DCS World Beginners Guide to Systems Coding

BETA ACCESS

Table of Contents

Modeling

Contributions

List of Contributors

by Sirius

We would like to thank a list of people being able to help with the Beginners Guide to Systems Coding (BGSC) guide. We as the group have always been wanting to be able to thoroughly, yet simply describe the details and process of making an External Systems Model (ESM) but have never been able to find the easiest approach. With the dedication of these people, that goal has been successfully accomplished.

- **Sirius**. Sirius has led progress of the BGSC guide for the majority of the time. 3 months were spent on dedicating to the researching of how ESM works with the DCS API, functions, and data environment. Various subjects, such as the MB-339A/PAN mod, AJS-37 Lua code, and TF-51D Lua code were used to grapple an understanding of how an ESM is coded and works.
- **CubanAce**. CubanAce, receiving help from the creators of the MB-339A/PAN mod: Frecce Tricolori Virtuali, was able to give simple details and help on proper code details on element pointers. CubanAce's work on the *Making a clickable cockpit* Guide led by Level and himself were also helpful to give further extent in details on this guide. Cuban is well known for his Su-57 "PAK FA" mod, along with simple AI flyable assets.

Setting up

Software

by Sirius

Before you begin learning about how to program your own set of systems, avionics, or instruments in DCS World for a particular aircraft, you need to have the correct software on your computer or laptop device installed. Here is a current list of appropriate software that you are able to use for this:

- Sublime Text
- Microsoft Visual Studio Code (VSCode)
- Notepad++
- Atom Editor

There are other types out there than those 4, but here is an example of what it may look like:

For this particular type of software (Atom Editor) I prefer it due to the ability of it having many packages set up for you and easy to install to run the code.

Prerequisite knowledge

by Sirius

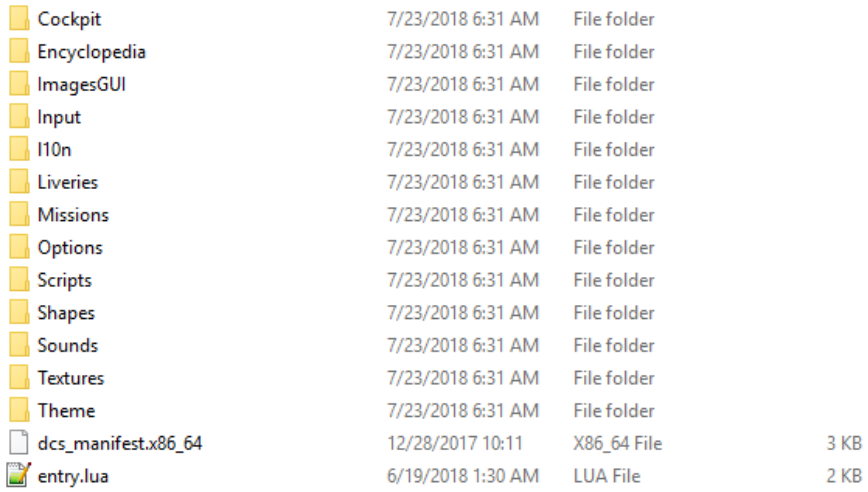
It is known to many modders that ESM is not a fairly easy subject, and for the most part, that is the truth. A lot of time is put into researching how the DCS API functions and interprets code. Having a decent understanding of how the Lua code interacts with the mods and modules inside of DCS can give you a big benefit. With that being said, there are some things you need to know before delving into this type of said subject:

1. You need to know how to make a basic mod structure. This includes the gear positions, canopy, pylons, cockpit implementation, and collision model apart of the folder structure.
2. You need to know how Lua works. If you have absolutely no idea or are foreign to the subject of programming itself, then it may be very unlikely that you are going to get far in learning how to program an ESM for DCS World. It is *highly recommended* for someone to understand how to code anything in a programming language, more preferably Lua itself in order to understand what code you are going to be dealing with.
3. You need to know how to work with debugging and working with certain code statements within DCS. The core infrastructure that the DCS API works with, especially when dealing with Lua, acts as a framework. (A framework is a predefined set of functions to write inside of a script that is being inherited by an overall class like script or package) Being able to debug your own code (ie. using the dcs.log folder in your C:\Users\<< insert username here >>\Saved Games\DCS\Logs directory) is a huge must in order to find out the issue in your script(s).

Folder Structure

by Sirius

There is a specific folder structure that almost virtually every mod should adhere to by, whether or not it is being created for ESM. The picture is shown below shows the areas of the folder paths and structure.



↓ *Inside of Cockpit* ↓

↓ *Inside of Cockpit/Resources* ↓

↓ *Inside of Sounds* (later on for warnings) ↓

The rest of these paths are optional and are the basic mod structure layouts.

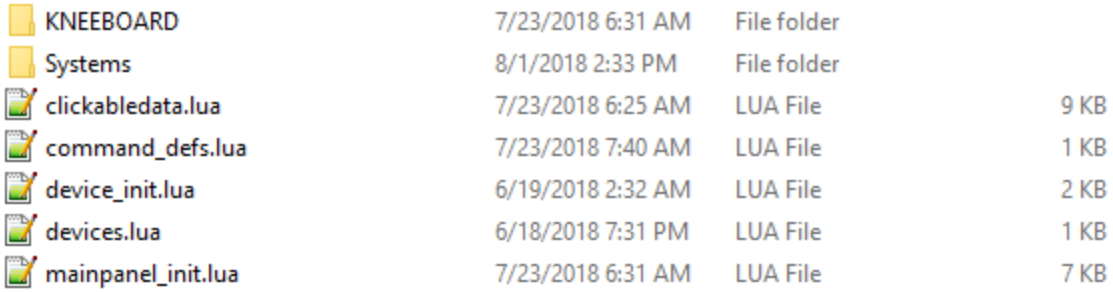
Programming

Lua setup

by Sirius & CubanAce

Before you can open up your text editor and begin writing your first lines of code for an ESM, (for others it may just be refreshing up on the topic of it) we are going to need to create a set of files in order to help us out when it comes to defining things like pointer elements, devices, and clickable data. Here is the list of files you are going to need to create. (Place this in your Cockpit/Scripts directory)

[Fig. 1.1]



And before we get any farther on this, we need to edit the entry.lua file so that it calls the Cockpit/Scripts directory path.

[Fig. 1.2.]

```
62  dofile(current_mod_path .. "/Scripts/modname.lua")
63  dofile(current_mod_path .. "/Scripts/Weapons.lua")
64  make_flyable('modname', current_mod_path..'/Cockpit/Scripts/', nil, current_mod_path..'/Scripts/comm.lua')
65  dofile(current_mod_path .. "/Scripts/Views.lua")
66  make_view_settings('modname', ViewSettings, SnapViews)
```

Line 64 is of most relevance.
current_mod_path .. '/Cockpit/Scripts', nil
is what is needed.

We have put together the following links for you so that you can easily get the base files shown in fig. 1.1 without having to make them yourself which may be hard for a beginner.

- clickabledata.lua: https://pastebin.com/k0LRHF6L
- command_defs.lua: https://pastebin.com/edYQ4nNP
- device_init.lua: https://pastebin.com/9F9qYseL
- devices.lua: https://pastebin.com/06uarV2g
- mainpanel_init.lua: https://pastebin.com/LKeDfMw6

All of these files that have been specified the links are *are absolutely vital* in order for you to run this properly and code an ESM. Here are the meanings of all of the files:

- **clickabledata.lua**: This file is needed in order to specify your point helpers and elements of the cockpit. Without this, it is virtually impossible for you to hover over switches and knobs in the cockpit or whatever accessory and be able to click it. (This is also needed for making training missions in DCS)
- **command_defs.lua**: This file is responsible for holding specified internal numbers for the arguments and actions of certain things in DCS. (An example such as the landing gear requires the internal ID of 68.)
- **device_init.lua**: This file helps you declare the system type of your systems, the file path, the mainpanel_init file, and your 3D representations (ie. MFD, HUD)
- **devices.lua**: This file is well responsible for creating the IDs for each device in the systems model.
- **mainpanel_init.lua**: This file is responsible for containing your cockpit coordinates with ESM (your Views.lua file is ignored with a systems model) and your gauges. Either base gauges, which are the main animations ie. Mach Number or Stick Roll, or the parameter gauges.

Section 1-A

avLuaDevice

by Sirius

The term "avLuaDevice" in DCS refers to a "device" (derogatory script file) within DCS that runs Lua statements. This can have many uses, such as, but not limited to:

- External animations
- Cockpit statements
- Startup/weapons systems
- Gauges/instrumentation, and also the movement of switches

avLuaDevice is the base for most, if not all code that is written for an ESM in DCS. However, other types, such as the avSimpleElectricSystem are specially designed for coding electrical systems for the aircraft that the DCS API targets when triggered events are called for those actions. We are going to write a new line in our Cockpit/Scripts/device_init.lua and Cockpit/Scripts/devices.lua:

↓ *device_init.lua* ↓
(involves creating a new file in Systems folder of Cockpit/Scripts path)

```
34  creators[devices.TEST] = {"avLuaDevice", LockOn_Options.script_path .. "/Systems/test.lua"}
```

↓ *devices.lua* ↓

```
8  devices["TEST"] = counter() --1
```

Now that we have initialized these lines into the specified 2 files above, we may now begin coding the test.lua file:

```
dofile(LockOn_Options.script_path.."command_defs.lua")
dofile(LockOn_Options.script_path.."devices.lua")

local test_system = GetSelf()
local dev = GetSelf()

local update_time_step = 0.01
local sensor_data = get_base_data()

make_default_activity(update_time_step)
```

To get an understanding of what is going on, we need to understand what these functions we are calling for actually mean.

1. **GetSelf()**: this is a function that is called in a device file in order to start initializing and get the name of the instance.
2. **update_time_step**: a local, file-based variable that is created in order to help with the script refresh loop at 0.01 seconds, or also known as 10 milliseconds. The script always updates by a millisecond, so this will update 10 times every second. This is vital to know if in the future you are going to do animations with an avLuaDevice to help make it smooth and do correct transitions.
3. **sensor_data**: inside of this variable is a function being called, get_base_data(). This function is done in order to initialize the script and get the information needed to input from the DCS API. (This includes things like your pitch, roll, and yaw values; and others like AoA.)
4. **make_default_activity()**: This is a variable that begins a refresh loop inside of the device Lua script. This is needed for when you when to constantly check the script for action. If this was not done, then the script would not be regularly checked for an event to have happened.

With these simple variables and functions being called, we are able to begin to touch other areas of the avLuaDevice code. (This can also be used in other areas but is mainly used in this one.) First off, we will go over how the **sensor_data** variable and **set_aircraft_draw_argument_value()** function works.

```
update = function()
  -- upvalues: sensor_data
  local mach = sensor_data:getMachNumber()
  local gforce = sensor_data:getVerticalAcceleration()
  local pitch = sensor_data:getStickRollPosition() / 100
  local roll = sensor_data:getStickPitchPosition() / 100
  local rudder = sensor_data:getRudderPosition() / 100

  set_aircraft_draw_argument_value(11, roll)
  set_aircraft_draw_argument_value(12, -roll)

  set_aircraft_draw_argument_value(15, roll)
  set_aircraft_draw_argument_value(16, -roll)

  set_aircraft_draw_argument_value(15, pitch)
  set_aircraft_draw_argument_value(16, pitch)

  set_aircraft_draw_argument_value(17, rudder)
  set_aircraft_draw_argument_value(18, rudder)
  set_aircraft_draw_argument_value(2, rudder)
end
```

Here, we have a basic script that is checking for the positions of my aircraft controls from the DCS API (in-game of DCS) and animating my external model controls with the function that controls the external animations, set_aircraft_draw_argument_value(). (specified above) I am also creating a variable function, named update. This function is going to be loaded automatically. (since earlier we declared the 4th function mentioned below for updating times) We have created 4 local variables:

1. Mach
2. Gforce
3. Pitch
4. Roll
5. Rudder

Each and every variable includes sensor_data:function(). This variable we declared earlier acts much like a class, and you are calling for a direct function to give back a value, either string or integer. With this, we are able to get all of these variables to display back an integer value whenever the script loops over again, and then draw the position of the

argument number that is specified. Here is a full list of functions that you can use for the data calling method:

(Get rid of the --[[]]-- commenting in Lua to be able to use them.)

```lua
--[[
local aoa = sensor_data:getAngleOfAttack()
local speed = sensor_data:getSelfVelocity()
local sas = sensor_data:getSelfAirspeed()
local tas = sensor_data:getTrueAirSpeed()
local ias = sensor_data:getIndicatedAirSpeed()
local mach = sensor_data:getMachNumber()
local vv = sensor_data:getVerticalVelocity()
local gforce = sensor_data:getVerticalAcceleration()
local balt = sensor_data:getBarometricAltitude()
local ralt = sensor_data:getRadarAltitude()
local pitch = sensor_data:getStickRollPosition() / 100
local roll = sensor_data:getStickPitchPosition() / 100
local rudder = sensor_data:getRudderPosition() / 100
local throttle = sensor_data:getThrottleLeftPosition()
local gear = sensor_data:getLandingGearHandlePos()
local speedbrake = sensor_data:getSpeedBrakePos()
local rpm = sensor_data:getEngineLeftRPM()
local flaps = sensor_data:getFlapsPos()
local cpy_state = sensor_data:getCanopyState()
local cpy_pos = sensor_data:getCanopyPos()
local wown = sensor_data:getWOW_NoseLandingGear()
local wowl = sensor_data:getWOW_LeftMainLandingGear()
local wowr = sensor_data:getWOW_RightMainLandingGear()
local gun_fire = get_aircraft_draw_argument_value(350)
local wing_fold = get_aircraft_draw_argument_value(8)
local nlg_dump = get_aircraft_draw_argument_value(1)
local stall = 0
--]]
```

The next example that we will highlight over is the use of the **get_param_handle()** and **setParam()** functions. These functions are used for when creating a new variable that involves an object inside of (usually) your cockpit model that is unable to be already animated by a "base gauge animation." (Pre-defined setups inside of your mainpanel_init.lua file) Here is an example of a script of how we would use this.

(On next page.)
↓ *test.lua* ↓

```lua
local Parameter = get_param_handle("ParameterName")
```

↓ *mainpanel_init.lua* ↓

```
44  CustomParameter                 = CreateGauge("parameter")
45  CustomParameter.parameter_name  = "ParameterName"
46  CustomParameter.arg_number      = 0
47  CustomParameter.input           = {0, 1}
48  CustomParameter.output          = {0, 100}
```

So let's try and boil things down here as simple as we can. Inside your test.lua file, you are getting the preliminary defined values of the parameter named "**ParameterName**". Inside of your mainpanel_init.lua file, you are creating a new "gauge" for your devices to use with the variable **CustomParameter**. The variable has the argument number of the cockpit of 0, with an input of a range of 0 to 1, and an output of a range from 0 to 100. With the local variable **Parameter** defined in your test device file, you are able to use the functions **set()** and **get()**. These variables will give you back an integer value of the exact position that it is set at. By default, the value will be at 0. Here is an example code:

↓ *test.lua* ↓

```
12  local Parameter = get_param_handle("ParameterName")
13
14  Parameter:set(1)
```

What happens here is that afterward, we define us getting the parameter handle values, we are setting it to a value of *1.* With this, the **set()** function calls into the input range and converts it into the output range in terms of keyframes. This means that with a value of 1 being put into our input, we are getting a keyframe of 100 or 100% of the movement animation for the cockpit model. Here is another example, but us using the **get()** function:

↓ *test.lua* ↓

```
14  Parameter:set(1)
15
16  if (Parameter:get() == 1) then
17    print_message_to_user("Parameter at 1!")
18  else
19    print_message_to_user("I'm lonely at 0. :(")
20  end
```

As we are setting our handle parameter value to 1, we have now introduced an if statement within the device to check if the parameter actually has been set to 1. If it has, it is going to use the function **print_message_to_user()**, which is an internal DCS call to sends a string of text in the top right corner. (This is similar of to a trigger event that prints a message to the user or coalition side in Mission Editor.) Otherwise, if the check fails for the integer value of 1 being set, it prints otherwise, and the statement ends.

However, this is nothing but only learning how to move around custom defined parameters for base gauges that already do not exist within the functions of the mainpanel_init.lua. This time, we will show something a little bit more complex, a landing gear system.

↓ *mainpanel_init.lua* ↓

```
98  LandingGear                 = CreateGauge()
99  LandingGear.arg_number      = 10
100 LandingGear.input           = {0, 1}
101 LandingGear.output          = {100, 0}
102 LandingGear.controller      = controllers.base_gauge_LandingGearHandlePos
```

↓ *devices.lua* ↓

```
8  devices["TEST"] = counter() --1
9  devices["GEAR_SYSTEM"]        = counter()--2
```

↓ *device_init.lua* ↓

```
34  creators[devices.TEST]              = {"avLuaDevice", LockOn_Options.script_path .. "/Systems/test.lua"}
35  creators[devices.GEAR_SYSTEM]       = {"avLuaDevice", LockOn_Options.script_path .. "/Systems/gear_system.lua"}
```

↓ *command_defs.lua* ↓

```
1  Keys =
2  {
3    iCommandPlaneGear = 68,
4  }
```

(You are able to get a full list of the ID's here: https://pastebin.com/nGEXymD8)

All we have done here is created a new creator in the initialization device Lua file for our gear system. After that, we declared it as an actual device within the devices with its corresponding name and being counted. Finally, we are creating a new gauge, but this time it is a base gauge instead of a parameter handle with a predefined variable that a function writes to, **base_gauge_LandingGearHandlePos**. The next line of code below is optional for if you want to make a clickable element within your cockpit. (which is needed for hovering over and being able to click to move and declare the actions of the file)

↓ *clickabledata.lua* ↓

```
266  elements["PNT_01"] = default_2_position_tumb(_("Landing Gear"), devices.GEAR_SYSTEM, 3001, 10)
```

Let's break this code down.

(Requires you to create the gear_system.lua file in Cockpit/Scripts/Systems directory path.)

↓ *gear_system.lua* ↓

```
dofile(LockOn_Options.script_path.."command_defs.lua")
dofile(LockOn_Options.script_path.."devices.lua")

local gear_system = GetSelf()
local dev = GetSelf()

local update_time_step = 0.01
local sensor_data = get_base_data()

make_default_activity(update_time_step)

local GEAR_COMMAND = 0
local GEAR_STATE = 0
local Gear = 68 -- internal DCS ID
local GearElem = 3001 -- clickable pointer element ID

```

This is the same old code that we have been using before for our test.lua file, except there are some differences.

1. Re-named test_system variable to **gear_system**.
2. Introduced 4 new variables

a. **GEAR_COMMAND** is in charge of us writing a 1 or 0 integer value to tell the Gear whether or not to go up or down.

b. **GEAR_STATE** controls a number from 0 to 1 (later in a decimal format) of the precise number of animation when we are setting the external model animations.

c. **Gear** is a variable of us calling for the internal DCS ID of the gear. (This is absolutely vital for being declared in the command_defs.lua file.) Since we are currently using an SFM in our ESM example, it is calling for an iCommand, which are needed for basic preparatory values and actions.

d. Lastly, **GearElem** is a variable for us getting the ID for our clickable pointer element. If you look back into our clickabledata.lua file, you will recognize the 3rd number in line inside of the **()** for the type of switch we want to be declared, a **default_2_position_tumb()** that acts as a 2-way switch inside of DCS. This line below is absolutely crucial to understanding how elements work and are called.

```
--SWITCHOFF elements["POINTER"] = default_2_position_tumb(LOCALIZE("Test Command"),devices.TEST, device_commands.Button_1,444)
```

Lastly, we are going to introduce a line with the function **listen_command()**. This function is used to tell the DCS API within our device Lua script to begin listening for if that iCommand is activated. (Which means if you have an Input for it in your keys, and the letter "G" assumably is hit, it will listen and tell you whether or not is declared.)

↓ *gear_system.lua* ↓

```
17  dev:listen_command(Gear)
```

(You are able to instead rather just call the number 68 as the "the internal ID" than having to create a local variable of the named instance, but is more recommended for keeping the code organized and knowing what is happening.)

We are now going to include a new framework-like function into our gear_system.lua device, called **SetCommmand()**. This is needed in order to listen for new commands (earlier to when we declared **listen_command()**) and tell us when one actually has been declared.

↓ *gear_system.lua* ↓

(On next page.)

```
19  function SetCommand(command, value)
20    if command == Gear or GearElem then
21      if (GEAR_COMMAND == 0) then
22        GEAR_COMMAND = 1
23      else
24        GEAR_COMMAND = 0
25      end
26    end
27  end
```

We are introducing 2 variables/values to check for within our function: **command** and **value**. Inside of the function is an if statement checking whether or not the landing gear was declared, from the internal ID and the pointer element ID. This means that if the landing gear button (whatever is assigned for it, assumably "G") has been pressed or the landing gear element in the cockpit has been clicked, it is going to turn our **GEAR_COMMAND** variable

to an integer value of 1. Otherwise, assuming that it has already been at 1 since the first if check within the command has failed, it will write it as an integer value of 0. Like our test.lua file, we are going to introduce back the **update()** function and create 2 if checks that will basically create the animation process of our landing gears.

↓ *gear_system.lua* ↓

(On next page.)

```
29  function update()
30    if (GEAR_COMMAND == 1 and GEAR_STATE > 0) then
31      GEAR_STATE = GEAR_STATE - 0.10
32    end
33
34    if (GEAR_COMMAND == 0 AND GEAR_STATE < 1) then
35      GEAR_STATE = GEAR_STATE + 0.10
36    end
37
38    set_aircraft_draw_argument_value(0, GEAR_STATE)
39    set_aircraft_draw_argument_value(3, GEAR_STATE)
40    set_aircraft_draw_argument_value(5, GEAR_STATE)
41  end
```

In this, we are beginning our update function. Then, we are creating a first if statement check to see if the **GEAR_COMMAND** variable has an integer value of 1. If so, it will also check if the **GEAR_STATE** which represents the position of the gear in terms of animation is more than 0. If so, it will keep on decrementing by a value of 0.10 until it hits 0. This is the same case for our 2nd if statement check, however, it checks if the commanding variable is at an integer value of 0 and is state of our landing gear is less than 0, which means it should be inside of our landing gear bays, at that point it will also increment by a value of 0.10 until it is fully at 1.

Finally, we are declaring 3 of the **set_aircraft_draw_argument_value()** functions as learned earlier for our update function back in the test.lua example. The integers 0, 3, and 5 stand each for the landing gear argument numbers of the aircraft as they are the default

LOMAC arguments. You can mimic this exact code in order to do other types of systems, such as, but is not limited to:

- Flaps
- Airbrake
- Canopy
- Wheel brakes (apart of gear system)

Additionally, now that you've learned how to use declaring functions to get and set the animation values of your parameter handles, you are now able to do:

- Custom coded gauges
- Knobs
- Switches/Levers that move "smoothly" ie. AJS37 or F/A-18C
- *and more to come!*

NEW: *Now adding an in-depth look for the gear system! You are able to check out the full script here:* https://pastebin.com/eMdX2DRW

Section 1-B

avSimpleElectricalSystem

by Sirius

NOTE: A *lot* of information given on how to create the electrical system is found in the avLuaDevice section. Those who have not read it/understand the topic are highly recommended to follow this section.

The electrical system is beyond one of the most important parts of the aircraft. Next in line is the hydraulics system (which will be addressed at a later point in time) which controls what controls of the aircraft operate. Your electrical system is going to cover the basics of all electrical power, electrical failures, and any electrical equipment such as HUD power, MFD, etc. Having even the uttermost basic electrical system set up will go a long way when building more onto your mod, or even future module.

With our prior knowledge on both subjects of using parameter handles and listening for commands, we will apply both for the electrical system. Let us begin with a basic, simplified script consisting of a switch that acts as our battery power.

↓ *mainpanel_init.lua* ↓

```
BatterySwitch                = CreateGauge("parameter")
BatterySwitch.parameter_name = "Battery_Switch"
BatterySwitch.arg_number     = 1
BatterySwitch.input          = {0, 1}
BatterySwitch.output         = {0, 100}
```

As seen here, we have declared a parameter gauge in our mainpanel_init.lua file. We will now write another basic script that will comprise as a separate a Lua device to handle the parameter handle, and listening for the command ID.

↓ *electric_system.lua* ↓

```lua
local electric_system = GetSelf()
local dev = GetSelf()

local update_time_step = 0.01

local sensor_data = get_base_data()

make_default_activity(update_time_step)

local BatteryParam = get_param_handle("Battery_Switch")
local Battery = 315 --Keys.iCommandPlaneBatteryOnOff
local BATTERY_COMMAND = 0

dev:listen_command(Battery)

function post_initialize()
  BatteryParam:set(0)
end
```

As we can see here, we have begun with a basic **avLuaDevice** structure. We have declared our parameter handle as **BatteryParam**, our internal ID as **Battery**, and the **BATTERY_COMMAND** as our on/off sequence boolean. (Boolean is a meaning that depicts whether a statement is true or false) Finally, we are listening to the internal ID, and setting the argument number keyframe to start at 0 once the system is loaded in DCS. The comment (depicted as **--** symbol next to Battery variable with number 315) is as a reference to the key you need to type in your **command_defs.lua** file.

With these preparatory steps in mind, we may now begin on the **SetCommand()** function.

(On next page.)

↓ *electric_system.lua* ↓

```
20  function SetCommand(command,value)
21    if command == Battery or 3009 then
22      -- If the battery switch is in the UP position
23      -- Moves battery switch to DOWN position
24      if (BATTERY_COMMAND == 0) then
25        BATTERY_COMMAND = 1
26        BatteryParam:set(1)
27        dev:DC_Battery_on(true)
28        -- print_message_to_user("Battery: ON")
29      -- Moves battery switch to UP position
30      else
31        BATTERY_COMMAND = 0
32        BatteryParam:set(0)
33        -- print_message_to_user("Battery: OFF")
34      end
35    end
36  end
```

With this script, we have 2 basics if statements set in place. Our first if statement inside of our function is whether or not our battery has been activated. The internal ID is represented by the variable Battery, while our cockpit ID value is represented by the number 3009. Additionally, as it is to be default in your Input folder with the mod, the RShift + L keys should automatically be your battery iCommand. However, if you would like to also have the switch an actual interactable object that you can click on the cockpit, then add this additional line of code.

↓ *clickabledata.lua* ↓

```
266   elements["PNT_01"] = default_2_position_tumb(_("Battery Power"), devices.ELECTRIC_SYSTEM, 3009, 11)
```

Now that we have our basic battery in place to act as electrical power, we can get more in-depth with other things, such as APU power, inverters, and etc. This should be followed by similar steps re-producing these steps. Here is the result as shown.

(On next page.)

↓ *mainpanel_init.lua* ↓

```
74   ThrottleLeftCutoff                  = CreateGauge("parameter")
75   ThrottleLeftCutoff.parameter_name   = "Left_Throttle_Cutoff"
76   ThrottleLeftCutoff.arg_number       = 3
77   ThrottleLeftCutoff.input            = {0, 1}
78   ThrottleLeftCutoff.output           = {0, 100}
79
80   ThrottleRightCutoff                 = CreateGauge("parameter")
81   ThrottleRightCutoff.parameter_name  = "Right_Throttle_Cutoff"
82   ThrottleRightCutoff.arg_number      = 5
83   ThrottleRightCutoff.input           = {0, 1}
84   ThrottleRightCutoff.output          = {0, 100}
```

```
110  LeftACGenSwitch                 = CreateGauge("parameter")
111  LeftACGenSwitch.parameter_name  = "Left_AC_Gen_Switch"
112  LeftACGenSwitch.arg_number      = 11
113  LeftACGenSwitch.input           = {-1, 1}
114  LeftACGenSwitch.output          = {-100, 100}
115
116  RightACGenSwitch                = CreateGauge("parameter")
117  RightACGenSwitch.parameter_name = "Right_AC_Gen_Switch"
118  RightACGenSwitch.arg_number     = 12
119  RightACGenSwitch.input          = {-1, 1}
120  RightACGenSwitch.output         = {-100, 100}
121
122  APUSwitch                       = CreateGauge("parameter")
123  APUSwitch.parameter_name        = "Auxillery_Power_Unit_Switch"
124  APUSwitch.arg_number            = 13
125  APUSwitch.input                 = {-1, 1}
126  APUSwitch.output                = {-100, 100}
```

(Note: until said, all below are as described)

↓ *electric_system.lua* ↓

```
local BatteryParam = get_param_handle("Battery_Switch")
local Battery = 315 --Keys.iCommandPlaneBatteryOnOff
local BATTERY_COMMAND = 0

local LeftACParam = get_param_handle("Left_AC_Gen_Switch")
local RightACParam = get_param_handle("Right_AC_Gen_Switch")
local LEFT_AC_COMMAND = 0
local RIGHT_AC_COMMAND = 0

local APUSwitch = get_param_handle("Auxillery_Power_Unit_Switch")
local APU_COMMAND = 0

local LEFT_ENG_COMMAND = 0
local RIGHT_ENG_COMMAND = 0

dev:listen_command(Battery)
dev:listen_command(LeftACParam)
dev:listen_command(RightACParam)
dev:listen_command(APUSwitch)

function post_initialize()
  BatteryParam:set(0)
  LeftACParam:set(0)
  RightACParam:set(0)
  APUSwitch:set(-1)
end
```

(Continues on next page.)

NOTE: you do not need to have a listen_command() statement for the parameter handles in this example.

↓ *electric_system.lua* ↓

NOTE: The code that formerly existed for this section has been deemed outdated and is currently being worked on coding wise. Sorry!

Section 1-C
Training Missions

*by Sirius (formerly **DISBANDED USER**)*

One of the many advantages of having an ESM with a clickable cockpit (including helpers) is the ability to create training missions. Creating a training mission is very simple, and involves little to no experience in programming! The only reason it is currently in this category is for the organization and a follow-along type of section.

Before we begin, let's discuss a few of the core items that are found in the Mission Editor that are used strictly in a single player training environment.

<< insert image here. >>

As we can see here, inside of the Mission Editor is a tab called "trigger events." This is used commonly within many missions made inside of DCS, especially popular in the multiplayer servers to "trigger" an event once it has been committed to. This can include, but is not limited to: flying in/out of a zone, a pilot crashing, reaching a certain altitude, and etc. We are able to manipulate certain conditions within our trigger events to establish a listening-like command to our cockpit environment. Let us know to create an example of what it would look like to check if a switch was moved to the position we wanted it in:

<< insert image here. >>

As seen from the image, we are able to recognize 3 distinct sections. 1 section is responsible for organizing the whole trigger event, the next one (moving accordingly to the right) is responsible for the conditions needed to trigger the event. In this case, we are checking if the cockpit argument number "**1**" is in a range of 1:1. (Meaning it is regardless at keyframe 100) Finally, the action calls to highlight the element pointer "**PNT_01**."

Section 1-D
Pointer types
by Sirius

When trying to set up your cockpit code, you will recognize in the **clickabledata.lua** file that there are a bunch of functions, each signifying a different type of switch. Do not fret! These switches all have different purposes and there are many ways that you can set them up. In the file, you are given these:

1. Push/momentary button
2. 1-way switch
3. 2-way switch
4. 3-way switch
5. Lever
6. Movable rotaries
7. Limited rotaries (ie. lights knob)
8. Multi-position switch (> 3)
9. Multi-position switch limited (> 3, can be stopped)
10. Pushable rotating button
11. 2-way lever
12. Toggleable button (ie. circuit breaker)

Modeling

Software

by Sirius

This is a part of the guide that is necessary for if you want to animate and be able to click certain parts of the cockpit. (This refers to as in interacting with the elements and helpers of the cockpit too.) There is currently only 1 3D modeling software that is capable of handling both tasks:

- Autodesk 3D Studio Max (or 3DS Max)

At this time, only 3DS Max has the capability of being able to handle animating parts correctly in argument number format and being able to export it into an .EDM format without being broken. 3DS Max is also the official choice of 3D modeling software that is used by various 3rd party developers and modders and is regularly updated by ED with a plugin for .EDM conversion.

Prerequisite knowledge

by Sirius

Just like the programming prerequisites, there are of course prerequisites for using 3D modeling software in order to make this ESM for DCS aircraft. The current requirements in order to ensure that you will be able to follow this guide properly include:

1. You need to know how to animate models in 3DS Max by using the argument number method. If you are unaware of this or have no idea whatsoever on how to do this, then please click this link here that is a video on how to guide you to successfully animating for DCS: https://www.megabotix.com/wp-content/uploads/dcs/animation-dcs.mp4
2. You need to have the current.EDM conversion plugin or one that works for 3DS Max. As of requirement no. 1, here is a link for this also: ftp://srv0files.eagle.ru/mods/edm_plugins/

Helpers

by Sirius

When it comes to having a Human cockpit, (a derogatory term for an aircraft with an interactable cockpit, in terms of ESM or DOF) helpers are very crucial to be able to have a clickable cockpit model in DCS World. There are 2 known helpers that are used for the majority of cockpit models in ESM:

- Point helper. The point helper is a helper that defines the center point of whatever in the software acts as a target for interaction. In DCS, you are able to use point helpers to interact with the Lua device code.
- Dummy helper. An alternative of point helper, the dummy helper acts similar to how the point helper works. Dummies, as the nickname implies, are used to give the software a 3D confined environment for interaction. This is not used as much as the point helper but is still used throughout main mods and modules.

A helper also has to follow under a set of requirements before it will work properly in the DCS ESM.

1. Correct naming. The best way to go about naming helpers (preferably point helpers) is by using the **PNT_** format. An example of this would be PNT_001. This indicates that you are using the point helper of 001. This keeps the 3D model organized and helps you later on the code the element pointers in clickabledata.lua
2. Correct size. One of the biggest issues about the interaction of helpers and the DCS ESM is the size. The DCS API (for internal reasons unknown) uses the size of **4 meters** or 4 millimeters. The correct sizing will let DCS properly be able to identify the exact space of the helper.

Here is an example of what the helper being set up would look like in Max:

(On next page.)

<< insert image here. >>