Contents

1. Introduction 02
2. Skeleton of Aegisub Scripts 02
3. Aegi 03
3.1. ffm 03
3.2. mff 03
3.3. progressTitle 04
3.4. progressTask 04
3.5. progressSet 05
3.6. progressReset 05
3.7. progressCancelled 06
3.8. progressLine 06
3.9. videoIsOpen 07
3.10. getFramerate 07
4. Logger 08
4.1. Fatal 08
4.2. Error 08
4.3. Warn 08
4.4. Hint 08
4.5. Debug 08
4.6. Trace 08
4.7. Log 09
4.8. Assert 09
4.9. windowError 010
4.10. windowAssertError 010
4.11. lineWarn 011
4.12. lineError 011
5. Ass 012
5.1. Line Collection 013
5.2. Loop through collected lines 014
5.3. Remove Lines 015
5.4. Insert Line 015
5.5. Commit changes to the subtitle 016
5.6. Create Lines 017
5.7. Add Style 017
5.8. Update Metadata 017
5.9. Parse Line 017
5.10. Get selection 017

1. Introduction

ILL is a module that aims to make working with subtitle objects efficient. It does most of the heavy lifting so that we can do more with fewer lines of code in our script. It provides various functions that allow us to work with lines, tags, drawings, text and comments. This makes it unnecessary to reinvent the wheel and write your own functions to do most of the common task while still allowing you to do complex tasks more easily.

2. Skeleton of Aegisub Scripts

```
export script_name = "name of the script"
export script_description = "description of your script"
export script_version = "0.0.1"
export script_author = "you"
export script_namespace = "namespace of your script"

DependencyControl = require "l0.DependencyControl"
depctrl = DependencyControl{
    {
        {
            "ILL.ILL"
            version: "0.0.1"
            url: "https://github.com/TypesettingTools/ILL-Aegisub-Scripts"
            feed: "https://raw.githubusercontent.com/TypesettingTools/ILL-Aegisub-
Scripts/main/DependencyControl.json"
        }
    }
}
ILL = depctrl\requireModules!
{:Aegi, :Ass} = ILL

functionName = (sub, sel, act) ->
    -- stuff goes here

depctrl\registerMacro functionName
```

This is the framework that all your scripts will have. Here we import ILL as well as classes that ILL offers that we need in this script. Here, we only import `Aegi` and `Ass` class but ILL offers much more and user should import them as they are needed. Finally, we define a function called `functionName`. This function name is what we register in Aegisub in the last line, and it gets executed as soon as we run the script. Everything we do in the guide below will go inside the function where `--stuff goes here` is written.

3. Aegi

Class `Aegi` deals with various Aegisub api and provides convenience function wrappers around them.

3.1. ffm

`ffm` uses the loaded framerate data to convert absolute time in milliseconds to a frame number.

3.1.1. Arguments

Argument	Description	Type	Default
ms	Time in milliseconds	Integer	-

3.1.2. Returns

If video is not open, it returns nil.

Returns	Description	Type
frame or nil	Frame corresponding to time	integer or nil

3.1.3. Usage

```
frame = Aegi.ffm 1000
frame = Aegi.ffm line.start_time
```

3.2. mff

`mff` uses loaded framerate data to convert a frame number of the video into an absolute time in milliseconds.

3.2.1. Arguments

Argument	Description	Type	Default
frame	Frame number of video	integer	-

3.2.2. Returns

If video is not open, it returns nil.

Returns	Description	Type
millisecond or nil	Time in milliseconds	integer or nil

3.2.3. Usage

```
ms = Aegi.mff 100
line.start_time = Aegi.mff 500
```

3.3. progressTitle

A progress dialog box is always shown when an Aegisub script is run. Aegisub allows user to modify what is shown in it. `progressTitle` sets the title for the progress window. Title is the large text displayed above the progress bar. This text should usually not change while the script is running. By default, title is set to the name of the macro running.

3.3.1. Arguments

Argument	Description	Type	Default
title	Title to set int the progress window	String	-

3.3.2. Returns

Returns
Nil

3.3.3. Usage

```
Aegi.progressTitle "Title of the progress window."
```

3.4. progressTask

`progressTask` sets the text for the progress window. Task is the small text below the progress bar showing what the script is currently doing. It is updated by the user as the task of the script changes.

3.4.1. Arguments

Argument	Description	Type	Default
task	Task that the script is currently executing	String	“”

3.4.2. Returns

Returns
Nil

3.4.3. Usage

```
Aegi.progressTask "Plotting World Domination."
```

3.5. progressSet

`progressSet` generates a progress bar in the progress window ranging from 0% to 100%.

3.5.1. Arguments

Argument	Description	Type	Default
i	Current index	Integer	-
n	Total index	Integer	-

3.5.2. Returns

Returns
Nil

3.5.3. Usage

```
for i = 1, 20
    Aegi.progressSet i, 20
```

3.6. progressReset

`progressReset` resets all the progress by setting the progress bar to 0% and progress task to empty string.

3.6.1. Arguments and Returns

Argument	Returns
Nil	Nil

3.6.2. Usage

```
Aegi.progressReset!
```

3.7. progressCancelled

`progressCancelled` checks if the user has cancelled the execution of script. In such a case, it stops any further execution of the script. It should be used inside loops or callback functions such that the user does not have wait until it is completed to stop the execution of the script.

3.7.1. Arguments and Returns

Argument	Returns
Nil	Nil

3.7.2. Usage

```
Aegi.progressCancelled!
```

3.8. progressLine

`progressLine` is a combination of a couple of functions that we have seen above so that we can execute all of them at once. It is recommended to use this unless for some reason you need to execute them separately.

This sets the progress bar, sets the progress task and checks for user cancellation. This is ideal to be used inside a loop or callback function.

3.8.1. Arguments

Argument	Description	Type	Default
Line	Line table as given by `Ass` class	Table	-
i	Current index	Integer	-
n	Total index	Integer	-

More info about getting line table from `Ass` class is discussed later.

3.8.2. Returns

Returns
Nil

3.8.3. Usage

```
ass = Ass sub, sel
ass\iterLines (l, i, n) ->
    Aegi.progressLine l, i, n
```

3.9. videoIsOpen

`videoIsOpen` is used to find if video is currently open in Aegisub or not.

3.9.1. Arguments

Argument
Nil

3.9.2. Returns

Returns	Description	Type
videoState	State of video	True if open and false if closed

3.9.3. Usage

```
if Aegi.videoIsOpen!
    -- video is open
else
    -- video is not open
```

3.10. getFramerate

`getFramerate` returns the framerate of the video that is currently open. If the video is not open, it returns the default value of 24000 / 1001.

3.10.1. Arguments

Argument
Nil

3.10.2. Returns

Returns	Description	Type
framerate	Framerate of the video	Float

3.10.3. Usage

```
framerate = Aegi.getFramerate!
```

4. Logger

Class `Logger` allows you to log messages to the progress window. If messages are logged by a script, the progress window stays open after the script has finished running until the user clicks the `Close` button.

Logger has two components: Log level and messages. Log level indicates the severity level of the message while messages are the string that you want to log.

By default, Aegisub's log level is set to 3 which means that the message above 3 won't be seen by end user unless they set the log level higher themselves.

4.1. Fatal

These message indicate something really bad happened and the script cannot continue. The log level of these messages is 0. Level 0 messages are always shown regardless of trace level settings in Aegisub. The execution of the script will end after this message is logged.

4.2. Error

An error indicates the user should expect something to have gone wrong even though you tried to recover. A fatal error might happen later. The log level of error messages is 1.

4.3. Warn

A warning indicates something is wrong and the user ought to know because it might mean something needs to be fixed. The log level of warning messages is 2.

4.4. Hint

A hint indicates something that the user should know that is not necessarily a cause for alarm. The log level of hint messages is 3.

4.5. Debug

A debug message includes information meant to help fix errors in the script, such as dumps of variable contents. Since the default trace level of Aegisub is 3, debug messages, which has the log level of 4, won't be seen by average user of the script unless they manually changed it. This is useful for script writers during the debugging of their scripts.

4.6. Trace

A track message includes extremely verbose information about what the script is doing, such as a message for each single step done with lots of variable dumps. The log level of trace message is 5.

4.7. Log

This is simply a wrapper around hint message who don't want to use `hint` and want to use `log` which seems synonymous to simply logging messages.

Arguments

Argument	Description	Type	Default
messages	Message you want to show	string / number / boolean / table	-
...	Parameters to the format string	lua string.format parameters	-

Returns

Returns
Nil

4.7.1. Usage

```
Logger.fatal "This is a fatal message."     -- log level 0
Logger.error "This is an error message."    -- log level 1
Logger.warn "This is a warning message."    -- log level 2
Logger.hint "This is a hint message."       -- log level 3
Logger.debug "This is a debug message."     -- log level 4
Logger.trace "This is a trace message."     -- log level 5

Logger.log "This is a simple message."      -- log level 3
```

You can also use Lua's string.format method.

```
Logger.fatal "There are %d %s messages.", 5, "fatal"
```

This will yield the message: `There are 5 fatal messages.`

4.8. Assert

`assert` will only show a message if a condition is false. After showing the message, further execution of script is terminated.

4.8.1. Argument

Argument	Description	Type	Default
condition	Condition to check	lua statement or boolean	-
message	Message you want to show if condition is false	string / number / boolean / table	-

Argument	Description	Type	Default
...	Parameters to the format string	lua string.format parameters	-

4.8.2. Returns

Returns
Nil

4.8.3. Usage

```
Logger.assert 1 > 2, "Unfortunately, 1 was not greater than 2."
```

4.9. windowError

Instead of using progress window, if the user wants to use Aegisub dialog to show messages, this can be used. Currently this only supports text messages. The execution of the script will be terminated after the message is shown.

4.9.1. Arguments

Argument	Description	Type	Default
message	Message you want to show	string / number / boolean / table	-
...	Parameters to the format string	lua string.format parameters	-

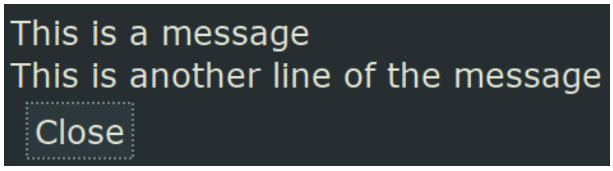
4.9.2. Returns

Returns
Nil

4.9.3. Usage

```
Logger.windowError "This is a message\nThis is another line of the message"
```

The above command will yield:



4.10. windowAssertError

`windowAssertError` is the same as the Section 4.9 `windowError` and Section 4.8 `assert` combined.

4.10.1. Arguments and Returns

Same as Section 4.8

4.10.2. Usage

```
Logger.windowAssertError line.start_time < line.end_time, "This is a message\nThis
is another line of the message"
```

4.11. lineWarn

`lineWarn` is used for showing warning for a particular line.

4.11.1. Arguments

Argument	Description	Type	Default
line	line collected by `Ass` class	table	-
message	Message you want to show	string	"not specified"

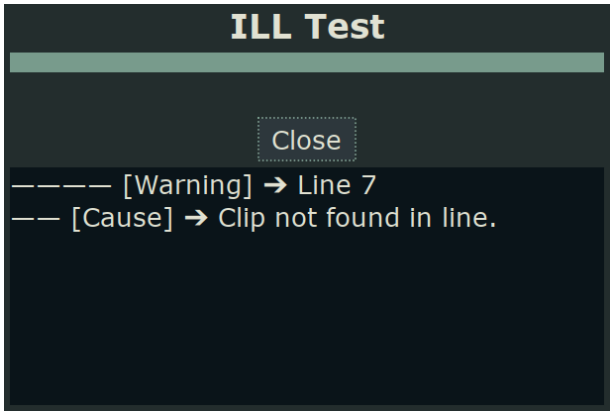
4.11.2. Returns

Returns
Nil

4.11.3. Usage

```
ass = Ass sub, sel
ass\iterLines (l, i, n) ->
    Logger.lineWarn l, "Clip not found in line."
```

The code above yields:



4.12. lineError

`lineError` is used for showing critical warning for a particular line. The execution of the script is terminated after showing this message.

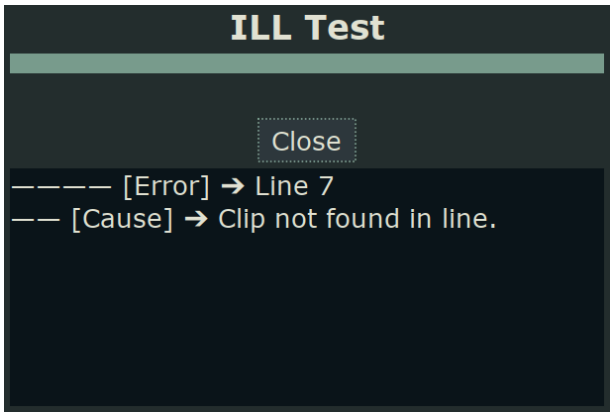
4.12.1. Arguments and returns

Same as Section 4.11

4.12.2. Usage

```
ass = Ass sub, sel
ass\iterLines (l, i, n) ->
    Logger.lineError l, "Clip not found in line."
```

The code above yields:



5. Ass

One of the first things we do when we write a script is we initiate the `Ass` instance. It's job is to collect the basic information of the subtitle files, lines of the files and provide various methods to act on them.

```
main = (sub, sel, act) ->
    ass = Ass sub, sel, act
```

Arguments

Argument	Description	Type	Default
sub	Subtitle object given by Aegisub	userdata	-
sel	(Optional) Selected lines	table[integer]	-
act	(Optional) Active line	integer	-
callback	(Optional) condition for line collection	function	`(l) -> not l.comment`

5.1. Line Collection

As we initialize the `Ass` class, we are collecting lines to work on at the same time. By default, the condition `(line) -> line.comment` suggests that only uncommented lines are collected but that can be changed.

The collected line table will have lines with all the fields of a normal line table but it adds other fields.

Argument	Description	Type
duration	duration of line in milliseconds	integer
startFrame	start frame of a line	integer
endFrame	end frame of a line	integer
frameCount	total number of frames	integer
styleRef	style table of current line	table
absoluteIndex	actual index of line in subtitle file	integer
naturalIndex	index of line as seen in Aegisub	integer

5.1.1. Collect all dialogue lines

If you only send sub as an argument, all uncommented lines will be collected.

> **Note**
>
> All lines does not mean all lines of the subtitle. It means all the dialogue lines as seen in Aegisub.

```
main = (sub, sel, act) ->
    ass = Ass sub
```

5.1.2. Collect all selected lines

If you send both `sub` and `sel` as arguments, only the uncommented dialogue lines among the selected lines will be collected.

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
```

5.1.3. Choosing conditions for collection

There may be cases where you want to only collect certain types of lines. In that case, you can override the default condition for line collection.

If you don't like that it skips over commented lines and want it to be included, collect lines as shown below. It will select all selected lines without checking anything.

```
ass = Ass sub, sel, act, -> return true
```

In fact, you can give any condition here and if that condition is fulfilled, only those lines will be collected i.e. if the callback function returns `true`, the line will be collected and vice versa. For example, if you want to only collect commented lines.

```
ass = Ass sub, sel, act, -> return line.comment
```

If you want to only collect lines whose layer is 1:

```
ass = Ass sub, sel, act, ->
    return line.layer == 1
```

If yo want to only collect lines whose text contain certain substring:

```
ass = Ass sub, sel, act, ->
    return line.text\match "text"
```

5.2. Loop through collected lines

Now that we have collected lines, we might want to iterate through it and work on individual lines.

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0          -- Return early if no lines were collected.
    ass\iterLines (l, i, n) ->
        -- do things to individual line here.
```

5.2.1. Arguments

One of the argument of `ass\iterLines` is a callback function which means it will offer you a few arguments that you can use.

Argument	Description	Type	Default
l	line table	table	-
i	current iteration of line	integer	-
n	total number of lines in collection	integer	-

The argument `l` is the line table which contains all the information about current line. However, the other arguments `i` and `n` are purely for progress reporting. An example was shown in Section 3.8.

5.2.2. Returns

While iterating through the lines, if we want to stop the iteration, we can return false.

Returns	Description	Type
stop iteration	if iteration should be stopped	boolean

For example:

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0
    ass\iterLines (l, i, n) ->
        if condition
            return false
            -- As soon as this callback function returns false, the iteration
will stop.
```

5.3. Remove Lines

If you want to remove lines from the collection, you can mark them for removal during the iteration.

> **Warning**
>
> This only marks the line for removal. Line has not been actually removed from the subtitle yet. Look Section 5.5 for more.

5.3.1. Arguments

Argument	Description	Type	Default
line	line collected by `Ass` class	table	-

5.3.2. Returns

Returns
Nil

5.3.3. Usage

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0
    ass\iterLines (l, i, n) ->
        if i % 3 == 0
            ass\removeLine l
```

5.4. Insert Line

You want to insert lines to the subtitle at some point. If you are iterating through the collection, make a copy of the line before inserting it or you can create a line from scratch to insert at any time. Look Section 5.6 for how to create lines from scratch.

> **Warning**
>
> This only marks the line for insertion. Line has not been actually inserted to the subtitle yet. Look Section 5.5 for more.

5.4.1. Arguments

Argument	Description	Type	Default
line	line collected by `Ass` class	table	-
index	(Optional) index of the collection at which to insert the line	integer	#ass.lines

In case the user gives the index too low or high, the index is clamped to be 1 and total number of lines so that it is always within the collection. If index is not provided, it will add the line at the end of the collection.

5.4.2. Returns

Returns
Nil

5.4.3. Usage

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0
    ass\iterLines (l, i, n) ->
        line = Table.deepcopy l
        -- Make some changes to the copy
        line.layer += 1
        line.text = "This is a copy of line #{line.naturalIndex}"
        if i % 3 == 0
            ass\insertLine line  -- inserts line at the end of collection
        elseif i % 5  == 0
            ass\insertLine line, i  -- inserts line right after current line
```

5.5. Commit changes to the subtitle

After we make various changes to the line collection, this is where we actually commit the changes to the subtitle file. This removes all the lines that were marked for removal, inserts lines at the index the user asked them to insert and apply the changes the user makes on the line.

5.5.1. Arguments

Argument	Description	Type	Default
updateRefs	update references of the line after comitting	boolean	false

> **Note**
>
> When the user commits and makes changes to the subtitle file, the line collection will become outdated. The indices may point to wrong lines. If the user changes times of the line, the fields related to frames also become wrong. If the user does not want to work with the line collection again (i.e. the script no longer has to iterate through lines again), there is no need to update references. However, if the user wants to iterate through lines again, we need to fix the references so that we're sure we're working with proper lines.

5.5.2. Returns

Returns
Nil

5.5.3. Usage

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0
    ass\iterLines (l, i, n) ->
        if i % 3 == 0
            line = Table.deepcopy l
            line.text = "This is a copy of line #{line.naturalIndex}"
            ass\insertLine line, i
        elseif i % 5  == 0
            ass\removeLine l
    ass\commit!
```

```
    ass\commit true -- if you want to update the line collection
```

5.6. Create Lines

5.7. Add Style

5.8. Update Metadata

5.9. Parse Line

5.10. Get selection

This gets the table of selected lines. This is most useful to be returned by the main function of the script and whatever indices are present in the table, those lines will be selected after the script is run.

5.10.1. Arguments

Argument	Returns
Nil	Nil

5.10.2. Returns

Returns	Description	Type
sel	indices of selected lines	table

5.10.3. Usage

5.10.4. Usage

```
main = (sub, sel, act) ->
    ass = Ass sub, sel
    return if #ass.lines == 0
    ass\iterLines (l, i, n) ->
        -- make changes here
    ass\commit!
    return Ass\getSelection!
```

As the user removes line or inserts line, the selection table gets updated. Therefore, `getSelection` returns the proper indices after working with lines.