RetroScript Handbook v4

By Rubberduckycooly + stxtic

Last updated: Jan 18th, 2024

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.
- Foreach statements:
  - `foreach (TypeName[objectName], store, type)`
    - iterates every object of type `TypeName[objectName]` and sets `store` to the object’s slotID.
    - `type` is either `ALL_ENTITIES` or `ACTIVE_ENTITIES`
  - `next` - use as the “ending brace”

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`

Pseudo-code	RetroScript (with custom variables)

`foreach (arrayPos2 in TypeName[Ring]) {`
`  object[arrayPos2].value0 = 1;`
`}`
 	
`foreach (arrayPos2 in TypeName[Player Object]) {`
`  object[arrayPos2].xpos = object.xpos;`
`  object[arrayPos2].ypos = object.ypos;`
`}`
	`foreach (TypeName[Ring], arrayPos2, ALL_ENTITIES)`
	`  object[arrayPos2].value0 = 1`
	`next`
	
	`foreach (TypeName[Player Object], arrayPos2, ACTIVE_ENTITIES)`
	`  object[arrayPos2].xpos = object.xpos`
	`  object[arrayPos2].ypos = object.ypos`
	`next`

Events and Functions

Events

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

Event	Description
`ObjectUpdate`	Called once every frame per object if 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 `private / public function [name]` to start a function and `end function` as the "closing brace." Functions can be forward declared using the preprocessor directive `reserve 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	RetroScript (with custom variables)

`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`
`}`
	`reserve function MyFunc`
	
	`event ObjectUpdate`
	`  x += 5`
	`  y = x`
	`  CallFunction(MyFunc)`
	`end event`
	
	`private function MyFunc`
	`  y += 2`
	`  return`
	`  y += 5 //this line doesn't hit`
	`end function`

Preprocessor Directives

RetroScript v4 has 1 preprocessor directive that is available to use. This 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_F_FEEDBACK` or `NO_F_FEEDBACK` The following `types` are exclusive to the decompilation: • `USE_STANDALONE` or `USE_ORIGINS` • `USE_NETWORKING` • `USE_MOD_LOADER` • `USE_DECOMP`

Variables

RetroScript v4 has 3 formats for extra variables that are available to use. These use the keywords `public` and `private`. `public` means this variable can be accessed by any script compiled after the current one, while `private` means the variable can only be accessed by the script it was created in. the formats for the variables are as follows:

Directive	Description
`[public]/[private] alias [val] : [name]`	Creates a new alias that gets replaced by `val` on compile time. Example: private alias 1 : myAlias
`[public]/[private] value [name] = [val]`	Creates a new static variable with the value of `val`. example: public value myValue = 0 static variables are not tied to an object and thus should not be used when a value is needed for every instance of an object. they are regular values that can be accessed the same as any other built-in one Note: Tables can't have their values defined with aliases, as it's not supported by the compiler

`[public]/[private] table [name] [values] end table`	Creates a new table, fills the table with any values it reads until it hits the `end table` keyword. Values should be separated by `,` character, unless there is a newline. E.g.: public table colourTable 0x600020, 0xC00040, 0xE04080 0x802040, 0xE04060, 0xE060A0 end table Tables can also be created with a set number of blank spaces for later storage. E.g.: private table positionTable[8] Tables are closer to functions than variables, as values from them are accessed via `GetTableValue` and can be set via `SetTableValue.` Like functions, their ids can be assigned to other variables for "pointer-like" functionality. E.g.: GetTableValue(temp0, temp1, myTable) temp0 += 0x10 SetTableValue(temp0, temp1, myTable) E.g. 2: switch temp1 case 0 temp0 = myTable1 break case 1 temp0 = myTable2 break end switch GetTableValue(temp2, 2, temp0) Note: Tables can't have their values defined with aliases, as it's not supported by the compiler

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()`

`SwapMusicTrack(string filePath, int trackID, int loopPoint, int ratio)`	Works similar to `SetMusicTrack()` & `PlayMusic()` but starts at a position based on `ratio`. `ratio` is using an 10000-based value, so 10000 = 1.0 music speed, 5000 = 0.5, etc. Commonly used with speed shoes.
`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 + StageConfig. 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 + 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
`DrawObjectAnimation()`	Draws the 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`
`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 palBankID, int startIndex, int endIndex, bool rotRight)`	Rotates all colors in `palBank` 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`

`SetPaletteEntry(int palBankID, int index, int color)`	Sets the palette entry in `palBankID` at `index` to the value of `color`
`GetPaletteEntry(int palBankID, int index, var palStore)`	Gets the palette entry from `palBankID` at `index` and stores it in `palStore`
`SetPaletteFade(int dstPalID, int palA, int palB, int blendAmount, int startIndex, int endIndex)`	Blends `srcPalA` with `srcPalB` by `blendAmount` amount, starting at palette index `startIndex` and continuing through to `endIndex` and stores the resulting colors in `dstPalID`
`CopyPalette(int srcPal, int srcPalStart, int dstPal, int dstPalStart, int count)`	Copies `count` colors from `srcPal`, starting at `srcPalStart`, to `dstPal`, starting at `dstPalStart`
`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

`temp0`
`temp1`
`temp2`
`...`
`temp7`
	Temporary values used to store values during arithmetic or other similar operations

`arrayPos0`
`arrayPos1`
`...`
`arrayPos5`
`arrayPos6`
`arrayPos7`
	Variables used for storing indexes to be used with arrays.

`tempObjectPos`	Set when `CreateTempObject()` is called, can only be used as an `arrayPos`
`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 it's name. (e.g. Ring has an `objID` of 10 in Sonic 1, so using `TypeName[Ring]` would be the same as using 10. If a match isn't found, it will default to Blank Object)

`object[index].value0`
`object[index].value1`
`object[index].value2`
`...`
`object[index].value47`
	Integer values used for long-term storage. What they are used for varies on an object-by-object basis.

`object[index].entityPos`	The object's slot in the object list
`object[index].groupID`	The object's typeGroup. By default, it matches its type, but can be set to another one (0x100, 0x101 & 0x102 are never assigned by default so they're good for using for custom groups)
`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].xvel` `object[index].yvel`	The object's speed on the X & Y axis (world-space)
`object[index].speed`	The object's general speed (world-space)
`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`
`object[index].drawOrder`	The object's drawing layer: is 3 by default. Manages what `drawList` the object is placed in after `ObjectUpdate`
`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 PRIORITY_BOUNDS_SMALL = 6 PRIORITY_ACTIVE_SMALL = 7	Aliases for IDs of `object.priority`. (needs to be manually implemented) 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 (or frozen) 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: object will update as long as it's within 128 pixels of the screen border left/right, if this check fails the object type will be set to `Blank Object` PRIORITY_INACTIVE: never updates PRIORITY_BOUNDS_SMALL: object will update as long as it's within 32 pixels of the screen border left/right and 128 pixels up/down PRIORITY_ACTIVE_SMALL: object will always update, unless paused (or frozen), the object is considered out of bounds if it's within 32 pixels of the screen border left/right and 128 pixels up/down
`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].lookPosX` `object[index].lookPosY`	The camera offset from the player's position.
`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
`ProcessObjectControl()`	Handles control inputs
`ProcessObjectMovement()`	Handles all of object tile collisions (used almost only for player)

`C_TOUCH`
`C_SOLID`
`C_SOLID2`
`C_PLATFORM`
	Collision types for `BoxCollisionTest`
	C_TOUCH: will set `checkResult` to true when collision happens
	C_SOLID / C_SOLID2: will set `checkResult` to 1 (Floor), 2 (LWall), 3 (RWall) or 4 (Roof) depending which side collided. C_SOLID2 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

`BoxCollisionTest(int collisionType, int thisObject, int thisLeft, int thisTop, int thisRight, int thisBottom, int otherObject, int otherLeft, int otherTop, int otherRight, int otherBottom)`	Checks for a collision between `thisObject` and `otherObject` using the hitbox values passed. Values can be set to `C_BOX` and they will instead be loaded from the object's active hitbox. 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_LENTITY = 4 [Origins] CSIDE_RENTITY = 5 [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
object[index].angle	Object's tile angle. Usually set via ProcessObjectMovement()
object[index].collisionPlane	Object collision plane (only 0 or 1)
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)
object[index].collisionMode	Object's active collision mode
object[index].controlMode	Object control mode (0 for normal)

`object[index].controlLock`	Object control lock timer
`object[index].pushing`	Object pushing flag usually set via collision functions
`object[index].visible`	Determines if the object is visible or not
`object[index].tileCollisions`	Determines if the object will interact with tiles or not
`object[index].interactions`	Determines if the object will interact with other objects or not
`object[index].gravity`	The object's gravity state. True if gravity is being applied (falling)

`object[index].up`
`object[index].down`
`object[index].left`
`object[index].right`
`object[index].jumpPress`
`object[index].jumpHold`
	Object input buffer values, generally set via `ProcessPlayerControl()`

`object[index].scrollTracking`	Determines if the camera will track the object's position or just follow it

`object[index].floorSensorL`
`object[index].floorSensorC`
`object[index].floorSensorR`
`object[index].floorSensorLC`
`object[index].floorSensorRC`
	Collision sensor result values when on floor.
	True if there was no collision, false if there was

`object[index].collisionLeft`
`object[index].collisionTop`
`object[index].collisionRight`
`object[index].collisionBottom`
	The object's active hitbox values based on the loaded animation and `object.animation`/`object.frame` values

`GetObjectValue(var store, int valueIndex, int entitySlot)`	Get `object.value[valueIndex]` of `entitySlot` and stores it in `store`
`SetObjectValue(int value, int valueIndex, int entitySlot)`	Set `object.value[valueIndex]` of `entitySlot` to the value of value
`SetObjectRange(int range)`	Sets the update ranges for all objects, `range` is how wide the "screen" is. the default values are the same as `SetObjectRange(424)`
`CopyObject(int destSlot, int srcSlot, int count)`	Copies `count` objects from `srcSlot` to `destSlot` Note: the size of the object list is 1184 entities, however there is another 1184 slots beyond that to be used for storage Example: `CopyObject(1184, 0, 2)` copies the objects in slot 0 & slot 1 into slot 1184 & 1185 respectively
`[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 1, so using `PlayerName[SONIC]` would be the same as using 0

Camera

Function/Variable/Alias	Description
`camera[screenID].xpos` `camera[screenID].ypos`	The camera's position in screen-space
`camera[screenID].target`	The object that will be tracked by the camera
`camera[screenID].style`	Determines how the camera will display
`camera[screenID].enabled`	Determines if the camera will keep track of it's target
`camera[screenID].adjustY`	Offsets vertically the camera

`CAMERASTYLE_FOLLOW = 0`
`CAMERASTYLE_EXTENDED = 1`
`CAMERASTYLE_EXTENDED_OFFSET_L = 2`
`CAMERASTYLE_EXTENDED_OFFSET_R = 3`
`CAMERASTYLE_HLOCKED = 4`
`CAMERASTYLE_FIXED = 5` [Origins*]
`CAMERASTYLE_STATIC = 6` [Origins]
 	
*Behavior present on the engine in v4, but inaccessible
	Aliases of IDs for camera.style (needs to be manually implemented)
	`CAMERASTYLE_FOLLOW` the camera will follow the target when possible
	`CAMERASTYLE_EXTENDED` keeps the camera ahead of the target as long as minimal speed is maintained
	`CAMERASTYLE_EXTENDED_OFFSET_L` shifts the camera to the left of the target
	`CAMERASTYLE_EXTENDED_OFFSET_R` shifts the camera to the right of the target
	`CAMERASTYLE_HLOCKED` keeps the camera locked horizontally
	`CAMERASTYLE_FIXED` the camera will always center to the target when possible
	`CAMERASTYLE_STATIC` the camera will stop moving

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 - GREEN HILL ZONE 1" has a `stgID` of 0 in Sonic 1, so using `StageName[R - GREEN HILL ZONE 1]` would be 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.actNum`	The stage's current act ID

`stage.curXBoundary1`
`stage.curXBoundary2`
`stage.curYBoundary1`
`stage.curYBoundary2`
	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_FROZEN = 3 STAGE_2P_MODE = 4	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_FROZEN: the object update event is paused, unless its priority is set to PRIORITY_ALWAYS STAGE_2P_MODE: similar to STAGE_RUNNING, but considers the active bound areas of 2 objects (Players)
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
stage.entityPos	The current slotID of the object being run
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 TILEINFO_DIRECTION TILEINFO_VISUALPLANE TILEINFO_SOLIDITYA TILEINFO_SOLIDITYB TILEINFO_FLAGSA TILEINFO_ANGLEA TILEINFO_FLAGSB TILEINFO_ANGLEB	Aliases of IDs for infoType for Get/Set16x16TileInfo (needs 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 the info of infoType of the tile at tileX, tileY and stores it in store
Set16x16TileInfo(int value, int tileX, int tileY, int infoType)	Sets the info of infoType of the tile at tileX, tileY and sets it based on value
Copy16x16Tile(int dst, int src)	Copies the tileset image data of src into dst, used for animated tiles
CheckCurrentStageFolder(string folderName)	Reads the name at the end of the current stage's folder and compares it with folderName, sets checkResult to true if there's a match
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 similarly to Mode-7 LAYER_3DSKY: The layer will render similarly to 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].buttonX`
`keyDown[index].buttonY`
`keyDown[index].buttonZ`
`keyDown[index].buttonL`
`keyDown[index].buttonR`
`keyDown[index].start`
`keyDown[index].select`
	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].buttonX`
`keyPress[index].buttonY`
`keyPress[index].buttonZ`
`keyPress[index].buttonL`
`keyPress[index].buttonR`
`keyPress[index].start`
`keyPress[index].select`
	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`)
`Abs(var value)`	Gets the absolute number of `value` and updates `value` with it
`GetTableValue(var store, int index, table)`	Gets a value from `table` at `index` and stores it in `store`
`SetTableValue(int value, int index, table)`	Sets the value in `table` at `index` to `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`
	RSDKv4 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

`scene3D.vertexCount` `scene3D.faceCount`	Amount of active faces/vertices in each buffer respectively (max of 1024 faces and 4096 vertices)
`scene3D.projectionX` `scene3D.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)
`scene3D.fogColor` `scene3D.fogStrength`	The colour of the fog in RGB format and the strength of the fog (0-255). Used with FADE_FADED flag

`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 FACE_FLAG_FADED = 4 FACE_FLAG_TEXTURED_C = 5 FADE_FLAG_TEXTURED_C_BLEND = 6 FACE_FLAG_3DSPRITE = 7	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` FACE_FLAG_FADED: render face on 3D Space with a solid color based on `faceBuffer.color` and with opacity based on `scene3D.fogStrength` and depth FACE_FLAG_TEXTURED_C: render face on 3D Space with a texture, using `vertexBuffer.u` and `vertexBuffer.v` of `faceBuffer.a` to determine the center of the sprite, and `vertexBuffer.u` and `vertexBuffer.v` of `faceBuffer.c` to determine how far it is from the X and Y edges of the sprite FACE_FLAG_TEXTURED_C_BLEND: similar to FACE_FLAG_TEXTURED_C, but the texture is blended FACE_FLAG_3DSPRITE: render face on 3D Space like `DrawSpriteScreenFX` using `SpriteFrames`, using `vertexBuffer.v` of `faceBuffer.a` to determine the `inkEffect` and `vertexBuffer.u` to select the sprite frame
`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

`MatrixInverse(int matrix)`	Performs an inversion on the values of `matrix`
`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
`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
`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_WAIT = 3`
`ENGINE_SCRIPTERROR = 4`
`ENGINE_INITPAUSE = 5`
`ENGINE_EXITPAUSE = 6`
`ENGINE_ENDGAME = 7`
`ENGINE_RESETGAME = 8`
`* 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_WAIT: The game is waiting for the engine to resume the game loop, normally used while on HW Menus
	ENGINE_SCRIPTERROR: Game stopped due to script compíler error
	ENGINE_INITPAUSE: Pause the game and trigger the HW Pause Menu
	ENGINE_EXITPAUSE: Resume the game and exit the HW Pause Menu
	ENGINE_ENDGAME: The game will close
	ENGINE_RESETGAME: Resets the game

`engine.state`	The current engine game loop state
`engine.language`	The language the engine is actively using
`engine.onlineActive`	Whether or not online functionality is enabled for the engine
`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)
`STANDARD` `MOBILE`	Names for the values of `engine.deviceType`
`engine.deviceType`	The current device type the game is currently running on

`CallNativeFunction(int functionID)`
 	
`CallNativeFunction2(int functionID, int param1, int param2)`
 	
`CallNativeFunction4(int functionID, int param1, int param2, int param3, int param4)`
	Calls the native engine function with the ID of `callbackFuncID` using no params, 2 params, or 4 params respectively. Valid function names are shown below. Adding a global variable with a name that matches any of the valid function names will result in its value being set to the function id internally. e.g: A global variable with the name "SetAchievement" will have its value set to the function ID of the "SetAchievement" function

`Print(message, bool isInt, bool addNewLine)`	Prints a message to the console & the log, if `isInt` is set then `message` will be treated as an int, otherwise it will be treated as a string. If `addNewLine` is set a `\n` character will be added to the end of the message when printed.
`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
`[Decomp Only] AchievementName[name]`	Use this to get the ID of an Achievement based on its name. (e.g. "Ring King" has an `achID` of 4 in Sonic 1, so using `AchievementName[Ring King]` would be the same as using 4

Native Functions

Documentation on valid values for CallNativeFunction/2/4. Any parameters named “unused” should always be set to 0.

Function/Variable/Alias	Description
`SetAchievement(int id, int status)`	Sets an achievement’s status to `Status`. Status 100 is achieved, anything else is unachieved. Use AchievementName[] to get achievement IDs.
`SetLeaderboard(int leaderboard, int score)`	Sets the leaderboard `Leaderboard`‘s score to `Score` if it’s lower than the stored score. Valid leaderboard IDs range from 0-127.
`Connect2PVS(int gameLength, int itemMode)`	Initializes a 2P VS session. `gameLength` and `itemMode` is how many matches and what type of item boxes to use respectively
`Disconnect2PVS()`	Ends the currently active 2P VS session. `SendEntity`(int entityID, int unused): sends the entity in slot `entityID` to the other player in the VS session.
`SendValue(int value, int unused)`	Sends `Value` to the other player in the VS session
`ReceiveEntity(int entityID, bool incrementSlot)`	Receives (and loads) the next entity in the stack from the other player. If `incrementSlot` is true then the entity is removed from the stack.

`ReceiveValue(int value, bool incrementSlot)`	Receives the next value in the stack from the other player. If `incrementSlot` is true then the value is removed from the stack.
`TransmitGlobal(int value, string name)`	Sends a global value to the other player in the VS session. In most cases `Value` should be the global variable and `name` should be the name of the variable. The other player does not have to manually receive this value as it will be set automatically based on the name.
`ShowPromoPopup(int id, string promoName)`	Attempts to display a promotional popup. **Note:** This function does nothing on the decompilation.
`NotifyCallback(int callbackID, int val1)` `NotifyCallback(int callbackID, int val1, int int val2, int val3)`	Sends a callback of type `callbackID`, along with data `val1-val3`. `Note:` behavior of this function on the decompílation is limited to only what strictly affects gameplay, other functions only serve a purpose in Sonic Origins.
`HapticEffect(int id, int unknown1, int unknown2, int unknown3)`	Plays a haptic effect on the controller/mobile device. The parameters of this function are unknown and therefore this does nothing on the decompilation. `[Requires haptics. This can be checked with #Platform: USE_F_FEEDBACK]`
`SetNetworkGameName(int unused, string name)`	Sets the game name of the network to `name`. `[Requires networking. This can be checked with #Platform: USE_NETWORKING]`

	[The following native functions requires the mod loader. This can be checked with #Platform: USE_MOD_LOADER]
ExitGame()	Exits the game.
FileExist(int unused, string filePath)	Checks if a file exist in filePath, if it does, sets checkResult to true.
OpenModMenu()	Opens the devmenu's mod menu.
AddAchievement(int unused, string name)	Adds a new achievement with the name name.
SetAchievementDescription(int id, string description)	Sets the description of the achievement with id, to description. Use AchievementName[] to get achievement IDs.
ClearAchievements()	Clears all loaded achievements
GetAchievementCount()	Gets the amount of loaded achievements and stores the value in checkResult.
GetAchievement(int id, int unused)	Gets the status of achievement id and stores the value in checkResult . Use AchievementName[] to get achievement IDs.
GetAchievementName(int id, int textMenu)	Gets the name of the achievement id and adds it as a new row to textMenu. Use AchievementName[] to get achievement IDs.
GetAchievementDescription(int id, int textMenu)	Gets the description of the achievement id and adds it as a new row to textMenu. Use AchievementName[] to get achievement IDs.

`GetScreenWidth()`	Gets the current internal screen width of the game, and updates `checkResult` to its value.
`SetScreenWidth(int width, int unused)`	Allows `ApplyWindowChanges` to change the internal screen width of the game, based on `width`.
`GetWindowScale()`	Gets the window scale multiplier of the game, and updates `checkResult` to its value.
`SetWindowScale(int scale, int unused)`	Allows `ApplyWindowChanges` to change the scale of the window size based on `scale`. any value below 1 is invalid and will break the window.
`GetWindowScaleMode()`	Gets the current scale mode of the game, and updates `checkResult` to its value.
`SetWindowScaleMode(int mode, int unused)`	Allows `ApplyWindowChanges` to set the scale mode of the game based in `mode`. any value greater than 1 is invalid.
`GetWindowFullScreen()`	Checks if the game is on fullscreen, sets `checkResult` to 1 if true.
`SetWindowFullScreen(bool fullscreen, int unused)`	Allows `ApplyWindowChanges` to turn the game into fullscreen or windowed, depending on the value of `fullscreen`.
`GetWindowBorderless()`	Checks if the game window is borderless, sets `checkResult` to 1 if true.
`SetWindowBorderless(bool borderless, int unused)`	Allows `ApplyWindowChanges` to set the window to borderless, or not, depending on the value of `borderless`.
`GetWindowVSync()`	Checks if VSync is enabled, sets `checkResult` to 1 if true.

`SetWindowVSync(bool vSync, int unused)`	Allows ApplyWindowChanges to turn on or off VSync, depending on the value of `vSync`.
`ApplyWindowChanges()`	Updates the game's width, scale, scale mode, vsync, fullscreen and borderless states based on the received values from the native functions above.
`GetModCount()`	Gets the amount of loaded mods and stores the value in `checkResult`.
`GetModName(int textMenu, bool highlight, int id, int unused)`	Gets the name of the mod `id` and adds it as a new row to `textMenu`. If `highlight` is set, the row will be highlighted.
`GetModDescription(int textMenu, bool highlight, int id, int unused)`	Gets the description of the mod `id` and adds it as a new row to `textMenu`. If `highlight` is set, the row will be highlighted.
`GetModAuthor(int textMenu, bool highlight, int id, int unused)`	Gets the author of the mod `id` and adds it as a new row to `textMenu`. If `highlight` is set, the row will be highlighted.
`GetModVersion(int textMenu, bool highlight, int id, int unused)`	Gets the version of the mod `id` and adds it as a new row to `textMenu`. If `highlight` is set, the row will be highlighted.
`GetModActive(int id, int unused)`	Gets the active flag of mod `id` and stores it in `checkResult`.
`SetModActive(int id, bool active)`	Sets the active flag of mod `id` to `active`.
`MoveMod(int id, int up)`	Moves position of mod `id` on the priority list up or down depending on `up`.
`RefreshEngine()`	Reloads the engine. Must be called for any mod active related changes to take effect

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!