Papyrus for Skyrim Guide

This document is a small guide to develop code in Skyrim using the internal language called Papyrus.

Author: CPU

Last Update: May 10th 2015

Contents

Introduction

What is Papyrus

Papyrus is the scripting system used inside the Skyrim game. Papyrus in not really a “scripting” language. It is not decoded in real time. It has to be compiled.

Papyrus is an object oriented language. You define your objects with inheritance, the properties, the functions, and then you can use them (similar to a super-simplified Java or Javascript or C#)

The source files of Papyrus have extension .psc

The compiled files have extension .pex

The name of the script is extremely important. Through the name the CK and Skyrim will know how to correlate the scripts and attach them to all objects inside the CK. (Of course names can be more or less whatever you want, with a max length of 80 characters, and no special characters.)

Papyrus is object oriented and is an “Imperative” language. Each item in the code will tell the game engine to do something. When completed the next instruction will be executed.

The reference guide for all script objects and function (including SKSE) is available at www.creationkit.com

To quickly get the full definition of a script object on the creationkit web site, just type the name of the object in the search tool, and add “script” at the end. (e.g. “Actor Script” will bring you immediately to the page describing the Actor script object.)

In these pages you will find all the properties, all the functions, all the events managed. In some cases also a few examples about how to use the object.

Conventions used in this document

The code is written in `Courier`.

Where something in the code is a generic statement then it is written in italic between angular brackets: *`<place here an example>`*

The script keywords are in blue and bold: **`while`** `(num > 0)`

Comments in the code are in green: `; This is a script`

Script Objects are listed in pink and bold: **`Actor`**

Examples of BAD code use the same formatting used for the normal code, but the lines have a strikethrough: ~~**`Actor`** `bad = 5 + 7`~~

When something important (usually a source of problems) is reported, a warning sign is added in front of the paragraph: ⚠

Where it is used

Papyrus scripts can be attached to about all the items of the CK. If you can attach a script, then in the CK there will be a special place in the edit window of the object to attach the script.

Scripting

Script logic

The very first line of all scripts is something like:

```
Scriptname <the name of the script> extends <another object>
```

The name of the script has to be the same name you used for the file.

The other object has to be one of the possible vanilla script objects or a script object provided by another mod, or script objects you defined with another script. It is also possible to define new scripts that extends nothing.

You can define inside the script "variables" that will hold the values you want.

```
Int num = 10

Float distance = 1000.0

String message = "This is a message"
```

There is no end of statement delimiter like in Java or Javascript. Every statement ends at the end of the line, and it is not possible to have two statements on the same line.[1]

You can define "functions" that are the methods available in your script to execute the logic.

```
Function doSomething()

      <some code here>

EndFunction
```

Functions can be called from other functions and also from other scripts is the script implementing them is referenced.

Some special "Functions" are called when an "Event" is generated by the game engine.

An Event usually starts with the preposition `On` and then the full name of the event.

The actual code inside the event is executed when the event is generated.

[1] It is possible to continue a statement on a new line if the last character of the previous line is the back-slash \

Papyrus has no code sections. Every statement that starts something has an equivalent statement to close it. (Similar to Visual Basic 6.)

`Function` is closed by `EndFunction`

`While` is closed by `EndWhile`

`Event` is closed by `EndEvent`

`If` is closed by `EndIf`

`State` is closed by `EndState`

Pretty much the closing statement is exactly the statement with and End at the beginning.

Papyrus is NOT case sensitive. The following variables are all the same:

Myvariable, MyVariable, myVariable, MYVARIABLE, mYvArIaBlE.

The list of all scripts can be accessed through the Creation Kit menu Gameplay→Papyrus Script Manager.

Only the scripts that are NOT marked as Hidden will be visible.

To have a script hidden (common for scripts that contains fragments) you can add the Hidden flag at the end of the first line of the script.

```
Scriptname myAwesomeInvisibleScript extends Actor Hidden
```

Hello World Example

From the www.creationkit.com

```
Scriptname HelloWorldScript extends Actor

Event OnInit()

      Debug.MessageBox("Hello World!")

EndEvent
```

If this code is attached to a NPC (the basic object is `Actor`) then, when the NPC is loaded for the first time the OnInit event is generated and the code inside the event is executed. It will show a popup window with the message: Hello World!

Let's analyze this basic code:

The first line is just declaring the script. Because the script extends the script object Actor, it can be attached to an actor (a NPC). The name of the script has to be *HelloWorldScritp.psc*

The code has just the definition for one event: OnInit. This event is generated only once when the script is initialized the first time.

There is just one line inside: `Debug.MessageBox("Hello World!")`

`Debug` is a global object that is used to call some special functions.

`.MessageBox` is a function provided by the Debug script object. The arguments of the functions are between parentheses.

`"Hello World!"` is a string that is the actual parameter.

To try this example, just attach this script to a NPC in the Creation Kit (edit the Actor, and in the Papyrus section create a new script. Make sure the script extends Actor.

Then copy/paste the code in the script (the name of the script has to match what is written on the first line.)

Save (this will compile if you are editing inside the CK.) and start Skyrim.

As soon as the NPC will be loaded the first time you will see the message box on the screen.

Papyrus Language

Commenting the code

Providing comments in code is always a good idea. For you and for the other people.

Papyrus allows comments at any line by using the semi-colon character `;`

Everything from this character up to the end of the line is considered a comment and ignored by the compile and the execution.

```
myNPC.Diable(True) ; Let’s get rid of the NPC by disabling it… Using the fade option here.
```

Another way to do comments is to insert them between brackets: `{ my comment }`. This is usually used to describe what a function you wrote does.

It is possible to do multi line comments by starting them with `;/` and ending them with `/;`

```
;/
  This is a multi-line comment
  Can be used if you want to write long descriptions.
  It is always good to add comments to your code.
/;
```

Keywords

These are the language keywords available in Papyrus.

`Function``(<parameters>)`: used to start a function definition. Will end with **`EndFunction`**.

```
Function pushAway(Actor anActor, Int amount)
      anActor.PushActorAway(anActor, amount)
EndFunction
```

Functions can return a value. To return a value the type of the script object returned has to be specified before the **`Function`** keyword, and the value has to be returned through the **`return`** keyword.

```
Cell Function getPlayerCell()
      Return Game.getPlayer().getParentCell() ; This code is not really efficient, do not use it.
EndFunction
```

`If` *<condition>* `ElseIf` *<condition>* `Else` `EndIf`: Used to execute a statement only if a condition is true.

```
If PlayerRef.getItemCount(Gold001) > 1000
	Debug.Notification("You are rich")
ElseIf PlayerRef.getItemCount(Gold001) > 200
	Debug.Notification("You have some money")
Else
	Debug.Notification("You need to do some adventures…")
EndIf
```

The condition of an If statement has to evaluate to true or false. It does not need to be put inside parentheses, but if the condition is inside parentheses they will not impact performance and may improve readability.

When there is an `ElseIf` statement, it will be executed ONLY if the previous if conditions are false and the current one is true.

The `Else` statement will be executed ONLY if ALL previous if conditions are false.

`While <condition> EndWhile`: is used to execute the code inside the statement until the condition is true.

```
Int numTimes = 1
While (numTimes <= 10)
	Debug.Notification("This is the iteration " + numTimes)
	numTimes += 1
EndWhile
```

This code defines an integer variable numTimes and assign the value 1 to it.

The while statement will run until the variable numTimes is less or equal to 10.

The body of the statement will print a notification on the screen (This is the iteration 1, This is the iteration 2, … , this is the iteration 10.) And then will increase the variable *numTimes* by 1.

As soon as the variable *numTimes* will reach 11 the cycle will end.

The parentheses around the condition are not required. But may improve the readability and will be stripped out by the compiler.

Sadly in Papyrus the `While` is the only cycle statement. The "`For`" statement is not available.

Operators

Here the description of all the operators available in Papyrus.

Parenthesis: `()`

When used in conditions and expressions, the parentheses allows to give priority in the evaluation.

```
Int result = (1 + 1) * 15
```

(1 + 1) will be evaluated before multiplying the result by 15.

The parentheses are used also to specify and call `Function`s.

```
Bool Function doSoething(int aNum, Actor aNPC)
```

Will define a function.

```
Bool res = doSomething(15, PlayerRef)
```

Will call the function.

Square Brackets: `[]`

Square brackets are used to define and index arrays. Put after a Script Object without anything inside will define an array of script objects. Put after an object script preceded by the new keyword and a static number inside the square brackets will define the size of the array. Used after a variable with an integer variable or an integer number will index the array to get the actual item.

```
Actor[] myListOfActors = new Actor[16]
```

Defining the array variable and initializing the array with 16 elements.

`Actor` aDude = myListOfActors[5]

Getting the element in position 5 (this is the 6th element!) of the array.

Comma: ,

The comma operator I used to separate the parameters in a function and events definition, or on function calls.

```
Function doSomething(Int param1, Int param2, String text)
```

The comma is used to separate the three parameters in the function definition.

```
doSomething(1, 2, "Hello")
```

The comma is used to separate the three parameters.

Math Operators: `+ - * / %`

Math operators are used to calculate an expression.

+ is used to add numbers and to concatenate string.

- Is used to subtract numbers and negate value (unitary minus.)

* is used to multiply numbers.

/ is used to divide numbers.

% is the modulo of two numbers.

Division and modulo by zero will produce an error in the error log.

If the values are both an integer then the result is an integer. If any of the values is a float the result is a float.

Dot: .

The dot operator is used after a variable to get a property or call a function.

```
myObject.aFunction()
```

```
myObject.name = "Bendu Ono"
```

Double Quotes: `""`

Double quotes are used to define strings.

Logical Operators: `!  ||  &&`

A logical operator evaluates to true or false.

`!` is the “not” operator and will transform False to True and True to False. A None object, and the numbers 0 and 0.0 are equivalent to False.

|| is the “or” operator. It is true when at least one of the expressions are true.

&& is th e”and” operator. It is true when all the expressions are true.

While evaluating conditions it is possible to short circuit the evaluations. The items are evaluated from left to right. As soon as the condition is fully determinate, then the remaining part is ignored.

```
If (anActor && anActor.getDistance(PlayerRef) > 100)
```

Is a good way to handle the checking in case the `Actor` anActor is null (equals to `None`.)

In case the variable anActor is `None`, then the expression is false for sure, and the second part of the expression will not be considered.

The following code:

```
If (anActor.getDistance(PlayerRef) > 100)
```

May produce errors in case myActor is `None`.

Comparison Operators: `==  !=  <  >  <=  >=`

Equality (==) is true if both expressions have equal values.

Inequality (!=) is true if expressions are different.

Less-than (<) is true if the first expression is smaller than the second expression. If the two are equal it returns false.

Greater-than (>) is true if the first expression is larger than the second expression. If the two are equal it returns false.

Less-than or equal to (<=) returns true if the first expression's value is smaller than or equal to the second expression's value.

Greater-than or equal to (>=) returns true if the first expression's value is larger than or equal to the second expression's value.

If floating-point values are being compared, a tiny epsilon value is accounted for to counteract floating-point error: `1.000000013 == 1.000000012` is true.

Assignment: `=`

Assignment operator assigns the value from the right expression to the variable (or property) from the left expression.

```
Int num = 10
```

The assignment operator can be used in Function definition to provide a default value for a parameter

```
Function doSomething(Actor npc, Actor anotherOne = None, int numOfTime = 3)
```

In this case is possible to call the function with just

doSomething(anActor)

The other two parameters will get the default value.

When calling a function is possible to name a parameter and use the assignment operator to provide the actual value:

```
doSomething(anActor, numOfTimes = 7)
```

Assignments with Operation: `+= -= *= /= %=`

These operators will quickly execute an operation on the variable on the right and then assign the result to the variable itself.

`num += 1` is equivalent to `num = num + 1` (but it is way faster.)

These operators will not work on arrays.

~~`anArray[7] += 1`~~

Will not compile.

Cast: `as`

The cast operator attempts to cast the value from the left expression to the type to the right of it, and returns the resulting value.

```
Float f = 12.3
Int i = f as int
```

Operator Precedence

()	Mathematical parenthesis
.	Dot operator (for accessing functions and properties)
As	Cast operator
- !	Unary minus and logical NOT
* / %	Multiplication, division, and modulus
+ -	Addition and subtraction
== != < > <= >=	Comparison operators
&&	Logical AND
||	Logical OR
= += -= *= /= %=	Assignment

Operator precedence is listed in the table above. The operators higher in the table will have priority on the operators below.

Objects

Papyrus as a few basic types and a huge amount of script objects that can be used. New Script objects can be defined by the modders.

Basic Types

The basic types available are:

`Int`: a signed integer number, can contain numbers from -2^{31} to $2^{31}-1$

`Float`: a floating 32 bits number. They are numbers with decimals. The dot is used to separate the mantissa from the decimal part. *123.456*

`String`: a sequence of characters. They are defined by starting them with a double-quote and ending them with another one. *"This is a string"*

`Bool`: a simple variable that can be only True or False. Any item that is `None` an empty string or a value of 0 or 0.0 will evaluate to False. Any other object will evaluate to True.

`None`: a null script object. Used when a variable that expect an object has no values inside.

Script Objects

Aside from the basic types all script objects can be used to define a variable. Some items defined by the game are[2]:

`Actor`: this script object represent all NPCs in the game, including creatures. This is NOT the definition of the NPC, is a NPC inside a location. This class inherits from `ObjectReference`.

`ActorBase`: this is the basic definition of a NPC, more or less what you see in the Creation Kit inside the Actor tab. To get the `ActorBase` from an `Actor` you can call the function `myActor.GetActorBae()` or `myActor.getLeveledActorBase()`

`Armor`: all types of clothes, armors, and wearable objects are referenced by this script object.

`Book`: represents the books, the notes, the cards, etc.

`Cell`: represent a cell defined in the game, can be an Interior cell or a part of the Worldspace. All `ObjectReference`s are always inside a cell, and the cell can be get with the function `myRef.getParentCell()`

`Debug`: this is a special script object that can be very useful to provide notifications and to execute very useful functions like `aNPC.SendAnimationEnvent("IdleApplaude2")`

`Door`: any kind of door and portcullis that can be open and closed.

`Faction`: special object used to give specific behaviors to NPCs.
`myFriend.addToFaction(CurrentFollowerFaction)`

`Flora`: plants and trees.

`Form`: the very basic Script object. All other objects will inherit from this one.

`Game`: a special object that can be used to get and set something in the game.
`Debug.Notification("The number of mods you have is: " + Game.GetModCount())`

`GlobalVariable`: used to get a global game variable that can be read and written.

`Key`: all keys used in the game by the player.

`Keyword`: a special item that is added to other object to quickly select them or to do some quick checking (`myNPC.hasMagicEffectWithKeyword(aKeyword)`)

`Math`: a special object that provides some mathematical functions (`sin`, `cos`, `tan`, `abs`, etc.)

`MiscObject`: Used to reference some common objects. One notable example is Gold001 that is used to reference the gold of the players, NPCs, and containers.

`ObjectReference`: this Script Object is used to reference all objects that are placed inside a Cell. They can be actors, doors, statics, etc.

`Package`: used to reference the packages that are associated to Actors.

[2] This is NOT an exhaustive list!

`Potion`: used to reference all the potions.

`Quest`: the script object used to reference and manipulate quests.

`Race`: used to reference races. `Debug.Notification("The race of Bendu Olo is" + benduOlo.getActorBase().getRace().getName())`

`ReferenceAlias`: used to reference the aliases used inside the quests. `myRefAlias.forceRefTo(myActor)`

`Scene`: used to start and stop scenes by code. `myScene.ForceStart()`

`Spell`: used for all kind of magic. `Fireball.cast(theTarget, PlayerRef) ; The player will throw a fireball to the target.`

`Utility`: a very useful object for some very common functions:

- `Utility.wait(5.0) ; stops this script for 5 real-time seconds.`
- `If (Utility.randomInt(1, 100) < 50) ; executes the statement only in 50% of cases.`

Arrays

Papyrus supports also arrays. An array is a set of objects all with the same type.

They are defined by a couple of square brackets, and the objects inside can be accessed by defining the object number inside the square brackets.

```
Actor[] myArrayOfActors = new Actor[10]

myArrayOfActor[0] = PlayerRef

myArrayOfActor[1] = benduOno
```

To create the array you need the keyword `new`. And you need to specify the size of the array.

Sadly the size has to be a static number. Arrays in Papyrus are NOT dynamic.

This will fail:

`Int mySize = 19`

~~`Actor[] myArray = new Actor[mySize]`~~

Arrays can have as maximum size 128 elements. The first entry of an array is always the position 0 (zero).

Arrays offer two really useful functions:

`.find()` and `.length`

`anArray.find(<object to find>)` will return the position of the `<object to find>` inside the array. If the object is NOT found then the result will be -1.

`.length` will simply return the length of the array. Useful when you receive the array from another function and then you have to do something on all elements of the array.

Object inheritance

All the script objects of Papyrus support inheritance.

What is the inheritance? Think about this: you have a dog. A dog is a mammal, so it will have something that came from the mammals. Mammals are animals with a spine (back bone), so mammals will get something from the "Animals with a spine" (the back bone.) At the end the dog has a back bone, because it is a mammal and a mammal has a spine.

Let's have a look to some real Papyrus objects.

The main object is called Form. If you look at the Form object you will see that it implements a function called .getName(); this function will provide the name of the object.

Another object is called ObjectReference. ObjectReference represent something inside a cell (an Actor, a furniture, a static, etc.).

ObjectReference inherits from Form. If you look at its definition (Data\Scripts\Source\ObjectReference.psc), you will see this:

```
Scriptname ObjectReference extends Form Hidden
```

The keyword **`extends`** means that the object defined in the script inherits from the other object.

`ObjectReference` inherits from **`Form`**.

And so you can call the functions and the properties of a **`Form`** over an **`ObjectReference`**.

`myObjectRef.getName()` is fully valid. Also if **`ObjectReference`** does not actually define this function.

`Actor` is another object, and it inherits from **`ObjectReference`**. So if you want to call the function `.Enable()`, that is defined by **`ObjectReference`** over an **`Actor`** you can!

The same applies to events. **`Form`** defines an event called `OnUpdate`. You can use this event to all script objects (they all inherits from **`Form`**.)

Transforming objects

Basic types can transform automatically one to another.

`Int` can be transformed to `Float` and vice versa. Of course some rounding will happen.

`Int`, `Float`, and `Bool` can be transformed to `String`s. `Bool` will transform to TRUE and FALSE. `String`s cannot be converted to numbers.

Some function (many of them) define parameters. For example the `Actor` defines this function:

```
Bool Function HasLOS(ObjectReference akOther)
```

This function will return true if the actor can see (LOS = Line of Sight) the other `ObjectReference`.

Now, suppose you have an `Actor` called benduOno, and you want to check if he has the line of sight to the player (PlayerRef is the common name for the player actor).

If you write ~~`if (benduOno.HasLOS(PlayerRef))`~~ you will get a compile error.

Why? Because PlayerRef is an `Actor`, not an `ObjectReference`. But `Actor` inherits from `ObejctReference`, so you can "cast" the `Actor` to an `ObjectReference`.

This code will work:

```
Actor benduOno = …

Actor PlayerRef = …

If (benduOno.HasLOS(PlayerRef as ObjectReference))
```

The "`as`" keyword allows to transform a script object to another. Of course only if the cast is possible.

This will miserably fail:

```
Actor PlayerRef = …

Door myDoor = PlayerRef as Door
```

One very common casting case is the function getNThRef() defined by the Cell script object. It will return an nth occurrence of an ObjectReference of the specified type.

For example this will list all the NPCs on a cell.

```
Cell myCell = …

Int num = myCell.getNumRefs(62)

While (num)

      num -= 1

      Actor act = myCell.getNThRef(num, 62) as Actor

EndWhile
```

Extending scripts

You can extend any available script to create a more specialized one. The new script may add functions and replace functions from the extended one.

Let's start with our main script:

```
Scriptname ParentScript

String Property name Auto

String Function getName()
      Return name
EndFunction
```

Now let's create another script that will extend the previous one:

```
Scriptname ChildScript Extends ParentScript

Int Property age Auto

Int Function getAge()
      Return age
EndFunction
```

Now, inside the ChildScript you can call both getAge() and getName(). getAge() is defined by the child script, while getName() is inherited by the parent.

The child functions can overwrite the parent ones:

```
Scriptname ChildScript Extends BaseScript

Int Property age Auto

String Function getName()
      Return Parent.name + " age= " + age
EndFunction
```

In this case calling `getName()` on the child will not invoke the parent function but the child function.

There are some special objects that you can use to handle this:

`Child` → The script that is extending its parent.

`Parent` → The original script that is being extended.

Inside the ChildScript, then you can call **`Parent`**`.getName()` to execute the function defined by the parent script.

A child script has access to all the functions and events from its parent, also the one defined in “States”. Any function or event that you do not implement in the child script is inherited by the child. If a function or event is implemented in the parent, and you implement it in the child as well, then the child one overrides the parent, and any calls to that function or event will use the child's version instead. Note that when you implement the function or event in the child you must match the parameters and the return type, along with the name. If you don't the script will not compile.

Casting and inheritance

Imagine that you have a set of Actors, and you wish to define what happens when they got hit.

There is an Event that is generated when an Actor (better an ObjectReference) is hit. The event is:

```
Event OnHit(ObjectReference aggr, Form src, Projectile prj, bool pattk, bool
snattk, bool baattk, bool hitb)
```

Now, let's imagine that you have three types of NPCs: cowards (they will run away), injured (they will die immediately), feral (they will attack back).

Let's create three different scripts extending Actor:

```
Scriptname cowardNPC extends Actor

Event OnHit(ObjectReference aggr, Form src, Projectile prj, bool pattk, bool snattk, bool
baattk, bool hitb)

      Debug.Notification("I will run away.")

EndEvent


Scriptname injuredNPC extends Actor

Event OnHit(ObjectReference aggr, Form src, Projectile prj, bool pattk, bool snattk, bool
baattk, bool hitb)

      Debug.Notification("I will die right now.")

EndEvent


Scriptname feralNPC extends Actor

Event OnHit(ObjectReference aggr, Form src, Projectile prj, bool pattk, bool snattk, bool
baattk, bool hitb)

      Debug.Notification("I will fight back.")

EndEvent
```

And now you attach these three script to three different Actors in CK.

When one of the actors will get hit, the appropriate message will be shown. Simple and easy.

Now, imagine that you have another script that will "hit" the NPC by code:

```
Actor akTarget = …

akTarget.OnHit(…)
```

In this case also if the script object is `Actor`, the specific implementation will be used based on the current script attached to the actual instance of the actor. You do not have to worry about it.

Properties

Properties are special variables that can hold a basic value or a reference to an object.

The associated values can be defined through the “Properties” button that is available on scripts.

The format to define a property is to define first the type of the object, then use the keyword `Property`, and then the name of the variable.

```
Int Property aNumber
```

Property variables can have two functions to set and read the value. They can be defined just after the property.

```
Int numTimes = 0 ; Here something can be stored internally

Int Property aSpecialNumber

      Int Function get()

            If (numTimes < 10)

                  numTimes += 1

                  Return Utility.RandomInt(0, 100)

            Else

                  Return Utility.RandomInt(0, 10)

            EndIf

      EndFunction

      Int Function set(Int value)

            numTimes = 0 ; To reset

      EndFunction

EndProperty
```

To close the definition of the property, when the getter and setter are defined, you have to use the `EndProperty` statement.

If the get() function is not defined, then the property will be write-only.

If the property will just store and provide a value, without doing anything specific, then the automatic getters and setters can be added automatically through the `Auto` keyword.

```
Int Property aNumber Auto

Float Property aFloat = 12.5 Auto
```

Will allow to read and write automatically the property, without requiring to define the get() and set() functions.

A property can be a constant static value. A value of such properties will never change and cannot be updated. The keyword `AutoReadOnly` has to be added to the end and a value has to be provided.

```
Int Property clothesSlot = 32 AutoReadOnly
```

A special note is for Conditional Properties.

A Conditional Property is a property that can be used inside the Conditions of CK objects (Topics, Dialogues, Quests, Packages, etc.) as argument.

Using these special properties allows to dramatically reduce the number of `GlobalVariable`s required to make the different parts of a mod collaborate.

To use conditional properties the script itself has to have the Conditional keyword at the end of the first line. Then the Conditional keyword can be added to the end of the properties that will be used in external conditions. Warning only properties that have the `Auto` flag can be defined as `Conditional`.

```
Scriptname myMainQuestScript extends Quest Conditional

Int Property numFollowers Auto Conditional
```

Now the property numFollowers can be used as conditional statement. The condition should be of type *GetVMQuestVariable*, then you can select the quest and then you can pick the actual variable (in the list they will have :: just before the name.)

Setting the property values

⚠ Please remember that just defining a property in the code WILL NOT set the value inside.

After adding a property to a script you ALWAYS need to go in the edit properties window and fit the value of the property.

When you add a property that is an actual script, in the fill value you will not see the script but the object where the script is attached (a quest, an actor, etc.) That is normal.

To benefit of the auto-fill capability, you need to name the property variable exactly as the object that the property is referring. Works in about 50% of cases but it is better than always getting the object by hand.

Removing a property

⚠ Please note that one main source of problems in mods is when a new version of a mod has one property removed from a script, and the removal was not done correctly.

To remove a property from a script you MUST first empty the property value, close the edit window of the object, and then you can go back and edit the code by removing the property itself.

At load time, if the mod was in the save game, the player will see a warning message, but it will disappear in the next save.

Functions

Functions are used in Papyrus to do all the magic[3].

A function is defined by the keyword Function, and the list of parameters between parentheses.

A function may return a value, in this case the type of the function has to be specified before the Function keyword.

```
Int Function getSomeNumbers(Actor aNPC)
	If (!aNPC)
		Return 0
	EndIf
	Return aNPC.getLevel() * aNPC. GetSitState()
EndFunction
```

A function may return a value. To specify the type of the return value, you have to put the type before the Function keyword.

When a function returns a value is mandatory to have the `Return` statement(s) inside the function body.

When a function is expected to return a value, it has to end by returning a value. Or you will receive a compile error.

A function may have also parameters. To specify the parameters you have to define, for each of them the type and then the name of the variable that will hold the parameter. Multiple parameters can be added by separating them with the comma (`,`).

Parameters may also have a default value. To specify a default value, after the name of the parameter you can add `=` *`<default value>`*.

```
Function pushAwayFromMe(Actor npc, int howFar = 1)
	PlayerRef.pushActorAway(npc, howFar)
EndFunction
```

[3] I am not referring to Spells, or Magic Effects… ;-)

You can call the function either:

```
pushAwayFromMe(benduOlo)

; This will make it fall without big consequences (the hit will be equal to
1)


pushAwayFromMe(benduOlo, 500)

; This will throw him very far away, and he probably will die…
```

A Function defined by a mod must be always ended by the `EndFunction` keyword. An exception is on `Native` functions that are exposed by the Game Engine or by SKSE.

A `Native` function has no Papyrus code, the code is implemented internally by the game or by a plugin. The definition of the function with its parameters is used to make it accessible to other papyrus scripts.

Example from Actor.psc (SKSE 1.7 version):

```
; Returns this actor's race

Race Function GetRace() Native
```

It is possible to specify "Global" functions. These functions can be called directly without requiring the object to be specified. Of course such functions cannot use any other variable or property that is defined outside the function. They are similar to Java "static" functions.

```
Int Function getMinutesFromGameHour(Float gamehour) Global

	Return 60.0 * (Math.floor(gamehour * 100.0) - (Math.floor(gamehour *
100.0) % 100)) / 10000.0

EndFunction
```

Function in a script that extends another script can overwrite the same functions from the parent script. The type, the name, and the parameters have to match.

Events

Events are a special type of function that is called automatically by the game engine when the corresponding event will happen in the game.

Events function names follow the rule: `On<Event name>(<parameters>)`

Because events are actual functions, they share exactly the same definition (including the options, like `native`) and logic.

An event will NOT return a value.

Event functions can be called as regular functions by other scripts.

An Event definition is closed by the `EndEvent` keyword.

```
Event OnHit(ObjectReference akAggressor, Form akWeapon, Projectile
akProjectile)

        If (akAggressor)

                Debug.Notification("Damn you," + (akAgressor as
Actor).getActorBase().getName())

        endIf

EndEvent
```

Warning, the definition of events, including the parameters is extremely important. It is safer to copy/paste the event definition from the actual object that you are extending (`Actor`, `ObjectReference`, etc.) or from the pages of the Creation Kit wiki (www.creationkit.com/Category:Events).

For some events you do not need to do anything special. If the event function is defined, and the event happens the function will be called. (Like the `OnInit()` event.)

For other events you need to register to have the event function called.

Here is a list of common events[4]:

Event	Script Object	Description
OnActivate	ObjectReference	Event called when the object reference is activated.
OnAnimationEvent	Form	Event called when the active magic effect/alias/form receives one of the animation events it was listening for. The receive such events the script has to register for them: `RegisterForAnimationEvent(aNPC, “IdleFornitureExit”)`
OnClose	ObjectReference	Event called when the object has finished closing. (Like a door that has finished animating shut, but not a book that has finished closing.)
OnDeath	Actor	Event called when the actor dies.
OnEffectStart	ActiveMagicEffect	Event called when a magic effect is starting on a specific target.
OnEffectFinish	ActiveMagicEffect	Event called when a magic effect ends on a specific target.
OnObjectEquipped	Actor	Event called when the actor equips an object, both the base objects and the reference are passed as parameters.
OnGainLOS	Form	Event called when an actor has [recommended: only for the first time] a Line of Sight (LOS) to the target object. To receive this event is require to register: RegisterForSingleLOSGain(PlayerRef, woderfulWeapon)
OnHit	ObjectReference	Event called when the object reference is hit by a weapon or projectile.
OnInit	Any and all scripts	Event called when the script has been created and all its properties have been initialized.
OnItemAdded	ObjectReference	Event received when an item is inserted into a container.
OnItemRemoved	ObjectReference	Event received when an item is removed from a container.
OnLoad	ObjectReference	Event called when the object's 3d is loaded and ready.
OnMagicEffectApply	ObjectReference	Event called when a magic effect is about to be applied to the object reference.
OnObjectEquipped	Actor	Event called when the actor equips an object.
OnObjectUnequipped	Actor	Event called when the actor unequips an object.
OnOpen	ObjectReference	Event called when the object (a door a portcullis, etc.) has finished opening.
OnPlayerLoadGame	Actor	Event called when the player loads a save game. This event is only sent to the player actor.
OnSit	Actor	Event called when an actor sits on a piece of furniture.
OnSleepStart	Form	Event called when the player goes to sleep. It is required to register for this event: RegisterForSleep()
OnSleepStop	Form	Event called when the player wakes up
OnTriggerEnter	ObjcetReference	Event called when the object reference is a trigger volume and has been entered.
OnTriggerLeave	ObjcetReference	Event called when the object reference is a trigger volume and has been leaved.
OnUpdate	Form	Event called from time to time. It is required to register to it with the number of seconds to wait for the update. RegisterForSingleUpdate(float seconds)

[4] For the full list of known events please refer to the CreationKit.com web site.

Managing Events

If you need to do something to initialize your mod, one of the best event you can use is OnInit().

On your main quest add a script like this:

```
Scriptname myQuestScript extends Quest

Int Property amIActive = 0 Auto

Event OnInit()
	amIActive = 1
	Debug.Notification(“My Mod is active.”)
EndEvent
```

The OnInit event will be generated the first time the script is initialized. You can do all required initializations here. Warning: it will be called only once.

If you want to do something every time the game with your mod is started, you can use the OnPlayerLoadGame() event. This event works only if it is attached to the Player actor or to a ReferenceAlias attached to the player. Consider creating a Reference Alias to your main quest, and specifying the alias to be a single reference to the player. Then you can attach this code to the ReferenceAlias.

```
Scriptname myModManagementScript extends ReferenceAlias

Int Property myModVersion Auto
Actor[] arrayFromVersion200
Armor[] armorsFromVersion300

Event OnPlayerLoadGame()
	If (myModVersion < 200)
		arrayFromVersion200 = new Actor[16]
		myModVersion = 200
	endIf
	If (myModVersion < 300)
		armorsFromVersion300= new Armor[32]
```

```
                myModVersion = 300
        endIf
EndEvent
```

Consider a main script of your quest that will do something from time to time, or a Magic Effect that will also do some checking and actions from time to time. This can be achieved by using the OnUpdate() event. To receive this event, you should register for it.

```
Scriptname myQuestMainScript extends Quest
Actor Property PlayerRef Auto
MiscObject Property Gold001 Auto
Event OnInit()
        RegisterForSingleUpdate(1.0) ; We register and the OnUpdate will be
called after 1 second
EndEvent
Event OnUpdate()
        If (PlayerRef.getItemCount(Gold001) < 1000)
                ; The player has now less than 1000 gold, let’s wait another
minute
        `       RegisterForSingleUpdate(60.0)
                Return ; We will not do anything else for now…
        EndIf
        Debug.Notification(“You got rich!”)
        ; Do not register again, the event will not be sent anymore
EndEvent
```

When the quest script will be initialized the first time, we call RegisterForSingleUpdate(1.0), this will trigger the OnUpdate event after one second wait.

Then inside the OnUpdate we check if the player has more or less 1000 gold. If he/she has less than 1000 gold, we register again for the same event, with a one minute wait (60 seconds.)

If he/she has 1000 gold or more we display a message and then we will NOT register again.

At the end the code will be run every 60 seconds until the player will have 1000 gold or more.

Another example is done on a furniture that can be activated. For example a PullChain.

```
Scriptname myPullChainManagerScript extends ObjectReference


Event OnActivate(ObjectReference activator)

      Debug.Notification("The pull chain was activated by: " + (activator as
Actor).getActorBase().getName())

EndEvent
```

As soon a NPC or the player will activate the object, then the event will be triggered.

Suppose you want to start doing something when the player enters a specific area. You can create a trigger and attach to it a script like this:

```
Scriptname myTriggerManagerScript extends Activator

Actor Property PlayerRef Auto

Event OnTriggerEnter(ObejctReference akActionRef)

      If (akActionRef != PlayerRef)

            Return ; We want only the player…

      EndIf

      PlayerRef.unequipAll() ; Get it naked…

EndEvent
```

By doing this, when the player will enter the trigger area, it will be stripped of everything.

Custom Events (using SKSE)

SKSE offer a way to create custom events for your mod. The events create through SKSE can be handled between different mods.

First you have to create your event function, as usual.

```
Event OnYourEventManagerName(String eventName, String strArg, Float numArg,
Form sender)

	; Do something here

EndEvent
```

The arguments that will be sent have to be defined exactly in the way described.

After defining the event manager function, you can now register it to actually handle the events.

```
RegisterForModEvent(String eventName, String yourEventMamangerName)
```

The event can now by triggered by other scripts (also from other Mods) with the function:

```
SendModEvent(String eventName, String strArg = "", Float numArg = 0.0)
```

Let’s imagine that you want to define an event that will be triggered to simulate a “call” from a NPC.

```
Scriptname myMainQuestScript extends Quest


Event OnInit()

	RegisterForModEvent(“SomebodyCallsEvent”, “OnSomebodyCallsMe”)

EndEvent


Event OnSomebodyCallsMe(String eventName, String strArg, Float numArg, Form
sender)

	If (eventName == “SomebodyCallsEvent”)

		Debug.Notification(strArg + “ is calling me”)

	endIf

EndEvent
```

Now in a different script you can send these events:

```
SendModEvent("SomebodyCallsEvent", aNPC.getActorBase().getName())
```

This will make a call to your event management function by passing as parameter your event name, and the name of a NPC.

The key here is to have the second parameter of RegisterForModEvents to be exactly what your event management function is.

And then use in SendModEvent as first parameter exactly the event name you defined as first parameter of RegisterForModEvents.

States

You can define different states for your script. Each state may have its own functions and events; different states of the same script may define the same functions and events. Of course with different code inside.

A script can be in one and only one state in a point in time.

To define the states of a script you can use the keyword `State`. And place all the Functions and Events inside the block `State` / `EndState`.

```
State aState
  ; Functions and Events
endState
```

Each script has also a so called "Empty state". The functions and the events that are defined outside all states go in this default "Empty state".

```
Scriptname statesExampleScript
Function aFunction()
      Debug.Notification("The script is in the empty state.")
EndFunction

State myState1
      Function aFunction()
            Debug.Notification("The script is in the state: myState1.")
      EndFunction
endState

State myState2
      Function aFunction()
            Debug.Notification("The script is in the state: myState2.")
      EndFunction
endState
```

Note that the function aFunction() is defined three times. This is possible because the function definitions are inside different states.

If you call the aFunction() function, the actual function that will be called will depend on the actual state of the script.

The same function defined in different states must have always the same type and the same parameters.

Every function that is defined in a state has to be defined also in the empty state. If the function is not required then the code block of the empty state function can be left empty.

This does not apply to events. They are defined already by the compiler, without requiring you to define them again in the empty state.

To set the current state for a script you can use the function `GoToState()`. This function has only one mandatory argument that is a string that identifies the state you want to assign to the script.

```
GoToState("myState1")
```

Calling the function `GoToState` will trigger two events: `OnEndState` and `OnBeginState`.

`OnEndState` is sent to the event defined in the previous state the script was in.

`OnBeginState` is sent to the event defined in the new selected state.

```
Scriptname myStatefulScript

State myState1
        Event OnEndState()
                Debug.Notification("Ending state 1")
        EndEvent
EndState

State myState2
        Event OnBeginState()
                Debug.Notification("Starting state 2")
        EndEvent
EndState
```

If you call GoToState(“myState1”) you will not receive any notification, but then when you call GoToState(“myState2”) you will receive both notification.

One because the state1 is ending, and another because the sate2 is starting.

If you want to assign a default state to the script you can add the Auto keyword at the begin of the state definition.

```
Scriptname myStatefulScript


Auto State myDefaultState

	; Blah, blah, blah

EndState


State anotherState

	; Blah, blah, blah

EndState
```

This script, when it starts, will start in the state myDefaultState.

How the function/event is selected?

1. If there is a definition for the function/event in the current state of the script it will be executed.
2. ElseIf there is a definition for the function/event in the current state of the extended script it will be executed.
3. ElseIf there is a definition for the function/event for the empty state of the script it will be executed.
4. ElseIf there is a definition for the function/event for the empty state of the extended script it will be executed.

The current state of a script can be get through the GetState() function.

Scripts and Fragments

About all main objects that can be defined in CK can have a script attached to them (also more than one.)

Some objects (Topic infos, Packages, Scenes …) may also have fragments of code. A fragment is just a small piece of code that is executed on specific conditions. There is no need to define a function or an event for fragments. It will be generated automatically by the Papyrus compiler.

Scripts attached to objects

A script attached to an object need to have the full definition, including a name. The first line has to start with the keyword **`Scriptname`**.

These scripts support Properties, Functions, Events, States. Just define and use them.

Package Fragments

When you edit a Package you will see three small edit boxes to edit three fragments that can be attached to the package.

One fragment will be run when the package starts, one when the package ends, one when the package changes.

When you close the package a script will be attached to it. The script will have a special structure and will define functions/events inside.

Inside these functions/events body you will see the code you put in the edit window.

By default a special **`Actor`** script object is passed to the fragment code. It will contain the NPC that is executing the package. The name of this variable is `akActor`.

Topic Fragments

A topic fragment will be attached to a topic. Two fragments are defined: one that is executed when the NPC start saying something, and another is executed when the NPC ends talking.

The convenience **`Actor`** script object is passed to the fragment code. It will contain the NPC that is speaking. The name of this variable is `akSpeaker`.

Scene Fragments

A scene fragment is attached to a scene. In this case multiple fragments are stored inside the generated script. Two main fragments are used when the scene starts and when the scene ends. You can define them in the “Edit Data” window of a scene.

If you add actions to the scene (you usually do…) then you can add fragments to the actions. These fragments will be stored in the same script of the scene (and the can share some common property, variable, and function.)

Adding properties and extra functions to script fragments

If you edit the script created by CK for the fragments, you will see something like this:

```
;BEGIN FRAGMENT CODE - Do not edit anything between this and the end comment

;NEXT FRAGMENT INDEX 2

Scriptname mycode_PF_blahblah_0402F412 Extends Package Hidden


;BEGIN FRAGMENT Fragment_1

Function Fragment_1(Actor akActor)

;BEGIN CODE

Debug.Notification(“NPC is ” + akActor.getActorBase().getName())

;END CODE

EndFunction

;END FRAGMENT


;END FRAGMENT CODE - Do not edit anything between this and the begin comment
```

The actual code you wrote in the edit field is stored inside the function “Fragment_1” (when multiple fragments are possible each will be named with a different number.)

After the last comment you can add your own stuff: extra functions, variables, properties, etc.

Let's imagine that you have a Scene, and in the scene a few actors will participate by doing something. When the scene ends you may be interested in how long the scene lasted.

You can achieve this by adding a fragment to the scene (just start by putting a single ; in one fragment edit box, and then close and re-open the scene.)

Then edit the full script and you will get something like:

```
;BEGIN FRAGMENT CODE - Do not edit anything between this and the end comment
;NEXT FRAGMENT INDEX 2
Scriptname mycde_SF_myScene_0405527D Extends Scene Hidden

;BEGIN FRAGMENT Fragment_0
Function Fragment_0()
;BEGIN CODE
;
;END CODE
EndFunction
;END FRAGMENT

;BEGIN FRAGMENT Fragment_1
Function Fragment_1()
;BEGIN CODE
;
;END CODE
EndFunction
;END FRAGMENT

;END FRAGMENT CODE - Do not edit anything between this and the begin comment
```

Now add at the very end a float variable and a property:

```
Float startedAt
GlobalVariable Property GameHour Auto
```

And then edit the fragments by adding the following code.

Begin Scene Fragment:

```
startedAt = GameHour.getValue()
```

End Scene Fragment:

```
Float timePassed = GameHour.getValue() - startedAt

Debug.Notification("The scene lasted for " + (timePassed as int) + " game
hours and" + (timePassed - (timePassed as int)) * 60.0)
```

When the scene will end you will get the notification about how long (in game time) the scene lasted.

The fragment function can share the global variables and all the properties defined.

Advanced

This chapter is for modders that have already some level of knowledge with Papyrus.

- Performance tricks is a set of recommendations to improve the speed of the code.
- Advanced features is to use some really advanced methods to execute something not ordinary.
- PapyrusUtils is a reference guide for this amazing toolkit that extends what you can do with Papyrus.
- MCM is a reference guide to develop MCM menu in SkyUI.
- Examples is a list of functions that can be used as is.

Performance tricks

Papyrus is a compiled language and each script that is executed consumes resources and computation time.

Keep you scripts clean and efficient is the difference between a bad mod and a good mod.

Do not use Game.GetPlayer()

The function `Game.getPlayer()` is time consuming and really not efficient.

A faster way to get the player is to define a property called:

```
Actor Property PlayerRef Auto
```

This property will auto fill, and is the quickest possible way to have the player actor.

This code:

```
Actor[] myLongActorList = new Actor[120] ; The list is empty but it is just
for explanation purpose

Int i = 0

While (i < 120)

      If (myLongActorList[i] == Game.GetPlayer())

            Debug.Notification(“Gotcha: ” + i)

            Return

      EndIf

      i = i + 1

EndWhile
```

Takes more than 15 times compared to the following code (that is still not perfect):

```
Actor Property PlayerRef Auto

Actor[] myLongActorList = new Actor[120]

Int i = 0

While (i < 120)

      If (myLongActorList[i] == PlayerRef)

            Debug.Notification("Gotcha: " + i)

            Return

      EndIf

      i = i + 1

EndWhile
```

Simplify as much as possible the conditions

A simple condition like (I < 100) takes about twice the time to just check if a value is empty or not.

If you need to browse an array for some reasons, do it from the last item to the first, and use as condition just the variable without operators.

```
Actor Property PlayerRef Auto

Actor[] myLongActorList = new Actor[120]

Int i = 120

While (i)

      i = i - 1

      If (myLongActorList[i] == PlayerRef)

            Debug.Notification("Gotcha: " + i)

            Return

      EndIf

EndWhile
```

This code is equivalent to the previous code but execute about 50% faster because the condition inside the while statements is simplified.

Use Assignments with Operation operators

Try to avoid code like `i = i - 1` or `i = i + 1`.

You can use `i -= 1` and `i += 1` respectively. The improvement in performance is marginal (about 1% gain) but still it will increase the script performance and improve readability.

```
Actor Property PlayerRef Auto

Actor[] myLongActorList = new Actor[120]

Int i = 120

While (i)

	i -= 1

	If (myLongActorList[i] == PlayerRef)

		Debug.Notification("Gotcha: " + i)

		Return

	EndIf

EndWhile
```

Use native functions when possible

If you are searching for an element inside an array, you can use the .find() function that is available on arrays. The following code completely replaces the previous ones:

```
Actor Property PlayerRef Auto

Actor[] myLongActorList = new Actor[120]

Int pos = myLongActorList.find(PlayerRef)

If (pos != -1)

	Debug.Notification("Gotcha: " + pos)

EndIf
```

And it executes about 50 times faster than the previous code.

Avoid to do in a loop code that can be done outside

Consider that you want to find the first NPC in the current cell that has the same race of the player.

This code is bad:

```
Actor Property PlayerRef Auto
Actor Function getSameRaceClosestNPCs(Cell cell)
	Int num = cell.getNumRefs(62)
	While (num)
		num -= 1
		If (cell.getNthRef(num, 62) && (cell.getNthRef(num, 62) as
Actor).getActorBase().getRace() == PlayerRef.getActorBase().getRace())
			Return (cell.getNthRef(num, 62) as Actor)
		EndIf
	EndWhile
	Return None
EndFunction
```

Why it is bad? Because inside the cycle you are calling always the same function on always the same object (and you will get always the same result): PlayerRef.getXXX().getYYY().

You can do it just once outside the loop:

```
Actor Property PlayerRef Auto
Actor Function getSameRaceClosestNPCs(Cell cell)
	Int num = cell.getNumRefs(62)
	Race playerRace = PlayerRef.getActorBase().getRace()
	While (num)
		num -= 1
		If (cell.getNthRef(num, 62) && (cell.getNthRef(num, 62) as
Actor).getActorBase().getRace() == playerRace)
			Return (cell.getNthRef(num, 62) as Actor)
		EndIf
	EndWhile
	Return None
EndFunction
```

This increases the speed of about 75%.

But it is just the beginning. In the code there are three times the same function call .getNthRef(). This can be done just once.

```
Actor Property PlayerRef Auto

Actor Function getSameRaceClosestNPCs(Cell cell)

        Int num = cell.getNumRefs(62)

        Race playerRace = PlayerRef.getActorBase().getRace()

        While (num)

                num -= 1

                Actor aNPC = (cell.getNthRef(num, 62) as Actor)

                If (aNPC && aNPC.getActorBase().getRace() == playerRace)

                        Return aNPC

                EndIf

        EndWhile

        Return None

EndFunction
```

This will double the performance of the script.

Optimize the conditions

Conditions in the If and While statements are executed from left to right. As soon the condition evaluates to true or false the other parts of the condition are no more considered. Use this at your advantage.

```
Actor aNPC = …

If (aNPC != None && aNPC.getLeveledActorBase().getRace() = myRace &&
aNPC.HasLOS(PlayerRef) && aNPC.GetEquippedWeapon().GetEnchantment() ==
myEnchantment && aNPC.isInFaction(myFaction))
```

This condition is pretty heavy and will check for heavy items while other light-weight items are at the end.

`aNPC != None` → Light and can be improved by writing just "`aNPC`". (the `!= None` can be implicit)

`aNPC.getLeveledActorBase().getRace() = myRace` → Average

`aNPC.HasLOS(PlayerRef)` → Heavy

`aNPC.GetEquippedWeapon().GetEnchantment() == myEnchantment` → Pretty heavy

`aNPC.isInFaction(myFaction)` → Light

If you run this in a cycle, replacing it with:

```
If (aNPC && aNPC.isInFaction(myFaction) &&
aNPC.getLeveledActorBase().getRace() = myRace  &&
aNPC.GetEquippedWeapon().GetEnchantment() == myEnchantment &&
aNPC.HasLOS(PlayerRef))
```

Will make the code to run way faster, because the heavy items of the condition are to the end of the condition.

Do not use shortcut functions, use the basic ones

Papyrus offers some shortcut functions that allow you to save some key types.

For example the function `GetActorReference()` of a **`ReferenceAlias`** has the short form `GetActorRef()`

You save a few keyboard strokes. But at execution time, the VM will call the other function, so it will require a little bit more time.

Try to avoid the functions:

Do not use **`Actor`**`.DamageAV()`, use `DamageActorValue()`

Do not use **`Actor`**`.ForceAV()`, use `ForceActorValue()`

Do not use **`Actor`**`.GetAV()`, use `GetActorValue()`

Do not use **`Actor`**`.GetAVPercentage()`, use `GetActorValuePercentage()`

Do not use **`Actor`**`.GetBaseAV()`, use `GetBaseActorValue()`

Do not use **`Actor`**`.ModAV()`, use `ModActorValue()`

Do not use **`Actor`**`.RestoreAV()`, use `RestoreActorValue()`

Do not use **`Actor`**`.SetAV()`, use `SetActorValue()`

Do not use **`ReferenceAlias`**`.GetActorRef()`, use `(GetReference() As` **`Actor`**`)`

Do not use **`ReferenceAlias`**`.GetActorReference()`, use `(GetReference() As` **`Actor`**`)`

Do not use **`ReferenceAlias`**`.ObjectReference GetRef()`, use `GetReference()`

Do not use **`ObjectReference`**`.IsEnabled()`, use `!IsDisaled()`

Do not use **`ObjectReference`**`.X`, use `GetPositionX()`

Do not use **`ObjectReference`**`.Y`, use `GetPositionY()`

Do not use **`ObjectReference`**`.Z`, use `GetPositionZ()`

Advanced features (Threads and ModEvents and SKSE)

float[] Function CreateFloatArray(int size, float fill = 0.0) global native

int[] Function CreateIntArray(int size, int fill = 0) global native

bool[] Function CreateBoolArray(int size, bool fill = false) global native

string[] Function CreateStringArray(int size, string fill = "") global native

Form[] Function CreateFormArray(int size, Form fill = None) global native

TODO

PapyrusUtils

PapyrusUtil is a wonderful set of tools that enables file access from papyrus scripts. It contains also some utility tools to manage Actors, Objects, camera, etc.

PapyrusUtil is a SKSE plugin and requires at least version 1.7.1; PapyrusUtil was developed by Ashal from an original implementation done by h38fh2mf.

PapyrusUtil is integrated in SexLab. If you have SexLab you do not need to install it.

The file Papyrusutils itself provides only some basic functions to get an array of Actors with a given size.

All the other functions are defined in: StorageUtil, JsonUtil, ActorUtil, ObjectUtil, and MiscUtil.

```
Actor[] Function ActorArray(int size) global
```

All these functions work in the same way: they will allocate for you an array of the size you want for the specific object type. Because the functions are global, you can call them directly from the `PapyrusUtil` object.

Equivalent functions, for different types, are implemented by SKSE:

```
float[] Function CreateFloatArray(int size, float fill = 0.0) global

int[] Function CreateIntArray(int size, int fill = 0) global

bool[] Function CreateBoolArray(int size, bool fill = false) global

string[] Function CreateStringArray(int size, string fill = "") global

Form[] Function CreateFormArray(int size, Form fill = None) global
```

```
Actor[] myArray = PapyrusUtil.ActorArray(37) ; PapyrusUtils

Int[] myOtherArray = CreateIntArray(Utility.RandomInt(5, 100)) ; SKSE
```

StorageUtil

Stores variables and lists of data on a form that can be pulled back out using the form and variable name as keys.

`StoregeUtil` contains a set of global functions for saving and loading any amount of values of type `int`, `float`, `form`, and `string`. The values are accessed by name on a form or globally. These values can be accessed and changed from any mod.

The values are stored on a specific `Form` or globally. The values persist during save games. To remove a values you have to explicitly UnSet or Clear it. In case the Form is deleted, the value will be removed from the save game.

The values that are saved take little memory. Expect to use less than 500 KB of physical memory even when setting thousands of values.

Value names are not case sensitive, the variables "abc", "ABC", and "Abc" are the save variable.

Variables of different type may share the same name. An `Int` variable ABC and a `Sting` variable ABC will not share the same value and can live together.[5]

⚠ When choosing names for variables try to remember that all mods access the same storage, so if you create a variable with name "Name" then many other mods could use the same variable but in different ways that lead to unwanted behavior. It is recommended to prefix the variable with your mod name or code. In this way you can be sure nobody else will try to use the same variable in their mod. For example SexLabArousal can prefix with **sla**, NonSexLab Animation Pack can prefix with **nsap**, etc.

All the following function are `global` and `native`. This definition is not shown in the example.

```
int Function SetIntValue(Form obj, string key, int value)

float Function SetFloatValue(Form obj, string key, float value)

string Function SetStringValue(Form obj, string key, string value)

Form Function SetFormValue(Form obj, string key, Form value)
```

These functions will set a value on a Form of the specified type. The parameter key is the actual name of the value you want to set and value is the value itself.

Each function will return the previous value if any.

If the `Form obj` is set to `None`, then the variable is stored globally. If it is set to something then the value is stored on the specified `Form`.

[5] Also if it possible to have variables of different type with the same name, this is not recommended to have your code clean.

```
bool Function UnsetIntValue(Form obj, string key)

bool Function UnsetFloatValue(Form obj, string key)

bool Function UnsetStringValue(Form obj, string key)

bool Function UnsetFormValue(Form obj, string key)
```

These functions will remove a value on a Form of the specified type. The parameter key is the actual name of the value you want to remove.

Each function will return True if it was possible to remove the value, they will return False is the value was not set.

If the `Form obj` is set to `None`, then the variable is removed from the global list. If it is set to something then the value is removed from the specified `Form`.

```
bool Function HasIntValue(Form obj, string key)

bool Function HasFloatValue(Form obj, string key)

bool Function HasStringValue(Form obj, string key)

bool Function HasFormValue(Form obj, string key)
```

These functions will check if a specific value is available on the object.

Each function will return True if the value is present, False otherwise.

If the `Form obj` is set to `None`, then the checking is done on the global list. If it is set to something then the value checked on the specified `Form`.

```
int Function GetIntValue(Form obj, string key, int missing = 0)

float Function GetFloatValue(Form obj, string key, float missing = 0.0)

string Function GetStringValue(Form obj, string key, string missing = "")

Form Function GetFormValue(Form obj, string key, Form missing = none)
```

These functions will return the value for the specified key. If the value is missing the default value will be returned (It can be specified in the `missing` parameter.)

If the `Form obj` is set to `None`, then the value is got from the global list. If it is set to something then the value get from the specified `Form`.

```
int Function AdjustIntValue(Form obj, string key, int amount)
```

```
float Function AdjustFloatValue(Form obj, string key, float amount)
```

These functions will add the specified `amount` to the specified key. The value is initialized with the specified `amount` if it was not existing.

If the `Form obj` is set to `None`, then the value is got from the global list. If it is set to something then the value get from the specified `Form`.

The values of a key can be also a list.

```
int Function IntListAdd(Form obj, string key, int value, bool allowDuplicate
= true)
```

```
int Function FloatListAdd(Form obj, string key, float value, bool
allowDuplicate = true)
```

```
int Function StringListAdd(Form obj, string key, string value, bool
allowDuplicate = true)
```

```
int Function FormListAdd(Form obj, string key, Form value, bool
allowDuplicate = true)
```

These functions will add a value to the specified key. If `allowDuplicate` is true, then in the list is possible to have twice the same value. If it is set to false, then adding an already existing value will produce no results.

The return value is the index of the added value in the list. If it was not possible to add the value then -1 is returned.

If the `Form obj` is set to `None`, then the value is added to the key of the global list. If it is set to something then the value is added to the key of the specified `Form`.

```
int Function IntListGet(Form obj, string key, int index)
```

```
float Function FloatListGet(Form obj, string key, int index)
```

```
string Function StringListGet(Form obj, string key, int index)
```

```
Form Function FormListGet(Form obj, string key, int index)
```

These functions will get a value from the specified key in the specified index position. If there was an error getting the value the returned value will be 0, 0.0, “”, or None respectively.

If the `Form obj` is set to `None`, then the value is got to the key of the global list. If it is set to something then the value if got from the specified `Form`.

```
int Function IntListSet(Form obj, string key, int index, int value)
float Function FloatListSet(Form obj, string key, int index, float value)
string Function StringListSet(Form obj, string key, int index, string value)
Form Function FormListSet(Form obj, string key, int index, Form value)
```

These functions will set a value in the specified key in the specified index position.

In case of problems the returned value is 0, 0.0, “”, or None respectively.

If the `Form obj` is set to `None`, then the value is set on the key of the global list. If it is set to something then the value is set in the specified `Form`.

```
int Function IntListAdjust(Form obj, string key, int index, int amount)
float Function FloatListAdjust(Form obj, string key, int index, float amount)
```

These functions will add the specified `amount` to the specified key in the specified index. The value is initialized with the specified `amount` if it was not existing.

If the `Form obj` is set to `None`, then the value is got from the global list. If it is set to something then the value get from the specified `Form`.

```
bool Function IntListInsert(Form obj, string key, int index, int value)
bool Function FloatListInsert(Form obj, string key, int index, float value)
bool Function StringListInsert(Form obj, string key, int index, string value)
bool Function FormListInsert(Form obj, string key, int index, Form value)
```

These functions will insert a value in the specified key in the specified index position. If the action fails False is returned.

If the `Form obj` is set to `None`, then the value is inserted on the key of the global list. If it is set to something then the value is inserted in the specified `Form`.

```
int Function IntListRemove(Form obj, string key, int value, bool allInstances
= false)
int Function FloatListRemove(Form obj, string key, float value, bool
allInstances = false)
int Function StringListRemove(Form obj, string key, string value, bool
allInstances = false)
int Function FormListRemove(Form obj, string key, Form value, bool
allInstances = false)
```

These functions will remove a value from the specified key. If the parameter `allInstances` is true then all occurrences of the value will be removed from the list.

If the `Form obj` is set to `None`, then the value is removed from the key of the global list. If it is set to something then the value is removed from the specified `Form`.

The functions return how many values are removed.

```
int Function IntListClear(Form obj, string key)
int Function FloatListClear(Form obj, string key)
int Function StringListClear(Form obj, string key)
int Function FormListClear(Form obj, string key)
```

Clear the whole list from an object. The list is not removed, it will just contains no more values. It returns the previous size of the list.

If the `Form obj` is set to `None`, then the list is removed from the global list. If it is set to something then the list is removed from the specified `Form`.

```
bool Function IntListRemoveAt(Form obj, string key, int index)
bool Function FloatListRemoveAt(Form obj, string key, int index)
bool Function StringListRemoveAt(Form obj, string key, int index)
bool Function FormListRemoveAt(Form obj, string key, int index)
```

Remove the value at the specified index. It returns false in case of problems.

If the `Form obj` is set to `None`, then the list value is removed from the global list. If it is set to something then the list value is removed from the specified `Form`.

```
int Function IntListCount(Form obj, string key)
int Function FloatListCount(Form obj, string key)
int Function StringListCount(Form obj, string key)
int Function FormListCount(Form obj, string key)
```

Returns the size of the specified list.

```
int Function IntListFind(Form obj, string key, int value)
```

```
int Function FloatListFind(Form obj, string key, float value)
```

```
int Function StringListFind(Form obj, string key, string value)
```

```
int Function FormListFind(Form obj, string key, Form value)
```

Returns the position of the value specified as parameter. Returns -1 if the value is not available.

```
Function IntListSlice(Form obj, string key, int[] slice, int startIndex = 0)
```

```
Function FloatListSlice(Form obj, string key, float[] slice, int startIndex =
0)
```

```
Function StringListSlice(Form obj, string key, string[] slice, int startIndex
= 0)
```

```
Function FormListSlice(Form obj, string key, Form[] slice, int startIndex =
0)
```

Fill the array passed as argument with the element at the specified index. The copy will end as soon the array or the list will run out of positions/elements.

```
int Function IntListResize(Form obj, string key, int toLength, int filler =
0)
```

```
int Function FloatListResize(Form obj, string key, int toLength, float filler
= 0.0)
```

```
int Function StringListResize(Form obj, string key, int toLength, string
filler = "")
```

```
int Function FormListResize(Form obj, string key, int toLength, Form filler =
none)
```

Updates the size of the list to the new length by truncating or extending it. If the list is extended the parameter `filler` will be used for the new part of the list.

Returns the number of elements truncated (negative values) or added (positive values) onto the list.

```
bool Function IntListCopy(Form obj, string key, int[] copy)
```

```
bool Function FloatListCopy(Form obj, string key, float[] copy)
```

```
bool Function StringListCopy(Form obj, string key, string[] copy)
```

```
bool Function FormListCopy(Form obj, string key, Form[] copy)
```

Overwrite any existing list for the specified key, with the values passed in the parameter array.

⚠ Please note that lists are limited to 500 values.

Examples:

```
int Function SetIntValue(Form obj, string key, int value)
```

Will save the value in the save game.

Some examples:

```
StorageUtil.SetIntValue(None, "myGlobalKey", 125)
```

Will set a global key (stored in the save game) of type `Int` and value equals to 125.

```
StorageUtil.SetStringValue(PlayerRef, "theTodayQuote", "I love wandering in
Skyrim")
```

Will add a string value, with the key “theTodayQuote” to the player Actor.

```
Float aNumber = StorageUtil.GetFloatValue(aBook, "TheValueOfPi")
```

Gets the value of the key “TheValueOfPi” from a Form.

```
Float aNumber = StorageUtil.GetFloatValue(aBook, "TheValueOfPi", 3.1415926)
```

Gets the value of the key “TheValueOfPi” from a Form, and returns a default value if the key is not available in the Form.

```
Int numNPCKilled = StorageUtil.FormListCount(PlayerRef, "allKilledNPCs")

While(numNPCKilled)

      numNPCKilled -= 1

      Debug.Notification((StorageUtil.FormListGet(Playerref, "allKilledNPCs",
numNPCKilled) as Actor).getLeveledActorbase().getName)

EndWhile
```

Will get the full list of Forms (as Actors) attached to the player in the key allKilledNPCs and will print their names.

JsonUtil

Similar to StorageUtil.psc but saves data to custom external .json files instead of forms, letting them be customizable out of game and stored independent of a user save file.

The functions provided are about the same as the ones provided by StorageUtil.

These functions will NOT require the Form to be specified, but require a filename where the values are read and write. The file will have an extension .json and will be stored in Data/SKSE/Plugins/StorageUtilData/. There can be only one file per installation, so different save games will share the same file.

Some important notes on usage to keep in mind:

- You may specific a folder path in the filename, i.e. "../MyData/config" will save to Data/SKSE/Plugins/MyData/config.json
- If not given in the filename argument, the filename will have the extension .json appended to it automatically.
- When the player saves their game any modified file will be automatically saved, written to, or created, if it does not exist.
- You do not need to call Load() or Save() manually unless you have a specific need to.
  - The save is automatic when the player saves the game.
  - When the player loads another save without saving the game and the function Save() was not invoked, then all the loaded data will be discarded and revert back to whatever the contents of the current saved file are.

```
bool function Load(string FileName)
```

Force load from the json file.

```
bool function Save(string FileName, bool minify = false)
```

Force a save to the json file. If minify is true then the json file will be a single long line (avoid it because it will be hard to update the file with a text editor.)

```
function ClearAll(string FileName)
```

Removes all the values from the JSON file.

All the StorageUtil functions are available in JsonUtil.

What is a JSON file?

A JSON file (commonly used in Javascript) is a quick and easy way to store structured data.

It looks like this example:

```
{
        "VersionKey": 12.5,
        "ModName": "My Amazing Mod",
        "ListOfNumbers": [0, 1, 2, 3, 4, 5],
        "ListOfStrings": ["Hello", "World", "This", "Is", "An", "Example"
}
```

JSON files can be edited with a normal text editor like Notepad++.

ActorUtil

This global object provides some utilities to override a package for an actor.

These overrides persist through save games. If you override package on same actor more than once then the package with highest priority will run, if multiple overrides have same priority then last added package will run. Priority ranges from 0 to 100 with 100 being highest priority.

```
Function AddPackageOverride(Actor targetActor, Package targetPackage, int
priority = 30, int flags = 0)
```

Calling this function will attach to the targetActor a package that will override all other packages added by the game (AI Packages, ReferenceAlias packages, Scene action packages, etc.)

The flag is set to something different from 1 will make the package to execute only if it met its conditions. If set to 0 the package will not perform the checking of its conditions and will execute immediately.

```
bool Function RemovePackageOverride(Actor targetActor, Package targetPackage)
```

This function will remove an override package from an actor. It will return false if the actor has not the specified package in its override package list.

```
int Function CountPackageOverride(Actor targetActor)
```

Counts how many package overrides are currently on this actor. It will also count ones that were assigned but are not running because the conditions are not met.

```
int Function ClearPackageOverride(Actor targetActor)
```

Removes all package overrides on this actor, including ones that were added by other mods.

```
int Function RemoveAllPackageOverride(Package targetPackage)
```

Removes the package passed as parameter from all actor that have package overrides.

ObjectUtil

This utility Object is providing functions to override the animations.

It can replace an animation on an object by using an animation event name.

If you run this code:

```
Idle Property laughIdle Auto ; "IdleLaugh" here from CK
SetReplaceAnimation(akTarget, "moveStart", laughIdle)
```

As soon as the actor akTarget will start moving, the laugh animation will be played.

Animation replace is checked once, that means if you replace the animation event “moveStart” with the idle IdleLaugh and the animation event “IdleLaugh” with the idle idleEatingStandingStart then whenever the actor will try to move it will start laughing only. The other replacement will not be considered.

This replacement persists through save games.

```
Function SetReplaceAnimation(ObjectReference obj, string oldAnimEvent, Idle
newAnim)
```

Replaces the animation on an object associated with the event name passed as parameter with the new animation. If `obj` is `none` then it will replace globally, be careful with this!

```
bool Function RemoveReplaceAnimation(ObjectReference obj, string
oldAnimEvent)
```

Removes a previously replacement set for an animation.

`int Function ClearReplaceAnimation(ObjectReference obj)`

Clears all animation replacements on an object.

`int Function CountReplaceAnimation(ObjectReference obj)`

Returns how many animation replacements have been set on object.

`String Function GetKeyReplaceAnimation(ObjectReference obj, int index)`

Returns the animation event that was replaced on an object by index. The function `CountReplaceAnimation` can be used to iterate on all replacements.

`Idle Function GetValueReplaceAnimation(ObjectReference obj, string oldAnim)`

Returns the idle animation that is replacing an animation event.

MiscUtil

This global object provides some miscellaneous commands.

`Function ToggleFreeCamera(bool stopTime = false)`

Toggles the free-fly camera.

`Function SetFreeCameraSpeed(float speed)`

Set the free-fly camera speed.

`Function SetFreeCameraState(bool enable, float speed = 10.0)`

Set the current free-fly camera state and set the speed when enabling.

```
float Function GetNodeRotation(ObjectReference obj, string nodeName, bool
firstPerson, int rotationIndex)
```

Undocumented function.

```
Function PrintConsole(string text)
```

Print some text to the console.

```
Function SetMenus(bool enabled)
```

Set the HUD (Head-Up Display) on and off.

```
String function GetRaceEditorID(Race raceForm)
String function GetActorRaceEditorID(Actor actorRef)
```

These functions can be very useful to check for races that are provided by a mod, without requiring the mod to be enabled or loaded.

Consider the following example:

```
Actor somethingLikeADog = None
String[] dogRaces = new String[22]
dogRaces[0] = "WerewolfBeastRace"
dogRaces[1] = "DLC2WerebearBeastRace"
dogRaces[2] = "_00GreaterShoggothRace"
dogRaces[3] = "_00WerebearBeastBlackRace"
dogRaces[4] = "_00WerebearBeastSnowRace"
dogRaces[5] = "_00WereSkeeverBeastRace"
dogRaces[6] = "_00DaedrothRace"
dogRaces[7] = "_00DramanBeastRace"
dogRaces[8] = "_00DwarvenPunisherRace"
dogRaces[9] = "_00WerewolfKingBeastRace"
dogRaces[10] = "WolfRace"
dogRaces[11] = "DLC1DeathHoundCompanionRace"
dogRaces[12] = "DLC1DeathHoundRace"
dogRaces[13] = "_00AspectRace"
dogRaces[14] = "DogRace"
dogRaces[15] = "DogCompanionRace"
dogRaces[16] = "MG07DogRace"
dogRaces[17] = "DA03BarbasDogRace"
dogRaces[18] = "DLC1HuskyArmoredCompanionRace"
```

```
22    dogRaces[19] = "DLC1HuskyArmoredRace"
23    dogRaces[20] = "DLC1HuskyBareCompanionRace"
24    dogRaces[21] = "DLC1HuskyBareRace"
25    ; Scan for all dogs and wolves around the player
26    Cell c = PlayerRef.getParentCell()
27    int num = c.GetNumRefs(43)
28    While (num)
29          num -= 1
30          Actor d = c.GetNthRef(num, 43) as Actor
31          If (d && !d.IsDisabled())
32                Race r = d.getLeveledActorBase().getRace()
33                If (dogRaces.find(MiscUtil.GetRaceEditorID(r)) != -1)
34                      somethingLikeADog = d
35                      Return
36                EndIf
37          EndIf
38    EndWhile
```

This code starts by defining an array of Strings with all possible dog and wolf races defined by Skyrim, Dawnguard and other mods. (Lines 2 – 24.) They are NOT actual races, so you DO NOT need to put a dependency to these mods.

Then all the NPC in the same cell of the player are checked. Line 26 is getting the cell where the player is, then we ask SKSE for the number of NPCs in the cell at line 27.

The cycle will run on all NPCs. As soon as we get a valid, non-disabled actor (31) we will get its race (32.)

At line 33 the magic happens: instead of searching for the race inside an array of races (this will require all the mods providing these races to be added in CK and then you will make a dependency to all these mods), we search on the array of strings we defined before.

The `GetRaceEditorID` will return a string with the id of the race, so this code will run in all version of Skyrim also if the mods providing these extra races are not installed.

The code can be optimized a little by replacing the lines 32 and 33 with just:

```
      If (dogRaces.find(MiscUtil.GetActorRaceEditorID(d)) != -1)
```

SkyUI and MCM

This chapter describes how to crete a MCM menu for your mod using SkyUI and MCM APIs.

SkyUI is an excellent mod that enhances the user interface of Skyrim. SkyUI provides also the MCM menu to provide configuration pages for mods.

A wiki for the MCM menu, made by *schlangster* can be found here: MCM-Quickstart.

How the MCM works? It requires a quest with a couple of items inside: an alias to the player to manage the load and updates, and some code to build the page (and do whatever you want with the values that the user will set.)

Also if the quest that contains the code and the alias for the MCM can be shared with another quest of your mod, it is a good idea to create a specific MCM quest. In this guide it will be supposed that you have a quest called **myModConfigQuest**.

Inside the quest you need to create a ReferenceAlias, set it to the player (specific reference, cell = any, Ref = playerRef) and add to it a specific script provided by SkyUI: `SKI_PlayerLoadGameAlias`.

Now inside your myModConfigQuest quest create a script and call it myModConfig and make it extends the script provided by SkyUI called `SKI_ConfigBase`.

Now we can begin to define its content. The SKI_ConfigBase script will define a few properties for you. One is called ModName, set it to the name of your mod. Another is called Pages and is an array of strings. You can have as many pages as you want in your configuration menu (max 128), just create an entry in Pages for each page. Page names have to be different.

MCM is fully handled by events. The main event is OnPageReset.

```
Event OnPageReset(string page)
          …
EndEvent
```

It will be called any time a page in your configuration menu has to be refreshed.

The parameter that will be passed is the name of the page. It can be one of the values you defined in the Pages array, or can be an empty value. The empty value identifies the main page of the configuration (where often there is just the logo of you mod.)

Let’s define your myModConfig to have two pages inside. The first will be called “My mod config” and the second will be called “Dynamic enable”. Edit the Pages property of your script to match this definition.

Now in the code of the script add this:

```
Event OnPageReset(string page)
      If (page=="")
            Debug.trace("Main logo page")
      ElseIf (page=="My mod config")
            Debug.trace("This is the config page that is being generated")
      ElseIf (page=="Dynamic enable")
            Debug.trace("This is the second page that is being generated")
      EndIf
EndEvent
```

Ok, not too much right now, but it will already work. When you open the pages they will be empty, but you will see traces in the Papyrus.log

Now, how a page is structured?

The pages have a table layout with two columns. Each entry of the table has an index from 0 to 127. The first cell is 0, the cell on the top of the second column is 1, the left cell of the first row is 2, the right cell of the second row is 3, and so on.

0	1
2	3
4	5
6	7
…	…

In each cell you can add an entry (called Option):

EmptyOption → Keeps the cell empty

HeaderOption → Adds a static text header

TextOption → Adds a label and some text inside the option

ToggleOption → Adds a label and a checkmark

SliderOption → Adds a label and a numeric slider

MenuOption → Adds a label and a drop-down menu

ColorOption → Adds a label and a color picker option

KeyMapOption → Adds a label and a way to setup a keyboard map to define the key shortcuts

All options are managed through events, and they can be created in two ways: direct option generation with IDs, and options generated with states.

Options generated by states make the code way easier to be organized and updated. Direct generation with IDs can be used when the list of options you need is not static and will be generated depending on some factors. Our first page will use states, and the second one will generate the list of options dynamically.

There are some functions provided by the MCM API to insert the different options in the cells. Each function will put the requested option in the current cell and then will move to the next one.

There are two ways to define "which cell will be the next one":

LEFT_TO_RIGHT → When a cell is used then next one will be the one on the right. If the cell was already on the right then the next one will be the left cell of the next row.

TOP_TO_BOTTOM → The next cell will be the cell just below the current one.

The sequence mode can be defined and changed any time through the function:

`SetCursorFillMode(TOP_TO_BOTTOM)` and `SetCursorFillMode(LEFT_TO_RIGHT).`

Let's see the different type of options one by one:

Empty Option

Not too much to say, the cell will be just empty. No events are generated for the cell because the user cannot interact with it.

`int AddEmptyOption()`

An id for the option is generated and returned, but usually you can simply ignore it for empty options.

Header Option

Also this one is pretty easy, you just need to specify the text that will be represented in the header. The header will not generate events because the user cannot interact with it.

`int AddHeaderOption(string text, int flags = 0)`

An id for the option is generated and returned, but usually you can simply ignore it for headers.

Text Option

A text option will add a label and some text visible to the user. Up to version 4.1 of SkyUI the text option will NOT give to the user a way to type some text. This option is commonly used to implement buttons.

It comes in a state version and in a stateless version:

```
AddTextOptionST(string stateName, string text, string value, int flags = 0)
```

```
int AddTextOption(string text, string value, int flags = 0)
```

The first one is the stateful version (the function mane ends with ST), the latter is the stateless version. We will see later how to handle the events generated in both cases.

`stateName` is the name of the state associated with the option. (Only for stateful.)

`text` is what will be represented as label for the option

`value` is the actual text that is shown after the label (the user can click on it)

`flags` is used to define if the option is enabled, read-only, etc. We will see this later.

The stateless function will return the id of the generated option. This number is important and you have to keep it somewhere.

Now let's go back to the code for your MCM menu and let's add something inside:

```
Event OnPageReset(string page)
	SetCursorFillMode(TOP_TO_BOTTOM)
	If (page=="")
		AddHeaderOption("This is my wonderful mod.")
	ElseIf (page=="My mod config")
		AddHeaderOption("This is my wonderful mod.")
		AddTextOptionST("modStatus", "Mod status is", "Unknown")
		AddEmptyOption()
		AddTextOptionST("modAuthor", "This Mod was developed by", "<put 
here your name>")
	ElseIf (page=="Dynamic enable")
		AddHeaderOption("This page will be done later...")
	EndIf
EndEvent
```

Now if you run your mod you will see the MCM menu main page with just an header, and in the first page an header and two text fields (separated by an empty line), all in the left part of the page.

Toggle Option

A toggle option will add a checkmark on the page. Something that can be True or False, Checked or Unchecked, Flagged or Not-flagged, Set or Unset. Just two possibilities, like a Boolean value.

```
AddToggleOptionST(string stateName, string text, bool checked, int flags = 0)
```

```
int AddToggleOption(string text, bool checked, int flags = 0)
```

The first one is the stateful version (the function mane ends with ST), the latter is the stateless version. We will see later how to handle the events generated in both cases.

`stateName` is the name of the state associated with the option. (Only for stateful.)

`text` is option label, visible to the user

`checked` defines if the checkmark has to be checked or unchecked.

`flags` is used to define if the option is enabled, read-only, etc. We will see this later.

The stateless function will return the id of the generated option. This number is important and you have to keep it somewhere.

Slider Option

A slider option will add a scroll bar where the user can pick a value in a defined range of numeric values.

```
AddSliderOptionST(string stateName, string text, float value, string
formatString = "{0}", int flags = 0)
```

```
int AddSliderOption(string text, float value, string formatString = "{0}",
int flags = 0)
```

The first one is the stateful version (the function mane ends with ST), the latter is the stateless version. We will see later how to handle the events generated in both cases.

`stateName` is the name of the state associated with the option. (Only for stateful.)

`text` is option label, visible to the user

`value` is the current value of the slider

`formatString` is optional and defines how the value will be shown to the user

`flags` is used to define if the option is enabled, read-only, etc. We will see this later.

The stateless function will return the id of the generated option. This number is important and you have to keep it somewhere.

Menu Option

A menu option will show a drop-down to the user and allows to pick one of the values from the drop-down.

```
AddMenuOptionST(string stateName, string text, string value, int flags = 0)
```

```
int AddMenuOption(string text, string value, int flags = 0)
```

`stateName` is the name of the state associated with the option. (Only for stateful.)

`text` is option label, visible to the user

`value` is the current value of the dropdown menu (the list is defined inside the events.)

`flags` is used to define if the option is enabled, read-only, etc. We will see this later.

The stateless function will return the id of the generated option. This number is important and you have to keep it somewhere.

Let's go back to the page to add a few more option. Only the part "My Mod Config" is listed here:

```
      ...
      ElseIf (page=="My mod config")
            AddHeaderOption("This is my wonderful mod.")
            AddTextOptionST("modStatus", "Mod status is", "Unknown")
            AddEmptyOption()
            AddTextOptionST("modAuthor", "This Mod was developed by", "<put
here your name>")
            AddToggleOptionST("LikeIt", "Do I like this mod?", True)
            AddSliderOptionST("numSecs", "How many seconds?", 0)
            AddMenuOption(armorSelect", "Select you favorite armor", "<none
      selected>")
      ElseIf (page=="Dynamic enable")
      ...
```

Now you will have a checkbox, a slider, and a drop down menu. They still do nothing. We did not created the events to manage them.

Color Option

Adds an option that opens a color dialog to select a color.

```
AddColorOptionST(string stateName, string text, int color, int flags = 0)
```

```
int AddColorOption(string text, int color, int flags = 0)
```

`stateName` is the name of the state associated with the option. (Only for stateful.)

`text` is option label, visible to the user

`color` is the current value of the color. The format is rgb. Hex values cab be set as 0xRRGGBB, int values can be set as r*65536 + g*256 + b.

`flags` is used to define if the option is enabled, read-only, etc. We will see this later.

The stateless function will return the id of the generated option. This number is important and you have to keep it somewhere.

KeyMap Option

Adds a key mapping option to allow the user to select a keyboard key.

```
AddKeyMapOptionST(string stateName, string text, int keyCode, int flags = 0)
```

```
int AddKeyMapOption(string text, int keyCode, int flags = 0)
```

`stateName` is the name of the state associated with the option. (Only for stateful.)

`text` is option label, visible to the user

`keyCode` is the SKSE keycode. It is represented as an image on the MCM option.

`flags` is used to define if the option is enabled, read-only, etc. We will see this later.

The stateless function will return the id of the generated option. This number is important and you have to keep it somewhere.

Now let's make this page roll by defining the events. Right now we defined only Stateful options, so we will see how to manage the events in a stateful mode. Refer to the States section to understand what they are and how to use them.

Please replace the page code with the following example:

```
Actor Property PlayerRef Auto
String[] theArmors = new String[10]
Int selectedArmor = -1
Bool addIfMissing = True
```

```
Int howManyArrows = 0

Event OnPageReset(string page)
    SetCursorFillMode(TOP_TO_BOTTOM)
    If (page==””)
        AddHeaderOption(“This is my wonderful mod.”)
    ElseIf (page==”My mod config”)
        AddHeaderOption(“Please select the options you like.”)
        String armorName = “None Selected”
        If (selectedArmor!=-1)
            armorName = theArmors[selectedArmor]
        EndIf
        AddMenuOption(“MENU_armorSelect”, “Select you favorite armor”, armorName)
        AddToggleOptionST(“CHECK_AddIt”, “Add the armor if missing?”, addIfMissing)
        AddSliderOptionST(“SLIDER_numArrows”, “How many arrows to add?”, howManyArrows)
        AddEmptyOption()
        AddTextOptionST(“TEXT_dress”, “Click to give the armor to the player”, “<do it>”)
    ElseIf (page==”Dynamic enable”)
        AddHeaderOption(“This page will be done later...”)
    EndIf
EndEvent

State MENU_armorSelect
    event OnMenuOpenST()
        SetMenuDialogOptions(theArmors)
        SetMenuDialogStartIndex(0)
        SetMenuDialogDefaultIndex(0)
    endEvent

    event OnMenuAcceptST(int index)
        selectedArmor = index
        SetMenuOptionValueST(theArmors[index])
    endEvent
```

```
35           event OnDefaultST()
36                  selectedArmor = -1
37                  SetMenuOptionValueST("None selected")
38           endEvent

39           event OnHighlightST()
40                  SetInfoText("You will give to the player the armor selected here")
41           endEvent
42     EndState
```

Let’s explain the code section by section.

We have a string array called theArmors, it contains a set of names for a set of armors (the array is empty but this does not matter, you can add some values inside the list.)

We have a drop down option (Menu Option) that will show a drop-down list. It is defined in a stateful way, so all the event management code it is inside the state defined.

The dropdown is defined at line 16, the state name is “MENU_armorSelect”.

There is a state called in the same way: MENU_ArmorSelect.

And there are 4 events defined for this state. Let’s check them one by one:

OnMenuOpenST is called to initialize the actual menu. It defines what the possible values are. We use the function SetMenuDialogOptions and we give to it as parameter the array that contains our list of armors.

OnMenuAcceptST is called when the user select an option from the drop down. The value passed is the index of the selected option. You should store this somewhere. And then you should update the visible value in the Menu Option using the function SetMenuOptionValueST().

OnDefaultST is called if the user select the default option (Tab by default.) You hould handle this in the same way you do for the OnMenuAcceptST event.

OnOptionHighlightST is called when the user goes with the mouse over the option. It is used to display some help text (that should be always defined) in the bottom section of the MCM menu.

Let’s see the full definition of the events:

```
Event OnMenuOpenST()
```

No arguments are passed. It is called to initialize the menu.

```
Event OnMenuAcceptST(int index)
```

Is called when an option is selected, the index of the selected option is passed as argument.

```
Event OnDefaultST()
```

Is called when the user select the default option.

```
Event OnOptionHighlightST()
```

Is called when the user goes with the cursor over the option.

To handle the toggle option we have some events that are the same and one that is different.

Toggle options do not require to be initialized. The OnDefaultST() and the OnOptionhighlightST() work in the same way.

When the user changes the status of the checkmark then the event OnSelectST() is called.

```
State CHECK_AddIt
	event OnSelectST()
		addIfMissing = ! addIfMissing
		SetToggleOptionValueST(addIfMissing)
	endEvent

	event OnDefaultST()
		addIfMissing = True
		SetToggleOptionValueST(True)
	endEvent

	event OnHighlightST()
		SetInfoText("If you add the armor to the player and the player does not
have the item, the armor will be added to the player's inventory")
	endEvent
EndState
```

Now let's manage the events for the Text Option. Remember they are used as buttons.

```
State TEXT_dress
	event OnSelectST()
		; Here do the code to process the action
		SetTextOptionValueST("DONE")
```

```
        endEvent


        event OnHighlightST()

                SetInfoText("By clicking this button you will add the selected armor to
the player.")

        endEvent

EndState
```

No special events here, we already saw them before.

Now the slider:

```
State SLIDER_numArrows

        Event OnSliderOpenST()

                SetSliderDialogStartValue(howManyArrows)

                SetSliderDialogDefaultValue(0)

                SetSliderDialogRange(0, 100)

                SetSliderDialogInterval(1)

        endEvent


        event OnSliderAcceptST(float a_value)

                howManyArrows = a_value as int

                SetSliderOptionValueST(howManyArrows)

        endEvent


        event OnDefaultST()

                howManyArrows = 0

                SetSliderOptionValueST(0)

        endEvent


        event OnHighlightST()

                SetInfoText("In addition to the armor the player will receive the
selected number of arrows.)

        endEvent

EndState
```

The event OnSliderOpenST() is used to initialize the slider. The minimum and maximum values are set by the function SetSliderDialogRange(), the default value is set by SetSliderDialogDefaultValue(), the actual current value is set by SetSliderDialogStartValue(), and you can specify also the interval for the slider through the function SetSliderDialogInterval().

```
Function SetSliderDialogRange(Float minValue, Float maxValue)

Function SetSliderDialogDefaultValue(Float defaultValue)

Function SetSliderDialogStartValue(Float currentValue)

Function SetSliderDialogInterval(Float interval)
```

If you want that the slider for the arrows goes 5 by 5 arrows, you have to call the function like: SetSliderDialogInterval(5).

If you want a range that goes from 10 to 50, then you call the function SetSliderDialogRange(5, 50).

The other events are pretty much the same as the other options.

Translation of the MCM

The MCM offers a quick way to translate the strings. If a string starts with the dollar symbol, then a file in Data\Interface\Translation\<mod name>_<LANGUAGE>.txt is open.

If in this file there is a line that starts with the string you used (the one that starts with a dollar symbol), then whatever is after a tab character is used for the actual string.

Example:

Suppose you mod is called MyMod.esp, and in the code of your MCM you have a line like: AddHeaderOption(“$HeaderTitle1”).

When the MCM process the line, it will search for a set of txt files in the translation directory. Let’s say you have:

MyMod_ENGLISH.txt, MyMod_SPANISH.txt, MyMod_FRENCH.txt, MyMOD_JAPANESE.txt

And inside each file you have

Welcome!

MyMod_ENGLISH.txt	$HeaderTitle1 Welcome!
MyMod_SPANISH.txt	$HeaderTitle1 ¡Bienvenido!
MyMod_FRENCH.txt	$HeaderTitle1 Bienvenue !
MyMod_JAPANESE.txt	$HeaderTitle1 ようこそ！

Please note that a tab character is required between the key and the actual value.

What about the stateless options?

Stateless options are useful when you don't know exactly how many options will be in the page.

Let's imagine that you what a mod that has only one MCM page that will list all NPCs in the player cell, and allows you to enable/disable them. Let's call the mod "The Disabler" (code **tdr**.)

Crate a quest in the md and call it tdrConfigQuest. Then add the ReferenceAlias to the player and attach to it the `SKI_PlayerLoadGameAlias`.

Then attach a script to it:

```
Scriptname tdfConfig extends SKI_ConfigBase


Actor Property PlayerRef Auto

Actor[] allActorsHere = new Actor[128]

Int numActors = 0

Int[] optionsIDs = new Int[128]

ModName = "The Disabler"

Pages = new Arry[1]

Pages[0] = "Disable!"

Event OnPageReset(String page)

    If (page=="The Disabler")

        ; Get all NPCs in the player cell

        Cell c = Player.getParentCell()

        Int num = c.getNumRefs(62)

        If (num > 128)

            num = 128 ; Arrays in Papyrus are limited to 128 entries

        EndIf

        numActors = 0

        SetCursorFillMode(LEFT_TO_RIGHT)

        While (num)

            Num -= 1

            Actor a = c.getNthRef(num, 62) as Actor

            If (a) ; Only if it exists

                allActorsHere[numActors] = a

                String name = a.getLeveledActorBase().getName()
                optionsIds[numActor] = AddToggleOption(name, !a.isDisabled())
                numActor += 1
            EndIf
        EndWhile
    EndIf
EndEvent

Event OnOptionSelect(int optionId)
    Int pos = optionIDs.find(optionID)
    Bool isDisabled = allActorsHere[pos].isDisabled()
    allActorsHere[pos].enable(isDisabled) ; No need to invert the value…
    SetToggleOptionValue(optionId, isDisabled)
EndEvent

Event OnOptionHighlight(int optionId)
    Int pos = optionIDs.find(optionID)
    String name = allActorsHere[pos].getLeveledActorBase().getName()
    If (allActorsHere[pos].isDisabled())
        SetInfoText(name + " is currently disabled. Click to enable it.")
    Else
        SetInfoText(name + " is currently enabled. Click to disable it.")
    EndIf
EndEvent
```

How this code works?

It will initialize an array of actors. Every time the MCM page is open, the code will check for all the actors in the cell of the player. For each of them a toggle option will be added, with the option set to true if the actor is enables (= !isDisabled()) and false if it is disabled.

Because we don't know how many actor to manage, then we cannot really use the states to handle the options. In this case is better to create an array of `Int`egers that will hold the each option id, in the same place where the actual `Actor` will be stored.

Then we use the global stateless events OnOptionSelect and OnOptionHighlight to do the job. When we receive the event we will get the optionID, we will find its position in the array of IDs, and then do job.

Where to store the values?

You have two main options to store the configuration values of your mod.

One is to use a set of Properties, another is to use some sort of storage, like StorageUtil or JsonUtil.

Properties are quicker and easier (you just use them directly), while using StorageUtil or JasonUtil requires that you use temporary variables and then read or update the value in the file storage.

But Properties are a serious problem when you upgrade the mod by adding/changing existing properties.

A special case can be storing the values in a game object, like a GlobalVariable or an actor value. They persist over the saves but should be used rarely and on very special conditions.

Other events of MCM

```
Event OnConfigInit()
```

Is called when the script is initialized.

```
Event OnConfigRegister()
```

Is called when the configuration menu is successfully registered.

```
Event OnConfigOpen()
```

Is called when the configuration menu is opened.

```
Event OnConfigClose()
```

Is called when the configuration menu is closed.

What about version management?

To manage the version of your mod, and execute all the logic to handle the updates you need to implement one function and one event:

```
Int function GetVersion()
```

Should return the current version of your mod as integer. Common practice is to use the major number of the version multiplied by 1000, the minor number multiplied by 10, and then add a number from 0 to 9 for minor patches (version a, version b, c, etc.)

So if your mod is in version 2.13b the number returned should be something like 2132.

2 * 1000 + 13 * 10 + 2 (where a = 1, b = 2, c= 3, etc.)

```
Event OnVersionUpdate(int version)
```

Is called when a version update of this script has been detected.

You can do then some code like:

```
Event OnVersionUpdate(int version)
    If (version < 1000)
        Debug.Messagebox("Sorry, beta versions cannot be upgraded.")
        Return
    EndIf
    If (version < 1100)
        myNewAromrList = new Armor[2]
        myNewAromrList[0] = bigEnterrpiseArmorProperty
        myNewAromrList[1] = fulgentArmorProperty
    EndIf
    If (version < 1200)
        If (PlayerRef.getItemCount(superKey) > 0)
            PlayerRef.removeItem(superKey)
        endIf
        PlayerRef.addItem(anotherKey, 1)
    EndIf
    If (version < 1202)
        ; Fix the bug of the version 1.20a by restarting the quest
        myQuest.stop()
        myQuest.start()
    EndIf
EndEvent
```

Adding an image/logo to your mod

If you do a MCM menu for your mod, probably you will add also an image describing you mod.

Images can be added in MCM through the function LoadCustomContent().

See the following example:

```
Event OnPageReset(String page)
    If (page == "")
      LoadCustomContent("My Mod\MyWonderfulPic.dds")
      Return
    EndIf
    UnLoadCustomContent()
    ...
EndEvent
```

If you place a DDS picture called *MyWonderfulPic.dds* in the folder Data\Interface\My Mod\ then you will see your picture on the main MCM page of your mod.

You can use GIMP or Photoshop to convert a normal image to a DDS image.

Please note that the content has to be unloaded for other pages, o it will be always visible.

```
Function LoadCustomContent(string source, float x = 0.0, float y = 0.0)
```

When the image has been loaded, then other options are hidden. You have to unload it with UnloadCustomContent() to show back the normal options.

The optional parameters x and y can be used to move the image in the page, to better fit in the correct position.

Option Flags

All options support some flags:

OPTION_FLAG_NONE → to have a normal read/write option.

OPTION_FLAG_DISABLED → to have the option disabled and grayed out.

OPTION_FLAG_HIDDEN → Options with this flag will not be visible and they will be replaced by an empty option.

OPTION_FLAG_WITH_UNMAP → Will enable the unmap button for keymap options.

The flags can be set in all AddOptions as last parameter during option creation. Or they can be set with the functions:

```
SetOptionFlagsST(int flags, bool noUpdate = false, string stateName = "")
```

```
SetOptionFlags(int optionId, int flags, bool noUpdate = false)
```

`flags` is the flag you want to set (OPTION_FLAG_NONE, OPTION_FLAG_DISABLED, OPTION_FLAG_HIDDEN, OPTION_FLAG_WITH_UMAP)

`noUpdate` set to true will not execute the update immediately. It has to be explicitly called after.

`stateName` is used to identify a stateful option.

`optionId` is used to identify a stateless option.

Examples

Some quick examples that can be used as reference.

TODO

Code Step by Step

This section will contain a full long example of creating a mod (no tutorials here about how to use the Creation Kit, create worlds, cells, NPCs, etc., search them on Youtube if you need.) and adding some code behind.

Actor

Package fragments

Controlling a quest

Controlling a scene

ReferenceAlias

Magic Effects

Activators and Triggers

Communication between scripts

Add a MCM page

Use some advanced features

TODO

Using the SexLab Framework

This chapter describes how to use the functions provided by the SexLab Framework by *Ashal* (ashal@loverslab.com).

SexLab is a framework. It provides a way to play some sex oriented animations inside Skyrim.

The framework is very well written, it is stable and reliable. And alone does pretty much nothing (it is a framework.)

To play the animations other mods needs to have it as dependency and call the functions provided by SexLab. All the interactions with SexLab are Papyrus based.

All the information here is updated to SexLab version 1.59c.

Basic features

The main functions of SexLab is to start a sex animation between actors. The number of actors may vary from 1 (masturbation) to 5 (orgies.)

Usually in the animations the very first involved actor is the "submissive" one. In a Male to Female straight action the first actor is the female, and the other one is the male. Please note that all animations can be played by both sex without any restriction.

Animations are split in two different main categories: Human animations (involving only human-like races (biped-models)), and Creature animations (involving at least one non-human participant.)

The animations are split in stages (from just one to up to 6 stages), each stage will involve the full list of participants (Actors.)

Each animation has its own Tags to allow who is playing the animation to pick the most appropriate one.

Animations may also have a "Victim", defining a victim in the animation usually just affects the Stats for the involved actors.

Because SexLab is a framework you need to reference a Property pointing to it. The usual way to add it is to define, in your code a property like this:[6]

```
SexLabFramework Property SexLab Auto
```

And then remember to fill the property with the value inside the property window of the script. You cannot fail, there will be just a single option to pick.

Then to start a sex animation you just need a single line of code:

[6] In this manual the variable **SexLab** will be always represented as an object, also if it is not really an object.

```
SexLab.QuickStart(PlayerRef)
```

This will make the player to masturbate.

To have two NPCs have sex you can just execute this line:

```
Actor aFemaleNPC = …

Actor aMaleNPC = …

SexLab.QuickStart(aFemaleNPC, aMaleNPC)
```

The function `QuickStart` gives you very little control but is pretty effective to quickly start an action.

```
sslThreadController Function QuickStart(Actor Actor1, Actor Actor2 = none,
Actor Actor3 = none, Actor Actor4 = none, Actor Actor5 = none, Actor Victim =
none, String Hook = "", String AnimationTags = "")
```

As you can see in its definition, you may pass up to 5 involved actors. One of them, if you like, can be set as “Victim”.

`AnimationTags` will give you some control on the animations that will be selected (by specifying the tags for the possible animations.)

`Hook` will give you the ability to run some code on the different stages of the animation. We will see this later.

The function returns a *Thread Controller*. It can be used to control and manipulate the animation. We will see this later.

Another quick way to start an animation is to use the function StartSex:

```
Int Function StartSex(Actor[] Positions, sslBaseAnimation[] Anims, Actor
Victim = none, ObjectReference CenterOn = none, Bool AllowBed = true, String
Hook = "")
```

`StartSex()` requires an array of Actors to define who is involved.

The Victim can be specified passing an actor as parameter.

Then you have a couple of extra options that were not available with `QuickStart`: `CenterOn` and `AllowBed`.

`CenterOn` is used to pass a `ReferenceObject` (a marker usually) where the animation will be centered. In `QuickStart` the closest marker to the first actor will be selected to center the animation.

`AllowBed` will allow/deny the utilization of beds to play the animation over (only the animations marked as non-“Standing” will be used on beds by default.

The `Hook` behaves exactly as QuickStart and will be discussed it later.

If the animation can be started you will receive a ThreadID as result. If the animation cannot be started for some reasons you will receive -1 as result.

To have the full control of everything in the animation, you need to do some more advanced coding by requesting a ThreadMode from SexLab.

```
sslThreadModel Function NewThread(Float TimeOut = 30.0)
```

This is more advanced and we will see it later.

Data structure

SexLab has a full set of objects that are involved, here the list of them and the main functions you can call on them.

Actors

Actors are the NPCs (and the Player) that will participate in a sex animation.

SexLab accepts pretty much any actor that is a biped-model, is NOT a child[7], is not dead, and has at least one animation that supports its `Race`.

```
Int Function ValidateActor(Actor ActorRef)
```

This function will check if the actor is valid, and it is playing another animation.

A result of 1 means that the actor is good, a negative number means that it is not valid:

-1 → The actor is a None object.

-10 → it is are already playing a SexLab animation

-11 → For forbidden actors (no children, no actors with a scale that is too low, no forbidden biped races)

-12 → If the 3D of the actor is not yet loaded

-13 → If the actor is dead

-14 → if the actor is disabled

-15 → If the actor is flying (no sex on the clouds…)

-16 → if the actor is riding a horse

-17 → if the race of the actor is not supported (or creatures are disabled.)

[7] Pedophilia is against the rules and will NEVER be accepted.

```
bool function IsValidActor(Actor ActorRef)
```

This function calls the `ValidateActor` and the just returns a true/false value.

```
bool function IsActorActive(Actor ActorRef)
```

Will return True if the actor is currently playing a SexLab animation.

```
Actor[] function MakeActorArray(Actor Actor1 = none, Actor Actor2 = none,
Actor Actor3 = none, Actor Actor4 = none, Actor Actor5 = none)
```

This is a commodity function to get an array of actors with the actors passed as parameters.

```
Actor function FindAvailableActor(ObjectReference CenterRef, float Radius =
5000.0, int FindGender = -1, Actor IgnoreRef1 = none, Actor IgnoreRef2 =
none, Actor IgnoreRef3 = none, Actor IgnoreRef4 = none)
```

This function will search for an actor that is close to the `ObjectReference` passed as parameter, in the specified radius. You can set some other actors inside the parameters to avoid to pick them. You can also specify the sex of the actor. The sex is not the ActorBase sex but the one managed by SexLab. (0 male or female that behave as males, 1 females or males that behave as females, 2 creatures (in SexLab 1.6 there will be also gender for creatures.))

```
Actor[] function FindAvailablePartners(actor[] Positions, int TotalActors,
int Males = -1, int Females = -1, float Radius = 10000.0)
```

This function expects as parameter an array of actors you want partners for. At least one actor has to be placed in the array to start. You have to specify how many actors to find (there should be enough empty places in the array.) You may specify how many males and females you want.

The result will be an array of actors filled with compatible actors (if actors are available.)

```
Actor[] function SortActors(Actor[] Positions, bool FemaleFirst = true)
```

This function requires an array of actors as source and will return a sorted array of actors based on their gender.

```
function ApplyCum(Actor ActorRef, int CumID)
```

This function adds a cumshot skin texture to the actor.

CumID == 1 → Oral

CumID == 2 → Vaginal

CumID == 3 → Anal

CumID == 4 → Oral and Vaginal

CumID == 5 → Oral and Anal

CumID == 6 → Vaginal and Anal

CumID == 7 → Oral, Vaginal and Anal

```
function AddCum(Actor ActorRef, bool Vaginal = true, bool Oral = true, bool
Anal = true)
```

This function adds a cumshot skin texture to the actor.

```
function ClearCum(Actor ActorRef)
```

Removes any cumshot skin texture that was applied by SexLab. The cumshot skin texture will expire alone by the time defined in the SexLab MCM menu.

```
form[] function StripActor(Actor ActorRef, Actor VictimRef = none, bool
DoAnimate = true, bool LeadIn = false)
```

This function will strip the actor and will return an array of `Form` (usually `Armor`s) to remember the clothes the actor was wearing, this array can be used to redress it later.

If the parameter `DoAnimate` is false then the actor will play a simple strip animation.

If the parameter `LeadIn` is true then the actor will partially strip and will fully strip only when the animation will start.

```
form[] function StripSlots(Actor ActorRef, bool[] Strip, bool DoAnimate =
false, bool AllowNudesuit = true)
```

This function is a little more advanced than `StripActor`, is requires an array of Booleans to identify which slot to undress (see later to understand what slots are.) The parameter `AllowNudeSuit` when set to True will replace the body of the actor with a “nude suit” to show naked NPCs in case the used body textures have no nudity.

The `Strip` `Bool` array has to be of size 33. Each item is exactly the Body Slot with the same number minus 30. See the Strip Slots table below.

For example the gauntlets has the body slot 33 so the index for the Strip array will be 3.

For example the circlet has the body slot 42 so the index for the Strip array will be 12.

`Function UnstripActor(Actor ActorRef, Form[] Stripped, bool IsVictim = false)`

This function redress an actor using the items returned by the StripActor or StripSlots function.

`Bool function IsStrippable(form ItemRef)`

This function will check if a specific `Form` (an Armor for example) is strippable or not, non-strippable items are defined by associating them to a keyword that contains “NoStrip”.

`Form function StripSlot(Actor ActorRef, int SlotMask)`

This function remove the item (mainly `Armor`s) that are wear on the specific slot that is passed as parameter. If there was an item on the slot it is returned by the function.

The `SlotMask` is the Body Slot mask (not the SexLab one 0-based); see the Body Slot table to find the Slot mask for each body part.

`Form function WornStrapon(Actor ActorRef)`

This function will return the stapon worn by the actor if it is in the strapon list of sexlab. It will return none if no strapon is worn. It will return none also if the user is wearing a strapon that is not in the sexlab strapon list.

`Bool function HasStrapon(Actor ActorRef)`

Same function as WornStrapon but returns true or false.

`Form function PickStrapon(Actor ActorRef)`

If the actor is already wearing a strapon it will be returned. If the actor is not yet wearing a strapon one from the SexLab Strapon list will be randomly chosen and returned. This function will NOT make the actor wear the strapon.

`Form function EquipStrapon(Actor ActorRef)`

This function will pick randomly one strapon from the SexLab strapon list and will add it to the actor and then will make the actor wear it. There is no way to specify the strapon to be used. It will be chosen randomly.

`Function UnequipStrapon(Actor ActorRef)`

Removes the strapon from the actor.

`Function ForbidActor(Actor ActorRef)`

This function will add the actor to the forbidden actor faction, and will make them to be always not valid when involved in SexLab animations.

`Function AllowActor(Actor ActorRef)`

This function will add the actor to the forbidden actor faction, and will make them to be always not valid when involved in SexLab animations.

`Bool function IsForbidden(Actor ActorRef)`

This function checks if an actor is in the forbidden faction.

`Function TreatAsMale(Actor ActorRef)`

SexLab by default uses the sex of the `Actor Base` to understand if they are males or females. It is possible to alter the gender of an actor with this function.

The actor, also if it is a female in the definition of the `Actor Base`, will be treated as a male.

`Function TreatAsFemale(Actor ActorRef)`

SexLab by default uses the sex of the `Actor Base` to understand if they are males or females. It is possible to alter the gender of an actor with this function.

The actor, also if it is a male in the definition of the `Actor Base`, will be treated as a female.

`Function` ClearForcedGender(`Actor` ActorRef)

This function will reset the actor gender to the standard sex defined in its `Actor Base`.

`Int function GetGender(Actor ActorRef)`

Returns the gender for the actor. Is no specific gender was defined in SexLab the sex defined in the Actor Base will be returned.

`Int[] function GenderCount(Actor[] Positions)`

By giving an array of `Actor`s you will receive an array of `Int` of 3 elements. [0] is the number of males, [1] is the number of females, [2] is the number of creatures.[8]

`Int function MaleCount(Actor[] Positions)`

Returns the number of actors in the array that have a male gender.

`Int function` FemaleCount(`Actor`[] Positions)

Returns the number of actors in the array that have a female gender.

`Int function` CreatureCount(`Actor`[] Positions)

Returns the number of actors in the array that are creatures.

[8] This may change in SexLab 1.60, with this version also creatures will have sex.

Slots (for stripping in SexLab)

A normal NPC of Biped-model has a set of body slots to wear pieces of clothes, boots, hoods, etc. Some of the slots are used to “wear” specific items like wings, tails, dicks, strapons, etc.

Slots in Skyrim starts at 30 and ends at 61. Some of them are assigned directly by the vanilla Skyrim, some of them are assigned by modders. Here the common list of body slots for the biped-model:

Body Slot	SexLab Slot	Slot Mask	Description	Who defined it
30	0	0x00000001	Head	Creation Kit
31	1	0x00000002	Hair	Creation Kit
32	2	0x00000004	Body	Creation Kit
33	3	0x00000008	Hands	Creation Kit
34	4	0x00000010	Forearms	Creation Kit
35	5	0x00000020	Amulet	Creation Kit
36	6	0x00000040	Ring	Creation Kit
37	7	0x00000080	Feet	Creation Kit
38	8	0x00000100	Calves	Creation Kit
39	9	0x00000200	Shield	Creation Kit
40	0	0x00000400	Tail	Creation Kit
41	1	0x00000800	Long hair	Creation Kit
42	2	0x00001000	Circlet	Creation Kit
43	3	0x00002000	Ears	Creation Kit
44	4	0x00004000	Face/Mouth	Nexus Wiki
45	5	0x00008000	Neck (cape, scarf, shawl, neck-tie, etc.)	Nexus Wiki
46	6	0x00010000	Chest primary or outer-garment	Nexus Wiki
47	7	0x00020000	Backpack/Wings	Nexus Wiki
48	8	0x00040000	Strapon	SexLab
49	9	0x00080000	Pelvis primary or outer garment	
50	0	0x00100000	Decapitated Head	Creation Kit
51	1	0x00200000	Decapitated Body ; Strapon	Creation Kit ; Mods
52	2	0x00400000	Pelvis secondary ; Strapon	Nexus Wiki ; Mods
53	3	0x00800000	Leg primary or outer garment or right leg	Nexus Wiki
54	4	0x01000000	Leg secondary or undergarment or left leg	Nexus Wiki
55	5	0x02000000	Face alternate or jewelry	Nexus Wiki
56	6	0x04000000	Chest secondary or undergarment	Nexus Wiki
57	7	0x08000000	Shoulders	Nexus Wiki
58	8	0x10000000	Arm secondary or outer garment or left arm	Nexus Wiki
59	9	0x20000000	Arm secondary or outer garment or right arm	Nexus Wiki
60	0	0x40000000	Strapon/Shlong	SOS
61	1	0x80000000	FX01	Creation Kit

Animations

The animations are the main item managed by SexLab. You may find them explicitly or by tag to play a specific animation with the list of actors you will define.

If you use the functions `QuickStart` or `StartSex` you can not specify directly the animations. You need to use SexLab threads if you want to force a specific animation (or a set of possible animations.)

Every animation is defined by the **`sslAnimationBase`** object.

This object is internal and should not be used alone.

A few functions can be used to get information about a specific animation (usually the use of these functions is not required in mods.)

The animations have two hidden properties that can be sometimes useful:

Name → The user name assigned to the animation

Registry → The ID of the animation

An animation has a set of other items inside, the stages, the positions the definition for the Shlong, the definition for where to apply the cumshot, the timing for each stage. Usually you don't need to access this properties.

But there is another property that is the key to pick a good animation: its tags.

A tag is a way to categorize the animation. See the dedicated section to learn more about tags.

Animations have also a ContentType property. It can be Misc, Sexual, Foreplay. These contents cannot be read directly. They can be queried using the GetAnimations functions. They are set up during animation creation.

Some functions that can be used on the animation (**`sslBaseAnimation`**) are:

```
bool function IsSilent(int Position, int Stage)
```

True if the animation stage is silent.

```
bool function UseOpenMouth(int Position, int Stage)
```

True if the animation stage has at least one position that has the mouth open.

```
bool function UseStrapon(int Position, int Stage)
```

True is the animation uses a strapon in the specified stage.

```
int function GetSchlong(string AdjustKey, int Position, int Stage)
```

Return the Shlong used by the stage, according to the SOS definition.

```
int function ActorCount()
```

Returns the number of positions for the animation.

```
int function StageCount()
```

Returns the number of sages defined for the animation.

```
int function GetGender(int Position)
```

Returns the expected gender for the specific position.

```
bool function MalePosition(int Position)
```

Returns True if the specific position is expected to be a male.

```
bool function FemalePosition(int Position)
```

Returns True if the specific position is expected to be a female.

```
bool function CreaturePosition(int Position)
```

Returns True if the specific position is expected to be a creature.

```
int function FemaleCount()
```

Counts how many females are expected in all positions.

```
int function MaleCount()
```

Counts how many males are expected in all positions.

```
int function GetCum(int Position)
```

Returns the type of cum that will be added for the specific position. The possible values are:

1 → Oral

2 → Vaginal

3 → Anal

4 → Oral and Vaginal

5 → Oral and Anal

6 → Vaginal and Anal

7 → Oral, Vaginal and Anal

```
bool function IsSexual()
```

Returns if the animation is defined as sexual.

```
float function GetRunTime()
```

Returns the number of seconds that the animation will last. It is adjusted with the stages timing defined in the SexLab MCM.

```
float function GetRunTimeLeadIn()
```

Returns the number of seconds for the lead in stage.

```
float function GetRunTimeAggressive()
```

Returns the number of seconds that the animation will last when playes as aggressive. It is adjusted with the stages timing defined in the SexLab MCM.

```
bool function HasActorRace(Actor ActorRef)
```

Returns True is the animation supports the race of the actor. The actor should be a creature.

```
bool function HasRace(Race RaceRef)
```

Returns True is the animation supports the race. The race should be a creature race.

```
bool function HasRaceID(string RaceID)
```

Returns true if the animation contains the race ID.

```
string[] function GetRaceIDs()
```

Returns an array of `String`s with all supported Race IDs.

The Tags System

The SexLab animations have a powerful mechanism to identify them. Each animation has a set of tags defined to characterize it.

When playing an animation in your mod using SexLab you can select the most appropriate animation using tags. This allows you to get all the possible animation that respect a given category and make your mod not repetitive.

There are a set of tags defined by SexLab, other mods that add SexLab animations (More Nasty Critters, Non-SexLab Animation Pack, Zaz Animation Pack, etc.) may had further tags to the animations.

The main SexLab tags are:

Authors tags: “Arrok”, “Leito”, etc. It is just the name of the creator of the animation.

Aggressive: The “Aggressive” tag is used to limit the aggressive animations. These animations can be limited by the user in the SexLab MCM menu.

Type of intercourse: “69”, “Anal”, “Blowjob”, “Breast”, “Boobjob”, Cowgirl”, “Cunnilingus”, “Doggy”, “Doggy Style”, “Doggystyle” (the doggystyle tag is not uniform in SexLab), “Feet”, “Footjob” (Footjob tags are not uniform in SexLab tags), “Fisting”, “Handjob”, “Licking”, “Masturbation”, “Missionary”, “Oral”, “Vaginal”

Intercourse variants: “Behind”, “Sideways”, “Cuddling”, “Dirty”, “Fetish”, “Hugging”, “Kissing”, “Loving”, “Rough”, “Foreplay”, “Knees”, “Laying”, “LeadIn”

Number of people: “Solo”, “Lesbian”, “MF”, “MFF”, “MMF”, “MMMF”, “MMMM”, “Threeway”, “Orgy”

Standing: There is the special tag “Standing” that is used by SexLab to avoid certain animations when playing the animation over a bed.

Stats related tabs: the tags “Oral”, “Vaginal”, “Anal”, “Loving”, and “Dirty” are used to alter the SexLab stats for the actors involved. (See SexLab stats below.)

Getting the animations by tags

```
sslBaseAnimation[] function GetAnimationsByTags(int ActorCount, string Tags,
string TagSuppress = "", bool RequireAll = true)
```

This function requires the number of actors involved in the animation and the tags that are required.

It will return the list of animations that contain the tags.

The string Tags is done by adding all the tags separated by commas; the order of tags is not important. Example: “Oral,Dirty”.

The parameter `RequireAll` will change the way the tags are considered. If it is True then an animation is added to the result list only if **ALL** the tags specified are contained in the animation. If it is False then an animation is inserted in the result list if it contains **AT LEAST** one of the tags.

You can also specify a list of tags that the animations **MUST NOT** contain. If an animation has at least one of the tags specified in the `TagSuppress` string then it will not be returned.

```
sslBaseAnimation[] function GetAnimationsByType(int ActorCount, int Males = -
1, int Females = -1, int StageCount = -1, bool Aggressive = false, bool
Sexual = true)
```

This function will return an array of animations that contain the specified amount of males and females and the number of stages. The Sexual parameter will filter animations that have a ContentType equal to Misc or Sexual or Foreplay.

```
sslBaseAnimation[] function PickAnimationsByActors(actor[] Positions, int
limit = 64, bool aggressive = false)
```

By passing an array of actors this functions return an array of animations that will respect the gender of each actor involved. (Not considering the placement of the actor, just the number of males and females.)

```
sslBaseAnimation[] function GetAnimationsByDefault(int Males, int Females,
bool IsAggressive = false, bool UsingBed = false, bool RestrictAggressive =
true)
```

Gets an array of appropriate animations by the number of males and females specified.

```
sslBaseAnimation function GetAnimationByName(string FindName)
```

Returns a single animation that has exactly the name specified.

A couple of functions can be used to refine the animation list:

```
sslBaseAnimation[] function MergeAnimationLists(sslBaseAnimation[] List1,
sslBaseAnimation[] List2)
```

This function is an utility function that will return an array of animations containing the two annray of animations passed as parameters.

```
sslBaseAnimation[] function RemoveTagged(sslBaseAnimation[] Anims, string
Tags)
```

This function will remove all the animations from the source list that have at least one of the specified tags.

Obtaining animation information by tags

```
int function CountTag(sslBaseAnimation[] Anims, string Tags)
```

Will count how many animations of the list passed as parameter contains the list of tags passed. At least one of the tags.

```
int function GetAnimationCount(bool IgnoreDisabled = true)
```

Returns the number of animations registered in SexLab. The list can be all the animations or just the ones that are enabled.

```
string function MakeAnimationGenderTag(Actor[] Positions)
```

This is a commodity function that will produce a tag string based on the genders of the actors passed in the array. The result is something like: “FM”, “FFM”, “MMM”, “FC”, “FCCCC”, etc.

```
string function GetGenderTag(int Females = 0, int Males = 0, int Creatures =
0)
```

This is a commodity function that will produce a tag string based on the number of different genders passed as parameters. The result is something like: “FM”, “FFM”, “MMM”, “FC”, “FCCCC”, etc.

Facial Expressions

SexLab gives you some control of the facial expressions.

The main object used is `sslBaseExpression`. It is internal and subject to change. Use at your own risk.

The following function specify if they are exposed by an `sslBaseExpression` (use with caution) or directly by `SexLab` (they are safe.)

`function OpenMouth(Actor ActorRef)`

Make the mouth open for the Actor passed as parameter. [Defined by `SexLab` → safe to use]

`function CloseMouth(Actor ActorRef)`

Closes the mouth of the actor passed as parameter. [Defined by `SexLab` → safe to use]

`bool function IsMouthOpen(Actor ActorRef)`

Checks if the mouth of the actor is open (True) or closed (False). [Defined by `SexLab` → safe to use]

```
sslBaseExpression function PickExpression(Actor ActorRef, Actor VictimRef =
none)
```

Returns an `sslBaseExpression` for the specified actor. [Defined by `SexLab` → safe to use]

If the actor is specified and the victim is not then a random expression is selected (based on the sex of the actor (not the gender).)

`sslBaseExpression function RandomExpressionByTag(string Tag)`

Gets a random expression that contains the specified tag. [Defined by `SexLab` → safe to use]

Possible tags are: Afraid, Aggressor, Angry, Consensual, Happy, Joy, Negative, Nervous, Normal, Pain, Pained, Pleasure, Sad, Scared, Shy, Upset, Victim.

`sslBaseExpression function GetExpressionByName(string findName)`

Gets an expression with the specified name. [Defined by `SexLab` → safe to use]

Possible names are: Pleasure, Happy, Joy, Shy, Sad, Afraid, Pained, and Angry.

`function ClearMFG(Actor ActorRef)`

Completely re-initialize the expression on the target actor. [Defined by `SexLab` → safe to use]

`function ClearPhoneme(Actor ActorRef)`

Reset the phoneme variation from the target actor. [Defined by `SexLab` → safe to use] `function ClearModifier(Actor ActorRef)`

Clears the expression modifier for the target actor. [Defined by `SexLab` → safe to use]

`function Apply(Actor ActorRef, int Strength, int Gender)`

Applies a preset to an actor of the specified strength and of the specified gender. [Defined by `sslBaseExpression` → unsafe]

`function ApplyPhase(Actor ActorRef, int Phase, int Gender)`

Applies a preset to an actor from the specified phase. [Defined by `sslBaseExpression` → unsafe]

`function OpenMouth(Actor ActorRef) global`

Makes the actor to open its mouth. [Defined by `sslBaseExpression` but global → afe]

`function CloseMouth(Actor ActorRef) global`

Makes the actor to close its mouth. [Defined by `sslBaseExpression` but global → safe]

`bool function IsMouthOpen(Actor ActorRef) global`

Returns a Boolean to understand if the mouth of the actor is open or closed. Makes the actor to open its mouth. [Defined by `sslBaseExpression` but global → safe]

`function ClearMFG(Actor ActorRef) global`

Completely re-initialize the expression on the target actor. [Defined by `sslBaseExpression` but global → safe]

`function ClearPhoneme(Actor ActorRef) global`

Reset the phoneme variation from the target actor. [Defined by `sslBaseExpression` but global→ safe]

`function ClearModifier(Actor ActorRef) global`

Clears the expression modifier for the target actor. [Defined by `sslBaseExpression` but global→ safe]

Voices

The voices (just a Moan of Mild, Medium, and Hot levels) are controlled by the object `sslBaseVoice`.

Only one function is available on this object:

```
Function Moan(Actor ActorRef, int Strength = 30, bool IsVictim = false)
```

This function will move the lips of the actor and will pay the moaning sound according to the parameters.

From SexLab interface you have access to these functions:

```
sslBaseVoice function PickVoice(Actor ActorRef)
```

```
sslBaseVoice function GetVoice(Actor ActorRef)
```

Both functions return a voice compatible with the actor. Try to avoid GetVoice because it is an alias and will reduce performances.

```
function SaveVoice(Actor ActorRef, sslBaseVoice Saving)
```

Assigns a specific voice to the actor.

```
function ForgetVoice(Actor ActorRef)
```

removes the custom voice for the actor.

```
bool function HasCustomVoice(Actor ActorRef)
```

Returns true if the actor has a custom voice set.

```
sslBaseVoice function GetVoiceByGender(int Gender)
```

Returns the base voice for the specific gender.

```
sslBaseVoice function GetVoiceByName(string FindName)
```

Returns the base voice that has the specified name, if existing.

```
sslBaseVoice function GetVoiceByTags(string Tags, string TagSuppress = "",
bool RequireAll = true)
```

Search a voice that matches the tags specified (same criteria as getAnimationByTags.)

The voice tags are: Female, Male, Classic, Normal, Breathy, Loud, Rough, Young, Stimulated, Excited, Excitable, Quiet, Timid, Average, Harsh, Mature, Old.

Using threads to run SexLab animations

Playing an animation is SexLab is quick and easy if you use QuickStart or StartSex. But you have very little control of what happens in the animation.

An animation that is running (or is going to run) is identified by a SexLab thread. The thread can be controlled (if required) through a Thread Controller.

To obtain a SexLab thread is enough to write this single line of code:

```
sslThreadModel Thread = SexLab.NewThread()
```

The object sslThreadModel is the thread. Getting a thread will not automatically start an animation or doing anything specific. As you can see there are no parameters for the NewThread() function.

Functions available on a SexLab Thread Model:

```
int function AddActor(Actor ActorRef, bool IsVictim = false, sslBaseVoice
Voice = none, bool ForceSilent = false)
```

Adds one actor to the sexlab animation. You can specify a specific voice or if the actor has to be silent. If you specify the option IsVictim then the animation will be tagged as Aggressive.

```
bool function AddActors(Actor[] ActorList, Actor VictimActor = none)
```

Works in a similar way to the AddActor() function but accept an array of actors. If you want to specify the victim you have to pass it as an actor (and the actor has to be in the array.)

```
sslThreadController function StartThread()
```

This function will make the animation to start. It should be the very last function called after the Thread is fully initialized and defined. It returns an `sslThreadController` that gives you some control on the playing animation. If for some reasons the animation cannot start you will receive `None` as result.

```
function SetStrip(Actor ActorRef, bool[] StripSlots)
```

This function permits to override the body slots that have to be stripped by SexLab for the specific actor specific animation that will be played by the controller. `StripSlots` has to be a Bool array of size 33. If one position is True the corresponding Body Slot will be stripped, if it is False the Body Slot will NOT be stripped.

```
function DisableUndressAnimation(Actor ActorRef, bool disabling = true)
```

This function will disable or enable for the specified actor the undress animation. If `disabling` is True the strip animation will be disabled, if it is False the strip animation will be performed.

```
function DisableRagdollEnd(Actor ActorRef, bool disabling = true)
```

This function will enable or disable the ragdoll (the actors collapsing to the floor) at the end of the animation. If `disabling` is True the regdoll will be disabled, if it is False the ragdoll will be performed.

```
function DisableRedress(Actor ActorRef, bool disabling = true)
```

This function will disable or enable the redress of the actor at the end of the sex animation. If `disabling` is True the actor will NOT redress, if it is False the actor will redress.

```
function SetVoice(Actor ActorRef, sslBaseVoice Voice, bool ForceSilent =
false)
```

This function will force a voice on the specified actor. The same function can be used also to make the actor silent for the animation.

```
sslBaseVoice function GetVoice(Actor ActorRef)
```

Returns the voice associated to the given actor.

```
bool function IsUsingStrapon(Actor ActorRef)
```

Return True is the actor is wearing a strapon.

```
function EquipStrapon(Actor ActorRef)
```

Makes the specified actor to wear a strapon. If a strapon was set it will be used, if not then a random one will be chosen.

```
function UnequipStrapon(Actor ActorRef)
```

Removes the strapon from the actor.

```
function SetStrapon(Actor ActorRef, Form ToStrapon)
```

Set a specific strapon to the actor. The strapon can be one that is not in the SexLab strapon list.

```
Form function GetStrapon(Actor ActorRef)
```

Returns the strapon associated to the actor, if any.

```
function SetExpression(Actor ActorRef, sslBaseExpression Expression)
```

This function associates to an actor a specific facial expression.

```
sslBaseExpression function GetExpression(Actor ActorRef)
```

Returns the current expression associated to the actor.

```
int function GetEnjoyment(Actor ActorRef)
int function GetPain(Actor ActorRef)
```

Calculate and return the enjoyment or the pain the actor is having. See SexLab Stats.

```
int function GetPlayerPosition()
```

Return the position index of the Player actor in the actor list of the animation. It returns -1 is the player is not participating.

```
int function GetPosition(Actor ActorRef)
```

Return the position index of the specified actor in the actor list of the animation. It returns -1 is the actor is not participating.

```
bool function IsPlayerActor(Actor ActorRef)
```

Return true if the passed actor is the player. (If you need to check this in your code, it is faster to use something like: `if ActorRef == PlayerRef`)

```
bool function IsPlayerPosition(int Position)
```

Checks if in the specified position there is the player

```
bool function HasActor(Actor ActorRef)
```

Checks if the specified actor in in one of the positions.

```
bool function IsVictim(Actor ActorRef)
```

Checks if the specified actor is marked as victim for the animation.

```
bool function IsAggressor(Actor ActorRef)
```

Checks if the specified actor is marked as aggressor for the animation.

```
int function GetHighestPresentRelationshipRank(Actor ActorRef)
int function GetLowestPresentRelationshipRank(Actor ActorRef)
```

These two functions will return the highest and lowest relationship rank between the specified actor and all the other actors in the positions. The relationship rank is defined by the Construction Kit as:

4: Lover

3: Ally

2: Confidant

1: Friend

0: Acquaintance

-1: Rival

-2: Foe

-3: Enemy

-4: Arch nemesis

```
function ChangeActors(Actor[] NewPositions)
```

This function change the actors in the sexlab animation positions with the ones defined in the array. Warning: if the actors are not matching the original ones, a new animation will be selected. Use it with caution.

```
function SetForcedAnimations(sslBaseAnimation[] AnimationList)
```

Set a list of animations to be used for this thread. One of them, which can apply to the list of actors in the positions, will be randomly selected when the animation scene will start. If a forced animation is defined it will be used in priority.

```
function SetLeadAnimations(sslBaseAnimation[] AnimationList)
```

Set a list of animations to be used for this thread as lead in. One of them, which can apply to the list of actors in the positions, will be randomly selected when the animation scene will start for the lead in.

`function SetAnimations(sslBaseAnimation[] AnimationList)`

Set a list of animations to be used for this thread. If no forced animations are defined, then one of the passed animations, which can apply to the list of actors in the positions, will be randomly selected when the animation scene will start.

`function AddAnimation(sslBaseAnimation AddAnimation, bool ForceTo = false)`

Adds an animation to the list of animations or to the list of forced animations.

`function DisableLeadIn(bool disabling = true)`

This function enables and disables the lead-in animation for the animation scene.

`function DisableBedUse(bool disabling = true)`

Disable or enable the utilization of beds for this thread. This will not overwrite the normal SexLab config.

`function SetBedFlag(int flag = 0)`

This function is a little bit more powerful of the previous one. Three possible values are possible:

-1 → Never use beds, no matter what.

0 → Use the normal SexLab configuration

1 → Force the use of a bed (of course a bed has to be found)

`function SetTimers(float[] setTimers)`

Replaces the times for each stage of the animation with the ones defined in the parameter.

`function CenterOnObject(ObjectReference CenterOn, bool resync = true)`

```
function CenterOnCoords(float LocX = 0.0, float LocY = 0.0, float LocZ = 0.0,
float RotX = 0.0, float RotY = 0.0, float RotZ = 0.0, bool resync = true)
```

These functions will make the SexLab animation to play centered on the specified `ObjectReference` or the specified coordinates. The parameter `resync` is not used.

`bool function CenterOnBed(bool AskPlayer = true, float Radius = 750.0)`

This function will search for a bed in the proximity of the player and will center the animation on it. If `AskPlayer` is True then if a bed is found a popup will be proposed to the player to use the bed or not.

Using hooks

What are hooks? Hooks are used in SexLab to call a function you defined when a specific animation event will happen inside a SexLab animation.

A hook itself is just a string that you use to tell SexLab to send a ModEvent to your mod when something happens.

Hooks work in conjunction with the SKSE function RegisterForModEvent.

To actually use them you need to ask SexLab to record your Hook, and then register for the mod event to specify which function to call when the event will be generated by SexLab Thread Controller.

The standard event managed by SexLab are:

AnimationStart → This event is sent at the very begin of the animation setup (Preparation state), nothing is yet started here.

AnimationEnding → Sent just before the animation is concluded. In this phase some small actions are performed, just after the animation will be completed and concluded.

AnimationEnd → This is the very last event sent. It is sent when the animation actually ends.

LeadInStart → This event is sent if there is a lead-in and when the lead-in part of the animation is starting.

LeadInEnd → This event is sent IF there is a lead-in and when the lead-in part of the animation is concluded.

StageStart → This event is sent at begin of each stage of the animation.

StageEnd → This event is sent at end of each stage of the animation.

OrgasmStart → This event is sent at the very end of the stages, when the actual stage starts. It is not sent for Lead-In animations.

OrgasmEnd → This event is sent at the very end of the stages, when the actual stage ends. It is not sent for Lead-In animations.

AnimationChange → This event is sent if the animation is changed by the hot-key.

PositionChange → This event is sent if the positions of the actors in the animation is changed by the hot-key.

ActorChangeStart → This event is sent when the actors are changed (by the equivalent function proposed by the Thread Model.)

ActorChangeEnd → This event is sent when the actors are changed (by the equivalent function proposed by the Thread Model.)

ActorsRelocated → This event is sent when the actors are re-aligned or re-centered while the animation is playing.

Do how can is receive an event by SexLab? This is a small piece of code (it is not a full example, see the last section of this chapter for full examples):

```
Function doSex(Actor aNPC)
    sslThreadModel thread = SexLab.NewThread()
    thread.AddActor(PlayerRef)
    thread.AddActor(aNPC)
    thread.SetHook("MyHook")
    RegisterForModEvents("AnimationEnd_MyHook", "SexIsCompleted")
    thread.Start()
EndFunction

Event SexIsCompleted(string eventName, string argString, float argNum, form 
sender)
    Actor[] Positions = SexLab.HookActors(argString)
    Int num = Positions.length
    While (num)
        num -= 1
        String name = Positions[num].getActorBase().getName()
        Debug.Notification("The SexLab Animation ended for Actor: " + name)
    EndWhile
    UnregisterForModEvents(eventName)
EndEvent
```

The function DoSex() will ask SexLab for a thread. On the thread we just add two actors, the Player and the other NPC that is a parameter for the function.

Then we define a hook. The hook is called "MyHook".

Just after this we are registering for a mod event, which will be sent by sexlab, for the event "AnimationEnd". To do this we do a RegisterForModEvents and the event name we are defining is "AnimationEnd", underscore, the name of your hook.

Then we start the animation.

When the event will be generated SKSE will call the function we specified (SexIsCompleted.)

This event will be called with a few parameters. We can use some functions to better understand what called us. (In the example we just list all the actors that were participating.)

The functions given by the Thread Model are:

```
function SetHook(string AddHooks)
```

Add a hook to the animation. All possible events will be sent for this hook. You have to register for the ones you want to manage.

```
string[] function GetHooks()
```

Returns an array of strings with all the hooks defined for the animation.

```
function RemoveHook(string DelHooks)
```

Removes a specific hook from the animation.

If you use the StartSex() function of SexLab you can specify an hook for it. This hook, if defined, will be associated to all events. Of course you have to register for mod events to actually receive them. Because with StartSex you have very little control, it is better to register for the mod events BEFORE calling the function StartSex() or you may not receive the first events.

Now, how you can get something in your event management function?

SexLab offers some functions to get information about the animation when an event is received. All these functions require as parameter the `argString` variable that is sent to the event function you define.

These functions can be called ONLY inside the event function you define to manage the hooks.

```
sslBaseAnimation function HookAnimation(string argString)
```

Returns the actual animation that is played.

```
int function HookStage(string argString)
```

Returns the stage number of the stage that is played.

```
Actor function HookVictim(string argString)
```

Returns the victim of the animation if defined.

```
Actor[]function HookActors(string argString)
```

Returns all participants (actors involved) in the animation.

```
float function HookTime(string argString)
```

Returns the total time of the animation.

Getting and settings SexLab stats

This may change in SexLab 1.60, it will be written in a future version of this document.

TODO

Defining and registering a new animation

If you want, you can define and add new animations to SexLab. Like Non-SexLab Animation Pack, or Zaz Animation Pack, or More Nasty Critters are doing.

To do this of course you need the actual animation files (*.hkx), and then you have just to create a registering function that will be called by SexLab to actually register the new animation, and last ask SexLab to register the animation.

After this your animation can be used in SexLab as any other animation.

To actually do this you need a script that extends `sslAnimationFactory`.

```
ScriptName myExtraSexLabAnimation extends sslAnimationFactory
```

Inside somewhere this script you have to call two SexLab functions (actually sslAnimationFactory functions):

PrepareFactory(), and RegisterAnimation().

PrepareFactory should be called only once at the very begin (you can then register one or more animations.)

RegisterAnimation is the actual registration of the animation.

Register animation requires a single parameter: the id of the animation.

By calling this function, SexLab will call, any time is needed, a function of your script that NEEDS TO HAVE exactly the same name as the id of the animation.

Let's imagine you want to add a new animation for two human actors that has the ID: "mySpecialAnim".

You need to call the function: `RegisterAnimation("mySpecialAnim")`

And then you need to have in the script a function defined as:

```
Function mySpecialAnim(int id)
```

Inside this function you have to fill up a sslBaseAnimation. This object can be obtained by using the function Create().

sslBaseAnimation anim = Create(id)

The id HAS TO BE the id that is the parameter of the function you defined to instantiate the animation.

At this point you can use the function from `sslBaseAnimation` to actually fill all the required values for your animation (content, sound, positions, stages, tags, etc.)

The code should be more or less like this one:

```
Function mySpecialAnim(int id)
	sslBaseAnimation Base = Create(id)
	Base.Name = "My Special SexLab Anim"
	Base.SetContent(Sexual)
	Base.SoundFX = Squishing

	int a1 = Base.AddPosition(Female, addCum=Oral)
	Base.AddPositionStage(a1, "AP_SkullFuck_A1_S1", 0)
	Base.AddPositionStage(a1, "AP_SkullFuck_A1_S4", 0, openMouth = true)

	int a2 = Base.AddPosition(Male)
	Base.AddPositionStage(a2, "Arrok_MaleMasturbation_A1_S2", 0, sos = 2)
	Base.AddPositionStage(a2, "Arrok_MaleMasturbation_A1_S4", 0, sos = 3)


	Base.AddTag("Oral")
	Base.AddTag("Masturbation")
	Base.AddTag("MF")


	Base.Save(id)
endFunction
```

The first thing to do it to have the animation. The first line will do it.

Then you provide a name for the animation.

The animation has content Sexual and will use Squishing sounds.

Then you are adding the first position. Remember the first position is usually the “receiver” (female usually.) Here you define also is some cum effect has to be applied at the end and where.

Then you add, for this position, two stages. For the first stage you define the .hkx animation only. For the second stage you define the animation and make the female to open the mouth.

Then a second position is added. The second position will use a couple of HKX animations used for male masturbation. The extra parameter is used when Shlongs of Skyrim is available and will define how to bend the penis.

At the end we are defining a few tags for the animation (Oral, Masturbation, and MF.)

Last we save the animation.

If you execute the code a new animation will be added to SexLab with what you defined.

To have good results you should also define correctly the positions and the alignment for the specific stages. But this is beyond this training guide.

Adding your own Strapon to SexLab

SexLab comes with default with a single strapon. It gets other strapons from well-known mods.

If your mod is adding a strapon you may want it to be added to the SexLab strapon list.

To do this you can use a function defined by `sslSystemConfig`.

```
Armor function LoadStrapon(string esp, int id)
```

This function will get a strapon from a mod specified as "esp" with the id specified, and will add it to the SexLab strapon list.

It should be something like:

```
sslSysConfig.LoadStrapon("MyWonderfulMd.esp", 0x021FE)
```

Remember to use only the last 5 hexadecimal digits of your strapon (it has to be an Armor). The previous digits are used to index the mods and are not required.

SexLab ready to use examples

In this section a few ready-to-use SexLab examples.

Start a quick sex with a dialogue

Create a new dialogue in your quest. Call it myModQuickSexDialogue.

Crate a branch and its topic inside: myModQuickSexBranch and myModQuickSexTopic.

Create a topic info inside the topic, as prompt write "Do sex with me" and whatever you want as response (you need one.)

In the Topic Info window you will see the papyrus fragment containers. Just put a semi-colon (;) inside the first one and then close the topic info (this will generate the script for you.)

Open the topic info, you will see a script on the right corner. Edit it. Do not touch any of the existing lines of code. At the very bottom add:

```
SexLabFramework Property SexLab Auto

Actor Property PlayerRef Auto
```

Save and close.

Now in the begin code, where you put the semi-colon, remove the semicolon and add these lines of code:

```
Actor[] Positions = new Actor[2]

Position[0] = PlayerRef

Position[1] = akSpeaker

SexLab.StartSex(Positions)
```

Close the topic info. And do not forget to assign the properties on the script in the properties window.

Congratulations you just added a new line of dialogue (works on all NPCs, we did not put conditions), and just after you say it to a NPC a sex animation will start.

Now let's make this a little bit more advanced. Create a new branch in your dialogue. In the topic create a new topic info. Use as Prompt: "Give me your ass", do a response (does not matter what you will write.)

Create again the script in the topic info (it will be a Topic Fragment), by putting just a semi-colon.

Add exactly the same properties as the previous example. Save and close.

Now in the code of the begin fragment write:

```
; In this case the player will be active, so it is the second position

SexLab.QuickStart(akSpeaker, PlayerRef, AnimationTags="Anal")
```

Save, set values for the properties. Congratulations, now you have a second line that will start an anal animation.

Start sex with a trigger

Define a trigger

Catch the OnTriggerEnter

Pick the actor and a few actors around

Start a sex scene

TODO

Do two sex animation one after another

Do a hook that will start a second animation

TODO

Start a sex scene inside a Quest Scene

TODO