Zelda Classic and ZQuest

v2.54 API Specification

Baseline

The 2.54 API is based entirely on the specification of ZC and ZQuest v2.53.0. All of the core components of 2.53.0 (Final) are the criteria for the base of the 2.54 specification; and all of the following represent changes to that specification, in the form of new features, or feature modifications.

Compatibility

As with prior versions of Zelda Classic and ZQuest, version 2.54.0 strives to permit the user to open older quests either in ZQuest, to edit and save them; or to play them in the ZC player.

The present specification supports quests made in the following ZC versions:

1.90

1.92 (various)

2.10

2.10.1

2.10.2

2.50.0

2.50.1

2.50.2

2.53.0

Unlike the various 2.50.x builds, and version 2.53.0, quests created in 2.54.0 will require 2.54 (or higher) to be usable, either to open them in the ZQuest editor, or to play them in ZC (quest player).

This is an integral component of the specification, and should there be a v2.54.1, then all quests made in that, will require 2.54.1 or higher. All support for back-loading (e.g., loading a quest *made in 2.50.2*, in 2.50.0) quests to *earlier* builds is dead, from this point onward.

Support for forward-loading will be maintained indefinitely. Future versions of ZC in the 2.xx series will continue to support 1.90 and higher quest formats, and we may be able to add 1.84 to this specification with some work; or possibly 1.50 with additional work, as the source for 1.50 is available.

Libraries

ZC and ZQuest v2.54 shall use the same libraries as v2.53.0, notably Allegro 4.4.3 (including our fixes), its dependencies (JPGALLEG v2.5, loadpng (1212?), libdumb, almp3, and alogg); GME 0.6, and ZLIB 123.

GME should be probably be updated to the current version. The other libs seem to be satisfactory for the present working environments under which we deploy our software.

Feature Additions

ZScript Parser

The specification for the ZScript parser now includes *comment blocks* using standard C syntax. No other parser changes are included in this spec, beyond those of the 2.53 API: Namely that scoped variables are destroyed when their scope exits.

ZScript Pointers

The addition of new pointers:

Audio-> : This is for all audio functions, and shall be used for all future audio features of the script engine. Old audio features shall be cross-ported, so that they work both under their old pointer (for legacy support), and under this new pointer.

Debug-> : Used both for experimental instructions (undocumented), and for instructions meant to be used for debugging scripts. /i

ZScript Instructions

The API spec now includes the following internal ZScript instructions (sorted by pointer type):

Global Instructions

`OverlayTile(int t1, int t2)`
This overlays one tile onto another, and only affects in-RAM quests. It does not write in a permanent manner, and the overlay is lost when a quest file is reloaded.

`int Version;`
This is an integer containing the present version of Zelda Classic (when running a script in a quest), to report it to scripts. It reports in a manner as follows: 25000 (2.50.0), 25001 (2.50.1), 25002 (2.50.2), 25300 (2.53.0) , 25400 (2.54.0). At present, this would report 0 or -1 for version of ZQuest that do not support this data, and it is also included in the 2.53.0 specification, albeit *silently*.

`int Build;`
Similar to *Version*, but it returns the build ID. Useful for debugging beta and release candidate scripts. May be useful in the future if small version differences occur, for interim support.

`int Beta;`
Might be dropped from the spec. This returns the Beta ID, or 0 if the ZC version is neither a beta, nor an alpha. Beta IDs return as positive integers, and Alpha IDs return as *negative* integers.

`bool DisableActiveSubscreen;`
A flag to prevent the active subscreen from falling when the player presses the Start key (or joypad button). This is useful for quests where the internal ZQuest subscreen is not used, such as with scripted subscreens; or during cutscenes; or during events where the questmaker wishes to disable it.

A similar flag for the passive subscreen is likely prudent, but can wait for the 2.55 spec.

`int SizeOfArrayBool(bool *ptr)`
Returns the size of a ZScript array with the *bool* type.

`int SizeOfArrayFFC(ffc *ptr)`
Returns the size of a ZScript array with the *ffc* type.

`int SizeOfArrayNPC(npc *ptr)`
Returns the size of a ZScript array with the *npc* type.

`int SizeOfArrayItem(item *ptr)`
Returns the size of a ZScript array with the *item* type.

`int SizeOfArrayItemdata(itemdata *ptr)`
Returns the size of a ZScript array with the *itemdata* type.

`int SizeOfArrayEWeapon(eweapon *ptr)`
Returns the size of a ZScript array with the *eweapon* type.

`int SizeOfArrayLWeapon(lweapon *ptr)`
Returns the size of a ZScript array with the *lweapon* type.

Note: In the 2.53 and earlier spec, it was only possible to read the size of a *float* typed array, despite that they are (internally), all the same format.

Game Pointer

`int MapCount()`

Returns the number of maps in use. THis has the format of a function to serve as a reminder that it is read-only.

`int GetScreenEnemy(int n1, int n2, int n)`

Returns the enemy ID for index *n* on screen n2 of dmap *n1.*

`void SetScreenEnemy(int n1, int n2, int n, int v)`

Sets the enemy index *n* on screen *n2* of dmap *n1* to a value of *v. Should that be the present screen blitted to the main bitmap,* the enemy does not change until the screen is refreshed (redrawn). Perhaps an instruction to reload enemies would be useful. This does not persist, and is lost when the quest exits.

`int GetScreenDoor(int map, int screen, int index)`

Returns the type of door used on a given map, screen, and Door[] index thereof.

`void SetScreenDoor(int map, int screen, int index, int type)`

Sets the given Door[] index on a map, and screen to a desired value. This value does not persist, and is lost when the quest exits.

`void GreyscaleOn()`

Changes the entire display to greyscale. The DMap palette is preserved when converting to greyscale.

`void GreyscaleOff()`

Disables greyscale output, restoring colour, using the present DMap Palette. .

`int DMapPalette[512];`

Allows writing a palette ID to any of the DMaps, to change the palette of a given DMap instantly. This value does not persist, and is lost when the quest exits.

`void SetMessage(int message, int string[])`

Copies the contents of string[] into a ZQuest Message String. If the string is too long, it is truncated prior to copying.

`void SetDMapName(int dmap_id, int *buffer)`

Copies the text of *buffer into the DMap Name string for the given dmap_id. If the string is too long, it is truncated.

`void SetDMapTitle(int dmap_id, int *buffer)`

Copies the text of *buffer into the DMap Title string for the given dmap_id. If the string is too long, it is truncated.

`void SetDMapIntro(``int dmap_id, int *buffer`**`)`**
Copies the text of *buffer into the DMap Intro string for the given dmap_id. If the string is too long, it is truncated.

`int GetItemScript(``int *ptr`**`)`**
Returns the numerical ID of an item script, by passing a string to *ptr. On a match for a valid item script, the ID of that script is returned. It returns -1 on a failure.

Audio Pointer (New)

`void PauseSound(int sfx);`

Pauses a sound that is enqueuer by Allegro and is playing.

`void ResumeSound(int sfx);`

Resumes a paused sound effect.

`void EndSound(int sfx);`

Kills a sound effect that is playing.

`void PlaySound(int sfx);`

Clone of Game->PlaySound() to the new pointer.

`void PauseMusic();`

Pauses the present music track; and *may* not affect GME music.

`void ResumeMusic();`

Resumes the present music track; and *may* not affect GME music.

Screen Pointer

`CreateLWeaponDx(int type, int itemid)`

Creates an *lweapon*, using the *itemdata* attributes from a specified *item*. This is an expansion of CreateLWeapon(), which passes -1 as its itemid, and it allows more control over the weapon properties that are automatically forwarded when an item ID is passed to the weapon.

```
void DrawBitmapEx
(int layer, int bitmap_id, int source_x, int source_y, int source_w,
int source_h, int dest_x, int dest_y, int dest_w, int dest_h,
float rotation, int cx, int cy, int mode, int lit,  bool mask);
```

This is a far superior bitmap drawing routine, that allows for a variety of modes, thanks to the allegro 4.4 updates, including translucent bitmaps, pivoting, rotating, flipping, and various combinations thereof. The mode used is based on a flagset: Stacking modes such as MODE_TRANS+MODE_PIVOT will result either in a translucent, pivoted bitmap, or if the combination is illegal, a report of the attempt to use an illegal mode will be reported to allegro.log.

*I have chosen not to support **gouraud** mode, as it has little purpose in 8-bit indexed blits.*

Internally, this is a huge switch-case block, where the modes are validated, and the appropriate allegro function is selected from a list. Many use draw_bitmap_ex(), but that does not support all of the legal modes, so various others are used, such as pivot_sprite().

`void WavyIn(); WavyOut();`

These call the internal warp transition effects of wavyin() and wavyout() on demand.

`void ZapIn(); ZapOut();`

These call the internal warp transition effects of zapin() and zapout() on demand.

`void OpeningWipe();`

This calls the internal function of openingwipe() used by warps, or by some combo collisions, on demand. It uses the wipe format set in quest rules.

These may be candidates for an Effects-> pointer or class.

`void TriggerSecret(``int type`**`)`**

This is an expansion of Screen->TriggerSecrets(), that affects a *specific* secret. To accomplish this, the **new FFScript class** has a revised version of the HiddenEntrance() functions within it, that I tailored for this application.

NPC Pointer

`int InvFrames;`

This returns or sets the number of invincibility frames for the npc. When the npc is invincible (after being hit), this is set to a positive value, that ticks down by one per frame. Writing a positive value to this makes the npc invincible, and modifying it when it is non-zero can shorten, or lengthen the invincibility duration.

This needs to be tested and sanity checked against negative values. Writing a negative value to it is illegal, and should be safeguarded by either converting all writes absolute, or clamping them to a range of 0, 214747.

`int Invincible;`

This should probably be a bool. It returns the 'superman' status of an npc, and may be dropped from the spec.

`bool HasItem;`

When read, this returns if the npc carries the room special item. Writing to this is legal, but it needs to be verified that having multiple enemies with this flag does not cause bugs.

`bool Ringleader;`

This flag denotes if the npc is a Ringleader. If it is, killing it also kills every other npc on the screen. Unlike HasItem, multiple npcs with this flag should not pose a problem. This is read-write.

`int Defense[];`

This has been expanded to cover the new defences. See the *Enemy Editor* for more information.

`float Misc[];`

This has been resized from 16 indices, to 32 indices.

//Other Enemy Editor Specification Variables Go Here

Item Pointer

`float Misc[];`

This has been resized from 16 indices, to 32 indices.

`int AClock;`

This represents the clock for the item animation cycle. It is read-write.

EWeapon Pointer

`float Misc[];`

This has been resized from 16 indices, to 32 indices.

LWeapon Pointer

`int Range;`

This represents the range, or duration of an item, for boomerangs, hookshots, and arrows. It is read-write.

`Float Misc[];`

This has been resized from 16 indices, to 32 indices.

FFC Pointer

`int ID;`

This represents the index on the screen for the FFC in question, effectively returning REFFC, but allowing the user to specify *this->ID* or *ffc->ID* in scripts, simplifying the process of targeting a specific ffc.

Note: The Misc[] for the *ffc* pointer has not been changed, due to RAM concerns, as every ffc in a quest is pre-loaded, and adding 16 longs * 32 instances per screen * 128 screens per map * map count, would cause a dramatic increase in RAM usage.

ItemData Pointer

This pointer has been greatly expanded, completing the work initially planned for 2.50. Writing values to itemdata affects all items of the ID that the user modifies, globally, and instantaneously. Writing to these fields, does how persist between game sessions, and is lost when the player exits the quest.

`int ID;`
This returns the ID of the item to which a script is attached, and effectively returns REFITEM. This capability is critical to making item scripts easier to use, as in 2.53.0 and earlier specs, when you create an item collect script and want to tell the script what item is running it, you needed to manually specify that in the InitD. This allows easily determining what item is attached to a script, and to do other things by item ID.

`int Modifier;`
This represents the Link Tile Modifier value for the item, in the Item Editor, and is read-write.

`int Tile;`
This represents the present tile used by the item, and is read-write. This requires a sanity check to clamp it in the valid range of tiles.

`int CSet;`
This represents the CSet used to draw the item to the screen bitmap, and is read-write. This requires being sanity checked in a valid range of CSets.

`int Flash;`
This represents the CSet used for the item flash animation, if any; and it is read-write. . This requires a sanity check to clamp it to a valid range of CSets. I believe that -1 is used for ‘no flash’, and this requires verification.

`int AFrames;`
This represents the number of frames in the animation cycle used for drawing the item to the screen bitmap, and it is read-write. It needs to be clamped to positive values, starting at 0.

`int ASpeed;`
This represents the clock used to time the item animation, and ist is read-write. It needs to be clamped to positive values, starting at 0.

`int Delay;`

This represents the number of frames before the animation cycle for the item begins. It is read-write, and must be clamped to a range of positive values, starting at 0.

`int Script;`

This represents the *action script* used by the item. It is read-write, and must be clamped between the range of 0 (no script), and the highest-used script slot.

`int PScript;`

This represents the *collect script* used by the item. It is read-write, and must be clamped between the range of 0 (no script), and the highest-used script slot.

`int MagicCost;`

This represents the MP cost of the item, in the *Item Editor*. It is read-write, and must be clamped between 0 and MAX_INT.

`int MinHearts;`

This represents the *Item Editor* field that allows the user to specify a minimum heart requirement to collect an item (e.g. , 5 hearts for SWORD2). It is read-write, and must be clamped between 0 and MAX_INT.

`int Attributes[10];`

This represents the ten misc attributes fields used by the *Item Editor*. Some of these have predefined functions, based on the class of the item, while many item classes—including all 'script' classes- have these fields blank. In ZC 2.50, when an item did not use one of these fields, the Item Editor refused to permit entering a value.

In the 2.54 spec, all *Item Editor* fields are always available, to permit tailoring scripted (global) item classes that the user may define. These fields are therefore, always available now. This array is read-write, and the range is clamped between (-)214747.9999 and 214747.9999.

`int Sprites[10];`

This represents the ten weapon sprites in the *Item Editor*. In ZC 2.50, only weapon classes where sprites were used allowed editing their fields, but in the 2.54 spec, the user may always set these fields, and thus, scripted items may now make use of this data for sprite assignment.

The range of this must be clamped between 0 and MAX_SPRITE.

`bool Flags[5];`

This represents the five flag tickboxes in the *Item Editor*. Under 2.50, only items with a flag that was hardcoded for the user to edit allowed ticking them, but in the 2.54 spec, these are always available, and thus item scripts may make use of them. Many items have a hardcoded effect with these flags. As an example, swords use Flag[3] for their 'Can Slash' attribute. These are read-write.

`bool Combine;`

This represents the *Item Editor* 'Upgrade when collected twice' flag. It is read-write.

`bool Downgrade;`

This represents the *Item Editor* 'Remove when Used' flag. It is read-write.

`bool KeepOld;`

This represents the Item Editor 'Keep Lower Level items' flag. It is read-write.

`bool RupeeCost;`

This represents the *Item Editor* flag 'Use Rupees instead of Magic', and is read-write.

`bool Edible;`

This represents the *Item Editor* 'Can be Eaten by Enemies' flag, and it is read-write.

`bool GainLower;`

This represents the *Item Editor* 'Gain All Lower Level Items' flag, and it is read-write.

Link Pointer

…

Enemy Editor

…

Defences

The enemy editor now includes new defences, and defence results, as follows:

Defence Types: Script 1, Script 2, Script 3, Script 4, Script 5, Script 6, Script 7, Script 8, Script 9, and Script 10.

These correspond to weapons of the same types.

Defence Results:

Double Damage

Triple Damage

Quadruple Damage

Trigger Screen Secrets

Block if < 5

Item Editor

Sprite Editor

Fonts

The 2.54 specification expands the number of internal fonts by a whopping 44 new entries, mainly based on old system fonts that are effectively in the public domain. All of the samples were gathered from sources that are posted under a creative commons license, with no special restrictions on deploying them in a product.

This is also in preparation for allowing the user to add fonts to a quest, which is not part of the 2.54 spec.