RetroScript Handbook

By Rubberduckycooly + RMGRich

Last updated: Jan 18th 2021

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. Because of this, v4-only keywords are marked as such and v3-only keywords are marked as such. 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)`
- `CheckNotEquals(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 (v4 ONLY):
  - `foreach TypeName[objectName], store`
    - iterates every object of type `TypeName[objectName]` and sets `store` to the object’s slotID
  - `foreach TypeGroup[index], store`
    - iterates every object in `TypeGroup[index]` and sets `store` to the object’s slotID
  - `floop` - 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--`
	`    endif`
	`endif`

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

Pseudo-code	RetroScript (with custom variables)

`foreach (ArrayPos2 in TypeName[Ring]) {`
`  Object[ArrayPos2].Value0 = 1;`
`}`
 	
`foreach (ArrayPos2 in TypeGroup[256]) {`
`  Object[ArrayPos2].XPos = Object.XPos;`
`  Object[ArrayPos2].YPos = Object.YPos;`
`}`
	`foreach TypeName[Ring], ArrayPos2`
	`  Object[ArrayPos2].Value0 = 1`
	`floop`
	
	`foreach TypeGroup[256], ArrayPos2`
	`  Object[ArrayPos2].XPos = Object.XPos`
	`  Object[ArrayPos2].YPos = Object.YPos`
	`floop`

Subs and Functions

Subs

Subs are easily thought of as “default functions,” and are all called periodically during gameplay. To define subs, you use `sub [name]` as the start and `endsub` 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.ObjectInteractions 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 and once when the stage loads. Used for loading assets and spriteFrames

Functions

Users can define functions by using `function [name]` to start a function and `endfunction` 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.

Examples

Pseudo-code

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

MyFunc(y) {
    y += 2; //increment x
}
```

RetroScript (with custom variables)

```
#function MyFunc

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

function MyFunc
    y += 2
endfunction
```

Preprocessor Directives

RetroScript has 4 preprocessor directives (in v4, 2 in v3) 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_Renderer` or `HW_Renderer` • `Use_Haptics` or `No_Haptics`
`#alias [val]:[name]`	Creates a new alias that gets replaced by `val` on compile time.
`#static [val]:[name]`	Creates a new variable with the starting value of `val`. This can be referenced and modified like any other variable. These can be referenced in other scripts, which means they cannot conflict with any script (2 statics can't be named the same thing).
`#array [name]:(val0, val1, ...)`	Creates a new array with the values in the list. `#table` can also be used.

Built-ins

Audio

Function/Variable/Alias	Description
`Music.Volume`	Current master volume for music
`Music.CurrentTrack`	Currently playing music track ID
`Engine.BGMVolume`	Sound FX Volume (ranges from 0-100)
`Engine.SFXVolume`	BGM 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 (has to 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 8000-based value, so 8000 = 1.0 music speed, 4000 = 0.5, etc. Used more commonly with speed shoes.
`SfxName[name]`	Use this to get the ID of an SFX based on it's name. (e.x Jump.wav has an `sfxID` of 0, so using `SfxName[Jump]` would be the same as using 0
`PlaySfx(int sfx, int loopCnt)`	v3: Plays the sfx with index of `sfx` in the gameconfig `loopCnt` times v4: Plays the sfx with index of `sfx` in gameconfig + stageconfig `loopCnt` times (recommended to use `SfxName[]`)
`StopSfx(int sfx)`	v3: Stops the sfx with index of `sfx` in the gameconfig v4: Stops the sfx with index of `sfx` in gameconfig + stageconfig (recommended to use `SfxName[]`)
`PlayStageSfx(int sfx, int loopCnt)`	Plays the sfx with index of `sfx` in the stageconfig `loopCnt` times
`StopStageSfx(int sfx)`	Stops the sfx with index of `sfx` in the stageconfig

`SetSfxAttributes(int sfx, int loopCnt, int pan)`	Sets the amount of times for `sfx` to loop to `loopCnt` (-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
`EditSpriteFrame(int frame, int pivotX, int pivot, int width, int height, int sprX, int sprY)` `(v3 mobile & v4 only)`	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 `DrawSpriteScreenXY`, the position is in screen-space (`0, 0` is top left, `0, 1` is 1 px to the right, etc), otherwise the position is in world-space (`0, 0` is top left, but `0, 0x10000` is 1px to the right)

`FX_SCALE`
`FX_ROTATE`
`FX_ROTOZOOM`
`FX_INK`
`FX_FLIP`
	IDs to be used for `DrawSpriteFX` and `DrawSpriteScreenFX` below.
	For `FX_INK`, see the IDs near

`DrawSpriteFX(int frame, int fx, int XPos, int YPos)`
 	
`DrawSpriteScreenFX(int frame, int fx, int XPos, int YPos)`
	Draws sprite `frame` to the specified X and Y position using the specified FX mode
	If using `DrawSpriteScreenFX`, the position is in screen-space (`0, 0` is top left, `0, 1` is 1 px to the right, etc), otherwise the position is in world-space (`0, 0` is top left, but `0, 0x10000` is 1px to the right)

`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

`DrawNumbers(int startingFrame, int XPos, int YPos, int value, int digitCnt, int spacing, int 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 align, int unknown, int unknown2, int spacing)`	Draws the loaded stage's act name using 26 frames starting from startingFrame (only uppercase english letters are supported), at `XPos` & `YPos` (screen-space), using alignment to determine where the text center is (0 = left, 1 = middle, 2 = right), with spacing pixels between each letter
`DrawRect(int XPos, int YPos, int width, int height, int R, int G, int B, int A)`	Draws a rect with a size of `width`, `height` at `XPos` & `YPos` (screen-space), with a colour of R, G, B and with an alpha of A
`LoadAnimation(string filePath)`	Loads an animation from `Data/Animations/[filePath]` for the object to use
`DrawPlayerAnimation()`	Draws the player at its X and Y position, based on the loaded animation and `Object.Frame`/`Object.Animation`
`DrawObjectAnimation()`	Draws the object at its X and Y position, based on the loaded animation and `Object.Frame`/`Object.Animation`

`LoadVideo(string path)`	If the filename contains ".rsv": an RSV file is loaded from path, otherwise a video with the name of `path` is loaded from `Videos/`
`NextVideoFrame()`	Advances the video frame if an RSV is 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 palID, int startPalIndex, int startIndex, int endIndex)`	Loads a palette from `Data/Palettes/[filePath]` into `palID` starting from `startPalIndex`, with a file offset of `startIndex` and reading all colors through to `endIndex`
`RotatePalette(int palID, int startIndex, int endIndex, int right)`	Rotates all colours in `palID` starting from `startIndex` through to `endIndex` left one index if `right` is 0, else rotates one right
`SetScreenFade(int r, int g, int b, in a)`	Sets the fade out effect based on `r`, `g`, `b` and `a`
`SetActivePalette(int palID, int startLine, int endLine)`	Sets the active palette to `palID` for all lines from `startLine` through to `endLine`
`SetPaletteEntry(int palID, int index, int colour)`	Sets the palette entry in `palID` at `index` to the value of `colour`

`GetPaletteEntry(int palID, int index, int store)`	Gets the palette entry from `palID` at `index` and stores it in `store`

`SetPaletteFade(int dstPal, int r, int g, int b, int blendAmount, int startIndex, int endIndex)`
 	
`SetPaletteFade(int dstPal, int srcPalA, int srcPalB, int blendAmount, int startIndex, int endIndex)`
	v3: Blends the currently active palette with `r`, `g`, and `b`, by `blendAmount` amount, and stores it in `dstPal`, starting at palette index `startIndex` and continuing through to `endIndex`
	
	v4: Blends `srcPalA` with `srcPalB` by `blendAmount` amount, starting at palette index `startIndex` and continuing through to `endIndex` and stores the resulting colours in `dstPal`

`CopyPalette(int srcPal, int dstPal)`
 	
`CopyPalette(int srcPal, int srcPalStart, int dstPal, int dstPalStart, int count)`
	v3: Copies `srcPal` into `dstPal`
	
	v4: Copies `count` colours from `srcPal`, starting at `srcPalStart`, to `dstPal`, starting at `dstPalStart`

`ClearScreen(int clrIndex)`	Clears all pixels on screen with the colour from `clrIndex` in the active palette

Object

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.

Function/Variable/Alias	Description

`TempValue0`
`TempValue1`
`TempValue2`
`...`
`TempValue7`
	Temporary values used to store values during arithmetic or other similar operations

`ArrayPos0`
`ArrayPos1`
 	
`ArrayPos2`
`ArrayPos3`
`ArrayPos4`
`ArrayPos5`
`PlayerObjectPos`
`PlayerObjectCount`
	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 type, int`	Creates a temporary object specified by `type`, `propertyValue`, XPos and YPos near the end of the object list and sets `TempObjectPos` to the created object's `slotID`.

`propertyValue, int XPos, int YPos)`	This should only be used for misc objects like FX and objects that are destroyed quickly
`ResetObjectEntity(int slot, int type, int propertyValue, int XPos, int YPos)`	Resets the object at `slot` to the type and position specified by `type`, `propertyValue`, `XPos` and `YPos`
`CheckResult`	A value that some functions set as the resulting value. Can be used with all sorts of arithmetic
`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` (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`

`PRIORITY_ACTIVE_BOUNDS`
`PRIORITY_ACTIVE`
`PRIORITY_ACTIVE_PAUSED`
`PRIORITY_XBOUNDS`
`PRIORITY_XBOUNDS_DESTROY`
`PRIORITY_INACTIVE`
`PRIORITY_BOUNDS_SMALL`
`PRIORITY_UNKNOWN`
	IDs for Object.Priority. Can be used in v3 but must be 0 through 5 integer values
	`PRIORITY_ACTIVE_BOUNDS`: object will update as long as it's within 0x80 pixels of the screen border left/right and 0x100 pixels up/down
	`PRIORITY_ACTIVE`: object will always update, unless paused (or frozen)
	`PRIORITY_ACTIVE_PAUSED`: object will always update, even if paused (or frozen)
	`PRIORITY_XBOUNDS`: same as PRIORITY_ACTIVE_BOUNDS however there's no Y check so as long as it's within XBounds it'll update
	`PRIORITY_XBOUNDS_DESTROY`: same as PRIORITY_ACTIVE_XBOUNDS, except if the check fails the object's type will be set to blank object
	`PRIORITY_INACTIVE`: never updates or draws
	
	v3 cannot use the following, even with numeric values:
	`PRIORITY_BOUNDS_SMALL`: object will update as long as it's within 0x20 pixels of the screen border left/right and 0x80 pixels up/down
	`PRIORITY_UNKNOWN`: object will always update, unless paused (or frozen), not entirely sure the difference between this and PRIORITY_ACTIVE

`Object[index].Priority`	The object's priority value, determines how the engine handles object activity, by default it's set to `PRIORITY_ACTIVE_BOUNDS` v3 has to use 0 through 5 instead of the aliases (`PRIORITY_BOUNDS_SMALL` and `PRIORITY_UNKNOWN` are not valid)
`Object[index].DrawOrder`	The object's drawing layer: is 3 by default. Manages what `drawList` the object is placed in after `ObjectMain`

`FACING_LEFT`
`FACING_RIGHT`
 	
`FLIP_NONE`
`FLIP_X`
`FLIP_Y`
`FLIP_XY`
	IDs for Object.Direction. `FACING_LEFT` is the same as `FLIP_X`, and `FACING_RIGHT` is the same as `FLIP_NONE`

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

`INK_NONE`
`INK_BLEND`
`INK_ALPHA`
`INK_ADD`
`INK_SUB`
	IDs for `Object.InkEffect`

`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 was processing during `ProcessAnimation()`
`Object[index].AnimationSpeed`	The object's animation processing speed
`Object[index].AnimationTimer`	The timer used to process the animations
`Object.OutOfBounds`	Read-only value that is true if the object is out of the camera bounds
`Object.SpriteSheet`	The spritesheetID of the active object

`Object[index].Value0`
`Object[index].Value1`
`Object[index].Value2`
`..`
`Object[index].Value7`
`(cont.)`
 	
`Object[index].Value8`
`Object[index].Value9`
`...`
`Object[index].Value47`
	Integer values used for long-term storage. What they are used for varies on an object-by-object basis.

`ProcessPlayerControl()`	Handles control inputs

`PlayerTileCollision()`	Handles all of object tile collisions (used almost only for player)

`C_TOUCH`
`C_BOX`
`C_BOX2 (see note)`
`C_PLATFORM`
	IDs for collision type below
	C_BOX2 is v3 **mobile** & v4 only

`PlayerObjectCollision( int type, int left, int top, int right, int bottom)`
 	
`PlayerObjectCollision( int type, int thisObject, int thisLeft, int thisTop, int thisRight, int thisBottom, int otherObject, int otherLeft, int otherTop, int otherRight, int otherBottom)`
	v3: Checks for a collision with the player using the hitbox values passed. Sets `CheckResult` to 0 if there wasn't a collision, otherwise it's set to 1 (floor), 2 (LWall), 3 (RWall) or 4 (Roof)
	
	v4: Checks for a collision between `thisObject` and `otherObject` using the hitbox values passed. Values can be set to 0x10000 and they will instead be loaded from the object's active hitbox. Sets `CheckResult` to 0 if there wasn't a collision, otherwise it's set to 1 (floor), 2 (LWall), 3 (RWall) or 4 (Roof)

`CSIDE_FLOOR`
`CSIDE_LWALL`
`CSIDE_RWALL`
`CSIDE_ROOF`
	IDs for cSide for the functions below

`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`. Sets `CheckResult` to true if there was a collision, false if there wasn't. 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`. Sets `CheckResult` to true if there was a collision, false if there wasn't. This function is better used to handle moving along surfaces
(everything below here until the end of the category is v4 only)	**(everything below here until the end of the category is v4 only)**
`Object[index].TypeGroup`	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].XVelocity` `Object[index].YVelocity`	The object's speed on the X & Y axis (world-space)
`Object[index].Speed`	The object's general speed (world-space)
`Object[index].Angle`	Object's tile angle. Usually set via `ProcessPlayerTileCollisions()`

`Object[index].CamOffsetX` `Object[index].LookPos`	The camera offset from the player's position. `Object.LookPos` controls Y Offset, while `Object.CamOffsetX` controls X Offset
`Object[index].CollisionPlane`	Object collision plane (only 0 or 1)

`CMODE_FLOOR`
`CMODE_LWALL`
`CMODE_ROOF`
`CMODE_RWALL`
	IDs for CollisionMode, not to be confused with `CSIDE`

`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 of the object is visible or not
`Object[index].TileCollisions`	Determines if the object will interact with tiles or not
`Object[index].` `ObjectInteractions`	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.TrackScroll`	Determines if the camera will track the object's position or just follow it

`Object[index].Flailing0`
`Object[index].Flailing1`
`Object[index].Flailing2`
`Object[index].Flailing3`
`Object[index].Flailing4`
	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 index, int objectPos)`	Gets the `Object[objectPos].Value` at `index` and stores it in `store`
`SetObjectValue(int value, int index, int objectPos)`	Sets the `Object[objectPos].Value` at `index` to the value of `value`
`CopyObject(int startIndex, int copy offset, int count)`	Copies `count` objects from `startIndex` to `offset`

Player (v3 only)

Everything listed in the object variables can be used here by replacing `Object[index]` with `Player`, including *most* v4 ones.

Function/Variable/Alias	Description
`Player.EntityNo`	The object's slot in the object list
`Player.ScreenXPos` `Player.ScreenYPos`	The player's position in screen-space, set via camera following functions

`Player.Value0`
`Player.Value1`
`Player.Value2`
`...`
`Player.Value15`
	Values used for long-term storage, values can be used for anything, what they are used for varies on an player-by-player basis

`Player.LookPos`	The camera Y offset from the player's position
`Player.Flailing[index]`	Collision sensor result values when on floor. True if there was no collision, false if there was one
`Player.Skidding`	Skidding timer
`Player.FollowPlayer1`	Flag that determines if the player will follow "Player1" (never used)

`Player.Water`	Flag that determines if the player is under water (never used)
`Player.Timer`	General purpose timer for the player object

`Player.TopSpeed`
`Player.Acceleration`
`Player.Deceleration`
`Player.AirAcceleration`
`Player.AirDeceleration`
`Player.GravityStrength`
`Player.JumpStrength`
`Player.JumpCap`
`Player.RollingAcceleration`
`Player.RollingDeceleration`
	Physics values that control various aspects of player movement

`BindPlayerToObject(int playerID, int objectID)`	Binds the player object playerID to an object slot objectID. This is almost certainly a requirement as without this the game will likely crash or at the very least not function well at all

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 amount of stages that are in stage list `index`

`PRESENTATION_STAGE`
`REGULAR_STAGE`
`BONUS_STAGE`
`SPECIAL_STAGE`
	IDs for the 4 stage lists that can be used to store stages in RSDK v3 & v4

`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 of the timer should increase or not
`Stage.PauseEnabled`	Determines whether or not the player is allowed to pause the game
`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. Any layers that are not set with this array or are set to 9 will not be drawn.
`Stage.Midpoint`	Any active layers above this value will draw only tiles on the high Visual Plane, otherwise they will only draw tiles on the low Visual Plane
`Stage.WaterLevel`	The height of the water relative to 0 in the stage layout

`STAGE_RUNNING = 1` `STAGE_PAUSED = 2`	Stage state IDs
`Stage.State`	The stage's current activity state
`Stage.PlayerListPos`	The current player ID, based on the gameconfig's player list
`Stage.ActivePlayer`	The current slotID of the player object being run
`Stage.DebugMode`	Determines if debugMode is active or not
`Stage.ObjectEntityPos`	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`*	IDs for infoType for `Get/Set16x16TileInfo` v3 uses 0 through 8 `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
`CheckStageFolder(string folder)`	If the loaded stage's folder matches `folder`, `CheckResult` is set to true, else it is set to false
`TileLayer[index].XSize` `TileLayer[index].YSize`	The width/height of the `tileLayer` in chunks

`TILELAYER_NOSCROLL`
`TILELAYER_HSCROLL`
`TILELAYER_VSCROLL`
`TILELAYER_3DFLOOR`
`TILELAYER_3DSKY`
	IDs for `TileLayer.Type`

`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.Up`
`KeyDown.Down`
`KeyDown.Left`
`KeyDown.Right`
`KeyDown.ButtonA`
`KeyDown.ButtonB`
`KeyDown.ButtonC`
`KeyDown.Start`
	True if the corresponding button/key has been held

`KeyPress.Up`
`KeyPress.Down`
`KeyPress.Left`
`KeyPress.Right`
`KeyPress.ButtonA`
`KeyPress.ButtonB`
`KeyPress.ButtonC`
`KeyPress.Start`
`KeyPress.AnyStart`
	True if the corresponding button/key was pressed on this frame.
	For `KeyPress.AnyStart`, true if any key was held down **AND the game is on the title screen**

`Menu1.Selection` `Menu2.Selection`	the current row selected by MENU_1/MENU_2
`CheckTouchRect(int x1, int y1, int x2, int y2)`	Checks if a touch input was detected between the inputted coordinates (based on screen)

`HapticEffect(int hapticID, int unknown1, int unknown2, int unknown3)`	Plays the haptic effect `hapticID` if haptics are enabled **HapticEffect exists in v4, but has no `hapticID` and remains disabled so it is not listed.**

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 and stores it in `store`
`Not(var value)`	Performs a NOT operation on `value` and updates it (`value = ~value`)
`Absolute(var value)`	Gets the absolute number of `value` and updates `value` with the new number

	RMG's note: this is v4 only?? why didn't CW add it in v3????
`GetArrayValue(var store, int index, arr array)`	Gets a value from `array` at `index` and stores it in `store`
`SetArrayValue(int value, int index, arr array)`	Sets the value in `array` 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`
	RSDK v3 & v4 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.
`3DScene.FogColor` `3DScene.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_TEXTURED_3D`
`FACE_TEXTURED_2D`
`FACE_COLOURED_3D`
`FACE_COLOURED_2D`
`FACE_FADED`
`FACE_TEXTURED_C`
`FADE_TEXTURED_D`
`FACE_SPRITE3D`
	The different face drawing flags that can be used with `FaceBuffer.Flag`.
	Only FACE_TEXTURED_3D, FACE_TEXTURED_2D, FACE_COLOURED_3D and FACE_COLOURED_2D are available in v3 using 0 through 3.

`FaceBuffer[index].Flag`	The active drawing flag for this face
`FaceBuffer[index].Color`	The colour to draw the face when drawing with `FACE_COLOURED_2D` or `FACE_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`
`TranslateMatrixXYZ(mat matrix, int x, int y, int z)`	Translates `matrix` to x, y, z, all shifted 8 bits (0x100 = 1.0)
`ScaleMatrixXYZ(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, similar to 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
`LoadTextFont(string filePath)`	Loads a bitmap font from `filePath` for use with textMenus
`MENU_1` `MENU_2`	Menu IDs for `menu` parameters
`LoadTextFile(int menu, string filePath)`	Loads a `menu` based on the file loaded from `filePath`
`SetupMenu(int menu, int rowCount, int selectionCount, int alignment)`	Sets up `menu` with `rowCount` rows, `selectionCount` active selections and aligning to `alignment`

`AddTextMenuEntry(int menu, string text, int highlightEntry)`
 	
`EditTextMenuEntry(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` *`TEXTINFO_TEXTSIZE`* *`TEXTINFO_ROWCOUNT`*	Types of data that can be fetched via `GetTextInfo()`. Can be accessed in v3 via 0, 1 & 2 respectively

`GetTextInfo(var store, int 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(int menu, int XPos, int YPos)`	Draws `menu` to `XPos` & `YPos` relative to the screen
`DrawText(int 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(int 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.State`	The current engine game loop state, can be set to 2 or `RESET_GAME` to restart the game
`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 or not `HapticEffect()` will play haptics
`Engine.FrameSkipTimer`	The timer for the frame skip feature
`Engine.FrameSkipSetting`	The setting for the frame skip feature
`Engine.TrialMode`	Whether or not the game is built as a “trial version” (basically always false, never used in v4)

`RETRO_WIN`
`RETRO_OSX`
`RETRO_XBOX_360`
`RETRO_PS3`
`RETRO_iOS`
`RETRO_ANDROID`
`RETRO_WP7`
 	
`RETRO_STANDARD`
`RETRO_MOBILE`
	Names for the values of `Engine.PlatformID`

`Engine.PlatformID`	The current platform id the game is currently running on

`EngineCallback(int callback)`
 	
`EngineCallbackFunc(int callbackFuncID)`
 	
`EngineCallback(int callbackFuncID, int param1, int param2)`
 	
`EngineCallbackExt(int callbackFuncID, int param1, int param2, int param3, int param4)`
	v3: Sends `callback` to the engine
	
	v4: Calls the native engine function with the id of `callbackFuncID` using no params, 2 params, or 4 params

`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
`ONLINEMENU_ACHIEVEMENTS` *`ONLINEMENU_LEADERBOARDS`*	Online menus that can be loaded via LoadOnlineMenu. Can be used in v3 via 0 and 1 respectively
`LoadOnlineMenu(int menuID)`	Loads the data for the specified online menu
`SetAchievement(int id, int status)` `SetLeaderboards(int id, int entry)`	Sets the status/entry of the achievement/leaderboard with index of `id` to `status/entry` (these were moved to an EngineCallback() call in v4)

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!