_x_pos *int*	Returns the current X Position of the party on the World Map.
world_map_y_pos *int*	Returns the current Y Position of the party on the World Map.

Script Actions Summary

Description	Object	A request to examine (extended-look) an object.
Combat	Critter	A combat action is occurring.
Create	Script	This script-object is being created. (UNIMPED).
Critter	Critter	A critter script is getting its heartbeat.
Damage	Object	This object is taking damage.
Destroy	Script	This script-object is being destroyed.
Drop	Item	An object is being removed from another object's inventory and is being dropped on the ground. (UNIMPED).
Look At	Object	A brief look at an object is being requested.
Map Enter	Map	This map is being entered (was just loaded).
Map Exit	Map	This map is being left (is being saved as a savegame).
Map Update	Map	This map is being updated (changing levels, lighting, etc.)
None	Script	No action.
Pickup	Item	An attempt is being made to pickup an object and place it in an object's inventory. Or, could be looting/stealing.
Spatial	Script	This scripts' spatial radius has been entered.
Start	Script	Starting up script for first time.
Talk	Critter	A script dialogue is being requested.
Timed Event	Script	A timed event has just reached activation.
Use	Object	Attempt to use an object.
Use Object On	Object	Use an object on another object.
Use Skill On	Object	Use a skill on an object.

Script Action Groups

Object:

Object actions are generic actions that can be done on any game object. For instance, if something requests to look at an object in the game, this would be valid on any of the normal (non-interface) objects. Individual prototypes of objects determine whether or not that object has certain actions available to it. So, for example, unless the prototype of the object is set to allow you to USE it, the player will never be given the option to do so.

Item:

Item actions are actions specific to those objects that the player can pick up, drop, and wear. In other words, inventory items.

Critter:

Critter actions are called on objects that represent active beings in the game. In a sense, these actions only occur with 'intelligent' objects, which in nearly all cases are capable of movement and entering combat, dialog, etc.

Map:

Map actions are specific to special map occurrences, such as when a map is loaded, saved, or updated. Originally map actions were only available to the **map script** itself, but it became clear that they could potentially be needed for any script. They are extremely helpful for setting up variables at the start of a map. When a map is first entered, the **map script** gets run first, and then every other script which is flagged as needing to be called on enter gets called. This can be used to store map-specific variables in the **map script** and to 'export' them from there. Then any other scripts that need to use these variables just need to 'import' them, since they are guaranteed to exist.

Script:

Script actions relate to script-specific calls, such as when a script is being destroyed, activated by spatial movement, or previously setup timed events. These actions are not directly activated by the player (unlike object actions), but by the script system itself.

Script Action Descriptions

Description:

The player is trying to examine this scripts' object. The default behavior is to print the long description of the object from its prototype in the small display window.

Combat:

Combat is occurring, and one of the combat sub-actions has occurred. These actions allow the script to react to combat (even though combat attack/etc. choices are actually handled in the engine). They are:

- HIT_SUCCEEDED -- The scripts' object successfully hit it's target. This can be used to do extra damage (radiation, for example), or to count the attacks that succeed.
- SEQUENCING – Combat sequencing is being checked (to see if critters want to enter/exit combat).
- TURN – A combat turn is just about to start. You can override the default turn behavior here to prevent a critter from reacting to combat (or to cause them to do something special).
- NONCOM_TURN – (UNUSED).

Create:

(UNUSED).

Critter:

The critter's heartbeat is occurring. Basically, each script gets hit every so often (should be several times a second) so that it can react to its environment or to changes in flag variables, etc. This is where the code can be put to have the critter walk, etc. on its own.

Damage:

Something has occurred to damage this scripts' object. This will almost always mean that a critter has been hit in combat. Here they could set hostile flags (that might affect dialog or quest statuses later), or they could heal themselves to prevent death (this isn't good practice, but may be useful in one or two cases where you want to make sure the critter says something, does something, etc. before they actually die).

Destroy:

This script is being destroyed, and it and all related events (timed events, for instance) are about to be removed from the system. This most likely means it is either a critter that has just been killed in combat, or that it is an inventory item that was just used up. Useful things to do when this action is called are to give the player experience points (for killing creatures or fulfilling quest objectives) and to update counts, such as the kill-counts (number of critters of a given type killed), or local counts (how many radscorpions are left in the cave, or how many gang members are left still in town).

Drop:

An inventory item is being dropped from a critter's inventory.

Look At:

The player is trying to look at the scripts' object. The default behavior is to print the short description, in other words the prototype name in the small display window.

Map Enter:

The map has just been entered, and this script is flagged as needing to be run before the normal game processing starts up. Here, initialization code can setup variables for the map.

Map Exit:

The map is about to be left. This occurs when the player goes to the town/world maps.

Map Update:

The map is getting a 'heartbeat' to let it update the map. Here ambient light levels can be changed, such as when it becomes nighttime or the player changes levels to an underground/above-ground area.

None:

Nothing is happening. This should never be called, but is there for the default case.

Pickup:

This procedure means different things for different types of scripts:

Item Scripts – A critter is attempting to pickup an inventory item. For containers, however, this means that a critter is trying to *open* it/loot it.

Critter Scripts – A critter is attempting to steal from this scripts' critter object.

Spatial:

An object has just moved in this scripts' spatial sphere of detection. The script may need to double-check that this object is of a particular type (a critter, the player's object, etc.).

Start:

A script is being run for the very first time.

Talk:

A critter object is starting to talk, either because the player requested it, or because the critter's script requested that it occur. Here the dialog system needs to be setup, and the actual script-language dialog system calls will be placed.

Timed Event:

A timed-event has just gone off for this script. Usually, this event was setup earlier by this script itself. It lets the script delay events or in effect give itself a heartbeat.

Use:

Something (usually a critter) is attempting to use this object. The command source_obj will return who it is. Nearly always this will mean a critter (most likely the player) is attempting to use it, but occasionally another script will make the call, which can be used to differentiate its behavior. For instance, this allows the vault door scripts to open the vault door when the player uses the vault door computer, but **not** when the player attempts to open the door directly.

Use Object On:

A critter is attempting to use an inventory item on this scripts' object. This could be lockpicks on a door, a medical kit on a critter, the iquana-on-a-stick on a dog, etc.

Use Skill On:

A critter is attempting to use a skill on this scripts' object. The default behavior is dependent on the specific skill itself.