Oxygen Engine

Handbook

Contents

Hotkeys

Window Mode

- Alt + Enter = Toggle fullscreen / window mode
- Shift + Alt + Enter = Switch between different full-integer multiples of game resolution

Rendering

- Alt + F = Previous filtering method
- Alt + G = Next filtering method
- Alt + H = Next frame sync method
- Alt + B = Change background blur setting
- Alt + 1 … Alt + 8 = Toggle rendering of layer:
  - Alt + 1 = Plane B, no priority
  - Alt + 2 = Plane A, no priority
  - Alt + 3 = VDP sprites, no priority
  - Alt + 4 = Custom sprites, no priority
  - Alt + 5 = Plane B, with priority
  - Alt + 6 = Plane A, with priority
  - Alt + 7 = VDP sprites, with priority
  - Alt + 8 = Custom sprites, with priority
  - Note that certain rendered objects like custom planes and debug draws can‘t be toggled

Simulation Time Control

The following hotkeys control the simulation speed:

- Numpad-1 = Normal game speed
- Numpad-2 = Fast-forward 3x game speed (with Ctrl: 2x game speed)
- Numpad-3 = Fast-forward 5x game speed
- Numpad-4 = Fast-forward 10x game speed
- Numpad-7 = Fast-forward at maximum speed
- Numpad-5 = Slowmotion 0.2x game speed (with Ctrl: 0.5x game speed)
- Numpad-6 = Slowmotion 0.05x game speed (with Ctrl: 0.01x game speed)
- Numpad-0 = Pause simulation
- Numpad-Period = Single-frame step
- Ctrl + Numpad-Period = Continue at normal speed until the next watch triggers or something gets logged (also Shift + Numpad-Period)

Audio Volume

Increase or reduce the global audio volume:

- Numpad-Plus = Increase global volume
- Numpad-Minus = Reduce global volume

Game Resolution

Change the horizontal game resolution:

- Numpad-Divide = Reduce width
- Numpad-Multiply = Increase width

Miscellaneous functions on the F keys

- F1 = Show hotkeys
- Shift+F1 = Open the config.json file
- F2 = Write a game recording file
- F3 = Rescan for connected controllers

- F4 = Switch controls for player 1 and player 2

Save States

- F5 = Go to the Save State menu
- F8 = Go to the Load State menu
- F7 = Quick load last loaded state - without changing current speed (so this can be used while single-stepping with Numpad-Period)

Performance

- Alt + P = Switch between performance displays
  - Warning: Don‘t leave the first one (with the flickering box) on for a longer period of time - like more than some minutes -, as it isn‘t too healthy for some LCD screens

Memory View

Keys I, O and L control the Memory View, see section on this Tool.

Side Panel

Keys Home and End control which Side Panel tab is shown.

Debug Keys

Numbers 1, 2 ..., 9 and 0 on the main keyboard can be read as Debug Keys by the scripts, see script binding "Key0".
E.g. for Sonic 3 A.I.R., key 1 disables camera bounds, and key 0 toggles a debug output for certain collision boxes.

Built-in Debugging Tools

Memory View

The Memory View is a simple output of a part of memory. It allows you to investigate both ROM and RAM data, as well as the extended RAM.
The data is shown in hexadecimal form, like in a hex editor. The leftmost text in each line is the address of the first byte shown.

Use the following keys:

- I = Show the Memory View, change number of lines shown, and hide it again
- O / L = Navigate through memory by changing the start address from where data is shown; O = up, L = down
  - Change by 0x20 if no other key is pressed
  - With Shift: Change by 0x100
  - With Ctrl: Change by 0x1000
  - With Shift + Ctrl: Change by 0x10000

Side Panel

The Side Panel displays various types of debug information arranged in several tabs.
Note that many items in there act as clickable buttons, to show or hide additional information.
Also, the side panel can be made smaller or larger by dragging its left border.

C = Call Frames

- Lists all functions called in the last simulated frame.
- They are ordered by their call hierarchy by default, but you can also sort them alphabetically or by source file name.
- For doing optimizations, profiling samples can be shown as a rough estimate how much script code got executed in each function.

U = Unknown Addresses

- In case simulation encountered jumps or calls to unknown addresses, these are listed here.

W = Watches

- Here you can find the active debug watches, i.e. memory locations whose changes are tracked during script execution.
- Watches with one or more changes in the last frame are highlighted. You can click on each change to get a call stack that shows where exactly the change happened.
- Note that you can add watches by using the "Global Defines" tab, or with the "debugWatch" script function.

G = Global Defines

- Shows all script defines with a fixed memory address, i.e. all global variables in memory.
- You can investigate on their values and set debug watches on them here.

R = Rendered Geometry

- Gives an overview over all planes, sprites, etc. that was rendered in the last frame.
- In most (but not all) cases, hovering your mouse over an entry highlights its bounding box in the viewport.

V = VRAM Writes

- This is some kind of persistent debug watch, specifically for the emulated video memory.
- Lists all writes to VRAM that got triggered in the last frame by the scripts.

L = Log

- In addition to the usual log output on screen, all logged events are displayed here again.
- Click on a log entry to get a call stack showing where it was logged.

Additional tabs

- More tabs can be defined by scripts (see "OxygenCallback.setupCustomSidePanelEntries" in Sonic 3 A.I.R. scripts)

Lemon Script

About the Lemon Script Language

Lemon script is a homebrew scripting language originally designed to be an alternative representation of M68K assembler code.
As such it is a quite simple language, originally including only features that were necessary for porting Sonic 3 A.I.R.

If you want to engage in scripting, make yourself familiar with the language first. It is somewhat similar in syntax to languages like C (or C++, C#, Java, whatever) - but with some differences, most notably:

1. Line break handling
   - Line breaks are regarded as end of a code line. That implies that you can't split expressions like function calls into multiple lines.
   - For that reason, there's no need to end a line with a semi-colon. Doing so results in a compile error.
   - Curly braces `{ }` always have to be placed in their own lines.
2. There is only a limited number of data types that can be used, and currently no way to define additional data types.
3. Definitions of functions and global variables have to start with keywords `function` and `global`, respectively. Other definitions like constants etc. start with their respective keywords.

Script Feature Level

By selecting a feature level for your scripts, you gain access to more recently added lemon script language features. By default, level 1 is used, unless you add the following line to your scripts:

```
//# script-feature-level(2)
```

Replace the number here with one of these two available feature levels so far:

(1) The default feature level 1, mainly for compatibility of older scripts. It includes the base script language features, but none of the extensions of higher levels, see below.

(2) Changes for level 2:
   - Introducing `string` as an actual data type (it is usable in level 1, but is pretty much just an alias for u64 there).

- The compiler checks for the correct syntax for if-clauses, namely you need to place parentheses around the condition (this was optional before).

Identifiers

Identifiers for variables, functions, defines, etc. consist of the following characters:

```
'A'..'Z', 'a'..'z', '0'..'9', '_', '.'
```

So unlike other languages, the dot can be a normal part part of an identifier.
Lemon script is case-sensitive.

Data Types

There are only a handful of basic data types for variables and constants that can be used:

- `u8` is a 8-bit / 1-byte unsigned integer, values ranging from 0 to 2^8-1 (0 to 255)
- `u16` is a 16-bit / 2-byte unsigned integer, values ranging from 0 to 2^16-1
- `u32` is a 32-bit / 4-byte unsigned integer, values ranging from 0 to 2^32-1
- `u64` is a 64-bit / 8-byte unsigned integer, values ranging from 0 to 2^64-1
- `s8` is a 8-bit / 1-byte signed integer, values ranging from -2^7 to 2^7-1 (-128 to 127)
- `s16` is a 16-bit / 2-byte signed integer, values ranging from -2^15 to 2^15-1
- `s32` is a 32-bit / 4-byte signed integer, values ranging from -2^31 to 2^31-1
- `s64` is a 64-bit / 8-byte signed integer, values ranging from -2^63 to 2^63-1

For boolean values, you can use the type `bool`, but note that this not an actual data type, instead only a synonym for `u8`. When evaluating a value to an actual bool (e.g. in an if-clause condition), all values != 0 are regarded as true, and 0 is regarded as false.

There is no floating point data type. Because of M68K code origins.

The `void` type exists, but only as return value type for functions that don't return anything.

Starting with script feature level 2, `string` is added as a proper data type for strings (before that, strings were referenced using u64 variables). A string can be casted to a u64 integer, giving you the 64-bit hash of the string, but you can't convert such a u64 hash value back to a string.
To create a string, use string literals like `"Hello World"` for fixed strings, and the stringformat function to build new strings if needed.
Note that strings are always 8-bits per character, usually interpreted as ASCII.

With script feature level 2, there's a few more features available for strings:

- operator `+` concatenates two strings (with level 1, this only results in a garbage value)
- operators `<`, `<=`, `>`, `>=` compare two strings, using their ASCII representation
- the method-like call to `.length()` returns a string's length

Examples:

```
string part1 = "This is "
string part2 = "a full sentence."
string text = part1 + part2         // text = "This is a full sentence."
if (part1 < part2)                  // That's true here, as character 'T' has a smaller ASCII value than character 'a'
{
    u8 length = text.length()       // Returns the length of text, which is 24 characters in this case
}
```

Variables

You can define variables to store data. There's two kinds of variables:

- **Global variables** need to be defined outside of functions, using the `global` keyword. As the name implies, they can be accessed from everywhere in your own scripts and also in modules compiled on top (i.e. mods using a higher mod priority).
- **Local variables** only exist inside the scope where they are created. This can be a whole function, or a block `{ }` like the if-clause's if-block and else-block.

The code line to define a variable uses this form:

- `global` *`<type>`* *`<name>`* for global variables
- and simply *`<type>`* *`<name>`* for local variables

Using the following placeholders:

- *<type>* can be any data type, except void
- *<name>* is an arbitrary identifier following the rules for identifiers as described above - you just need to avoid keywords and already existing names of variables, functions, constants, etc.

Examples:

```
global u8 winCounter          // Defines a global variable named "winCounter" of type u8; needs to be placed outside a function
s32 offset                    // Defines a local variable named "winCounter" of type s32; needs to be placed inside a function
string characterName          // Defines a local variable named "characterName" of type string; to be placed inside a function
```

Variables by default have a value of 0 for integer data types, false for `bool` variables, or an empty string in case of `string` variables.
In case of local variables you can directly assign them as value like this:

```
s32 offset = 60                     // Define one variable with a fixed initial value
s32 anotherOffset = offset * 2      // Define another one whose initial value is going to be twice of whatever offset is
                                    // when that line gets executed
```

Constants

Unlike variables, which can be assigned new values all the time, constants are meant to be fixed values that can't change at run-time at all.
To declare a constant, use this syntax:

```
constant <type> <name> = <value>
```

With these placeholders:

- *<type>* is a data type, which can be any integer type, `bool` and `string`
- *<name>* is the name if the constant and must be a valid identifier (the same rules as for variable names apply)
- *<value>* is the value of the constant, which needs to fit with `<type>`.

Examples:

```
constant u16 SCREEN_HEIGHT = 224
constant string WELCOME_MESSAGE = "Hey diddly ho neighborino!"
```

Constants can either be defined:

- in global scope - making them globally accessibly, like global variables
- or in a local scope inside a functions - in case they're needed only locally

Constant Arrays

For creating a fixed (in terms of size and content) lookup table or a similar list of constant values, you can make use of constant arrays:

```
constant array<<type>> <name> =
{
    <values>
}
```

Here, placeholders *<type>*, *<name>* and *<values>* have these meanings:

- *<type>* is the data type of the values in the array; allowed are all the same types as for constants
- *<name>* is the name of the array, following the usual rules for identifiers
- *<values>* is a comma-separated list of values, which can be multi-line if needed
- Note that in the syntax definition above, the `<>` characters around the type are *literally* the characters `<` and `>` and not part of placeholders like `<type>` etc.

Some examples:

```
constant array<u32> PRIMES =                    // A lookup of the first 10 prime numbers, no idea where that's useful
{
    2, 3, 5, 7, 11, 13, 17, 19, 23, 29
}

constant array<string> CHARACTER_NAMES =        // A list of character names
{
    "Sonic",                                    // You can place line breaks between values as you like
    "Tails", "Knuckles",
    "Motobug"
}
```

Just like with constants, you're free to define constant arrays either in global scope, or locally inside a function if needed only in there.

To use a constant array, access individual values like this:

```
string firstCharacterName = CHARACTER_NAMES[0]
string lastCharacterName = CHARACTER_NAMES[3]
```

The first value in the array always uses **index 0**, the last one the number of values minus 1.

To get the number of values in a constant array, you can use the array's `.length()` method right after the constant array name, like this:

```
u32 numCharacterNames = CHARACTER_NAMES.length()
string lastCharacterName = CHARACTER_NAMES[CHARACTER_NAMES.length()-1]     // Now index 3 is not hard-coded any more
```

Control Flow

Lemon script allows for the following control flow statements.
Note that the inner statements there can themselves be control flow statements like an if clause or one of the loop statements below.
Also note that in the syntax examples, *<single-statement>* / *<statements>* , *<condition>*, *<initialization>* and *<incrementation>* are just placeholder to be replaced with your own code.

If-clause

Executes a statement or block only if a given boolean condition is met.
The syntax is similar to many C-like programming languages. For a single conditional statement:

```
if (<condition>)
    <single-statement>      // This must not be exactly one statement; if you place more, only the first is executed conditionally
```

Or for a block of multiple statements:

```
if (<condition>)
{
```

```
    <statements>             // One or multiple statements, or none at all
}
```

You can add one or more statements to be executed if the condition is not met by adding an `else` part:

```
if (<condition>)
{
    <statements>
}
else
{
    <statements>
}
```

Or chain together multiple different conditions:

```
if (<condition>)
{
    <statements>
}
else if (<condition>)        // Should be a different condition obviously, otherwise this doesn't make much sense
{
    <statements>
}
else
{
    <statements>
}
```

While-loop

Executes a statement or block an arbitrary number of times, as long as a given condition is met.
Here again, a C-like syntax is used:

```
while (<condition>)
{
    <statements>
}
```

Just like for the if-clause, you can also have a single statement instead of the block.
Inside a loop, you can use the `break` and `continue` statements:

- `break` exits the loop immediately and continues script executing after it
- `continue` restarts the loop, including an evaluation of the condition

For-loop

Executes a statement or block, as long as a given condition is met, and does an increment step after each loop.

```
for (<initialization>; <condition>; <incrementation>)
{
    <statements>
}
```

Such a for-loop statement is basically a shorthand for the following loop:

```
<initialization>
while (<condition>)
{
    <statements>
    <incrementation>
}
```

Note that:

- Both `break` and `continue` are possible in there as well, with continue executing the incrementation before doing the condition evaluation.
- Each one of *<initialization>*, *<condition>* and *<incrementation>* can either be a single statement or just empty. If the condition is empty, the for-loop will loop without condition and can only be exited using `break`.

Memory Access

To access a piece of memory (including ROM and RAM), use a basic data type followed by the address in square brackets. This can be used for both reading and writing data.
Example: `u16[0x012345]` would access the bytes at addresses 0x012345 and 0x012346.

Some notes:

- Memory access is always **little endian**, just like the Mega Drive's / Genesis' M68K processor.
- Unlike with the M68K, it's allowed to access odd memory addresses with all data sizes, not just `u8` and `s8`.
- The address can be constant or calculated at runtime like in `u32[myAddressVariable + 4]`
- Memory access can use only the following data types: `u8, s8, u16, s16, u32, s32, u64, s64` - but not `string`

The total mapped memory space is limited to 24-bit addresses (0x000000 to 0xffffff), memory accesses only use the lower 24-bits. This means that addresses 0xffff1234 and 0xff1234 are identical.
Inside the memory space, not all addresses are valid to access. The following are:

- 0x000000 .. 0x3fffff = ROM -- this is where the game ROM content is stored
- 0x800000 .. 0x8fffff = "Shared memory", actually additional RAM
- 0xff0000 .. 0xffffff = RAM – for game ports like Sonic 3 A.I.R., this is mostly used the same way as by the original game

Registers

The registers a0 .. d7 of the M68K are represented in lemon script by the global variables `A0` .. `D7`.
These have some special sub-variables that allow for partial access. Here's an overview using `D0` as example:

- `D0` -> This is the full 32-bit register, interpreted as unsigned by default
- `D0.u8` -> Lowest 8 bits, interpreted as unsigned 8-bit value
- `D0.s8` -> Lowest 8 bits, interpreted as signed 8-bit value
- `D0.u16` -> Lowest 16 bits, interpreted as unsigned 16-bit value
- `D0.s16` -> Lowest 16 bits, interpreted as signed 16-bit value
- `D0.u32` -> Lowest 32 bits, interpreted as unsigned 32-bit value (same as `D0`)

- `D0.s32` -> Lowest 32 bits, interpreted as signed 32-bit value

Functions

Function definitions follow this syntax:

```
function <return-type> <name>(<parameters>)
{
    // Here comes the code to be executed when the function is called
}
```

The `function` keyword always goes first, followed by the return value - which can be `void` if the function should not return anything.
Next is the function name, which needs to follow the rules for identifiers (see above).
And finally a list of parameters. This can be empty `()` if no parameters are needed, otherwise it's a comma-separated list of parameters, each consisting of the parameter data type and the parameter variable name.

Examples:

```
function void myTestFunc()          // Defines a function named "myTestFunc" with no return value and no parameters
{
    // Do stuff here
}

function s16 getNiceNumber(bool negative)        // This function returns a u16 integer and takes a boolean as input
{
    // Just an example; returning either -42 or 42 depending on the input
    if (negative)
        return -42
    else
        return 42
}
```

```
function void addPlayer(string name, u16 age, u32 highscore)        // An example with multiple parameters
{
    // This example makes use of the functions "debugLog" that prints a text on the screen, and "stringformat" to build a string
    debugLog(stringformat("Player %s has highscore %d", name, highscore))
}
```

Functions with actual return types (not void) need to return a value on each possible path of execution.

To call a function inside your code, use this syntax:

```
myTestFunc()                              // Calls "myTestFunc" as defined in the example above
s16 myVariable = getNiceNumber(true)      // Calls "getNiceNumber" as defined in the example above & uses the return value
addPlayer("Guy123", 21, 271400)           // Calls "addPlayer" as defined in the example above
```

Function Overloading I

It's possible to create multiple versions of the same function with different parameters. When resolving calls, the compiler will choose the most fitting version of the function.

Function Overloading II

You can even overload a function with the *very same* parameters and return type. This is particularly useful if you're making a mod that wants to replace a function from the vanilla game with one that does things differently.
In such a case, your overloaded function might want to call the original version of the function at some point of execution. Maybe because you don't need to replace it all, but only add some code before or after the original version is executed. For such cases, just call the original function by its name with an additional prefix `base.` - including the dot.

In practise that looks like this:

```
function void helloWorld()          // Original version of the function in the main game scripts
{
```

```
    debugLog("Original function called")
}

function void helloWorld()          // Overloaded version - usually not directly below, but somewhere in your mod
{
    debugLog("Doing stuff before the original")
    base.helloWorld()
    debugLog("This is done after the original")
}
```

Address Hooks

Functions can have so-called address hooks. This is an associated ROM address for the function, to allow for indirect calls using addresses with the "call" keyword, like in `call 0x123456 + D0.u16`.
Whenever such a call is executed, the function with the resulting address's hook will get called. Or none at all, if there is no function with that hook, leading to the call having no effect.

Address hooks are declared with a line like this preceding the function header:

```
//# address-hook(0x123456) end(0x123478)
```

Here, the first curved brackets include the address for this function. The "end" part afterwards is optional and has only informative meaning for functions directly representing original M68K code.

Includes

There's usually only a single script file that gets loaded as a starting point, e.g. a "main.lemon" file.
In order to add more script files to compilation, `include` lines like the following examples are used:

`include myfunctions` → includes the contents of a script file named "myfunctions.lemon"
`include utils/helpers` → includes the contents of a single script file "helpers .lemon" in a subdirectory named "utils"
`include entities/player/?` → includes the contents of all script files in a nested subdirectory "entities/player"

The paths used in includes are always relative to the including script file's location.

Note that the wildcard `?` can only be used to include all *.lemon files in a directory, but not for additional filtering or recursive includes of subdirectories.

Preprocessor Directives

Before evaluating the actual contents of a script, the compiler performs some preprocessing on the script - especially filtering out parts of the code that aren't meant to be compiled, e.g. because they require a newer or older game version.
There's a few preprocessor directives you can use to control that.

#define *<name>*
#define *<name>* = *<value>*

Creates a new preprocessor definition, or if one with that name already exists, overwrites its value.
If no value is explicitly assigned, the value will be 1.

Examples:

```
#define HAS_EUKAS_AWESOME_MOD             // Sets the definition HAS_EUKAS_AWESOME_MOD to value 1
#define AWESOME_MOD_VERSION = 0x105       // Sets the definition AWESOME_MOD_VERSION to value 0x105
```

Note: Do not confuse the `#define` preprocessor directive with the `define` keyword (without #) that is used in the Sonic 3 A.I.R. scripts to declare shorthands for ceratin expressions. These are really two entirely different things (though both are derived from preprocessor defines / macros in the C programming language, hence the same naming).

#if *<condition>*
#else
#endif

Checks whether the condition evaluates to anything different from 0. If so, the scripts between `#if` and `#endif` (or `#if` and `#else`) are included in the compilation. If the condition evaluates to 0, the scripts in between are filtered out and handled as if they were empty lines.
The condition may include:

- Preprocessor definitions. These include the ones listed under "Default Preprocessor Definitions" below, as well as custom ones defined by the script module itself or other modules that were compiled before (via `#define`). If a preprocessor definition is unknown, it evaluates to 0 automatically.
- Operations between preprocessor definitions and constants.

Examples:

```
#if HAS_EUKAS_AWESOME_MOD
    // Code in between here is compiled if the HAS_EUKAS_AWESOME_MOD preprocessor definition is set
#endif

#if GAMEAPP >= 0x22020100
    // Code in between here is compiled if the GAMEAPP preprocessor definition is set to a value of 0x22020100 or higher
#else
    // Code in between here is compiled if the GAMEAPP preprocessor definition is set to a value of less than 0x22020100
#endif
```

Note: You can also filter out other preprocessor directives, e.g. create a preprocessor definition via `#define` only when a certain condition is fulfilled.
However, this only works in game versions from February 2022 on. For compatibility with older game versions, you can resort to including a different script file conditionally.
For example:

```
#if GAMEAPP >= 0x22020100                // This encodes the date of Feb 1, 2022
    include my_new_definitions_file
#endif
```

where a different script file - in this case named "my_new_definitions_file.lemon" would have the `#define` directives (which by the way also require a game version from at least February 2022).

#error "*<error-message>*"

Makes script compilation fail intentionally and print a message to the player.

Example:

```
#error "Something is wrong! Also, this message is not helpful :("   // Let's hope your error message is better than that
```

This can be useful if you're making a script mod that depends on a different script mod to be loaded first. Given that the other mod created a preprocessor definition that you can check against, you can output an error if that definition was not found.
For example:

```
#if !HAS_EUKAS_AWESOME_MOD
    #error "Euka's Awesome Mod is required. Make sure to activate it, using a lower mod priority."
#endif
```

Default Preprocessor Definitions

STANDALONE

In the Sonic 3 A.I.R. game scripts, you might stumble across usage of the `STANDALONE` preprocessor definition. This one is always set to 1.

But why does it even exist?
The idea is that when STANDALONE is 0, the code behaves exactly like the original game code. That means that all changes made to the game are marked with a `#if STANDALONE` block or similar.
(Nitpicking side-note: Most but not *all* changes are marked this way. There are few exceptions like calls to getScreenWidth() or getScreenExtend() that will return different values in the widescreen standalone than in the original game, where a fixed screen width of 320 and a screen extend of 0 is used.)

GAMEAPP

The `GAMEAPP` preprocessor definition makes the difference between the Oxygen Development Application ("OxygenApp.exe") and the actual game application ("Sonic3AIR.exe").
Unlike `STANDALONE`, its value is not 1, but a version code of the game executable, so you can do comparisons like `#if GAMEAPP >= 0x21010800` if e.g. you need to include a piece of script code for only the newer game versions that support it.
For Sonic 3 A.I.R. the value of `GAMEAPP` is something like 0x21010800 (this example representing version 21.01.08.0) – you can get the actual value from the "GameAppBuild" entry in the game's metadata.json.

In conclusion:

- Game application: `GAMEAPP = 0x21010800` `STANDALONE = 1`
- Oxygen dev app: `GAMEAPP = 0` `STANDALONE = 1`

Engine Script Functions

A general note on fixed point numbers

Some of the following functions use fixed point numbers as parameters or return values. This is done as replacement for floating point numbers, and can be one of the following:

- A 8.8 fixed point number is a 16-bit value (either u16 or s16), where the upper 8 bits represent the integer part, and the lower 8 bits represent the fractional part.
  Examples:
  - `s16 value = 0x700      // This represents 7 as a 8.8 fixed point number`
  - `s16 value = 0x340      // This represents 3.25 as a 8.8 fixed point number`
  - `s16 value = -0x55      // This represents ca. -0.332 as a 8.8 fixed point number`
- Similarly, a 16.16 fixed point number is a 32-bit value (either u32 or s32), where the upper 16 bits represent the integer part, and the lower 16 bits represent the fractional part.
  Examples:
  - `s32 value = 0x70000    // This represents 7 as a 16.16 fixed point number`
  - `s32 value = 0x34000    // This represents 3.25 as a 16.16 fixed point number`
  - `s32 value = -0x5555    // This represents ca. -0.3333 as a 16.16 fixed point number`

Basics

s8 min(s8 a, s8 b)
u8 min(u8 a, u8 b)
s16 min(s16 a, s16 b)
u16 min(u16 a, u16 b)
s32 min(s32 a, s32 b)
u32 min(u32 a, u32 b)

- Returns the smaller value of inputs a and b.

s8 max(s8 a, s8 b)

- Returns the larger value of inputs a and b.
- Also exists for types u8, s16, u16, s32, u32.

s8 clamp(s8 a, s8 b, s8 c)

- Limits the value in a into the range defined by b and c; the result is:
  - b if a <= b
  - c if a >= c
  - a if it's between b and c
- Also exists for types u8, s16, u16, s32, u32.

u8 abs(s8 value)

- Returns absolute value of the input, i.e. negates it if it's negative.
- Also exists for 16-bit and 32-bit data sizes.

u32 sqrt(u32 value)

- Calculates the square root of the input value, rounded down to the next integer value.

s16 sin_s16(s16 value)
s16 cos_s16(s16 value)

- Calculates the sine or cosine, respectively, of the input value.
- The input value is expected to be a signed 8.8 fixed point number in radians – so an input of 0x648 (rounded, or 2*Pi*0x100 to be exact) represents 360°.
- The return value is also a 8.8 fixed point number, so it's something between -0x100 and 0x100.

s32 sin_s32(s32 value)
s32 cos_s32(s32 value)

- Same as sin_s16 / cos_s16, but with 16.16 fixed point numbers, so it's more precise.

- So, an input value of 0x6487e (rounded, or 2*Pi*0x10000 to be exact) represents 360°; and the result is in range -0x10000 to 0x10000.

String Access / Manipulation

u32 strlen(string str)

- Returns the length of the string. Alternatively, you can instead use `str.length()` .

u8 getchar(string str, u32 index)

- Returns the character at the given index.

string substring(string str, u32 index, u32 length)

- Builds a substring, starting at index with the given length.

u64 stringformat(string format, ...)

- Creates a string by filling in values into a "printf"-like pattern.
- The format parameter has to be a string and supports the following placeholders:
  - %s for string parameter
  - %d for integer parameter, output as decimal number
  - %x for integer parameter, output as (unsigned) hexadecimal number; note that this will not add any prefix like "0x"
  - %b for integer parameter, output as (unsigned) binary number; note that this will not add any prefix
  - %02d or %02x for integer output (decimal or hexadecimal, respectively), and an enforced minimum number of digits to be filled up with zeroes; replace the 2 with any other single digit
  - Use %% if you like to output an actual percent character %
- This function supports up to 8 additional value parameters.

Flag Registers

Note: The internal flag registers of the engine – Zero and Negative Flag – only exist for easier compatibility of scripts with original M68K code. Use the following functions only where needed, and avoid using them in your own new script code.

bool _equal()

- Returns true if the Zero Flag is set.

bool _negative()

- Returns true if the Negative Flag is set.

void _setZeroFlagByValue(u32 value)

- Set the Zero Flag by the given value. It will be set if the input value is exactly zero, and cleared in all other cases.

void _setNegativeFlagByValue(s8 value)
void _setNegativeFlagByValue(s16 value)
void _setNegativeFlagByValue(s32 value)

- Set the Negative Flag by the given value. It will be set if the signed input value is below zero, and cleared if it is zero or positive.

Stack Access

Similar to the flag register functions, this is mostly for compatibility with M68K code.
Avoid using these, as there are usually better ways of temporarily saving values (namely using local variables) or passing values between functions.

void push(u32 value)

- Pushes the given value onto the stack.

u32 pop()

- Returns the topmost value from the stack and removes it there.

Memory Data Processing

void copyMemory(u32 destAddress, u32 sourceAddress, u32 bytes)

- Copies data from one address in memory (e.g. RAM or ROM) to another.

void zeroMemory(u32 startAddress, u32 bytes)

- Clears a part of memory by writing zeroes there.

void fillMemory_u8(u32 startAddress, u32 bytes, u8 value)
void fillMemory_u16(u32 startAddress, u32 bytes, u16 value)
void fillMemory_u32(u32 startAddress, u32 bytes, u32 value)

- Fills a part of memory by writing multiple copies of the given value there.

Persistent Data Access

Persistent data is stored in a file inside the user's data directory. It can be used for saving game progress, settings, etc.
The persistent data file internally is built up of multiple packages with arbitrary strings as identifying names / keys. This way, you can store multiple different types of data inside the same persistent data file.

u32 System.loadPersistentData(u32 targetAddress, string key, u32 maxBytes)

- Loads data from the persistent data file.
- Pass a string as key that identifies which persistent data package you want to load.
- Returns the number of bytes written to the target address, this can be 0 if no persistent data existed with the given key.

void System.savePersistentData(u32 sourceAddress, string key, u32 bytes)

- Saves data from memory into the persistent data file.
- Pass a string as key that identifies which persistent data package you want to overwrite. If the key is not used yet, a new package will be created.

u32 SRAM.load(u32 address, u16 offset, u16 bytes)
void SRAM.save(u32 address, u16 offset, u16 bytes)

- Deprecated. (This was part of the old SRAM-emulating system that got used before the more flexible persistent data functions were a thing.)

Raw Data Loading

bool System.hasExternalRawData(string key)

- Returns true if there is a raw data file registered under the given key string.

u32 System.loadExternalRawData(string key, u32 targetAddress)

- Loads data from a raw data file into memory.
- Returns the number of bytes loaded. This can be zero if no raw data was registered under the given key.

u32 System.loadExternalRawData(string key, u32 targetAddress, u32 offset, u32 maxBytes, bool loadOriginalData, bool loadModdedData)

- Same as above, but with more parameters.
- Parameters offset and maxBytes determine which part of the raw data to load. You can set maxBytes to zero to load everything starting at offset.
- Parameters loadOriginalData and loadModdedData can limit raw data loading to keys defined as part of the game itself or a mod for that game, respectively.

bool System.hasExternalPaletteData(string key, u8 line)

- Returns true if there is a custom palette defined in the game's or mod's palettes data directory with the given key.
- Use the palette's file name (without extension .png) as key.

u16 System.loadExternalPaletteData(string key, u8 line, u32 targetAddress, u8 maxColors)

- Loads data from a custom palette defined in the game's or mod's palettes data directory.
- Returns the number of colors loaded.
- Colors will get loaded in 32-bit ABGR format, i.e. with one byte per channel, and red in the least significant byte.

Game Resolution

u16 getScreenWidth()

- Returns the width in pixels of the simulated game screen.

u16 getScreenHeight()

- Returns the height in pixels of the simulated game screen.

u16 getScreenExtend()

- Returns the additional pixels on each side of the screen for widescreen display.
- This is calculated as:
  ```
  (getScreenWidth() - 320) / 2
  ```

Input

u16 Input.getController(u8 controllerIndex)

- Returns a controller's input bitmask, with each bit representing a button or d-pad direction.

u16 Input.getControllerPrevious(u8 controllerIndex)

- Returns a controller's former input bitmask from the previous frame.

u8 Input.buttonDown(u8 index)

- Returns true if the button with the given index is held down at the moment.
- Index ranges from 0x00 to 0x0f for player 1's input. Add 0x10 to query player 2's input instead.

u8 Input.buttonPressed(u8 index)

- Returns true if the button with the given index was just pressed this frame, i.e. is now held down but wasn't before.

Miscellaneous

u32 System.rand()

- Returns a random u32 number.
- Usage of this function breaks determinism of scripts, which can make debugging much harder. E.g. game recordings including a call to „System.rand" won't work properly and input recordings won't produce the same results each time.
- For that reason, if you need random number, better rely on a script-implemented pseudo-random number generator.

void setWorldSpaceOffset(s32 px, s32 py)

- Set the position of the upper left corner of the screen inside the world space coordinate system.
- This makes sense to set in 2D scrolling games to be able to use world space coordinates instead of screen space coordinates e.g. when drawing custom sprites.

s64 System.getGlobalVariableValueByName(string variableName)

- Returns the value of a global variable, given by its name.
- The variable name must refer to an actual global variable - not a local variable, constant or define, or anything else. If no global variable with the given name exists, the returned value will be 0.
- This function will return a s64 no matter what the data type of the global variable is. You might need to cast the result to the right type.
- Whenever possible, use global variables directly, instead of relying on this function.

void System.setGlobalVariableValueByName(string variableName, s64 value)

- Sets the value of a global variable, given by its name.
- The variable name must refer to an actual global variable - not a local variable, or anything else. If no global variable with the given name exists, nothing happens.
- This function works with all integer types as values.
- Whenever possible, use global variables directly, instead of relying on this function.

Debugging

Log = ...

- Not a function actually, but some kind of global variable. Though it can only be written to.
- This is a quick way of logging an integer value (up to 64-bit) onto the screen.
- Usage example: `Log = D0.u16`
- Note that one one logged value per script line that uses Log will stay visible on the screen. So when using this, you might want to step frame by frame, or run until a logging occurs.

void debugLog(string text)

- Writes the passed string to the debug log.
- This is the more sophisticated version of the Log variable, this time as an actual function.

void debugLogColors(string name, u32 startAddress, u8 numColors)

- Outputs multiple colors as palette debug box on the screen.
- It will only be visible for a single frame, so you might want to step frame by frame, or run until a logging occurs.

void System.writeDisplayLine(string text)

- Writes text into the display line at the bottom of the screen.
- Text written to the display line will stay there briefly before fading out.

void assert(bool condition)
void assert(bool condition, string text)

- Triggers an error message box if the condition is false.
- With the text parameter, you can define a custom text to display as error message.

void debugWatch(u32 ramAddress, u16 bytes)

- Adds a debug watch on the given address in RAM or shared memory.

void debugDumpToFile(string filename, u32 startAddress, u32 bytes)

- Save parts of memory to a file. The filename parameter must be a string.
- The file will be created in the game‘s working directory, which is usually the executable directory.

void Debug.drawRect(s32 px, s32 py, s32 sx, s32 sy)
void Debug.drawRect(s32 px, s32 py, s32 sx, s32 sy, u32 color)

- Draw a rectangle on the screen for debugging purposes.
- Uses world space coordinates.
- The optional color parameter is using the 0xRRGGBBAA format.

Control Flow

void yieldExecution()

- Signals the end of the current frame. The script will continue after this call in the next frame.
- Best practise: Call this only in a single script function that acts as a general function ending a frame. When modding a game, you usually wouldn‘t use this at all.

bool System.callFunctionByName(string functionName)

- Calls a script function directly by its name.
- The parameter must be the name of the script function to call. This only works for functions without parameters and return values.
- The return value is true if the function actually exists, otherwise the call has no effect and false is returned.

void System.setupCallFrame(string functionName)

- Creates a new entry on the call stack, at the start of the given script function.
- The parameter must be the name of the script function to use. This only works for functions without parameters and return values.

void System.setupCallFrame(string functionName, string labelName)

- Creates a new entry on the call stack, at a label inside the given script function.
- The parameters must both be a strings, the function name (see above), and the name of the label.

Direct VRAM Access

u16 getVRAM(u16 vramAddress)

- Reads a u16 value (2 bytes) from the given VRAM address.

void setVRAM(u16 vramAddress, u16 value)

- Writes a u16 value (2 bytes) to the given VRAM address.

Emulated VDP (Video Display Processor)

void VDP.setupVRAMWrite(u16 vramAddress)

- Set the VRAM address to use for subsequent calls to VDP.readData* and VDP.writeData*.

void VDP.setupVSRAMWrite(u16 vsramAddress)

- Set the address in VSRAM (vertical scroll offsets) to use for subsequent calls to VDP.readData* and VDP.writeData*.

void VDP.setupCRAMWrite(u16 cramAddress)

- Set the address in CRAM (colors) to use for subsequent calls to VDP.readData* and VDP.writeData*.

void VDP.setWriteIncrement(u16 increment)

- Set the number of bytes to advance after each call to VDP.readData16 and VDP.writeData16.

u16 VDP.readData16()

- Read a u16 (i.e. 2 bytes) from VRAM / VSRAM / CRAM at the address previously defined with one of the calls above.
- The address will be increased by the current write increment in each call, so multiple calls to this function read from different addresses each time.

u32 VDP.readData32()

- Same as VDP.readData16, but reads a u32 (i.e. 4 bytes).
- Note that internally, this is two calls to VDP.readData16, so write increment is applied once in between and once afterwards.

void VDP.writeData16(u16 value)

- Write a u16 value to VRAM / VSRAM / CRAM. See VDP.readData16 for more information.

void VDP.writeData32(u32 value)

- Write a u32 value to VRAM / VSRAM / CRAM. See VDP.readData32 for more information.

void VDP.copyToVRAM(u32 address, u16 bytes)

- Copy memory from the given address (in RAM, extended RAM, or ROM) into VRAM / CRAM / VSRAM.
- This acts like several subsequent calls to VDP.writeData16, so uses the previously defined VRAM / CRAM / VSRAM address.

void VDP.zeroVRAM(u16 bytes)

- Fill parts of VRAM with zeroes.
- This acts like several subsequent calls to VDP.writeData16, so uses the previously defined VRAM / CRAM / VSRAM address.

void VDP.fillVRAMbyDMA(u16 fillValue, u16 vramAddress, u16 bytes)

- Fill parts of VRAM with the given u16 value.

void VDP.copyToVRAMbyDMA(u32 sourceAddress, u16 vramAddress, u16 bytes)

- Copy memory from the given address (in RAM, extended RAM, or ROM) into VRAM.
- This is a more handy version of the VDP.setupVRAMWrite + VDP.copyToVRAM combination.

void VDP.copyToCRAMbyDMA(u32 sourceAddress, u16 vramAddress, u16 bytes)

- Copy memory from the given address (in RAM, extended RAM, or ROM) into CRAM.
- This is a more handy version of the VDP.setupVRAMWrite + VDP.copyToCRAM combination.

void VDP.Config.setActiveDisplay(u8 enable)

- Enables or disables rendering.
- Only a blank screen in the backdrop color (black by default) will be shown while rendering is disabled.

void VDP.Config.setNameTableBasePlaneB(u16 vramAddress)

- Define the plane B name table base in VRAM.

void VDP.Config.setNameTableBasePlaneA(u16 vramAddress)

- Define the plane A name table base in VRAM.

void VDP.Config.setNameTableBasePlaneW(u16 vramAddress)

- Define the plane W name table base in VRAM.

void VDP.Config.setVerticalScrolling(bool verticalScrolling, u8 horizontalScrollMask)

- Enable or disable vertical scroll mode, i.e. usage of multiple scroll offsets in vertical direction stored in VSRAM.
- Also defines the scroll mask for horizonal scroll offsets. Common values are 0x00 (i.e. there‘s only one horizontal scroll value shared for all pixel lines), 0x07, 0xf8, and 0xff (i.e. there‘s a horizontal scroll value for each pixel line individually).

void VDP.Config.setBackdropColor(u8 paletteIndex)

- Sets the palette index to be used as backdrop color.

void VDP.Config.setRenderingModeConfiguration(u8 shadowHighlightPalette)

- Switches to other rendering modes. Not yet supported!

void VDP.Config.setPlayfieldSizeInPixels(u16 width, u16 height)

- Defines the size in pixels of plane A and plane B.
- Use only powers of two, between 256 and 1024 pixels for width and height.

void VDP.Config.setPlayfieldSizeInPatterns(u16 width, u16 height)

- Same as VDP.Config.setPlayfieldSizeInPatterns, but using patterns as input, with one pattern being 8 pixels in size.
- Valid values are powers of two, between 32 and 128 patterns for width and height.

void VDP.Config.setupWindowPlane(bool useWindowPlane, u16 splitY)

- Setup usage of plane W.
- If plane W is enabled, the screen will get split horizontally at pixel position splitY, with plane W being shown above and plane A shown below.

void VDP.Config.setPlaneWScrollOffset(u16 x, u8 y)

- Set plane W scroll offset.

Oxygen Renderer

void Renderer.setPaletteEntry(u8 index, u32 color)

- Set a palette entry directly without using the usual VDP mechanisms (involving CRAM).
- This way, you can access all entries up to index 0xff and write a 32-bit color value in ABGR format (i.e. least significant byte is red). Note that the alpha channel will get ignored.
- This function needs to be called in each frame again, otherwise will only have an effect for a single frame.

void Renderer.setPaletteEntryPacked(u8 index, u16 color)

- Same as Renderer.setPaletteEntry, but using the packed 16-bit color format.
- Depending on its topmost bit, a color value can be one of the following:
  - the 9-bit VDP colors format 0000 BBB0 GGG0 RRR0
  - or an extended 15-bit format 1BBB BBGG GGGR RRRR

void Renderer.enableSecondaryPalette(u8 line)

- Enables a secondary palette to be used below the given pixel line.
- This is used e.g. in Sonic game projects for the split between normal and underwater colors.
- This function needs to be called in each frame again, otherwise will only have an effect for a single frame.

void Renderer.setSecondaryPaletteEntryPacked(u8 index, u16 color)

- Same as Renderer.setPaletteEntryPacked, but for the secondary palette.

void Renderer.setScrollOffsetH(u8 setIndex, u16 lineNumber, u16 value)

- Sets a horizontal scroll offset directly without using the usual VDP mechanisms.

- This allows you to access the usual plane A (setIndex == 1) and B (setIndex == 0) scroll offsets, but also additional scroll offset sets (indices 0x02 and 0x03) that can be useful for custom planes, see Renderer.setupPlane.
- This function needs to be called in each frame again, otherwise will only have an effect for a single frame.

void Renderer.setScrollOffsetV(u8 setIndex, u16 rowNumber, u16 value)

- Sets a vertical scroll offset directly without using the usual VDP mechanisms.
- See Renderer.setScrollOffsetH for more info.

void Renderer.setHorizontalScrollNoRepeat(u8 setIndex, bool enable)

- Enables or disables the "NoRepeat" mode for horizontal scrolling, for the given scroll offset set.
- If enabled, horizontal scrolling will not wrap back inside the plane, but fill everything outside with empty / transparent pixels.

void Renderer.setVerticalScrollOffsetBias(s16 bias)

- Set a bias value that the renderer will add to the screen x-position when determining which vertical scrolling offset should be used.
- Only used in vertical scroll mode.

void Renderer.enforceClearScreen(bool enable)

- If enabled, the screen will get completely cleared each frame.
- This is usually not enabled, as rendering of plane B will overwrite the whole screen anyway. But when using custom planes, it can make sense to enable this.

void Renderer.enableDefaultPlane(u8 planeIndex, bool enabled)

- Planes A and B will get rendered by default, but you can prevent that with this function. Useful if you have custom planes as a replacement, see Renderer.setupPlane.

void Renderer.setupPlane(s16 px, s16 py, s16 width, s16 height, u8 planeIndex, u8 scrollOffsets, u16 renderQueue)

- Define a custom plane for a particular rectangle on the screen.
- Parameters:
  - px, py, width, height define the rectangle on screen

    - planeIndex refers to: 0x00 = plane B non-prio layer, 0x01 = plane A non-prio layer, 0x02 and 0x03 = plane B / A priority layers
    - scrollOffsets is the index of one of the scroll offset set to use, usually 0x00 (plane B offsets) or 0x01 (plane A offsets), but can as well be a custom set (0x02 and 0x03)
    - renderQueue defines when this plane will get rendered in relation to other planes and sprites
- This function mainly exists to allow for more obscure things like e.g. rendering plane A with different scroll offsets on the upper and lower parts of the screen.

void Renderer.resetCustomPlaneConfigurations()

- Removed previously defined custom planes and restores the default planes again.

void Renderer.resetSprites()

- Clears the list of sprites to be rendered.
- This is meant to be done once per frame before starting to draw sprites again. However, there might be situations where the last sprites shown should just remain, like just fading out the last image – in this situation, do not call this function and do not draw any new sprites.

void Renderer.drawVdpSprite(s16 px, s16 py, u8 encodedSize, u16 patternIndex, u16 renderQueue)

- Draws a sprite like the VDP does on original Mega Drive / Genesis hardware (VDP using its data in the sprite attributes table = SAT).
- Parameters:
    - px and py is the position on screen. Note that this is using actual screen coordinates (without the 0x80 offset preset in the SAT)
    - encodedSize is a byte of the format 00WW00HH, where WW is the width and HH is the height between 00 (1 pattern = 8 pixels) and 11 (4 patterns = 32 pixels)
    - patternIndex is an encoded reference to the patterns to use for drawing; same format as in SAT
    - renderQueue defines when this sprite will get rendered in relation to other sprites and the planes

void Renderer.drawVdpSpriteWithAlpha(s16 px, s16 py, u8 encodedSize, u16 patternIndex, u16 renderQueue, u8 alpha)

- Same as Renderer.drawVdpSprite, but supporting an additional alpha transparency value (0xff for full opacity).

void Renderer.drawVdpSpriteTinted(s16 px, s16 py, u8 encodedSize, u16 patternIndex, u16 renderQueue, u32 tintColor, u32 addedColor)

- Same as Renderer.drawVdpSprite, but supporting a multiplicative tint color and an additive color
- Both color parameters use the RGBA format, as a hexadecimal number this looks like 0xRRGGBBAA (red, green, blue, alpha)

bool Renderer.hasCustomSprite(u64 key)

- Check if there is a custom / modded sprite with the given key. The key must be a string here.

void Renderer.drawCustomSprite(u64 key, s16 px, s16 py, u8 atex, u8 flags, u16 renderQueue)
void Renderer.drawCustomSprite(u64 key, s16 px, s16 py, u8 atex, u8 flags, u16 renderQueue, u8 angle, u8 alpha)
void Renderer.drawCustomSpriteTinted(u64 key, s16 px, s16 py, u8 atex, u8 flags, u16 renderQueue, u8 angle, u32 tintColor, s32 scale)

- Draw a custom / modded sprite.
- Parameters:
  - key is a string referencing a custom sprite loaded from a PNG or BMP file, or it's the result of a Renderer.setupCustom*Sprite call
  - px and py denote the position where to draw the sprite on the screen
  - atex is an offset added to palette indices inside palette sprites (not used for component sprites loaded from PNGs)
  - flags is a combination of:
    - 0x01 = flip horizontally
    - 0x02 = flip vertically
    - 0x20 = interpret px/py as world space coordinates (see setWorldSpaceOffset) instead of a screen position
    - 0x40 = draw sprite with priority flag
    - 0x80 = ignore global tint and added color, even if this is an component sprite, where those would usually be applied automatically
  - angle is the rotation angle to draw, covering the full u8 range (0 is 0°, and 0xff is just below 360°)
  - alpha is an alpha transparence value between 0 (completely invisible) and 0xff (fully opaque)
  - tintColor can be used to change the sprites colors by channel-wise multiplication

        - the format as hexadecimal number is 0xRRGGBBAA (red, green, blue, alpha)
        - a value of 0xffffffff is white with full opacity, i.e. keeping the original colors
    - scale is a scaling factor for the sprite as a 16.16 fixed point number – i.e. a value of 0x20000 would draw the sprite in double size

void Renderer.drawCustomSpriteWithTransform(u64 key, s16 px, s16 py, u8 atex, u8 flags, u16 renderQueue, u32 tintColor, s32 m11, s32 m12, s32 m21, s32 m22)

- Draw a custom / modded sprite, this time with a custom 2x2 transformation matrix.
- Parameters are the same as for the third Renderer.drawCustomSprite variant, except:
    - angle and scale are replaced by the four entries of a 2x2 matrix (m11, m12, m21, m22 in row-first order)
    - These are interpreted as 16.16 fixed point values

u64 Renderer.setupCustomUncompressedSprite(u32 sourceBase, u16 words, u32 mappingOffset, u8 animationSprite, u8 atex)
u64 Renderer.setupCustomCharacterSprite(u32 sourceBase, u32 tableAddress, u32 mappingOffset, u8 animationSprite, u8 atex)
u64 Renderer.setupCustomObjectSprite(u32 sourceBase, u32 tableAddress, u32 mappingOffset, u8 animationSprite, u8 atex)
u64 Renderer.setupKosinskiCompressedSprite(u32 sourceAddress, u32 mappingOffset, u8 animationSprite, u8 atex)

- Creates a custom sprite from the ROM or RAM.
- These functions are somewhat game-specific, see the existing scripts for examples.

void Renderer.addSpriteMask(s16 px, s16 py, s16 width, s16 height, u16 renderQueue, u8 priorityFlag)

- Add a mask that hides all sprites with a lower render queue.

void Renderer.addSpriteMaskWorld(s16 px, s16 py, s16 width, s16 height, u16 renderQueue, u8 priorityFlag)

- Same as Renderer.addSpriteMask, but uses world space coordinates.

void Renderer.resetViewport(u16 renderQueue)

- Reset a previous reduction of the viewport to a smaller portion of the screen, see Renderer.setViewport.

void Renderer.setViewport(s16 px, s16 py, s16 width, s16 height, u16 renderQueue)

- Reduce the viewport to a smaller portion of the screen.

void Renderer.setGlobalComponentTint(s16 tintR, s16 tintG, s16 tintB, s16 addedR, s16 addedG, s16 addedB)

- Define the global tint color and added color, which will get applied to component sprites.
- For tintR/G/B, a value of 0x100 would be a neutral value that does not introduce any change to the colors.
- For addedR/G/B, these are values to be added to the respective 8-bit pixel channel value, after applying tint. You can make use of negative values here as well, to darken component sprites on the screen.

void Renderer.drawText(string fontKey, s32 px, s32 py, string text, u32 tintColor, u8 alignment, s8 spacing, u16 renderQueue, bool useWorldSpace)

- Draw text to the screen.
- Parameters:
  - fontKey is the name of the font to use, e.g. "smallfont". This is the font's file name without path or extension.
  - px and py define the anchor position where to draw the text.
  - text is the actual text string to draw.
  - tintColor is the color to apply when rendering the text, using 0xRRGGBBAA (red, green, blue, alpha) format.
  - alignment can be one of the following:
    - 1: Left / top aligned text (i.e. the anchor position at px, py is the upper left corner of the text)
    - 2: Horizontally centered / top aligned text
    - 3: Right / top aligned text
    - 4: Left aligned / vertically centered text
    - 5: Horizontally and vertically centered text
    - 6: Right aligned / vertically centered text
    - 7: Left / bottom aligned text
    - 8: Horizontally centered / bottom aligned text
    - 9: Right / bottom aligned text
  - spacing can be used to add additional space between characters. It's measured in pixels and can also be negative. Use the default value of 0 for standard spacing.
  - renderQueue defines when this text will get rendered in relation to other sprites and the planes.

- useWorldSpace = true interprets px/py as coordinates in the world, otherwise screen coordinates (in pixels from the upper left corner) are used.

Audio Playback

void Audio.playAudio(u64 sfxId)
void Audio.playAudio(u64 sfxId, u8 contextId)

- Starts playback of a sound effect or music track.
- Parameters:
  - sfxId can be a string referring to a sound ID as defined in "audio_replacement.json" (e.g. "1F_sonic3"), or a u8 value that will be mapped automatically to the respective hex code string (e.g. 0x1c will be interpreted as the string "1C").
  - contextId defines whether the sound is played as an effect or a music track, and uses the respective volume setting:
    - 0x00 for music
    - 0x01 for sound effects - this is the default if calling this function without a contextId parameter
- Some sound IDs use a defined channel, which means it can only be played once at a time. Starting a new sound of a channel will stop playback of other sounds using the same channel.

bool Audio.isPlayingAudio(u64 sfxId)

- Checks if a sound effect or music track with the given sound ID is already playing.

void Audio.stopChannel(u8 channel)

- Stops the sound / music using the given channel.
- Can be used to e.g. stop the music, which is usually taking channel 0.

void Audio.fadeOutChannel(u8 channel, u16 length)

- Fades out and afterwards stops the sound / music using the given channel.
- The length parameter is expressed in 1/256 seconds, so a value of 0x280 would refer to 2.5 seconds of fade-out time.

void Audio.fadeInChannel(u8 channel, u16 length)

- Fades in the sound / music using the given channel. This only really makes sense for sounds / music that are right in the middle of fading out.

void Audio.pushBackChannel(u8 channel)

- Pauses playback of the given channel, so that a different sound / music using the same channel can be started. When its playback stops, the old sound / music will fade in and continue.
- A use-case for this would be the extra life jingle in Sonic games, which are meant to pause but not stop the main music, and let it continue as soon as the jingle finished playing.

void Audio.enableAudioModifier(u8 channel, u8 context, u64 postfix, u32 relativeSpeed)
void Audio.disableAudioModifier(u8 channel, u8 context)

- These are highly specialized functions only added to handle sped-up versions of music tracks in S3&K. See usage there for examples.

Modding

bool Mods.isModActive(string modName)

- Checks whether a mod is active.
- The parameter is the mod's display name, i.e. the "MetaData" → "Name" field inside the mod's "mod.json" file.

s32 Mods.getModPriority(string modName)

- Returns the mod's loading priority as set by the user. This is 0 for the lowest prioritized mod, and increasing for the mods with higher priorities. For inactive mods or unknown mod names, -1 is returned.
- The parameter is the mod's display name, i.e. the "MetaData" → "Name" field inside the mod's "mod.json" file.

Undocumented

There exist some more functions that can be used, but there's no documention. You can find the full list of available functions in the Sonic 3 A.I.R. scripts under "_reference/cpp_core_functions.lemon".
To get idea what they're for and how they are used, you might find results by searching for their usage in scripts.

Rendering

Render Queues Overview

Most script functions meant for rendering something on the screen – e.g. sprites and planes – have a render queue parameter.
This is used to determine the order of rendering independent of the order of script calls.
Render queues in theory range from 0x0000 to 0xffff, with lower values getting rendered first, and higher values overwriting them. In other words, if you want to render something in fron of everything else, use a high render queue like 0xf000.

Common render queue values:

- 0x1000 for plane B non-prio layer
- 0x1800 is where background blur gets applied to everything in lower render queues
- 0x2000 for plane A / W non-prio layer
- 0x3000 for plane B priority layer
- 0x4000 for plane A / W priority layer
- 0x9000 .. 0xa000 for game object sprites
- 0xe000 for UI elements

Palette Sprites vs. Component Sprites

Palette sprites

- This term covers all custom sprites loaded from 8-bit BMPs, plus those created from game data (see Renderer.setupCustom*Sprite script functions).
- Palette sprites are internally stored as one 8-bit index per pixel, i.e. as indexed image. They won't work without a palette.

Component sprites

- These are loaded from 32-bit PNGs.
- Component sprites are internally stored in 32-bit ABGR format, i.e. with 8 bits per red, green and blue channel and an 8-bit alpha channel that usually will get alpha blended when being rendered.
- The game automatically applies a global tint color and an added color when rendering a component sprites. This is used for effects like fading from / to a black or white screen. This behavior can be disabled in the draw script call, see Renderer.drawCustomSprite.