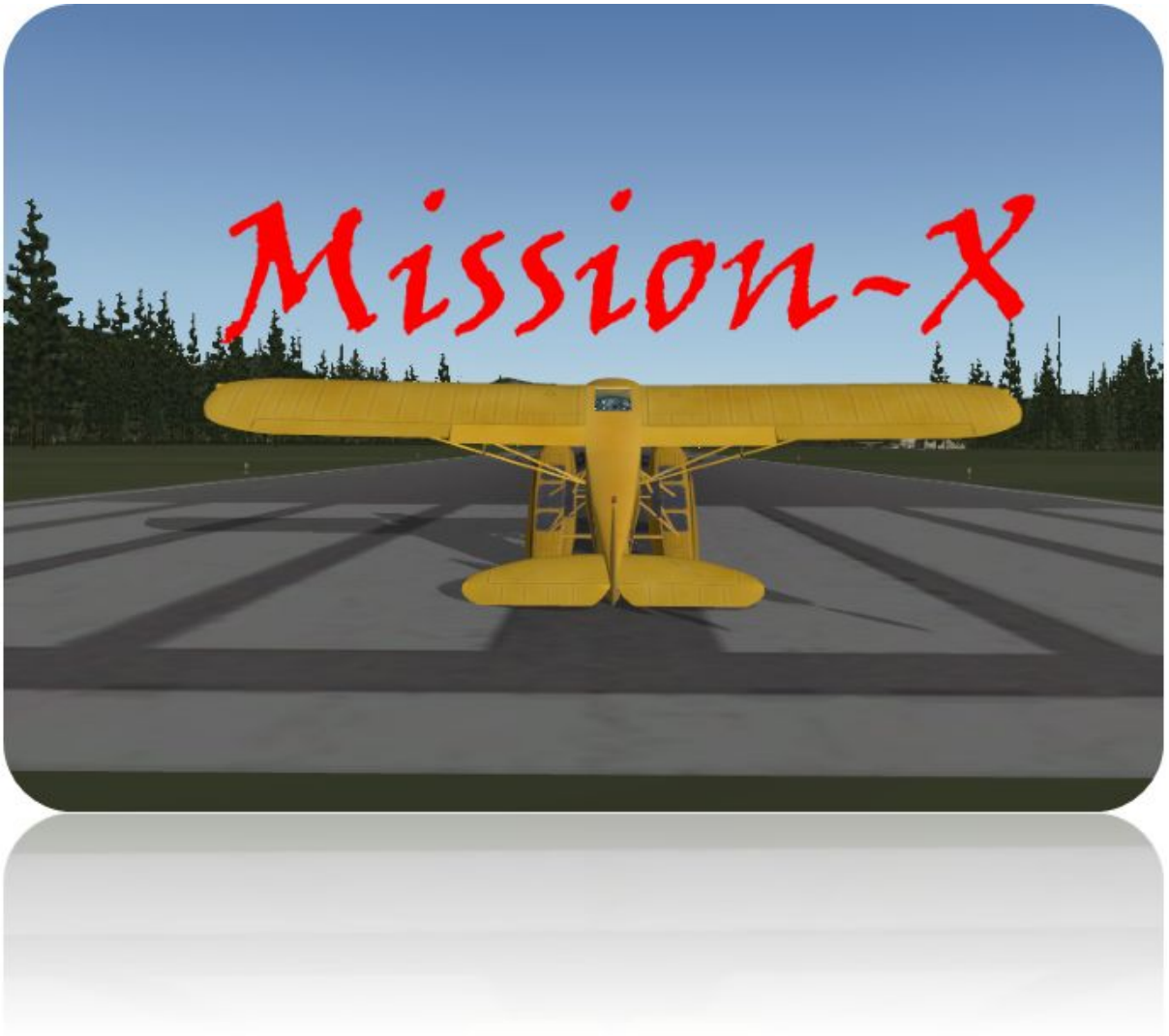
Mission-X v3 Scripting Concepts

Written by: Saar.N	Revision 0.06	18-feb-2020	Based on v3.0.242.2



Written by: Saar.N	Revision 0.06	18-feb-2020	Based on v3.0.242.2

Written by: Saar.N	Revision 0.06	18-feb-2020	Based on v3.0.242.2

Preface

Hello all fellow simmers.
The following document is intended to give you a brief guide to scripting in Mission-X with MY-BASIC.

What the document is all about: scripting and how to use it. We will use "actual examples" for this.
What this document is not: The document won't show you how to write a full mission and won't teach you MY-BASIC scripting and syntax, only the portion that is relevant for Mission-X scripting will be discussed.

You can read about MY-BASIC in its git project page.

Enjoy
Saar Nagar - Snagar

MY-BASIC implementation

Mission-X plugin has no intention to allow the script part to build a new application through it.
The main benefit of using scripts as part of the plugin workflow, is to handle logic in a flexible manner. It does require the designer to attain some scripting skills, but that can be learned too.

Mission-X uses only the logic syntax part of MY-BASIC language, it stripped most of the advanced features like: ~~Arrays and~~ Lambda.
The main syntax that we should deal with is:
If… Then… Else statements.
Variable = ***Value***.
def function_name()…enddef.

We can also use built in functions from MY-BASIC like converting numbers to string and vice versa, but the real use is to decide how to move forward the mission base on values we fetch from X-Plane.

Tip:
In some elements there will be attributes you will not use. You can ignore them if they are optional (you do not have to write them down).

How or when to call a script

There are few components from where you can call a script:

- Task
- Trigger
- Message

A task can be based on a script, while trigger has more options to call a script, either as a condition using "*cond_script*" attribute, or as a triggering action using "*script_name_when_fired*" or even as a post action using "*post_script*" attribute (for full range of options, read the designer guide).

When you want a more complex logic to test conditions and therefore decide if task/trigger is complete, you will probably use a "script", and it will have to modify task/trigger state, by altering the boolean property: "***script_conditions_met_b***". When we set this property as part of a task, then that means the task is complete. When we set this property at the trigger level, then it means the "*trigger*" rules have been fulfilled therefore the event: "*script_name_when_fired*" can be called or we can send a message using: "*message_name_when_fired*" (depends if the former did not alter the trigger state).

> ***"script_name_when_fired"**: this attribute **name** might be confusing.*
> *In triggers, this attribute **should always** be translated as: Call the script only when all conditions are met.*
> *Conditions defined by:* `<conditions> and <loc_and_elev_data>` *elements.*

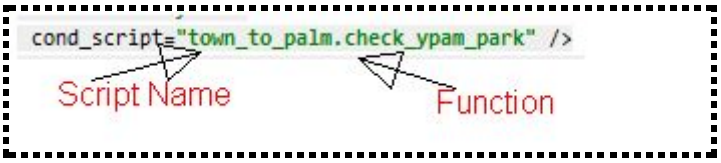
In a trigger you have more options to override former script decisions, and this complexity also allows flexibility. Please do remember that you do not need to fill in all scripts attributes, only the ones that assist in moving the mission forward.

In the following pages we will investigate a few script implementations and how they affect mission progress.

How to call a script or function from the mission XML file

When calling a script, it is best to use the format: "script_name.function_name". Example: "town_to_palm.check_ypam_park".

Script name: *"town_to_palm"* (also represent a file name or scriptlet name).
Function name: *"check_ypam_park"*.
The plugin will seed the argument: **mxFuncCall** with the function name, so you can lookup this name in the script.



Example 1 - One time action using Task based script

Writing a script does not mean that it needs to be based on a complex logic. In some cases, you might want to do one action only, and it is easier for you to do it inside a script.
Let's assume that you want to send a message to the simmer when an objective becomes active (basically the start of a flight leg, in most cases), or you want to check if inventory item exists in plane and then send a one time message.

A nice trick to send a message to the simmer is to create an optional "task" and only test it once. The task will call a script that calls a message.
From a plugin perspective, the task will always be fulfilled therefore it will execute the script every flight iteration, ***but***, we can disable this behavior by using - "*fn_send_comm_message("mesage_name", "track name", true)*" function. The third parameter is a "one time flag", "*true*" means run just once, and the "track name" is a unique name the plugin monitors (check designer guide).

What do we need to implement this one time behavior:

- Define a task based on a script.
- Create a script with a function by the name "*goal1_onetime_message*"

We will use the "townsville_to_palm_and_back" mission as a base mission for our example.
Here is a task example that calls a function in a script. The function will send the message and flag the task as success.
Remember: If we don't flag the task as success, then the plugin will continue to evaluate the task, therefore it will call the message continuously.

[v3.0.206] Since v3.0.206 "`fn_send_onetime_text_message`" has been deprecated. You should use "`fn_send_text_message`" and flag it as "one time" only message (third parameter). Check the designer guide for more details.

Snippet from mission file

```
<task name="starting Message"  base_on_script="town_to_palm.goal1_onetime_message"
eval_success_for_n_sec="" mandatory="no" force_evaluation="no"/>
```

Snippet from script file

```
elseif (mxFuncCall = "goal1_onetime_message") then

  ' Sending text as communication message
  fn_send_text_message ("Good morning pilot...fly to Palm Island...",
                        "GoodMorning01", true)

  ' Setting task as complete is important since plugin won't continue to evaluate
  ' TASK once it is flagged as complete.
  fn_set_task_property ("starting Message", "script_conditions_met_b", "true")
endif
```

Tip:
For a one time message at the beginning or end of a flight leg, you could just use: “<start_leg_message>” or “<end_leg_message>” elements.

Example 2 - Trigger based on a script

Adding a script to a trigger can be a complementary way to add complex conditions. A trigger allows you to design an event based on a physical area without writing any code, but once you want to add more conditions that are not supported by the trigger syntax this is when scripting comes in handy.

For example, let's assume we want the simmer to reach a certain area in the airfield, like parking or fuel area, and we want him/her to park the plane.
A trigger can define the following conditions without a script:

- Parking location and area (either as radius, or polygonal area).
- Plane needs to be on ground or airborne.

If you want to check if a simmer reached an event area, plane speed is less than 2 nautical miles and park-brake is set, you will have to add it in a script, since the trigger's XML syntax does not support these kinds of tests.

How to add a test to “ground speed” + “park-brake“.

Here is an example for Trigger declaration:

```
<trigger name="trig_park_in_YPAM" type="rad" enabled="yes" >
        <conditions plane_on_ground="yes" cond_script="town_to_palm.check_ypam_park" />
        <event_scripts post_script="" script_name_when_fired="" script_name_when_left="" />
        <messages message_name_when_fired="ParkedInYPAM" message_name_when_left="" />

        <loc_and_elev_data>
          <radius length_mt="20"/>
          <point lat="-18.7517052" long="146.579651"/>
        </loc_and_elev_data>
</trigger>
```

In this example, we see that the trigger defines a parking area (radius + location) But in order to evaluate the other conditions, we use “`cond_script`” attribute to call a “*script.function*” to handle the added rules (you can also just call a script. Please read the “designer guide” for more information).
If the script finds that simmer is fulfilling the rules, it will flag the “`script_conditions_met_b`” property as “*true*” thus flaging the trigger as “accomplished” or “complete” and this will lead to the call of “`message_name_when_fired`”.

The following snippet of code does not represent the order of implementation in the script file. This is only for readability sake.
MY-BASIC is an interpreter, therefore branching of calls should reside at the end of the script file and function definitions should reside prior to the branching.
To send a predefined message from a script, we will use: "`fn_send_comm_message`" function. It also allows us to flag the message as a one-time or not. Check "designer guide" for more details.

```
' Main branching code
if (mxFuncCall = "check_ypam_park") then
  call func_check_ypam_park();
elseif (mxFuncCall = "check_ybtl_park") then
  call func_check_ybtl_park();
elseif (mxFuncCall = "dummy_message1") then
  fn_send_comm_message ("Fly to Palm Island...", "dummy_message1", true ) 'can be scriptlet
elseif (mxFuncCall = "func_start_mxpad01") then
  call func_start_mxpad01();
endif
```

***mxFuncCall**: Holds the name of the function we are searching in trigger: "trig_park_in_YPAM". cond_script="town_to_palm.check_ypam_park" from the example above.*

The following script represents the function. It has many debug commands that should be commented out once code runs as expected.

```
' Functions
def func_check_ypam_park()

  ground_speed = fn_get_dref_value ("ground_speed");
  if (mxError = "") then
    print "ground_speed: ", ground_speed," "
    if (mxError = "") then
        print ", parkbrake: ", parkbrake;

        ' flag trigger trig_park_in_YPAM as complete
        if ground_speed < 2 and parkbrake = 1 then
          fn_set_trigger_property ("trig_park_in_YPAM", "script_conditions_met_b", "true")
        endif

    else
      print "Error: ", mxError;
    endif
  else
    print "Error: ", mxError;
  endif

enddef
```

Since v3.0.214 we have inventory capabilities too. If you would like to add Inventory tests, then use the following functions:

- `fn_is_item_exists_in_plane("barcode")` and
- `fn_is_item_exists_in_ext_inventory ("external inventory name", "barcode")`

In the function "`func_check_ypam_park`" we use the "print" command for debugging. Please remember to **remove** or **comment out** this command, before publishing your mission.

***mxError**: another seeded attribute.*
The plugin seed it after every function call. You should check if "mxError" is empty as part of success/failure outcome investigation, very useful during mission debug.

Consult the designer guide when a function returns success/failure, sometimes it is a boolean value and sometimes it is a number.

The plugin was designed to always write to "mxError" in case something is wrong or assume as wrong.

PRINT command has a high FPS hit. Use only when you test a mission

Example 3 - Task based on a script

Let's assume you define an “objective” a “run-up check”. You want the simmer to do it in a certain order for the default Cessna 172c provided by Laminar.
Let's break what we want into smaller parts:

1. We need information based on Datarefs, to read plane components status, like:
    a. Park-brake state.
    b. Transponder code.
    c. Prop_speed_rpm.
    d. Ignition_key state.
    e. Acf_takeoff_trim (won't be covered)
    f. Others…
2. We need a set of tasks that will be done serially and be dependent on the previous one. This can be achieved in two ways:
    a. by defining a “<leg>” for each task, so the next <leg> will be the next step in the “run-up” check, or
    b. you could create a set of tasks in the same objective that will be dependent against each other, that way only one <leg> is needed. The last task will be mandatory.

Gathering Dataref Data

To monitor planes or sim state we need Datarefs. To gather Dataref data, we should check the Dataref page in **xsquawkbox** site or just read the local “dataref” file in X-Plane install dir (just search for “dataref”).
Here is an example of a list of useful DataRefs we can monitor:

```
<xpdata>
  <dataref name="retract_gear" key="sim/aircraft/gear/acf_gear_retract"/>
  <dataref name="mixture" key="sim/cockpit2/engine/actuators/mixture_ratio_all"/>
  <dataref name="roll_ratio" key="sim/cockpit/gyros/phi_vac_ind_deg"/>
  <dataref name="pitch_ratio" key="sim/cockpit/gyros/the_vac_ind_deg"/>
  <dataref name="airSpeed2" key="sim/flightmodel/position/indicated_airspeed2"/>
  <dataref name="ground_speed" key="sim/flightmodel/position/groundspeed"/>
  <dataref name="gearForce" key="sim/flightmodel/forces/faxil_gear"/>
  <dataref name="days" key="sim/time/local_date_days"/>
  <dataref name="sec" key="sim/time/zulu_time_sec"/>
  <dataref name="parkbrake" key="sim/flightmodel/controls/parkbrake"/>
  <dataref name="m_fuel_total" key="sim/flightmodel/weight/m_fuel_total"/>
  <dataref name="barometer_setting" key="sim/cockpit/misc/barometer_setting"/>
  <dataref name="transponder_code"
key="sim/cockpit2/radios/actuators/transponder_code"/>
  <dataref name="transponder_mode"
key="sim/cockpit2/radios/actuators/transponder_mode"/>
  <dataref name="prop_speed_rpm_arr[1]"
key="sim/cockpit2/engine/indicators/prop_speed_rpm"/>
  <dataref name="ignition_arr[1]" key="sim/cockpit2/engine/actuators/ignition_key"/>
  <dataref name="pitch_trim_takeoff" key="sim/aircraft/controls/acf_takeoff_trim"/>
  <dataref name="user_elev_trim" key="sim/flightmodel/controls/elv_trim"/>
</xpdata>
```

This is an example to a useful set of Datarefs you will probably use in your scripts.

Once we define “DataRefs”, they will be available in our scripts, using the `fn_get_dref_value (“dataref name”)` function, the function returns a number value.

Creating a serial set of tasks to mimic run-up test - we display only 3 tasks from the full list

```
<objective name="doRunUpTest" >
  <task name="step1_parkbreak" depends_on_task="" base_on_trigger=""
base_on_script="runup_test.check_park_break" eval_success_for_n_sec="3" mandatory="no"
force_evaluation="no"/>
  <task name="step1_transponder" depends_on_task="step1_parkbreak" base_on_trigger=""
base_on_script="runup_test.check_transponder_vfr" eval_success_for_n_sec="3" mandatory="yes"
force_evaluation="no"/>
  <task name="step1_rpm1700_1" depends_on_task="step1_transponder" base_on_trigger=""
base_on_script="runup_test.check_engine_rpm_1700" eval_success_for_n_sec="3" mandatory="no"
force_evaluation="no"/>
...
</objective>
```

The example might seem complex, but it is very repetitive in its nature. The code behind it could have been simpler, but I wanted to display some script aspects and capabilities.
Let's examine the first two tasks:

```
<task name="step1_parkbrake" depends_on_task="" base_on_trigger=""
base_on_script="runup_test.check_park_brake" eval_success_for_n_sec="3" mandatory="no"
force_evaluation="no"/>
```

Action Required	XML Attribute
Check simmer pulled Park-brake and its speed is lower than 2kn	Use Script: `base_on_script` Call script “**runup_test**” Lookup function: “**check_park_brake**”
Check park-brake is set for at least 3 seconds	`eval_success_for_n_sec="3"`

The short version of the function:

```
ground_speed = fn_get_dref_value ("ground_speed"); ' read ground speed dataref value.
parkbrake = fn_get_dref_value ("parkbrake");       ' read park brake dataref value.
if ground_speed < 2 and parkbrake = 1 then
   ' "trig_park_in_YPAM": Trigger name from XML file.
   ' "script_conditions_met_b", "true": modify trigger property to true.
   fn_set_trigger_property ("trig_park_in_YPAM", "script_conditions_met_b", "true")
endif
```

The second task:

```
<task name="step1_transponder" depends_on_task="step1_parkbreak" base_on_trigger=""
base_on_script="runup_test.check_transponder_vfr" eval_success_for_n_sec="3" mandatory="yes"
force_evaluation="no"/>
```

Action Required	XML Attribute
Evaluate task only if "step1_parkbreak" is complete	depends_on_task="step1_parkbreak"
Check condition in script	base_on_script="runup_test.check_transponder_vfr"
Check conditions are valid for at least 3 seconds	eval_success_for_n_sec="3"
This step is mandatory	mandatory="yes"

The short version of the function:

```
transponder_code = fn_get_dref_value ("transponder_code");
if (mxError = "") then
  if transponder_code = 3000 then
    fn_set_task_property ("step1_transponder", "script_conditions_met_b", "true")
  else
    fn_set_task_property ("step1_transponder", "script_conditions_met_b", "false")
  endif
else
  print "Error: ", mxError;
endif
```

We need to set the property "`script_conditions_met_b`" to ***true*** or ***false*** states since we want to test transponder value for at least 3 seconds. Without modifying the state of the property, plugin won't be aware of the condition changes and won't alter the mission flow.

The rest of the tasks ?

- Task 3 until the end will define "`depends_on_task`", that way we build serial tasks in the objective.
- "Last task" should be "mandatory" or objective will be flagged as success before reaching the last task.
- Optional, we can modify other tasks "`mandatory`" and "`force_evaluation`" attributes to be more restrictive across the runup test.

You could have done the same by creating one <leg> for each task we wrote, but that might be too much overhead. Again, there is no one way to do things, it is up to you to make it work.

Creating Dialogs using <message> element

T.B.D

From here…

The combination of simple task/trigger elements with the <message> component can make the mission framework more accessible by the simmer.

Scripts are a huge advantage to control mission flow, but I strongly suggest to first try to leverage the core elements and their capabilities before adding scripts. If you decide to add scripts then I strongly suggest making baby steps when implementing for the first time. Don’t jump and create complex missions and logic scripts. Investigate, try with simple missions and demo files, and integrate what you understand and works, first.

Enjoy
Saar Nagar - Snagar

Revision Table

Revision No.	Date + Modifier	What was changes
0.06	18-feb-2020	Removed MX-Pad and the selection options. Will add <choices> element later.
0.05	29-aug-2019	Changed all <goal> examples to be flight leg oriented.
0.04	23-nov-2018	Minor adjustment to Mission-X v3.0.214. Added reference to inventory functions. Fixed Selection title to be in the menu Minor cosmetic modifications.
0.03	06-jul-2018	Adjustment for Mission-X v3.0.206 changes. Always_evaluate renamed to “force_evaluation” Added MXPad selection example.
0.02	13-jan-2017	Added MX-PAD scripting concepts
0.01		initial