RetroScript Handbook v3

By Rubberduckycooly + stxtic

Last updated: Nov 29th 2023

Table of Contents

Introduction to RSDK and RetroScript

About RSDK

The Retro Engine Software Development Kit (Retro-Engine or RSDK) is a primarily 2D game engine with many "old school" graphics effects, including functionality akin to "Mode 7" on an SNES and palette-based graphics. RSDKv3 (previously thought to be RSDKv2[1]), the 3rd version, was only used in the Sonic CD (2011) remaster (with a slight update for the mobile port of which will be addressed later) and was then upgraded to RSDKv4 (previously thought to be RSDKvB[1]) for the Sonic 1 and 2 mobile remasters (and likely the Sonic 3 proof-of-concept), using an updated version of RetroScript with more built-ins. Mania uses RSDKv5, the latest officially used version of RSDK, which uses a transpilable version of RetroScript[2]. Versioning for RSDK has followed the editor's version since v3[3]. RetroScript remains officially unnamed, though it was previously confused with TaxReciept[1].

[1] Christian Whitehead's reply to RDC's tweet: https://twitter.com/CFWhitehead/status/1341701486657433601

[2] CW has stated that v5 scripts get transpiled into C for use in the Game.dll file.

[3] When asked why Nexus and CD was named v3, CW stated that as of v3, the engine versions began to match the editor's.

About RetroScript

RetroScript's syntax is like that of Visual Basic. It does not use semicolons or braces and instead uses line breaks to mark expression endings. Because of it being a scripting language, it offers many benefits compared to a typical language such like C:

- Scripts are recompiled when a stage is loaded/restarted
    - Changes are incredibly easy to make and test almost instantly
- Specifically designed to create object code, making it easy to create objects

However, because of this, there are also many drawbacks which add a challenge to more experienced programmers:

- Custom variables cannot be defined. One must use the temporary built-in variables (discussed later in the handbook.)
- There are no data types other than integers. No decimal places (floats) or strings can be stored, except for passing some string constants to some built-in functions.
- User-defined functions cannot be passed any parameters. All variables are however kept the same, so it is possible to use the built-in variables as a "passing" method.
- You cannot have multiple expressions on one line. For example, `A = B + C` is invalid, but `A = B` then `A += C` is valid (discussed more in the next page).

Arithmetic

Mathematics

As previously mentioned, you cannot have more than 1 arithmetic expression in one line and they all must be done one by one. There can only ever be 1 variable on the right and another on the left. Because of this, the list of mathematical arithmetic operators is limited to the following assignment operators:

- `=` - regular assignment
- 4-function
  - `+=`
  - `-=`
  - `*=`
  - `/=` - division rounds *down* (flooring)
  - `%=` - modulo (used for remainder of division)
- Bit math
  - `<<=` - shift left
  - `>>=` - shift right
  - `&=` - AND
  - `|=` - OR
  - `^=` - XOR
- Unary
  - `++` - used as `Variable++`, equivalent to `Variable += 1`
  - `--` - used as `Variable--`, equivalent to `Variable -= 1`

Examples

Pseudo-code	RetroScript (with custom variables)

`i = 0;`
`j = 15;`
`i++;  //i is 1`
`i = j + 2; //i is 17`
	`i = 0`
	`j = 15`
	`i++  //i is 1`
	`i = j  //i is 15`
	`i += 2 //i is 17`

`x = 19;`
`y = 3;`
`d = 5;`
`x -= --d;  //x subtracted by 4`
`y -= d--;  //y subtracted by 4`
`//d is already 3`
	`x = 19`
	`y = 3`
	`d = 5`
	`d--`
	`x -= d //x subtracted by 4`
	`y -= d //y subtracted by 4`
	`d--  //d is now 3`

`i = 2;`
`i = i + 0.5; //i is 2.5`
	`i = 2`
	`i += 0.5 //oops! compiler error!`
	`//if decimals were allowed,`
	`//i would be 2.5`

Conditionals and Statements

Boolean Logic

Boolean operation is also possible but can only be used in control statements, and thus why they are in this section. There is no such boolean “or” or boolean “and” operator (`||` and && respectively). The list of operators are as follows:

- `==` - equal to (not = on its own)
- `>`
- `>=`
- `<`
- `<=`
- `!=`

There are, however, some functions that you can use to assign variables boolean expressions:

- `CheckEqual(A, B)`
- `CheckLower(A, B)`
- `CheckGreater(A, B)`
- `CheckNotEqual(A, B)`

All these set `CheckResult` to either 0 or 1 based on the result of the function, which can later be checked and ORed/ANDed with.

Control Statements

Since RetroScript does not use braces, there are specific keyword pairs that get used, along with small specifics for each:

- If statements:
  - `if [statement]` - `[statement]` is a single boolean expression as shown above
  - `else`
  - `endif` - use as the “ending brace”
  - **There is no such thing as a direct else-if in RetroScript.** To achieve an else-if, one must make a new if statement on a new line and close it properly.
- While statements:
  - `while [statement]` - `[statement]` is a single boolean expression as shown above
  - `loop` - use as the “ending brace”
- Switch statements:
  - `switch [variable]` - `[variable]` is the variable to check for
  - `case [int/alias]` - `[int/alias]` is an integer or alias to check if the variable is equal to
  - `endswitch` - use as the “ending brace”
  - Switches behave similarly as they do in C: `default` is optional and `break` is used in cases to stop fallthrough.

Examples

Pseudo-code	RetroScript (with custom variables)

`if (i == 0) {`
`    x++;`
`    y++;`
`} else if (i == 1) {`
`    x--;`
`}`
	`if i == 0`
	`    x++`
	`    y++`
	`else`
	`    if i == 1`
	`        x--`
	`    end if`
	`end if`

`while (x < 10) {`
`    x++;`
`    if (y == 5) break;`
`}`
	`while x < 10`
	`    x++`
	`    if y == 5`
	`        break`
	`    end if`
	`loop`

`switch (x) {`
`    case 1:`
`    case 2:`
`        y++;`
`    case 3:`
`        x++;`
`        break;`
`    default:`
`        z++;`
`        break;`
`}`
	`switch x`
	`case 1`
	`case 2`
	`    y++`
	`case 3`
	`    x++`
	`    break`
	`default`
	`    z++`
	`    break`
	`End switch`

Subs and Functions

Subs

Subs are easily thought of as "default functions," and are all called periodically during gameplay. To define events, you use `sub [name]` as the start and `end sub` as the "`closing brace`". The definable subs are as follows:

Sub	Description
`ObjectMain`	Called once every frame per object if priority allows for it [see priority notes]
`ObjectPlayerInteraction`	Called once every frame per object if `Player.ObjectInteraction` is enabled, and priority allows for it [see priority notes]
`ObjectDraw`	Called once every frame per object if priority allows for it [see priority notes]. The ordering is based the value of `Object.DrawOrder`
`ObjectStartup`	Called once per object type when the stage loads. Used for loading assets, initializing variables, etc.
`RSDKEdit`	Called once per object edit by the editor (RetroED v2), sets up the interface for variable editing
`RSDKDraw`	Similar to `ObjectDraw`, though only called by the editor (RetroED v2), called once a frame for each object
`RSDKLoad`	Similar to `ObjectStartup`, though only called by the editor (RetroED v2), used to load any spriteframes or variables for the object needed by the editor

Functions

Users can define functions by using `function [name]` to start a function and `end function` as the "closing brace." Functions can be forward declared using the preprocessor directive `#function [name]`. To call functions, you use the built in function `CallFunction(function)`, which means functions cannot have built in parameters, but there are ways to get around it in the example below. `return` can be used to preemptively end a function.

Examples

Pseudo-code

```
MyFunc(y);

ObjectUpdate() {
    x += 5; //x is 5
    MyFunc(x) //pass x (not it's value)
    //x is 7
}

MyFunc(y) {
    y += 2; //increment x
    return;
    y += 5; //this line doesn't hit
}
```

RetroScript (with custom variables)

```
#function MyFunc

sub ObjectMain
    x += 5
    y = x
    CallFunction(MyFunc)
end sub

function MyFunc
    y += 2
    return
    y += 5 //this line doesn't hit
end function
```

Preprocessor Directives

RetroScript v3 has 2 preprocessor directive that are available to use. These preprocessor directives are as follows:

Directive	Description
`#platform: [type]` `#endplatform`	Skips over lines of code if `type` does not match with what the bytecode is being compiled for. `type` can be: • `Standard` or `Mobile` • `Sw_Rendering` or `Hw_Rendering` • `Use_Haptics` or `No_Haptics` The following `types` are exclusive to the decompilation: • `Use_Standalone` or `Use_Origins` • `Use_Mod_Loader` • `Use_Decomp`
`#alias [val]: [name]`	Creates a new alias that gets replaced by `val` on compíle time

Built-ins

Audio

Function/Variable/Alias	Description
`Music.Volume`	Current volume for music
`Music.CurrentTrack`	Currently playing music track ID
`Music.Position`	Position of currently playing music
`Engine.SFXVolume`	Sound FX Volume (ranges from 0-100)
`Engine.BGMVolume`	BGM master volume (ranges from 0-100), combined with `Music.Volume` to get the final output volume
`SetMusicTrack(string filePath, int trackID, int loopPoint)`	Loads the music file (must be ogg format) from `Data/Music/[filePath]` into the `trackList` slot `trackID`, with a loop point of `loopPoint` (0 = no loop, 1 = loop from start, anything else is the sample to loop from)
`PlayMusic(int trackID)`	Plays the music track loaded into the slot `trackID`
`StopMusic()`	Stops the currently playing music track
`PauseMusic()`	Pauses the currently playing music track
`ResumeMusic()`	Resumes the music track that was paused using `PauseMusic()`

`[Decomp Only]` `SfxName[name]`	Use this to get the ID of an SFX based on it's name. (e.x Jump.wav has a `sfxID` of 0, so using `SfxName[Jump]` would be the same as using 0, if the game can't find a match, it will default to 0.
`PlaySfx(int sfx, bool loop)`	Plays the sfx with index of `sfx` in GameConfig. if `loop` is true, then the sfx will repeat, otherwise it will play once.
`StopSfx(int sfx)`	Stops the sfx with index of `sfx` in GameConfig
`PlayStageSfx(int sfx, bool loop)`	Plays the sfx with index of `sfx` in StageConfig. if `loop` is true, then the sfx will repeat, otherwise it will play once.
`StopStageSfx(int sfx)`	Stops the sfx with index of `sfx` in StageConfig
`SetSfxAttributes(int sfx, bool loop, int pan)`	Sets if the sfx should loop or not (-1 to leave it unchanged) and the panning of `sfx` to `pan` (-100 to 100 for left to right, with 0 being balanced)

Drawing

Function/Variable/Alias	Description
`LoadSpriteSheet(string path)`	Loads a spritesheet from `Data/Sprites/[path]` and sets `object.spriteSheet` to the sheet's ID
`RemoveSpriteSheet(string path)`	Removes a sheet that matches `path` if it exists
`SpriteFrame(int pivotX, int pivot, int width, int height, int sprX, int sprY)`	Creates a spriteframe with the specified values
`EditFrame(int frame, int pivotX, int pivot, int width, int height, int sprX, int sprY)`	Sets spriteframe `frame` to the new values
`DrawSprite(int frame)`	Draws sprite `frame` at the object's X and Y position
`DrawSpriteXY(int frame, int XPos, int YPos)` `DrawSpriteScreenXY(int frame, int XPos, int YPos)`	Draws sprite `frame` to the specified X and Y position If using `DrawSpriteXY`, the position is in world-space (0,0 is top left, 0,0x10000 is 1px to the right on the stage If using `DrawSpriteScreenXY`, the position is in screen-space (0,0 is top left, 0,1 is 1px to the right on the screen)

FX_SCALE FX_ROTATE FX_ROTOZOOM FX_INK FX_TINT (no alias, ID 4) FX_FLIP	IDs to be used for DrawSpriteFX and DrawSpriteScreenFX FX_SCALE allows sprite scaling based on object.scale FX_ROTATE allows sprite rotation based on object.rotation FX_ROTOZOOM allows for sprite scaling & rotation at the same time FX_INK allows for different ink effects based on object.inkEffect FX_TINT will render the sprite on a grayscale if object.inkEffect is INK_ALPHA, otherwise acts like FX_SCALE FX_FLIP allows for sprite flipping, depending on object.direction
DrawSpriteFX(int frame, int FX, int XPos, int YPos) DrawSpriteScreenFX(int frame, int FX, int XPos, int YPos)	Draws sprite frame to X and Y position using the FX mode If using DrawSpriteFX, the position is in world-space (0,0 is top left, 0,0x10000 is 1px to the right on the stage If using DrawSpriteScreenFX, the position is in screen-space (0,0 is top left, 0,1 is 1px to the right on the screen)
DrawTintRect(int XPos, int YPos, int width, int height)	Draws a tint rect with a size of width, height at XPos & YPos relative to screen-space
DrawRect(int XPos, int YPos, int width, int height, int red, int green, int blue, int alpha)	Draws a rectangle at XPos and YPos (screen-space) with a color based on a combination of red, green, blue and alpha
DrawNumbers(int startingFrame, int XPos, int YPos, int value, int digitCnt, int spacing, bool showAllDigits)	Draws values using startingFrame as the starting point at XPos & YPos (screen-space), with spacing pixels between each frame. Will only draw valid digits (or digitCnt digits if number is exceeded) if showAllDigits is 0, otherwise digitCnt digits will be drawn, with extras being 0

`DrawActName(int startingFrame, int XPos, int YPos, int drawMode, bool useCapitalLetter, int spaceWidth, int spacing)`	Draws the stage's act name using 26 frames starting from `startingFrame` (only english letters are supported), at `XPos` & `YPos` (screen-space), using `drawMode` to set word and alignment, with `spacing` pixels between each letter Possible drawMode values: 0 = Word 1 aligned from the right 1 = Word 1 aligned from the left 2 = Word 2 aligned from the left
`LoadAnimation(string filePath)`	Loads an animation from `Data/Animations/[filePath]` and assigns it to the current object type
`DrawPlayerAnimation()` `DrawObjectAnimation()`	Draws the player/object at its X and Y position, based on the loaded animation and `object.frame/object.animation`, this drawing can be flipped based on `object.direction`
`LoadVideo(string filePath)`	Loads a .ogv from `Videos/[filePath]` or a .rsv from `filePath`
`NextVideoFrame()`	Advances the video frame by 1 if an RSV was loaded
`ClearDrawList(int layer)`	Removes all entries in `drawList layer`
`AddDrawListEntityRef(int layer, int objectPos)`	Adds `objectPos` to the `drawList layer`
`GetDrawListEntityRef(var store, int layer, int objectPos)`	Gets the value in `drawList layer` at `objectPos` and stores it in `store`
`SetDrawListEntityRef(int value, int layer, int objectPos)`	Sets the value in `drawList layer` at `objectPos` to the value of `value`

Palettes

Function/Variable/Alias	Description
`LoadPalette(string filePath, int palBankID, int startPalIndex, int startIndex, int endIndex)`	Loads a palette from `Data/Palettes/[filePath]` into `palBank` starting from `startPalIndex`, with a file offset of `startIndex` and reading all colors through to `endIndex`
`RotatePalette(int startIndex, int endIndex, bool rotRight)`	Rotates all colors on the active palette starting from `startIndex` through to endIndex, moving left or right depending on 'rotRight'.
`SetScreenFade(int red, int green, int blue, int alpha)`	Sets the fade out effect based on `red`, `green`, `blue` and `alpha`
`SetActivePalette(int palBankID, int startLine, int endLine)`	Sets the active palette to `palBank` for all screen lines from `startLine` through to `endLine`
`SetPaletteFade(int dstPalID, int red, int green, int blue, int blendAmount, int startIndex, int endIndex)`	Blends the currently active palette with `red`, `green` and `blue` by `blendAmount` amount, starting at palette index `startIndex` and continuing through to `endIndex`
`CopyPalette(int srcPal, int dstPal)`	Copies `srcPal`, to `dstPal`
`ClearScreen(int clrIndex)`	Clears all pixels on screen with the color from `clrIndex` in the active palette

Object

A NOTE ABOUT `index`: appending a + or - to an array value or a constant will offset it + or - from that value or constant from the object's object position. `[index]` is also optional, and not including it will reference the current object.

Function/Variable/Alias	Description

`TempValue0`
`TempValue1`
`...`
`TempValue7`
	Temporary values used to store values during arithmetic or other similar operations

`ArrayPos0` `ArrayPos1`	Variables used for storing indexes to be used with arrays.
`TempObjectPos`	Set when `CreateTempObject()` is called, only for use as `Index`
`CreateTempObject(int objectType, int propertyValue, int XPos, int YPos)`	Creates a temporary object specified by `objectType`, `propertyValue`, `XPos` and `YPos` near the end of the object list and sets `TempObjectPos` to the created object's `slotID`. This should only be used for misc objects like FX and objects that are destroyed quickly
`ResetObjectEntity(int slot, int objectType, int propertyValue, int XPos, int YPos)`	Resets the object at `slot` to the type and position specified by `objectType`, `propertyValue`, `XPos` and `YPos`
`CheckResult`	A value that some functions set as the resulting value. Can be used with all sorts of arithmetic

`TypeName[name]`	Use this to get the ID of an Object based on its name. (e.g. Ring has an `objID` of 8 in Sonic CD, so using `TypeName[Ring]` would be the same as using 8. If a match isn't found, it will default to Blank Object)

`Object[index].Value0`
`Object[index].Value1`
`Object[index].Value2`
`...`
`Object[index].Value7`
	Integer values used for long-term storage. What they are used for varies on an object-by-object basis.

`Object[index].EntityNo`	The object's slot in the object list
`Object[index].Type`	The object's type
`Object[index].PropertyValue`	The object's propertyValue (subtype)
`Object[index].XPos` `Object[index].YPos`	The object's position in world-space (`0x10000 (65536) == 1.0`)
`Object[index].iXPos` `Object[index].iYPos`	The object's position in screen-space, truncated down from `XPos/YPos` (1 == 1)
`Object[index].State`	The object's state. Can be used any way the objects needs
`Object[index].Rotation`	The object's rotation, generally used with `DrawSpriteFX` and `FX_ROTATE` or `FX_ROTOZOOM` (ranges from 0-511)

`Object[index].Scale`	The object's scale, generally used with generally used with `DrawSpriteFX` and `FX_ROTATE` or `FX_ROTOZOOM` Uses a 9-bit bitshifted value, so `0x200 (512) == 1.0`
`FACING_RIGHT` `FACING_LEFT`	Aliases of IDs for `object.direction`

`FLIP_NONE = 0`
`FLIP_X = 1`
`FLIP_Y = 2`
`FLIP_XY = 3`
	Aliases of IDs for `object.direction` (need to be manually implemented)

`Object[index].Direction`	determines the flip of the sprites when drawing

`PRIORITY_BOUNDS = 0`
`PRIORITY_ACTIVE = 1`
`PRIORITY_ALWAYS = 2`
`PRIORITY_XBOUNDS = 3`
`PRIORITY_BOUNDS_DESTROY = 4`
`PRIORITY_INACTIVE = 5`
	Aliases for IDs of `object.priority`. (needs manual addition)
	`PRIORITY_BOUNDS`: Object will update as long as it's within 128 pixels of the screen border left/right and 256 pixels up/down
	`PRIORITY_ACTIVE`: Object will always update, unless paused
	`PRIORITY_ALWAYS`: Object will always update
	`PRIORITY_XBOUNDS`: Object will update as long as it's within 128 pixels of the screen border left/right
	`PRIORITY_BOUNDS_DESTROY`: Like `PRIORITY_XBOUNDS`, but if this check fails the object type will be set to `Blank Object`
	`PRIORITY_INACTIVE`: Object never updates

`Object[index].Priority`	The object's priority value, determines how the engine handles object activity, by default it's set to `PRIORITY_BOUNDS`

INK_NONE = 0 INK_BLEND = 1 INK_ALPHA = 2 INK_ADD = 3 INK_SUB = 4	Aliases of IDs for object.inkEffect, only takes effect when the object uses the FX_INK flag (needs to be manually implemented) INK_NONE will apply no ink effects (default) INK_BLEND will draw the sprite at 50% transparency (this is the same as doing INK_ALPHA with object.alpha at 128, but its faster) INK_ALPHA allows for alpha blending, how transparent it is will be determined by object.alpha INK_ADD allows for additive blending, how transparent it is will be determined by object.alpha INK_SUB allows for subtractive blending, how transparent it is will be determined by object.alpha
Object[index].InkEffect	Determines the blending mode used with DrawSpriteFX & FX_INK
Object[index].Alpha	The object's transparency from 0 to 255.
Object[index].Frame	The object's frame ID
Object[index].Animation	The object's animation ID
Object[index].PrevAnimation	The last animation the object processed in ProcessAnimation()
Object[index].AnimationSpeed	The object's animation processing speed
Object[index].AnimationTimer	The timer used to process the animations

`Object[index].DrawOrder`	The object's drawing layer: is 3 by default. Manages what `drawList` the object is placed in after `ObjectMain`
`Object[index].OutOfBounds`	Read-only value that is true if the object is out of the camera bounds
`Object[index].SpriteSheet`	The spritesheetID of the active object

`C_TOUCH`
`C_BOX`
`C_BOX2`
`C_PLATFORM`
`C_BOX3 [Origins, no alias, ID 4]`
`C_ENEMY [Origins]`
	Collision types for `PlayerObjectCollision`
	`C_TOUCH:` will set `checkResult` to true when collision happens.
	`C_BOX / C_BOX2:` will set `checkResult` to 1 (Floor), 2 (LWall), 3 (RWall) or 4 (Roof) depending which side collided. C_BOX2 doesn't consider velocities between both objects.
	`C_PLATFORM:` will set `checkResult` to true when collision with the top side of the hitbox happens, other sides will not have collision.
	`C_BOX3:` like C_BOX, but stores collision of objects above and below the player, used to help Knuckles climb multiple objects in a line
	`C_ENEMY:` like C_TOUCH, but adds an additional, hardcoded hitbox. Used by Amy during her Hammer Rush / Hammer Jump abilities.

`PlayerObjectCollision(int collisionType, int left, int top, int right, int bottom)`	Checks for a collision with the player using the hitbox values passed. Sets `CheckResult` to a value based on `collisionType` if there was collision, otherwise is set to false

`CSIDE_FLOOR  = 0`
`CSIDE_LWALL  = 1`
`CSIDE_RWALL  = 2`
`CSIDE_ROOF   = 3`
`CSIDE_ENTITY = 4 [Origins]`
	Aliases of IDs for cSide for the functions below (needs to be manually implemented)

`ObjectTileCollision(int cSide, int xOffset, int yOffset, int cPlane)`	Tries to collide with the FG layer based on the position of `iXPos` + `xOffset`, `iYPos` + `yOffset` in collision plane `cPlane`. Sets `CheckResult` to true if there was a collision, false if not. This function is best used to check if a tile is there, not to move along it
`ObjectTileGrip(int cSide, int xOffset, int yOffset, int cPlane)`	Tries to collide with the FG layer based on the position of `iXPos` + `xOffset`, `iYPos` + `yOffset` in collision plane `cPlane`. Sets `CheckResult` to true if there was a collision, false if not. This function is better used to handle moving along surfaces

`CMODE_FLOOR = 0`
`CMODE_LWALL = 1`
`CMODE_ROOF  = 2`
`CMODE_RWALL = 3`
	Aliases of IDs for CollisionMode, not to be confused with `CSIDE` (needs to be manually implemented)

`[Decomp Only]` `PlayerName[name]`	Use this to get the ID of a Player based on it's name. (e.g. "SONIC" has an `plrID` of 0 in Sonic CD, so using `PlayerName[SONIC]` would be the same as using 0

Player

Function/Variable/Alias	Description

`Player.Value0`
`Player.Value1`
`...`
`Player.Value15`
	Same as `Object.Value`

`Player.EntityNo`	The player's slot in the object list
`Player.XPos` `Player.YPos`	The player's position in world-space (`0x10000 (65536) == 1.0`) Note: this is a separate variable from `Object.X/YPos`
`Player.iXPos` `Player.iYPos`	The player's position in world-space (`1 == 1`) Note: this is a separate variable from `Object.iX/YPos`
`Player.ScreenXPos` `Player.ScreenYPos`	The player's position in screen-space, set via camera following functions
`Player.XVelocity` `Player.YVelocity`	The player's speed on the X & Y axis (`0x10000 (65536) == 1.0`)
`Player.Speed`	The player's general speed (`0x10000 (65536) == 1.0`)

`Player.TopSpeed`
`Player.Acceleration`
`Player.Deceleration`
`Player.GravityStrength`
`Player.JumpStrength`
`Player.JumpCap`
`Player.RollingAcceleration`
`Player.RollingDeceleration`
	Integer values intended for storage of physics attributes

Player.Rotation Player.Scale Player.Priority Player.DrawOrder Player.Direction Player.InkEffect Player.Alpha Player.Frame Player.Animation Player.PrevAnimation Player.AnimationSpeed Player.AnimationTimer Player.OutOfBounds	Same as objects
Player.CollisionMode	Player's active collision mode
Player.CollisionPlane	Determines if the player will be on plane A or plane B
Player.ControlMode	Player control mode (0 for normal)
Player.ControlLock	Player control lock timer
Player.Pushing	Object pushing flag usually set via collision functions
Player.Visible	Determines if the player is visible or not
Player.TileCollisions	Determines if the player will interact with tiles or not
Player.ObjectInteraction	Determines if the player will interact with other objects or not
Player.Gravity	The player's gravity state. True if gravity is being applied (falling)

`Player.Flailing`	Used to determine when to trigger ledge animations
`Player.Pushing`	Used to determine when to trigger pushing animations
`Player.Timer`	Used for frame counting
`Player.Angle`	Set by collision functions, determines the player's angle
`Player.FollowPlayer1` `Player.Water`	Sonic Nexus leftover variables

`Player.Up`
`Player.Down`
`Player.Left`
`Player.Right`
`Player.JumpPress`
`Player.JumpHold`
	Object input buffer values, generally set via `ProcessPlayerControl()`

`Player.LookPos`	Determines how much the camera will vertically offset if `Player.TrackScroll` is enabled
`Player.TrackScroll`	Determines if the camera can use Player.LookPos to shift position
`Player.CollisionPlane`	Determines if the player is on plane A or plane B

`Player.CollisionLeft`
`Player.CollisionTop`
`Player.CollisionRight`
`Player.CollisionBottom`
	The object's active hitbox values based on the loaded animation and Player`.Animation`/Player`.Frame` values

Stages

Function/Variable/Alias	Description
`LoadStage()`	Loads a stage based on `Stage.ListPos` & `Stage.ActiveList`
`Stage.ListPos`	The stage index in the active stage list
`Stage.ActiveList`	The active stage list to load stages from
`Stage.ListSize[index]`	The number of stages that are in stage list `index`

`PRESENTATION_STAGE = [P]`
`REGULAR_STAGE = [R]`
`BONUS_STAGE = [B]`
`SPECIAL_STAGE = [S]`
	IDs for the 4 stage category lists that can be used to store stages in RSDKv4
	The letters between brackets are used to select `category` in `StageName` below

`[Decomp Only]` `StageName[Category - Name]`	Use this to get the ID of a stage based on its category and name on the GameConfig. (e.g. "R - PALMTREE PANIC ZONE 1 A" has a `stgID` of 0 in Sonic CD, so using `StageName[R - PALMTREE PANIC ZONE 1 A]` is the same as picking stage ID 0 from the REGULAR_STAGE category

`Stage.Minutes`
`Stage.Seconds`
`Stage.MilliSeconds`
	The timer values for the current stage.
	These are automatically set for you as long as `stage.timeEnabled` is true

`Stage.TimeEnabled`	Determines if the timer should increase or not
`Stage.PauseEnabled`	Determines if the game can perform a 'Genesis' type of pause or not

`Stage.ActNo`	The stage's current act ID

`Stage.XBoundary1`
`Stage.XBoundary2`
`Stage.YBoundary1`
`Stage.YBoundary2`
	The stage's main camera boundaries, the camera will not go beyond these

`Stage.NewXBoundary1`
`Stage.NewXBoundary2`
`Stage.NewYBoundary1`
`Stage.NewYBoundary2`
	The stage's other camera boundaries, the camera will not go beyond these, however these are used when setting new camera boundaries

`Stage.DeformationData0[index]`
`Stage.DeformationData1[index]`
`Stage.DeformationData2[index]`
`Stage.DeformationData3[index]`
	The layer deformation data arrays.
	0 & 1 are used for the FG Layer (0 being for above water, 1 being for below water), while 2 & 3 are used for BG Layers (2 being for above water, 3 being for below water)

`SetLayerDeformation(int deformID, int deformA, int deformB, int type, int offset, int count)`	Sets the deformation of the deformation data array of `deformID` based on the deform values
`Stage.ActiveLayer[index]`	Drawable layer IDs, with index 0 being the lowest and index 3 being the highest. This is initially set during stage load
`Stage.Midpoint`	Any active layers above this value will draw only tiles on the high Visual Plane, otherwise only tiles on the low Visual Plane will draw
`Stage.WaterLevel`	The height of the water relative to 0 in the stage layout

`STAGE_RUNNING = 1` `STAGE_PAUSED  = 2`	Stage state IDs STAGE_RUNNING: object update and draw events will run if they're in bound or its priority is set to PRIORITY_ACTIVE/PRIORITY_ALWAYS STAGE_PAUSED: the object update and draw events are paused, unless its priority is set to PRIORITY_ALWAYS
`Stage.State`	The stage's current activity state
`Stage.PlayerListPos`	The current player ID, based on the GameConfig's player list
`Stage.DebugMode`	Determines if debug mode is active or not
`GetTileLayerEntry(var store, int layer, int chunkX, int chunkY)`	Gets the chunkID of the chunk at `chunkX`, `chunkY` on tileLayer `layer` and stores it in `store`
`SetTileLayerEntry(int value, int layer, int chunkX, int chunkY)`	Sets the chunkID of the chunk at `chunkX`, `chunkY` on tileLayer `layer` and sets the index to `value`

`TILEINFO_INDEX       = 0`
`TILEINFO_DIRECTION   = 1`
`TILEINFO_VISUALPLANE = 2`
`TILEINFO_SOLIDITYA   = 3`
`TILEINFO_SOLIDITYB   = 4`
`TILEINFO_FLAGSA      = 5`
`TILEINFO_ANGLEA      = 6`
`TILEINFO_FLAGSB      = 7`
`TILEINFO_ANGLEB      = 8`
	Aliases of IDs for infoType for `Get/Set16x16TileInfo`
	(requires manual implementation)
	`TILEINFO_FLAGSB` & `TILEINFO_ANGLEB` can only be used with `Get16x16TileInfo()` as they are read-only

`Get16x16TileInfo(int store, int tileX, int tileY, int infoType)`	Gets `infoType` of the tile at `tileX`, `tileY` and stores it in `store`
`Set16x16TileInfo(int value, int tileX, int tileY, int infoType)`	Sets `infoType` of the tile at `tileX`, `tileY` based on `value`
`Copy16x16Tile(int dst, int src)`	Copies the tileset image data of `src` into `dst`, used for animated tiles

`TileLayer[index].XSize` `TileLayer[index].YSize`	The width/height of the `TileLayer` in chunks

`LAYER_NOSCROLL = 0`
`LAYER_HSCROLL = 1`
`LAYER_VSCROLL = 2`
`LAYER_3DFLOOR = 3`
`LAYER_3DSKY = 4`
	Aliases of IDs for `TileLayer.Type` (needs manual addition)
	`LAYER_NOSCROLL`: No scroll
	`LAYER_HSCROLL`: Horizontal scroll based on camera
	`LAYER_VSCROLL`: Vertical scroll based on camera
	`LAYER_3DFLOOR`: The layer will render the lower half of the screen on "Mode 7"
	`LAYER_3DSKY`: The layer will render on "Mode 7", according to TileLayer size

`TileLayer[index].Type`	The type of rendering that the `TileLayer` uses
`TileLayer[index].Angle`	The angle of the TileLayer (used for 3DFloor & 3DSky rotations)

`TileLayer[index].XPos`
`TileLayer[index].YPos`
`TileLayer[index].ZPos`
	The position of the tileLayer (used for 3DFloor & 3DSky rotations)

`TileLayer[index].ParallaxFactor`
`TileLayer[index].ScrollSpeed`
`TileLayer[index].ScrollPos`
	The parallax values of the tileLayer (see parallax below for more info)

`TileLayer[index].DeformationOffset` `TileLayer[index].DeformationOffsetW`	The offset for the deformation data arrays when rendering (0,1 for FG & 2,3 for BG)
`HParallax[index].ParallaxFactor` `VParallax[index].ParallaxFactor`	The scroll info's parallax factor (relative speed), which determines how many pixels the parallax moves per pixel move of the camera
`HParallax[index].ScrollSpeed` `VParallax[index].ScrollSpeed`	The scroll info's scroll speed (constant speed), which determines how many pixels the parallax moves per frame
`HParallax[index].ScrollPos` `VParallax[index].ScrollPos`	The scroll info's scroll position, which is how many pixels the parallax is offset from the starting pos

Input

Function/Variable/Alias	Description

`KeyDown[index].Up`
`KeyDown[index].Down`
`KeyDown[index].Left`
`KeyDown[index].Right`
`KeyDown[index].ButtonA`
`KeyDown[index].ButtonB`
`KeyDown[index].ButtonC`
`KeyDown[index].Start`
	True if the corresponding button/key of `index` has been held.
	`Index` represents the controller expected for input, values 1 through 4 being a specific controller, while 0 is any controller

`KeyPress[index].Up`
`KeyPress[index].Down`
`KeyPress[index].Left`
`KeyPress[index].Right`
`KeyPress[index].ButtonA`
`KeyPress[index].ButtonB`
`KeyPress[index].ButtonC`
`KeyPress[index].Start`
	True if the corresponding button/key was pressed on this frame.
	Same note as above.

`CheckTouchRect(int x1, int y1, int x2, int y2)`	Checks if a touch input was detected between the inputted coordinates (screen-space) Returns a `checkResult` value based on input, if none, returns `-1`

Math

Function/Variable/Alias	Description
`Sin(int store, int angle)` `Cos(int store, int angle)`	Gets the value from the sin/cos512 lookup table based on `angle` and sets it in `store`
`Sin256(int store, int angle)` `Cos256(int store, int angle)`	Gets the value from the sin/cos256 lookup table based on `angle` and sets it in `store`
`ATan2(int store, int x, int y)`	Performs an arctan operation using `x` and `y` and stores the result in `store`
`GetBit(var store, int value, int pos)`	Gets bit at index `pos` from `value` and stores it in `store`
`SetBit(int value, int pos, int set)`	Sets bit at index `pos` to `set` and updates `value` accordingly
`Rand(var store, int max)`	Gets a random value from 0 to `max` (exclusive maximum) and stores it in `store`
`Not(var value)`	Performs a NOT operation on `value` (`value = ~value`)
`Interpolate(var store, int x, int y, int percent)` `InterpolateXY(var storeX, var storeY, int aX, int aY, int bX, int bY, int percent)`	Linearly interpolates (LERPs) x and y by `percent` and stores the result in `store`. `percent` is 0 through 256. `InterpolateXY` does 2 at once for points (`aX, aY`) and (`bX, bY`)

3D

Function/Variable/Alias	Description

`MAT_WORLD`
`MAT_VIEW`
`MAT_TEMP`
	RSDKv3 only allow use of 3 matrices: world, view & temp.
	Passing these should only be done to parameters of type `mat`.
	RSDK matrix values are shifted 8 bits, so `0x100` (starting vals) is 1.0

`3DScene.NoVertices` `3DScene.NoFaces`	Amount of active faces/vertices in each buffer respectively (max of 1024 faces and 4096 vertices)
`3DScene.ProjectionX` `3DScene.ProjectionY`	The width (X) and height (Y) of the 3DScene draw buffer. These values determine what base resolution to use for drawing functions. By default these values are 136(X) and 160(Y)

`FaceBuffer[index].a`
`FaceBuffer[index].b`
`FaceBuffer[index].c`
`FaceBuffer[index].d`
	The vertex indices to use to control this face's drawing

`FACE_FLAG_TEXTURED_3D = 0`
`FACE_FLAG_TEXTURED_2D = 1`
`FACE_FLAG_COLOURED_3D = 2`
`FACE_FLAG_COLOURED_2D = 3`
	Aliases of IDs for `FaceBuffer.Flag` (need to be manually added)
	FACE_FLAG_TEXTURED_3D: render face on 3D Space with a texture
	FACE_FLAG_TEXTURED_2D: render face on the object's `xpos` and `ypos` with a texture
	FACE_FLAG_COLOURED_3D: render face on 3D Space with a solid color based on `FaceBuffer.Color`
	FACE_FLAG_COLOURED_2D: render face on the object's `xpos` and `ypos` with a solid color based on `FaceBuffer.Color`

`FaceBuffer[index].Flag`	The active drawing flag for this face
`FaceBuffer[index].Color`	The colour to draw the face when drawing with `FACE_FLAG_COLOURED_2D` or `FACE_FLAG_COLOURED_3D` flags

`VertexBuffer[index].x`
`VertexBuffer[index].y`
`VertexBuffer[index].z`
`VertexBuffer[index].u`
`VertexBuffer[index].v`
	The vertex coordinates for the specified vertex

`SetIdentityMatrix(mat matrix)`	Sets the matrix of `matID` to the identity state
`MatrixMultiply(mat matrixA, mat matrixB)`	Multiplies `matrixA` by `matrixB` and stores the result in `matrixA`
`MatrixTranslateXYZ(mat matrix, int x, int y, int z)`	Translates `matrix` to x, y, z, all shifted 8 bits (0x100 = 1.0)
`MatrixScaleXYZ(int matrix, int x, int y, int z)`	Scales `matrix` by x, y, z, all shifted 8 bits (0x100 = 1.0)

`MatrixRotateX(mat matrix, int angle)`
`MatrixRotateY(mat matrix, int angle)`
`MatrixRotateZ(mat matrix, int angle)`
`MatrixRotateXYZ(mat matrix, int x, int y, int z)`
	Rotates `matrix` to `angle` on the specified axis, or all if using `MatrixRotateXYZ`.
	Angles are 512-based, like sin/cos

`TransformVertices(mat matrix, int startIndex, int endIndex)`	Transforms all vertices from `startIndex` to `endIndex` using `matrix`
`Draw3DScene()`	Draws the active 3DScene data to the screen

Menus

Function/Variable/Alias	Description
`MENU_1` `MENU_2`	Menu IDs for `menu` parameters
`Menu1.Selection` `Menu2.Selection`	the current row selected by MENU_1/MENU_2
`LoadTextFont(string filePath)`	Loads a bitmap font from `filePath` for use with textMenus
`LoadTextFile(menu, string path)`	Loads a `menu` based on the file loaded from `path`
`SetupMenu(menu, int rowCount, int selectionCount, int alignment)`	Sets up `menu` with `rowCount` rows, `selectionCount` active selections and aligning to `alignment`
`AddMenuEntry(menu, string text, int highlightEntry)` `EditMenuEntry(int menu, string text, int rowID, int highlightEntry)`	Adds or edits an entry to `menu` with the contents of `text`, and highlighted if `highlightEntry` is set to true

`TEXTINFO_TEXTDATA = 0`
`TEXTINFO_TEXTSIZE = 1`
`TEXTINFO_ROWCOUNT = 2`
	Aliases of types of data that can be fetched via `GetTextInfo()`. (needs manual addition)

`GetTextInfo(var store, menu, int type, int index, int offset)`	Gets the data of `type` from `menu` using `index`, using `offset` if the type is `TEXTINFO_TEXTDATA`
`DrawMenu(menu, int XPos, int YPos)`	Draws `menu` to `XPos` & `YPos` relative to the screen
`DrawText(menu, int XPos, int YPos, int scale, int spacing, int rowStart, int rowCount)`	Draws the contents of `rowCount` rows starting from `rowStart` in `menu` to `XPos` & `YPos` relative to the screen, using `spacing` pixels between them and using `scale` scaling
`GetVersionNumber(menu, int highlight)`	Adds a text entry with the game's version as the text, highlighted if highlight is set

Engine

Function/Variable/Alias	Description

`ENGINE_DEVMENU = 0`
`ENGINE_MAINGAME = 1`
`ENGINE_INITDEVMENU = 2 *`
`ENGINE_EXITGAME = 3`
`ENGINE_SCRIPTERROR = 4`
`ENGINE_ENTER_HIRESMODE = 5`
`ENGINE_EXIT_HIRESMODE = 6`
`ENGINE_PAUSE = 7`
`ENGINE_WAIT = 8`
`ENGINE_VIDEOWAIT = 9`
`* RESET_GAME will call this state`
	Aliases of IDs for `Engine.State` (needs manual implementation)
	ENGINE_DEVMENU: the game is currently on Dev Menu
	ENGINE_MAINGAME: normal game loop
	ENGINE_INITDEVMENU: the game will be forced to load Dev Menu
	ENGINE_EXITGAME: The game will close
	ENGINE_SCRIPTERROR: game stopped due to script compiler error
	ENGINE_ENTER_HIRESMODE: Turns on Hi-Res mode
	ENGINE_EXIT_HIRESMODE: Turns off Hi-Res mode
	ENGINE_WAIT: Pauses the game
	ENGINE_PAUSE: Same as ENGINE_WAIT
	ENGINE_VIDEOWAIT: Pauses the game until the video ends

`Engine.State`	The current engine game loop state
`Engine.Language`	The language the engine is actively using
`Engine.HapticsEnabled`	Determines whether `HapticEffect()` will play haptics or not
`Engine.TrialMode`	Whether or not the game is built as a “trial version” (basically always false)

`RETRO_WIN`
`RETRO_OSX`
`RETRO_XBOX_360`
`RETRO_PS3`
`RETRO_IOS`
`RETRO_ANDROID`
`RETRO_WP7`
	Names for the values of `Engine.PlatformID`

`Engine.PlatformID`	The current platform ID the game is currently running on
`EngineCallback(int callback)`	Sends callback to the engine
`saveRAM[index]`	an array of data capable of being written/read from file via `ReadSaveRAM()`/`WriteSaveRAM()`
`ReadSaveRAM()`	reads the contents of the save file on disk into SaveRAM (overwrites any existing values)
`WriteSaveRAM()`	writes the contents of SaveRAM to the save file on disk
`Engine.OnlineActive`	Whether or not online functionality is enabled for the engine
`ONLINEMENU_ACHIEVEMENTS = 0` `ONLINEMENU_LEADERBOARDS = 1`	Aliases for IDs for `LoadOnlineMenu` (needs manual implementation)
`LoadOnlineMenu(int menuID)`	Loads the data for the specified online menu (does nothing on decomp)
`SetAchievements(int id, int status)` `SetLeaderboards(int id, int entry)`	Sets the status/entry of the achievements/leaderboards with index of ID to `status/entry`
`[Decomp Only]` `AchievementName[name]`	Use this to get the ID of an Achievement based on its name. (e.g. "Take the High Road" has an `achID` of 3 in Sonic CD, so using `AchievementName[Take the High Road]` would be the same as using 3

Further Assistance

For any further questions relating to RetroScript or RSDK modding in general, join the Retro Engine Modding Server: your one stop for all RSDK modding!