Mission-X v3 Designer Guide

Written by: Saar.N	Revision 0.36	Based on: v25.02.1

!!! Breaking Changes since v24.x !!!

Although Mission-X is compatible with the older mission files (v3.0.0 - 3.0.222x), I suggest converting your mission files to the new standard (sorry for the inconvenience).

Before and After changes	
Since v3.304.12	
dataref_to_modify_when_fired and dataref_to_modify_when_left	Both attributes are now using the updated "dataref" syntax guidelines. For more information see: "dataref" topic

<dataref_start_cold_and_dark> now follows the new syntax guidelines.
 	
Instead of using ":" for equals and "," for delimiter, Use "=" for equality sign and "|" as a delimiter.
	Changed the text syntax.
	Before:
	sim/cockpit2/controls/yoke_heading_ratio:0,{next dataref=value}...
	After
	sim/cockpit2/controls/yoke_heading_ratio=0|{next dataref=value}...
	
	Before:
	fn_get_dref_value("main_rotor_speed");
	After:
	fn_get_dref_value("main_rotor_speed[0]");

Since v3.304.14 r1,r2	
is_virtual_b	Not really breaking change since the plugin always searches the 3D object in the virtual path and only then as a physical file. So this attribute is really mute.
All "*FN_STORE_GLOBAL_XXX*" functions will be deprecated	Use "*FN_SET_GLOBAL_XXX*" functions instead.
All "FN_GET_GLOBAL_XXX" functionality were extended, they now create the global parameter if it does not exists with default value (more detail see in this topic) Main reason to streamline basic programming tasks and not waste time on "testing before creating"	
Since v3.306.1b	
fn_is_mxpad_queue_empty() - renamed	fn_is_msg_queue_empty()
Since v3.306.1c	
timer_type	Deprecated, the background <mix> is always OS based.
Since v3.306.3	

Internal: "run_continuesly" renamed to "run_continuously"	Fixed syntax error. Affects saved files with timers and even then it might not affect the timer itself, since it is an edge case.
Not exactly breaking: When reading flight leg description the order was: "<leg><desc>[cdata]" and then "<leg>[cdata] if "<desc>[cdata]" was missing.	***New Lookup:*** First: "<leg>[cdata]" Then: "<leg><desc>[cdata]"
Since v24.03.2	
Not exactly a breaking change, but <base_weights_kg> child of "<global_setting>" attributes should be: "pilot_base_weight, passengers_base_weight, storage_base_weight". The original attributes: "pilot, weight and storage" might be deprecated in the future.	
Since v24.12.2	
The inventory layout has undergone a redesign in order to leverage the new X-Plane "station" feature. This means that older missions that use specific "inventory" code or are dependent on it, might not work as expected. Solution: • Use the new <compatibility inventory_layout="11" > in "<global_settings>" to force the correct inventory layout. • The new Mission file version is now 312. If you define <mission version="301" > then it will "force" the XP11 inventory layout too.	
v25.05.1	
Maximum inventory capacity was extended: 30 for planes and 100 for external storage.	

Working with array datarefs: Since v.3.0.225.2 your array numbering starts from 0..{n-1} (This to be inline with X-Plane XPSDK numbering).
If you have a dataref array of "**8**" elements, you should point to them with the numbers: 0..7, where "0" is the first element.

Example, if we want to check the rpm of a GA engine, I'll use the dataref: "*sim/cockpit2/engine/indicators/prop_speed_rpm*", but this is an array of 8 elements and we have only 1 engine, therefore, we will indicate the element of choice in the name attribute of the element "dataref":

```
<dataref name="prop_speed_rpm_arr[0]" key="sim/cockpit2/engine/indicators/prop_speed_rpm"/>
```

[v3.0.241.7] **Removed support** for the <mix> attributes: "*mxpad_label, mxpad_label_placement, mxpad_label_color*". Instead use: "**label, label_placement, label_color**".
See <message> topic.

Preface

Hello fellow simmers.
The following document is intended to give you an in depth insight of the latest Mission-X v3 internals and how you can best use it to create your own missions.

The documents will be divided into three parts.
The first: a simple "In Depth Overview" of each element.
The second: "Hands On" where I will rant and walk you through the creation of a simple mission.
The third: Scripting guide.

This document does **not** explain how to write a mission. It explains the missions XML elements, their meanings and gives some insight into each element usage.

Mission-X v3 is a blend of two paradigms - XML and scripting (minimal scripting capabilities are needed). The *pros* are mainly in flexibility - features can be implemented through scripting and not wait for the plugin programmer to implement them. But to achieve that, we need to expose interfaces (functions) by the plugin for the programmer. Once they are exposed, the designer can choose how to use them and when.
Cons: A Mission might be spread in more than one file. You need to keep track of where you wrote your code, the function names to call and debugging a script could be painful.
Although working with embedded scripts can overwhelm beginner designers, I believe it is a technicality that can be easily overcome and should not be a barrier.

If it is your first time creating a mission, you are more than welcome to share your experience and consult me directly at: snagar.dev@protonmail.com.

In depth overview of
Mission Data File Creation

Typographic Conversion in Data File

Try to use only lowercase letters

Symbol	Meaning	Example / Comment
int	Integer value	1,2,3
dec / decimal	Decimal value	1.0, 1.2, 345.34
bool	String value (plugin will translate)	[true|false] or [yes|no]
str	String	Hello, Latitude
[ ]	User defined values	
*****	**Needed ! otherwise Optional**	
_ (underscore)	**Default value**	[**0** | 1]
nm	nautical miles	
km	Kilometer	
mil	Miles	
mtr / mt	Meter	
ft	Feet	
kt/kts	Knots	Unit of speed. One nautical mile per hour
kg	Kilograms	1 Kg = 2.204 lbs
lbs	Libra	
cel	Celsius: a temperature scale	24.2 (Degrees C)
"|"	OR sign	a | b |c = a or b or c etc..
[value1|value2]	Pick one of the options, but only one of them. Underscore means - default value.	[0 | 1]: pick Zero or One Zero is the default.
Pct / %	Percentage	

How to read the overview

The overview was built like an XML file, to make it simpler to get around.
An XML file is constructed from elements (or tags) and sub elements.
For example:
<flight_plan> can encapsulate all "<leg>"s elements, therefore the key elements can be seen also as a grouping element for better readability.

The following are rules of thumb:

- **Tip:** You can use the "Content Page" to get a visual idea of how a mission file is structured.
- Please take time to read each Element you use, go over the attributes and comments. Read carefully the tips given in each table (if present) or at the end of each Element topic.
- In some cases, you will be pointed to other parts of the document to read the explanation. This was done to minimize duplications of info.
- For beginners I strongly encourage you to read the tutorial at the end of the "overview" part.

I hope you'll find the document helpful
Saar.N (Snagar)

It is best to read this document side by side with a mission file. That way you can better understand when and why a mission is constructed that way.

This document does **not** explain how to write a mission. It explains the missions XML elements, their meanings and gives some insight of each element usage.

Main component of each mission

The idea behind the mission file is to define a set of syntax and rules that make the logic and define the flow of a mission. The more you design missions, the more you will get familiar with the possibilities and limitations of the plugin.

The 5 main components of any mission are:

- **Task:** this represents the granular action you want the simmer to take. It can be mandatory - does this task have to be fulfilled before you can continue? or optional (nice to have).
  It can also be dependent on the success of other tasks (ordered serially like a checklist).

- **Objective:** Objectives incorporate one or more tasks.
  You can define order between tasks, by using dependency attributes, or define tasks that are mandatory or optional.
  There should be a mandatory task in at least one objective. If you won't define a mandatory task, then the plugin won't evaluate the objective and will probably flag it as success. This might finish the mission as soon as it was started.

- **Flight Legs:** define the parts of your flight plan.
  This is a high level representation of the mission.
  The plugin handles one "leg" at a time.
  You link objectives to one or more "flight legs", thus creating a simple or more complex flight plan. A "leg" should have at least one objective with a mandatory task, or the plugin will just flag it as success as soon as it starts.

- **Triggers:** Trigger is like an event. It is the simplest means to define logic and what will be the outcome of it. It is also a useful element to drive the mission forward.
  Usage examples:
  - send a message to simmer if the plane reaches a certain area.
  - Call scripts that will take the heavy load of logic and through simple calls will modify the state of the sim according to certain statistics or send dynamic messages.

  The trigger is a very powerful tool to use, but you do need to be organized especially when scripts are involved. It is also harder to debug.

- **Messages:** a message is not exactly a key component, but it will probably be used in any mission you will write, since it is the easiest way to send feedback to the simmer.
  The message acts like a simple mixture that allows you to send text messages and sound files at the same time. Your sound file can be background sound or the message itself (using your voice recording, instead of X-Plane narrator).
  More on that at the Message topic.

Before we dive-in

In the following pages we will go over the main elements that construct a mission file and try to understand what should or should not be part of it. Hopefully it will give you good enough insight on how to build your own mission.

I suggest reading this document side by side with a working mission file.
Any questions/suggestions you are more than welcome to send me directly to my email: snagar.dev@protonmail.com

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).

This document does **not** explain how to write a mission. It explains the missions XML elements, their meaning and gives some insight on each element usage.

Few words about XML

Mission-X uses files in XML format. The idea behind XML markup language is to describe a topic in hierarchical elements that are both readable and easy to understand. Anyone can define his/her format in a file but all needs to adhere to basic rules:

- every element should have a closing element.
  If the element is a stand alone one, then we should close it at the end of its declaration, example: `<link_to_objective name="FlyToKeyObjective" />`
  It will also be referred to as "***One liner element***" throughout the document.

- Every element can have attributes that describe it or sub-elements that can do the same.
  Attributes come in the format of *key="value"*

Here is a snippet that explain the concept:

```
<MISSION version="301" name="heli_rescue_v300" title="Helicopter Rescue">

<flight_plan>
    <leg name="reachKeyBiscyne" title="Fly to Key Biscayne" next_leg=""  >
      ...
      <start_leg_message name="starting_message" />
      <link_to_objective name="FlyToKeyObjective" />
      <desc><![CDATA[{some description};]]></desc>
      <map map_file_name="map01.png"/>
      <link_to_trigger name="{trigger name}" />
      ...
    </leg>
   ...
</flight_plan>
</MISSION>
```

<MISSION>	Is our root element
"version="301" is an attribute that describes MISSION elements compatibility. Since v24.12.2 the version was bumped to 312 but it still runs version 301, it just flag it as pre-v24.12.2 compatible. Example, the Inventory layout will be XP11 compatible. "name=heli_rescue" is an attribute that describes the mission name	
<flight_plan>	is a sub-element to MISSION. By itself it is just a container element for all <leg> elements that construct the course of the flight or actions need to be taken.
<leg> *Sub element to "flight_plan"*	The "leg" element has *attributes* of "name, title" and mainly *one liner sub-elements* that describe what constructs the current "leg", for example: "<map>, *<desc>, <link_to_objective> and more.

<![CDATA[{some text};]]	"CDATA" is a special ***sub-element*** that allows us to write free text. Usually used in descriptions or scriptlet elements.

By mixing elements with attributes and sub-elements we can create our own set of language that describe a topic and hierarchy of rules (what is and what is not). Not all elements or attributes are mandatory.

In this document the topic is: "**mission creation language**".

One liner element vs simple element

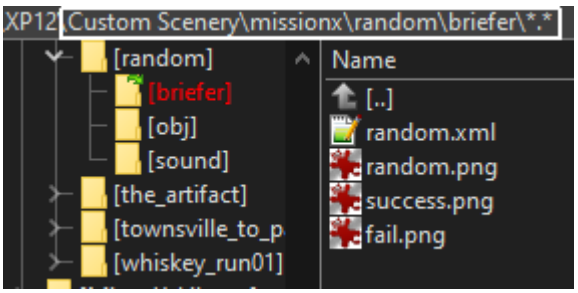
An element without sub-elements is referred to in this document as a:
"One liner element". Which means it enclosed itself.
Example:
<element attrib1="" />

Simple element or a non one liner element, will look like:
<element attrib1="">{content here or sub element/s}</element>

First things first - Mission Folder Structure

A mission folder has a specific structure with some flexibility. Here is a short explanation based on the "random" mission pack.
All mission folders must be located in "{XP}/Custom Scenery/missionx" folder.

Folder Structure



The root folder of this example is named: "*random*". Any mission folder must have a "*briefer*" sub folder that holds:

- The mission ".xml" file
- Images like the "success", "failure" and "mission image" files.

The main mission folder ("*random*") can hold other files like: scripts, 3D Objects and other images like "items, maps etc".

In the "*random*" mission folder we also see the "*obj*" and "*sound*" subfolders. The reason for that is because we predefined them in the "random.xml" mission file. They are not mandatory and we could place all files in the "root" folder of the mission (not the *briefer* folder).

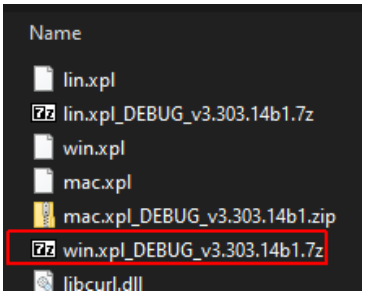
Images sizes for the "mission pick screen" can be a multiple of the following sizes:

- 240x190px (landscape)
- 145x210px (portrait)

Templated mission files: Once you master the mission syntax and concepts, you could write the same mission but with variants. You could do that using a "template" mission file. I do not suggest using this technique if you are a newbie for mission writing . You should read about it in the "Designing Templates" document first and then build a simple templated mission file (don't over complicate things).

The plugin Debug binaries

If you want to see more "log messages" and the "graphical cue" lines, you have to use the "debug build" of the plugin. It is usually located in the same folder as the "release" binaries. It should be compressed and its version number is uneven.
Example: v3.304.x is a release build while v3.303.x is a debug build sharing the exact same logic only with more "debug" information written to the Log.txt file.



Here is an example of a compressed "*debug*" build that needs to be extracted and replaced with the original "win.xpl" which is the "*release*" build.
You can use the "7zip" application to extract those files.

Troubleshooting Your Mission

- *I don’t see my mission file in the mission list screen:*
  - ☐ Check Log.txt file and check for any XML parsing errors when the plugin reads the file.
  - ☐ Check your mission file is in the "briefer" sub folder.

- *I don’t see the image I defined (mission image, map image, item image etc…)*
  - ☐ "mission image", "Success" and "failure" images must be in the "briefer" sub folder.
  - ☐ All the rest of the images - default location should be in the root mission folder (in this example the "random" folder).

- *I don’t see the 3D Object in X-Plane world or I see an error of missing 3D Object files when I load the mission*
  - ☐ Open the Log.txt file and check which 3D Object files are missing.
  - ☐ Make sure you placed your 3D Object files in the correct folder (did you define a dedicated folder name in the <global_setting> ?)
  - ☐ File name is incorrect or texture file is missing (open the ".obj" file and check the name of the texture file in it, usually in the first rows).
  - ☐ You are using a 3rd party 3D object file and the relative path you provided is wrong.

- *My script does not run*
  - ☐ Did you read about *embedded scripts* first in the designer guide ?
  - ☐ Did you place the script file in the correct folder (mostly the mission root folder)
  - ☐ Did you use the correct call for your function if it is in an external script file ("script file name"."function name") ?
  - ☐ If the script is an internal one: "<scriptlet>", did you use the correct name to call it ?
  - ☐ Check the Log.txt file for "interpreter" errors, should also be displayed in the mission screen.
    Try to figure out what you did wrong in the script syntax.
  - ☐ I receive "*Assign operator expected*" while there is no assign needed since we are calling a built in plugin function.
    Please check if you are using the correct "prefix" for your functions. Mission-X function calls start with "fn_{name}" and you might write only the function "{name}" without the "fn_" prefix.

- *Why don't I see more debug messages in the Log.txt file or the 3D Cue hints inside the X-Plane world ?*
  - ☐ Did you use the *debug* version of the plugin. Usually located in the plugin folder as a compressed file with the name: "win.xpl_DEBUG_xxx.zip" format or something similar.

Unzip this file and override the original "win.xpl" file (make sure to make a backup of it so you could rollback to the *release* binaries).

- [ ] If you want to see the "Cue graphical hints" in the X-Plane world, you must use XP11.
- [ ] Did you enable the "designer mode" and "cue mode" (in the menu) so you could see the "Graphical Cues" ?

TIPs FOR ANY DESIGNER

Decrease mission test times

When testing a mission, you ***do not*** need to fly the whole mission. You should use the X-Plane "flight configuration" to jump between airports instead of flying to them. If you need to "taxi" on the ground, just use X-Plane "map", click your plane, and move it around.
That will save you lots of time and frustration.
Remember, "Map" and "flight configuration" are your friends.

Finding plane coordinate (command "missionx/designer/xxx")

You have few custom commands that will write your plane or camera coordinate to the Log.txt file. Just copy the relevant data from it into your mission file.
See Appendix A for more explanations.

How to test a specific leg

Let's assume you have 10 flight legs in your mission and you want to test the 7th. You do not need to fly the whole 6 legs to test the 7th one, just modify your "briefer" "starting_leg" attribute, and the plugin will start from it.
Please do remember that your starting location can be changed too, or you will start from the original location (designed for starting "leg" only).
Another option is to modify the sub element: "<designer>" in the <global_settings> with the desired leg name. You will have to use the DEBUG build of the plugin:

```
<designer force_leg_name="leg7" />
```

Seeing is believing

As a designer, if you want to see the triggers or inventories area, you should set the <MISSION>s element attribute: "designer_mode" to "yes". You should see simple graphical cues highlighting the event areas in the X-Plane world.
As of v24.02.x you can see the same in X-Plane Nav Map, in XP11/12.

Dialogs with zero code

As of v3.0.224.x the <message> element has been enhanced and you can use it to call other messages to create a dialog like conversation. But it does not stop there, you can change the time, inject a metar file or even call a script/scriptlet code at the end of the dialog.
All this without writing a single line of code.

To be or not to be

Since v3.0.231x any designer can interact with the simmer using the <choice> element. Although you are limited by the number of options (6 for each choice), it is as flexible as a trigger, meaning, you can fire a script, call a message, fire a command or change datarefs. These actions allow for much diversity on each option element in the "choice window".

Organizing your Flight Legs

The plugin supports three mission styles. You can mix and match between them, but I think you should stick to the latest style if you write your mission file from scratch.
One of the features the plugin supports is the ability to read multiple <flight_plan>, <objectives>,<triggers> or <message_templates> elements.
So, instead of grouping all <leg>s in one <flight_plan> or all triggers in one <triggers> element, you can create for each <leg> its own <flight_plan>, <objectives> and <scriptlet>.
Example, let's assume we have a mission with 3 flight legs.
We could organize it the old way:
Everything in one root element: <flight_plan> for all legs, <triggers> for all triggers and <message_templates> for all messages.

Leg 1+2+3	
<flight_plan> ...{all flight legs} </flight_plan> <objectives> ...{all objectives} </objectives>	<triggers> ...{all triggers} </triggers> <message_templates> ...{all messages} </message_templates>

Debug Tab

Since v24.02.x you have a "debug" tab that allows you to monitor key aspects of the mission elements, like "tasks, triggers, datarefs, scripts and messages.".

- Triggers can be forced (Fire enter/leave events).
- Messages can be played (and their post action will be called too).
- Scripts can be modified "in-memory" to fix errors or to enhance them (it won't directly modify the original mission file, you will have to copy the modified code).

The **latest style** is to create one "<flight_plan>" for each <leg> element and its complement elements meaning: <objectives>, <triggers> and <message_templates> elements alike, for each waypoint.

Leg 1	**Leg 2**	**Leg 3**
<flight_plan> <leg> ... </leg> <objectives> ... </objectives> <triggers> ... </triggers> <message_templates> ... </message_templates> </flight_plan>	<flight_plan> <leg> ... </leg> <objectives> ... </objectives> <triggers> ... </triggers> <message_templates> ... </message_templates> </flight_plan>	<flight_plan> <leg> ... </leg> <objectives> ... </objectives> <triggers> ... </triggers> <message_templates> ... </message_templates> </flight_plan>

This way it is easier to find your way around and navigate between parts of the mission and make fewer mistakes.

Style difference: Side by side

Here are the different supported styles

Before v3.0.224.4	**v3.0.224.4**	**V3.0.241.7 and up**
<flight_plan> ...{all flight legs} </flight_plan> <objectives> ...{all objectives} </objectives> <triggers> ...{all triggers} </triggers> <message_templates> ...{all messages} </message_templates>	<flight_plan> <leg> ... </leg> </flight_plan> <objectives> ... </objectives> <triggers> ... </triggers> <message_templates> ... </message_templates>	<flight_plan> <leg> *. only 1 leg.* </leg> <objectives> *only objectives* *related to current leg* </objectives> <triggers> *only triggers related* *to current leg/tasks* </triggers> <message_templates> *only messages related* *to current leg/tasks* </message_templates> *Embedded scripts* <shared_variables>...</ > <scriptlet>... </ > </flight_plan>

You can mix and match the styles, but I suggest sticking to only one style per mission file.

Mission Header Elements

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

This tag describes the type of the file and encoding.
You can use iso-8859-1 or UTF-8. If you are not sure, then leave it as is.

<MISSION>

```
<MISSION version="300" name="Townsvile1_300" title="There and Back" designer_mode="no" >
```

The MISSION tag is the enclosing element of the mission data file.

Name	type	Values	Meaning
version *	str	301 | 312	Mission plugin compatibility. USe "301" to keep some screens with XP11 compatibility like the Inventory Layout. "312" supports the new Inventory layout or newer features that might not work with XP11.
name *	str	[Name_version] example: Alsaka_110	Unique name for Mission. Defines also the save file name. Avoid spaces. Add the version compatibility to the name.
title	str	"Madagascar Mission"	Short (around 40 characters) mission title, to display in the briefer.
designer_mode	str	["yes"| "no"] ["true" | "false"]	Designer mode is a flag that allows a designer to display "cues" in the x-plane world.

If you don't define the mission name, this might disable the option to save the mission.

<mission_info />

One liner element.
This element is important for the "mission pick" and "mission detail description" windows, before starting the mission.

```
<mission_info mission_image_file_name="t_to_palm.png" plane_desc="GA plane"
              estimate_time="~45 Minutes." difficulty="Easy."
              other_settings="Mission should not start in cold startup."
              weather_settings=""
              scenery_settings="OpenSceneryX v2.0, Townsville_Mission
                                (should be part of the mission package)" />
```

The setup information for our mission is listed in this element.
Always make sure you define this element to make mission settings easier for the simmer.

Images sizes could be: 240x180 or 145x210

<global_settings>

When a mission starts, it first apply the settings from "global_settings" element.
Let's take a look at a "global_settings" attributes:

```
<global_settings>
  <folders sound_folder_name="sound"  />
  <start_time day_in_year="100" hours="22" hours="" min="00" />
  <base_weights_kg pilot_base_weight="85" passengers_base_weight="150" storage_base_weight="35" />
</global_settings>
```

<folders />

One liner element.
The "folders" Element allows the designer a more constructive way to define file locations. The default folder is the root mission folder. This element is optional, which means if you don't define a folder for each file type, then the plugin will use the root folder.

"<folders>" element attribute:

Attribute Name	Type	Example Value	Meaning
sound_folder_name	str	"sounds"	Folder that holds all sound files.
obj3d_folder_name	str	"objects"	Folder that holds all 3D object files.
metar_folder_name	str	"metar01.txt"	Folder that holds all custom metar files.
script_folder_name	str	"scripts"	Folder that holds all script files.

<start_time />

One liner element.
Define the mission's local time.
Provide day in year + local hour

"<start_time>" element attribute:

Attribute Name	Type	Value	Meaning
day_in_year	int	0..364 **""**	Day from start of year (any year) **"" empty** or less than Zero, means: use current.
hours	float	0..23	Local time
min	int	0..59	

If you don't set a time, then the plugin will use the current time.

<base_weights_kg />

One liner element.
Designer could define the base payload for the plane or let the simmer manually move items from external inventory into the plane.
As of v24.12.2, there are two inventory layouts for the plane. The first is the XP11 compatible layout, were you have one container to move items into, and weight is evenly distributed by X-Plane, and the "newer" one is based on X-Plane 12 station implementation, which allows the simmer to directly place items in specific "locations" inside the plane, which directly affects the CG.
In case of the latter, only the "`pilot_base_weight`" will be calculated, and only if the "station 0 (Pilot)" is less than 40kg.

"<base_weights_kg>" element attributes:

Attribute Name	Type	Compat.	Value	Meaning
`pilot_base_weight`	float	xp11/12*	**85**, 70, 120	Pilot weight in Kg. **Default: 85kg**. Takes effect in both inventory layouts.
`passengers_base_weight`	float	xp11	**0**, 270.5	All passengers, including co-pilot/s
`storage_base_weight`	float	xp11	**0**, 5, 34	All storage items weigh: luggage, handbags etc...

* Pilot weight will override the “station 0” if its weight is less than 40kg.

Since Mission-X v24.12.2 and onward, the inventory is aligned with the X-Plane 12 "station" feature, which allows the plane designer to define the location in the plane where to place the "station". This directly affects the CG.

<compatibility />

X-Plane evolves, and some of the previous features it had might get better integration from Laminar, therefore the plugin might try to reflect the same.
One such feature is the plane "station" areas, where you store your "staff".
Until XP12 all weights were distributed evenly.
The plugin tries to reflect the new ability but it needs to allow mission designers to keep compatibility with their original mission intent.

Attribute Name	Type	Value	Meaning
`inventory_layout`	int	[""|11]	Which plane layout to display in the inventory screen.

inventory_layout = "11" or on X-Plane 11:
Payload is distributed evenly between the different station areas.

inventory_layout is not set, or set to any value except "11"
Each item will be placed, by the simmer, in its own "station" area, which directly affects the "Center of Gravity".

Behind the scenes, when the simmer commits the move, the plugin tests the first station weight, *it always assumes it is the pilot's station*. If the weight is less than 40kg, it will apply a 80kg pilot weight, overriding the “items” that were placed there.
Read more in the “inventory” topic.

<position />

One liner element.
Define how the auto-position acts.

"<position>" element attribute:

Attribute Name	Type	Value	Meaning
auto_position_plane	bool	**yes** | no **true** | false	When starting a mission the plugin will auto position the plane. Drawback, it screws the "start cold and dark". If the designer does not want the auto position to take place then he/she should flag this attribute as false. Please be advised that the simmer should be aware that they need to start in the correct location before mission start.

<designer />

One liner element.
Define which flight leg to start your tests based on the given name

"<designer>" element attribute:

Attribute Name	Type	Value	Meaning
force_leg_name	str	{valid leg name}	If we are in designer mode (can be altered in the menu too) then start the mission from the given leg name and ignore briefer settings.

<timer />

One liner element.
Designers can define a failure timer to limit mission or part of it at the global settings.

For more information check the <timer> topic in the <leg> topic.

<scoring>

Designers can define scoring for ad-hoc datarefs (this is not a custom list as of v3.304.9)
You can rate the simmer performance in the following
categories: "pitch / *roll* / *gforce* and *center_landing*" (if plugin could deduce the

runway information).
For each category you must define: "*best*, *good* and *average*".

Here is an example:

```
<global_settings>
...
  <scoring>
    <pitch best="-10|15|0.5" good="-15|18|0.4" average="-20|20|0.25" />
    <roll best="-25|25|0.5" good="-30|30|0.4" average="-35|35|0.25" />
    <gforce best="-0.5|1.55|0.5" good="1.55|1.82|0.4" average="1.82|1.95|0.25" />
    <center_line best="-1.3|1.3|0.5" good="-1.8|1.8|0.35" average="-2.3|2.3|0.25" />
  </scoring>
...
</global_settings>
```

For each category you have to define the "min | max and score" values.
The score will be given for "min" or the "max" values, which means that the score is the summation of both of them.
In the example above, the score value for "pitch" is "0.5", this means that the highest score for pitch is "*one*" ("0.5" for min plus "0.5" for max. This is achieved by the simmer if they did not exceed both of those pitch values during the whole flight).
Simmer will receive zero ("0") points if they exceed the "*average*" values. This means you can receive points for "min" but not for "max" if you exceed the worst case scenario (which is the "average" attribute) and vice versa.

Exceptions: The "`center_line`" should have two mirrored values, the "min" value should be the negative mirror of the "max" value, and the score should be halved, so if max score is "1" then use "0.5" instead (see example above).

Category Name <sub element>	Attributes format	Meaning
For each "sub element" you must define: "best", "good" and "average" attributes. Each attribute must define a string that reflects "min|max|score" values. Use pipe ("|") character to split between *min/max* and "*score*" values Example: <pitch best="-10|15|0.5" good="-15|18|0.4" average="-20|20|0.25" />		
pitch	"min|max|score"	Minimum and maximum pitch during the whole flight
roll	"min|max|score"	Minimum and maximum plane roll during flight
gforce	"min|max|score"	Minimum and maximum normal G-Force
center_line	"min|max|score"	Represent the distance to the center of the runway. "Minimum" value should be the negative of "Maximum". If "max=1" then "min=-1"

<weather />

```
<weather>
sim/weather/cloud_coverage[2]=1.|sim/weather/use_real_weather_bool=0|sim
/weather/cloud_base_msl_m[2]=10668.|sim/weather/shear_direction_degt[2]=
0|sim/weather/barometer_current_inhg=29.89|sim/weather/cloud_coverage[0]
=0.
</weather>
```

The weather element "text" value holds the datarefs to update X-Plane. Although you can set any dataref you want, it was designed mainly to focus on the weather dataref aspect.
For more information, see the "***weather***" topic in the "<leg>" section.

<briefer> element

The briefer is the "gateway" to your mission. In the briefer element you describe the background for the mission. You also define the starting location using ICAO code but I suggest using "latitude" and "longitude" for exact location instead.
Let's take a look at our demo "briefer" element:

```
<briefer starting_icao="YBTL" starting_leg="DeliverPalm" >
  <location_adjust lat="-19.2536716" long="146.770935" elev_ft="0" heading_psi="13"
                   pause_after_location_adjust="true" starting_speed_mt_sec=""/>
  <![CDATA[Some description]]>
</briefer>
```

<briefer> attributes

Attribute Name	Type	Value	Meaning
starting_leg*	str	"YbtlToYpam"	Which part of the flight plan is the starting one? Pick one of the "flight leg" name
starting_icao	str		Optional, ICAO starting location. It is better to use "lat/long"
position_pref	int	"10|**11**"	When positioning the plane, we have the option to use the **xp11** or xp10 code logic. The first will force the engine to start.
<![CDATA[]]>*	str	Mission description	Free text describing the mission

<location_adjust> attributes

Attribute Name	Type	Value	Meaning
lat / long	float	10.123650 -92.325641	exact starting position. For precise location use decimal location with 6 positions after the decimal sign
elev_ft	int	0, 100, 1234	Elevation in feet. Default is **0** = on ground
heading_psi	float	13.10	plane heading, true north
pause_after_location_adjust	bool	yes|**no** true|**false**	After the plane is positioned, does X-Plane start in "pause" mode or not. This is probably useful when starting in a flying position.
starting_speed_mt_sec	float	**0**, 51.44	Starting speed in meters per second. 1 Knot = 0.514444 meters per seconds

force_heading_b [mainly for designers]	bool	**[false**|true]	Designers can force only the heading to change even if the plane is less than 20 meter radius from the starting location. This is mainly for designers' use.

<datarefs_start_cold_and_dark>

When we start a mission the plugin positions the plane at the starting location. Unfortunately this action brings the plane to "running" condition regardless of your preferred preference (to start in "cold and dark").

As a workaround, I decided to add two options to bring the plane to cold and dark and the dataref option is the preferred one (IMHO). This element allows you to enter a set of dataref names (full path) and their initialized values to bring the plane to a "cold and dark" state. It might not be as good as Laminars implementation, but it is close. You can also use this to bring the plane to other desired states during the start of the mission, which is the main use of this element.
Other benefits: you can update custom made datarefs and the plugin will try to modify them.

Here is an example of a simple cold and dark implementation that I defined for the cessna 172. It can be tweaked and modified to include more systems that are not listed here:

```
<MISSION>
…
…
<datarefs_start_cold_and_dark>

sim/cockpit2/controls/yoke_heading_ratio=0|
sim/cockpit2/engine/actuators/prop_pitch_deg[0]=-4.9|sim/cockpit/radios/transponder_mode=0|s
im/cockpit/switches/pitot_heat_on=0|sim/cockpit2/controls/flap_ratio=0.0|
sim/cockpit/switches/pitot_heat_on2=0|sim/cockpit2/engine/indicators/prop_speed_rpm=0|sim/co
ckpit/electrical/taxi_light_on=0|sim/cockpit/electrical/nav_lights_on=0|sim/cockpit/electric
al/beacon_lights_on=0|sim/cockpit/electrical/landing_lights_on=0|sim/cockpit/electrical/stro
be_lights_on=0|sim/cockpit2/engine/actuators/ignition_key=0|sim/cockpit2/engine/actuators/ig
nition_on=0|sim/cockpit2/autopilot/autopilot_on=0|sim/cockpit2/electrical/battery_on=0|sim/c
ockpit2/electrical/generator_on=0|sim/multiplayer/controls/engine_prop_request=0

</datarefs_start_cold_and_dark>
…
…
</MISSION>
```

[v3.304.12 changed syntax] The format is: "{dataref}={value1[,value2,value3…]}|{dataref}=...". Example: `sim/cockpit2/controls/yoke_heading_ratio=0|sim/xxx/xxx`

It supports integer,float,double,integer array and float array.

Since **v3.304.12** if you define more than one value into an array, then it should apply it correctly.
Example:

```
sim/weather/region/cloud_coverage_percent=0.8,0.4,0.3|sim/cockpit2/controls/yoke_heading_ratio=0
```

The "`sim/weather/region/cloud_coverage_percent`" has three values in the array, you don't have to set all of the three for it to work, you can set a partial array value set. For example, Modify only the first two members.

Descriptive Elements

Elements that describe the mission objectives and how it should be constructed.

Flight Leg

Every mission is constructed from a set of one or more flight legs. A "leg" is constructed by one or more objectives and an objective is constructed by tasks we want the simmer to accomplish.
In Mission-X v3 we link the objectives to flight legs.

Please visit the tips section to read about suggestions on how to organize your mission data file.

Let's take a look at a "leg" element:

```
<flight_leg>
<leg name="DeliverPalm" title="Deliver goods to Palm Island" next_leg="FlyBackToYBTL" >
      <draw_script name="" />
      <pre_leg_script name="" />
      <start_leg_message name="Fly to Palm Message" />

      <desc><![CDATA[Some description]]></desc>
      <link_to_objective name="Land and Park in YPAM" />
      <link_to_trigger   name="trig_start_mxpad01" />
      <display_object instance_name="marker01_2" name="marker01" link_task=""
                      replace_lat="-18.751736" replace_long="146.579529"
                      replace_elev_above_ground_ft="30" />
      <post_leg_script name="" />
      <end_leg_message name="" />
 </leg>
[since v3.0.242.7]
 <scriptlet> ... </scriptlet>
 <shared_variables> ... </shared_variables>
</flight_leg>
```

You can remove elements and attributes that are not in use or with default values:

```
<leg name="DeliverPalm" title="Deliver goods to Palm Island" next_leg="FlyBackToYBTL" >
  <start_leg_message name="Fly to Palm Message" />

  <desc><![CDATA[Some description]]></desc>
  <link_to_objective name="Land and Park in YPAM" />
  <link_to_trigger   name="trig_start_mxpad01" />
  <display_object instance_name="marker01_2" name="marker01" replace_lat="-18.751736"
                  replace_long="146.579529" replace_elev_above_ground_ft="30" />
</leg>
```

It is the same elements with the same rules but without the attributes we do not need or their default values is good enough

Since v3.0.242.7 you can add "scriptlet" and "shared_variables" elements to the flight_plan, to make it easier on the designer managing their code especially for long or complex missions with many flight legs.

<leg>

Attribute Name	Type	Value	Meaning
`name *`	str	DelieverPalm	Unique name for our flight leg
`title`	str	Deliver goods	User readable title for the flight leg
`next_leg`	str	FlyBackToYBTL	Once a leg is finished, which next "leg" takes over ?
`is_dummy` `[v3.304.12]`	bool	[yes|**no**], [true,**false**]	You can flag a <leg> as a dummy one in order to do quick logic work or to move some of the previous leg actions into it. Think of it as a transition leg for cleanup or doing a background one time work.

A flight leg element defines the highest level of what you want to achieve. You still need to link it to action elements. They do not always have to be related to fly tasks, it can be related to any task you think can progress the mission.

"Dummy <leg>" (is_dummy="yes")

The attribute "`is_dummy`" is a flag that allows you to define an in between <leg> without needing an objective to succeed, it will ***always*** succeed after 2 flight cycles (which is around 1-2 seconds).
The use of a "dummy <leg>" is up to you, you don't need to use it, but you might want to offload some logic between two complex <leg>s.

What it supports:
The elements that will work for a <leg> with **is_dummy="yes"** are:

- <start_leg_message>
- <end_leg_message>
- <pre_leg_script>
- <post_leg_script>
- <link_to_trigger>
- <fire_commands_at_leg_start>
- <fire_commands_at_leg_end>
- <display_object> and <display_object_near_plane>
- <special_leg_directives>

You should not fill in any information regarding:

- <link_to_objective>
- <map>
- <gps>

<link_to_objective />

One liner element. Link a flight "*leg*" to an *objective*, hence giving it purpose.
You can define one or more objectives in a flight "leg".

Attribute Name	Type	Value	Meaning
name *	str	"Land and Park in YPAM"	Objective exact "name" to link to. Every flight loop, we will evaluate this objective.

<desc>

Simple element where you describe the flight "leg" objectives.
This description will be ***presented*** during mission load.

```
<desc><![CDATA[Some description]]></desc>
```

In v3.0.301 you can write the desc directly under <leg> and ignore <desc> element.

```
<leg>
  <subelement1 />
  <subelement2 />
.. .. ..
<![CDATA[Some description]]>
</leg>
```

<draw_script />

One liner element.
In rare cases where you need to do some action at frame level (every 4th frame or so), you can add a call to an external script. You should abide by the following rules:

1. Always think about performance - this function can hinder performance big time.
2. The scope of the function is only at the flight <leg> level. So *mxCurrentTask, mxCurrentObjective and mxCurrentTrigger* won't be available to you, meaning you are limited by the functions you choose to implement.
   On the other hand Global variables can play a handy role in passing values or keep progressing between scopes (example: trigger and flight leg).
3. Keep the code short and optimized.
4. Use the "*fn_stop_draw_script()*" function, when "draw_script" has concluded itself and it is no longer needed.
5. Use: "*fn_set_draw_script_name({name})*" function to replace or call draw script.
6. Only one draw script per flight leg.
7. The life of the script is within its Flight Leg. Once you transition to a new Flight Leg it is no longer active.
8. Test your implementation from a performance perspective too.

More functions in: "Flight Leg related Functions" table.

<timer />

One liner element.
Designers can define a failure timer to limit mission or part of it at the flight leg settings.

"<timer>" element attributes:

Name	type	Values	Meaning
name *	str	"timer_leg1"	Unique timer name
min *	float	**0.0**, 55.0, NN.nn	How many **minutes** to count down. Default is "zero", which means - ignore the timer.
run_until_leg	str	**""**, {some flight leg name}	Provide a flight leg name where the failed timers should stop running (means success for simmer).
success_msg fail_msg	str	"", "way_to_go_msg", "timeup_msg"	Optional a ***predefined*** message name. Can be empty. For failure cases, the plugin will provide a default message if not defined by the designer.
Since v3.306.3			
fail_on_timeout_b	bool	"**true**"|"false"	Default behavior is to fail the timer on time end.
post_script	str	"Some script name"	The post script will be called only on **failure**, when the timer reaches ***Zero (FAILURE)***, so basically you can treat the code as ***failure*** handling. Won't be called if the plane reached the target before the timer ends.
stop_on_leg_end_b	bool	"**false**"|"true"	Stop evaluating the timer at the end of the current leg.

Timer will start automatically at the beginning of the flight leg

You can only have one timer per flight leg. If you define more than one, the plugin will ignore them.

> Please remember that failing to define the "run_until_leg" attribute means that the timer will continue its countdown even on flight leg change.
> This might be OK in some cases but frustrating if not timed correctly.

<metar />

One liner element. Not supported in XP12, yet.
You can inject one custom metar file in each "flight leg" transition (if you want). In future builds you will be able to do the same through script function.
The custom metar file must adhere to X-Plane weather format, which is the same as NOAA METAR file format (as far as I understand).

```
<metar   metar_file_name="metar01.txt" force_custom_metar_file="no" />
```

Attribute Name	Type	Value	Meaning
metar_file_name *	str	"metar01.txt"	Mandatory Metar file name
force_custom_metar_file	bool	[yes|no] [true|false]	Allow or disallow a user to modify weather during a mission ? Will re-apply the same metar file every time a change will take place.

The metar file will be copied from the metar folder you defined in the "global_settings", it will default to the mission root folder if none was provided.

Useful elements to communicate with simmer - simplify messaging

<dynamic_message />

Full chapter can be read in the "Designing Templates" document, but you can read its first page in the Advanced Topics.

<start_leg_message />

One liner element. Will be called after the <pre_leg_script> element.
This will allow you to send an explanation (no need for triggers or special scripts).

Attribute Name	Type	Value	Meaning
name *	str	"start_leg_01_message"	Simplify messaging at the start of a flight leg. Fires only once.

<end_leg_message />

One liner element. Will be called after <post_leg_script> was triggered.
Designer can use that element to send a message to simmer, at the end of the flight "leg", explaining what should be done in the next leg or to conclude the flight.

Attribute Name	Type	Value	Meaning
name *	str	"end_leg_01_message"	Simplify messaging at the end of a "leg". Fires only once.

One time scripts at the start and end of a "leg" - optional and might be useful

<pre_leg_script />

One liner element.
One time script that will be called when the flight "leg" becomes active.

Attribute Name	Type	Value	Meaning
name *	str	"okavango.pre_leg_1"	

Read the topic "embedded script" to learn how to send "predefined parameters" if you need.

<post_leg_script />

One liner element. One time script that will be called when the flight "leg" becomes active

Attribute Name	Type	Value	Meaning
name *	str	"okavango.post_leg_1"	Fires after a flight "leg" is finished and before transitioning to "next_leg".

Read the topic "embedded script" to learn how to send "predefined parameters" if you need.

<link_to_trigger />

One liner element.
A trigger is a flexible means to execute anything from simple to complex logic.
In a nutshell, the trigger will allow us to call a message or to monitor simmer behavior and decide what to do according to the values we receive (more on that in a later topic)..

A trigger at the flight "<leg>" level, can act as:

a. One time message that you will trigger in a certain time/location.
b. An instructor - simple implementation for specific cases.
c. Start MXPad message sequence.

<link_to_trigger name="trig_start_mxpad01" />

Attribute Name	Type	Value	Meaning
name *	str	"trig_start_mxpad01"	Trigger "name" to link to. Every flight loop, we will evaluate this objective as part of the flight "leg".

<map />

One liner element.
Define one or more maps/images you want to display in the MAP layout.
You can see the map by mapping the command:
"missionx/general/missionx-toggle_map" to your joystick or keyboard, or through the Mission Briefer Window, after mission was started.
You can cycle between the maps (if there are more than one map in the flight "leg").
The map visible is: "***1180x480***" so plan the image in the correct aspect ratio.

<map map_file_name="map01.jpg" />

Attribute Name	Type	Value	Meaning
map_file_name *	str	"map01.png"	Map file name. Supports png/jpg and bmp. Other formats were not tested.

<weather />

(v3.304.12)

```
<weather>
sim/weather/cloud_coverage[2]=1.|sim/weather/use_real_weather_bool=0|sim
/weather/cloud_base_msl_m[2]=10668.|sim/weather/shear_direction_degt[2]=
0|sim/weather/barometer_current_inhg=29.89|sim/weather/cloud_coverage[0]
=0.</weather>
```

The weather element "text" value holds the datarefs to update X-Plane. You can set any dataref you want, but it was designed mainly to focus on the weather dataref aspect. It is being used as a "one time dataref modification" at the beginning of the flight leg or global settings, depending where you set it.
From the element name you can surmise that the main usage of this element is to manually modify the weather settings at the beginning of the flight leg.
In XP12 the weather setting might not change immediately and it might take ~60 seconds until the change will complete.

Since there is no dedicated attribute for this element, the format is always:

Attribute Name	Type	Value	Meaning
Example			
<weather> {key}={value}|{key}={value} <weather> <weather>sim/weather/cloud_coverage[2]=1|... </weather> Or <weather>sim/weather/region/cloud_base_msl_m,3048.0,6096.0,9144.0|... </weather>			

The **key** must hold the absolute dataref path.

> Most of the datarefs are supported, this includes arrays.
> For arrays use multiple "," between values for each array value.

<fire_commands_at_leg_start /> <fire_commands_at_leg_end />

The following "elements" will assist in executing X-Plane registered commands at the start and end of a flight leg.
Example from the Bell-412 by X-Trident:

```
<leg ....>
  <fire_commands_at_leg_start commands="412/buttons/PATIENT_off" />

  <fire_commands_at_leg_end commands="412/buttons/PATIENT_on_board,412/buttons/xxxx" />
</leg>
```

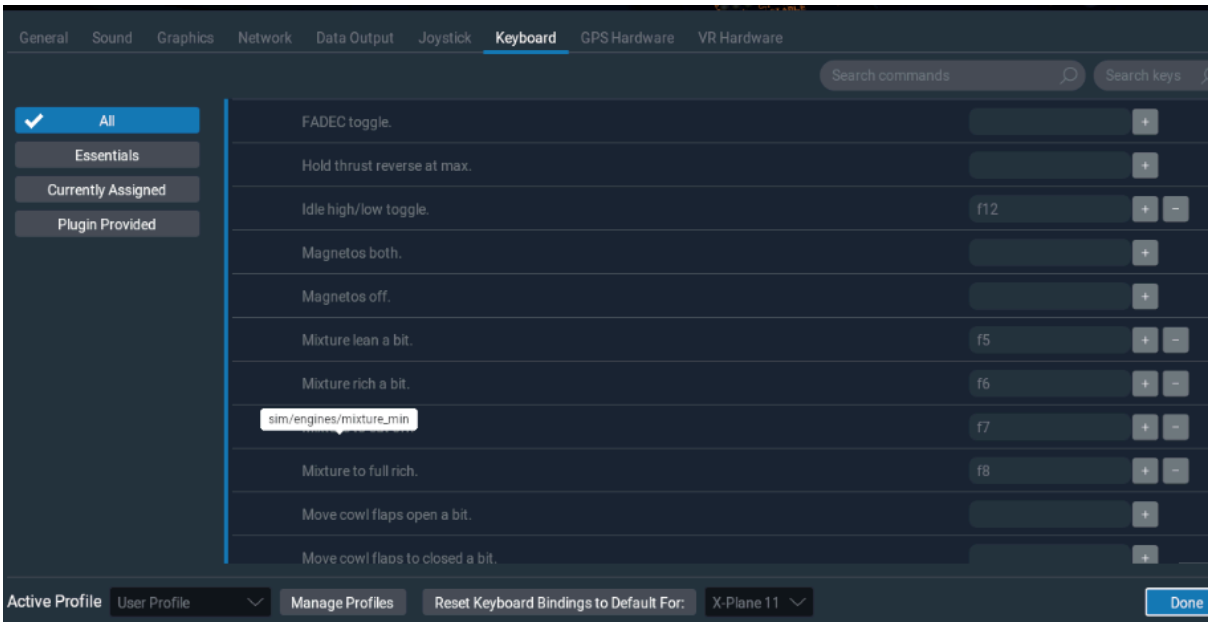
You can add timing to the command example:
`sim/engines/throttle_up:0.5`
The command will run for ~0.5sec and stop.

You will have to time it correctly, even 1 sec can have a large impact.

You can concatenate more commands using commas (",") between them.
You can even define a cold and dark set of commands, but I suggest using the dataref approach, since it manipulates the system directly.

> Tip: In some cases of "cold and dark" you should be able to use both techniques: "datarefs" and "commands" to prepare the desired scenario for a specific plane.

You can get the commands path by hovering above the desired command in the X-Plane keyboard binding screen.



The tooltip holds the commands path.

<display_object /> and <display_object_near_plane/>

One liner element.
You define in which flight "leg" you display a 3D Object using the "<display_object>" element. Basically you "ask" the plugin to create an instance (a copy) of the template object.

```
<display_object name="marker01" instance_name="intro_marker_01"
[replace_keep_until_leg="takeoff_from_rw34c"]  />
```

"display_object" attributes

Attribute Name	Type	Value	Meaning
name*	str	"metar01.txt"	Mandatory - object template name defined in <object_templates>
instance_name	str	"into_marker_01"	Unique name for the Object template.
link_task	str	"{objective}.{task}"	If you want 3D Objects to be linked to task state.
replace_{attrib_name}	str	replace_keep_until_leg	When we define a new instance we might need to modify some of its attributes, since the template one might not be relevant to the new instance. For example - location information. The attributes you can add "reaplce_" before them are: ***"lat", "long", "elev_ft", "keep_until_leg", "elev_above_ground_ft", "heading_psi", "pitch", "roll", "distance_to_display_nm"***
target_marker_b	bool	[true,**false**]	A flag that tells the plugin if the marker is above a target.
hide	bool	[true,**false**]	Mainly for internal use
v3.304.11			
relative_pos_bearing_deg_distance_mt	str	{bearing}|{distance} Example: 90|10 Or {acf_psi}|10 **Supports special keywords to make positioning more precise.** **Check below.**	Borrowed from "template" syntax. *Bearing* in degrees and *distance* in meters. This **only** applies to the location of the plane at the time the Object is in the pool of instances. Any <display_object> you will use with this attribute will be calculated relative to plane position and it will be done only once. It is best use on ground and at the start of a "flight_leg"
display_at_post_leg_b	bool	[true,**false**]	When to display the 3D Object, at the start or the end of the flight leg. Default is ***false*** (start of flight leg) Best use at the last flight leg.

<display_object_near_plane/>
"Display object near plane" is a special case of "<display_object>". It is here to be

compatible with the same element name in the template file. You can use it exactly like the <display_object> but you must set the "relative_pos_bearing_deg_distance_mt" attribute.

Keywords to use with "relative_pos_bearing_deg_distance_mt" attribute
If you use the "relative_pos_bearing_deg_distance_mt" attribute, you can use the following predefined keywords to dynamically calculate the bearing or position of the object relative to plane location **or** you can use one of your custom number <*dataref*> elements you defined in the <*xpdata*> root element.

Pre Defined Keyword	Supported	Explanation
{acf_psi}	xp11/12	The bearing of the plane
{wing_span}	xp11/12	The length of the wing (1 side of the wing)

You can write a simple mathematical expression as follow:

```
<display_object_near_plane name="crate01" instance_name="crate01_01"
relative_pos_bearing_deg_distance_mt="{acf_psi}+90|{wing_span}+1"
replace_lat="1" replace_long="1" replace_distance_to_display_nm="10.00"
target_marker_b="no" replace_elev_ft="" />
```

In this example we want to place the object 90 degrees from the plane bearing and position it one meter after wingtip (The position depends on the bearing).

<special_leg_directives />

One liner element.
This element is being populated mainly by the plugin. In generic mission file there is no real use for it, but in generated missions you can define the "base_on_external_plugin" attribute.

```
<special_leg_directives base_on_external_plugin="yes" />
```

Attribute Name	Type	Value	Meaning
base_on_external_plugin	bool	[yes|no] [true|false]	Will add to the generated task this attribute. It allows the task to bypass the trigger and script definition and it test success against the custom dataref: "xpshared/target/status" This will be used in special cases where we know an external plugin knows it needs to write to this dataref when it defines the action success or failure.

Flight Leg related Functions

The following functions can be called from a trigger/task or flight "leg" script.

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
<Leg> related functions	
fn_set_next_leg ("Leg name")	This is a dedicated function that you can call to modify your flight "leg" property "next_leg" value. You can call it from "mxpad"/"trigger" or "task" scripts. You must provide a valid flight "leg name".
fn_stop_draw_script()	Useful for performance. Will remove the script name so nothing will be called in the next plugin draw callback.
fn_set_draw_script_name ("script.func")	Modify the script to use for each ~4th frame. You can transition between draw scripts depending on circumstances. It can also help to optimize the code, since we will call smaller scripts rather than longer ones.
fn_set_leg_desc("text (s)") fn_set_leg_desc("leg name", "text") [v3.306.3]	Replace the current leg description text or specific leg name description. The function has two options, pick the one that suits you best.
fn_get_current_leg_desc() [v3.306.3]	Retrieve current flight leg description text.

Objectives - What are the mission objectives

The "<objective>" element is the main element to define the expected actions the simmer needs to take. An action is represented as a task, some are mandatory and some optional.

```
<objective name="Land and Park in YPAM" title="" >
   <task name="LandInYPAM" title="Land in YPAM" depends_on_task=""
         base_on_trigger="trig_land_in_YPAM" base_on_script="" eval_success_for_n_sec="3"
         mandatory="" force_evaluation="no"/>
   <task name="ParkInYPAM" title="Park in YPAM" depends_on_task=""
         base_on_trigger="trig_park_in_YPAM" base_on_script="" mandatory="yes"
         force_evaluation="yes" />
   <desc><![CDATA[Park plane in YPAM, and deliver the goods.]]></desc>
</objective>
```

You can remove attributes that are not in use or with default value:

```
<objective name="Land and Park in YPAM" >
   <task name="LandInYPAM" title="Land in YPAM" base_on_trigger="trig_land_in_YPAM"
         base_on_script="" eval_success_for_n_sec="3" />
   <task name="ParkInYPAM" title="Park in YPAM" base_on_trigger="trig_park_in_YPAM" mandatory="yes"
         force_evaluation="yes" />
   <desc><![CDATA[Park plane in YPAM, and deliver the goods.]]></desc>
</objective>
```

It is the same elements with the same rules but without the attributes we do not need or their default values is good enough

<objective>

Attribute Name	Type	Value	Meaning
name *	str	"Land and Park in YPAM"	Trigger exact "name" to link to. Every flight loop, we will evaluate this objective as part of the flight leg.
title	str	{enter title}	Internal string for designer, you can skip this attribute

When you design a mission, try to think in terms of "actions", you would like the simmer to do, and if they can be split into different objectives for better readability or to make the mission controllable.

The actions we will translate into tasks, some of them should be mandatory but others can be optional, whether it is to allow the designer to guide the simmer with a message or to monitor behavior and act accordingly.

<task />

One liner element.
The task is a simple placeholder where we link it to a more complex logic element like trigger or a script. A task can be dependent on other tasks' success before evaluating itself.

```
<task name="LandInYPAM" title="Land in YPAM" depends_on_task=""
      base_on_trigger="trig_land_in_YPAM" base_on_script="" base_on_command=""
      base_on_external_plugin=""  eval_success_for_n_sec="3" mandatory="" force_evaluation="no"/>
```

You can remove attributes that are not in use or with default value

Attribute Name	Type	Value	Meaning
name *	str	"LandInYPAM"	Task name must be unique in an objective
title	str	{enter title}	Optional, user readable name
depends_on_task	str	"Name of other task"	Enter the name of another task we depend upon.
			You must define one of these attributes. If you define all first three, the plugin will use "base_on_trigger". The hierarchy is: "trigger" or "script" or "command" When a 3rd plugin will use our shared dataref, then "base_on_external_plugin" will supersede all others. SAR plugin should support this feature in its future build (X-Trident) but for each plugin you will have to understand how they set the status, automatically or manually by simmer. The attribute that ***overrides them all*** is "is_placeholder", this is a special case where you can flag a task as a *placeholder* and *mandatory*, that way you can manage a set of mandatory actions that can't be fulfilled at the same time. We change the status through scripts that can be called from: trigger/message/choice/script in the same flight leg. ***Example:*** A simmer needs to conduct a mandatory task in two different locations at the same objective, that is not possible without a placeholder. To overcome this you can set a "placeholder" task to success after firing a specific message.
base_on_trigger *1	str	"Name of trigger"	
base_on_script *1	str	"Name of script"	
base_on_command *1	str	"sim/lights/taxi_lights_off"	
base_on_external_plugin *1	bool	yes|**no** true|**false**	
is_placeholder *1 *2	bool	Yes|**no** true|**false**	
base_on_sling_load	str	"**cargo01**"	This is specific for SlingLoad plugin integration. You need to provide the <sling name="" type="cargo"/>" element in the same <objectives> parent element.
eval_success_for_n_sec	int	3, 5, 10, **""**	How much time to evaluate success before flagging the task as complete. Default is "immediate" (empty value), but you can modify this as you see feet.
cumulative_timer_flag	bool	Yes|**no** true|**false**	Evaluating success ***will not reset a failed*** task once timer starts to tick, Instead will pause it and resume it once condition is met again (@Daikan).
mandatory	bool	Yes|**no** true|**false**	Is this task mandatory ? Flag only the real important tasks as mandatory so the other will be supporting tasks. Try to have at least one task as

			mandatory.
force_evaluation	bool	Yes|**no** true|**false**	Repeat task evaluation even if it was flagged as "success" ? If yes, then you should take into consideration that the task can "fail" the next time and probably alter the course of the mission. In some cases it is needed. Pay attention to "dependent" and "multi" mandatory tasks in the same objective.
For external plugin support - since v3.0.304			
is_target_poi	bool	[yes|**no**]	Is this Task the target of the flight leg ? The "Task" must be based on trigger and of type rad/poly. If yes then the plugin will pick the center of the area as a shared target position for the external plugins.
target_lat	decimal	-34.342564	Latitude value
target_lon	decimal	+20.543543	Longitude value
"target_lat" and "target_lon" is a way to manually define the target position instead of letting the plugin pick the center of the area OR *You can define a task as "is_target_poi" and also define the "target_lat/lon" attributes.*			

1 pick one of the attributes, "base_on_trigger", "base_on_script", "base_on_command", "base_on_external_plugin" or "is_placeholder"

Command is any action you can bind to the keyboard or Joystick.

*2

A task that is flagged as a **placeholder** is a special case since it is like a script but without the mandatory code it needs,
In such a case, you can define a few placeholder tasks that are mandatory and then flag each one as success based on another action in the same flight leg, that way you can define a few mandatory actions that are dependent on one another but can't be fulfilled at the same time. Using the "is_placeholder" task, will allow you to flag an action as success while also keeping it mandatory, think about it as a kind of a checklist that you have to do everything without a specific order.
Use: "**fn_set_task_property**" or "**fn_set_task_property_in_objective**".

When to call "fn_set_task_property" or "fn_set_task_property_in_objective" functions ?

It is safe to call "**fn_set_task_property**" from task based rules (the task is based on trigger, or script".

It is safe to call "**fn_set_task_property_in_objective**" from task based triggers or scripts, but also from "post_script" in <messages> or "script_name_when_fired" in <option> element outcome.

Task Properties to use in functions

`fn_set_task_property` ("task_name", "property_name", "property_value") or **`fn_set_task_property_in_objective`** ("objective_name"*, "task_name"*, "property_name", "property_value")

Property name	Will Convert To	Value Example
title	string	"Task Title"
base_on_trigger	string	"Trigger name"
base_on_script	string	"Script name"
eval_success_for_n_sec	int	"5"
force_evaluation	bool	"true", "false"
enabled	bool	"true", "false"
script_conditions_met_b	bool	"true", "false"
is_complete	bool	"true", "false"

```
fn_set_task_property (mxCurrentTask, "script_conditions_met_b", "true")
fn_set_task_property ("step1_parkbrake", "script_conditions_met_b", "true")
fn_set_task_property_in_objective ("fly_to_ypam", "step1_parkbrake",
"script_conditions_met_b", "true")
```

Use the "`fn_set_task_property_in_objective()`" function when you write code in Flight leg level, and there is no Objective or Task that is being evaluated yet. You can alternatively use it instead of "`fn_set_task_property()`".

`fn_get_task_info` ("task_name") or
`fn_get_task_info_in_objective` ("objective_name", "task_name")
*Will seed **all*** the following arguments:

Seeded Variable Name	Type	Expected Values
mx_state	string	success, was_success, need_evaluation
mx_type	string	trigger, script, undefined
mx_taskActionName	string	The name of the trigger/script to execute. Example: "trig_landed", "script01.func01"
mx_is_complete	bool	true, false
mx_enabled	bool	true, false
mx_script_conditions_met_b	bool	true, false
mx_TaskHasBeenEvaluated	bool	true, false
mx_force_evaluation	bool	true, false
mx_mandatory	bool	true, false

Use the "`fn_get_task_info_in_objective()`" function when you write code in Flight leg level, and there is no Objective or Task that is being evaluated yet. You can alternatively use it instead of "`fn_get_task_info()`".

Functions to get/set Task properties

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
Task related functions	
`fn_get_task_info ("task name")` `fn_get_task_info_in_objective ("objective name", "task name")`	See above for detail explanation. ***Returns "true"*** if found. This function will fail for <link_to_trigger> element.

`fn_set_task_property ("task name", "property_name", "property_value")`
 	
`fn_set_task_property_in_objective ("objective name", "task name", "property_name", "property_value")`
	Property value is a string representing a real type (check properties table)
	
	"fn_set_task_property" function will fail for <link_to_trigger> element but "fn_set_task_property_in_objective" will work

`fn_reset_task_timer ("task name")`	When a task evaluates: "eval_success_for_n_sec" it starts an internal timer. You can decide to reset it depending on the need.
`fn_set_cargo_end_position ("task name", lat(r), lon(r))` [v3.0.304] Will create 2 global params: mxCargoPosLat and mxCargoPosLon	Sling load tasks must have an end cargo position set. If you are using an init script you have to use this function to do so. This way the plugin will be able to determine the success coordinate.

Example:

```
fn_get_task_info ("LandInYPAM")
```

For the seeded script variables, see above.

Sling Load Based Tasks

```
<task name="cargo" title="Move cargo" base_on_sling_load="test_cargo" eval_success_for_n_sec="5" >
  <test_cargo type="cargo" start_lat="-19.25342199" start_lon="146.77093350" end_lat="-19.25257197"
end_lon="146.77090880" weight_kg="150"/>
</task>
```

If you define a task that is "base_on_sling_load" you will have to provide a sub element that will hold the following attributes:

Attribute Name	Type	Value	Meaning
type *	str	"cargo"	As of v3.0.304 only "cargo" type is supported
start_lat *	double	0.0, -19.345562	Cargo latitude start position
start_lon*	double	0.0, 41.345562	Cargo longitude start position
end_lat*	double	0.0, -19.345562	Cargo latitude target position
end_lon*	double	0.0, 41.345562	Cargo longitude target position
weight_kg	double	0.0, 100.0	Cargo mass in Kg. Default is 0 which means that the weight will be taken from the SlingLoad plugin.
init_script	str	"", "init_cargo"	A script that will initialize the SlingLoad plugin. This will override the attributes above, but it is up to the designer to initialize the end location using the function: "fn_set_cargo_end_position()", while the rest of the values you can initialize directly using pre defined DataRefs (see SlingLoad documentation for a known list).
cond_script	str	"", "flc_cargo"	You can use a script to replace the plugin task evaluation code. It is up to you to evaluate and decide the success of the task and when it is flagged as success or abort mission. See: "**fn_set_cargo_end_position()**"

The Sling Load based task is solely dependent on the **forked** "SlingLoad" plugin. Almost all the datarefs the plugin uses are basically created by the HSL (Heli Sling Load) plugin, the only exception for that is that the original cargo "position" dataref "HSL/Cargo/Position" is an array that is not being updated with the cargo movement, therefore I have forked the original plugin and added the missing information using 3 new datarefs: "HSL/Cargo/pos_lat_d, HSL/Cargo/pos_lon_d, HSL/Cargo/pos_elev_mt_d". Mission-X is only using the first two, and the "HSL/Cargo/Connected" to determine if the cargo is being hooked or not.

Currently Task success is being evaluated using the following logic:

1. Is cargo connected ("HSL/Cargo/Connected")
2. Is cargo in the target position (target position is stored internally).
3. If 1&2 are true, then evaluate the timer attribute "eval_success_for_n_sec"
4. Flag task as success if all 3 conditions are met.

With this knowledge, you could write something similar using a Task based script, but that one will be more complex to manage.

If you are using scripts to evaluate cargo position, you will have to depend on: "**fn_set_cargo_end_position()**" function and the global params returned: "**mxCargoPosLat** and **mxCargoPosLon**"

<triggers>

Triggers are the most complex element in Mission-X since they are the main executors of the mission logic, meaning, we can use them to call *messages*, *scripts*, *commands* or *datarefs* based on physical location or logical ones.
For physical location triggers we check distance relative to "plane" except "camera" trigger *type*. The "camera" trigger type test distance between the trigger and the camera position (useful if you want to mimic a person walking around while having minimal interaction with it).

Command in triggers behave like "flight leg" commands, meaning, they can also have a timer format in them. The trigger inherits the "`eval_success_for_n_sec`" from the task that is based on it as an implicit condition.
It is common to layer a few triggers on the exact same location to achieve a more flexible/complex behavior.
A trigger will fire the outcome element only when all conditions are met: physical location, script and timer (any combination you defined)

Here is an example for the new trigger format:

```
<trigger name="trig_land_in_YPAM" type="poly" rearm="" post_script="" enabled="yes" >
  <conditions plane_on_ground="yes" cond_script="" />
  <outcome message_name_when_fired="LandedYPAM" message_name_when_left=""
           script_name_when_fired="" script_name_when_left="" post_script=""
           commands_to_exec_when_fired="" commands_to_exec_when_left=""
           dataref_to_modify_when_fired="" dataref_to_modify_when_left=""  />
  <loc_and_elev_data>
    <point lat="-18.7510338" long="146.578293"/>
    <point lat="-18.7591095" long="146.584763"/>
    <point lat="-18.7591953" long="146.584625"/>
    <point lat="-18.7511101" long="146.578003"/>

    <!-- elevation volume is not needed since we defined the condition "on_ground"
         but it is here as an example -->
    <elevation_volume  elev_lower_upper_ft="100|4000"  />

  </loc_and_elev_data>
</trigger>
```

We have <condition> and <outcome>. The "`loc_and_elev_data`" assist in defining the physical location of the trigger.

You can remove elements and attributes that are not in use or with empty values:

```
<trigger name="trig_land_in_YPAM" type="poly" >
  <conditions plane_on_ground="yes" />
  <outcome message_name_when_fired="LandedYPAM" />
  <loc_and_elev_data>
    <point lat="-18.7510338" long="146.578293"/>
    <point lat="-18.7591095" long="146.584763"/>
    <point lat="-18.7591953" long="146.584625"/>
    <point lat="-18.7511101" long="146.578003"/>
  </loc_and_elev_data>
</trigger>
```

It is the same elements with the same rules but without the attributes we do not need or their default values is good enough

*In order for a Trigger to be triggered - **fired**, all **defined conditions must be met**:*

1. *Plane/Camera is in the physical area (`loc_and_elev_data` - if defined). "Camera" based triggers will ignore elevation tests.*
2. *Script must flag the trigger as success too (if `cond_script` was defined).*
3. `"eval_success_for_n_sec"` *is inherited from the task the trigger is linked to (if it was defined).*

A Trigger will fire by calling: the <outcome> attributes that were defined (if any).

None of them are mandatory.

You can use triggers to send simple *messages*, or to make more complex decisions based on *DataRef and command* values.

<trigger>

The trigger assists in driving the mission forward. We can use it to call a message or scripts to handle simple or complex logic.
"<trigger>" attributes:

Attribute Name	Type	Value	Meaning
name *	str	"Trigger Name"	Unique trigger name
type *	str	"rad","poly","slope", "script", "camera"	Each trigger must have a type. The type allows the plugin to correctly parse and validate the trigger.
rearm	bool	"true|false" "yes|no"	Trigger will be rearm once left. Good for repeating messages based on area.
post_script	str	{script name}	A script you want to call after the plugin finishes handling the trigger. Allow you to change the status of current or other triggers.
enabled	bool	"true|false" "yes|no"	Default true.

"post_script" - will always be called just after we test the trigger. It is also called if the trigger is disabled, so you can re-enable it (for example).

"Camera" trigger type will allow you to evaluate location based on "camera" position and not the "plane".
The trigger itself behaves like a "radius" based trigger.
The "camera" trigger ignores elevation tests.
You can interact with inventory areas even in camera mode.

<conditions />

One liner element.
The condition element allows you to define simple or complex logic to assist in evaluating if a trigger should fire.
"<conditions>" attributes:

Attribute Name	Type	Value	Meaning
plane_on_ground	str as bool	"" Empty = ignore "true|false" "yes|no"	Does the plane need to be on the ground ? Default is "" ***EMPTY*** = ***ignore***. When the value is empty, the plugin just checks the physical location (is plane in trigger physical area). This is the simplest form of a trigger..
cond_script	str	{name of script} or {script}.{func name}	Call a script to test and decide if the trigger should be triggered. The script will update the trigger property: "script_conditions_met_b" with "true or false" values. *Read the topic "embedded script" to learn how to send "predefined parameters" if you need.*
Inherited from a linked task (internal, no need to define it):			
eval_success_for_n_sec		The <task> can define the timer success rule, if so then the trigger inherits it. This means that the timer is added to the other conditions you defined in the trigger	

plane_on_ground optional attribute. If not defined then the plugin will only test if the plane is in the physical area of the trigger.

<outcome />

One liner element.
In order not to break the trigger element, the outcome will hold attributes that will define what should be done once a trigger area is fired or left but for X-Plane commands, datarefs, messages and scripts.

Attribute Name	Type	Value	Meaning
Messages			
message_name_when_fired	str	{message name}	Call message in "message_templates" element when **all conditions are met**
message_name_when_left	str	{message name}	Call message in "message_templates" element
message_name_when_entering_physical_area	str	{message name} *This attribute could be very useful for one time messages based only on physical location.*	***One time message.*** Call a message when the plane is in the physical area + elevation. Will fire even if ***not all conditions are met***, and will fire only once.
Scripts - replace <event_scripts> *Read the topic "embedded script" to learn how to send "predefined parameters" if you need.*			
script_name_when_fired	str	{Script name}	Call script or scriptlet from <embedded_scripts> element.
script_name_when_left	str	{script name}	Call script or scriptlet from <embedded_scripts> element.
commands			
commands_to_exec_when_fired	str	sim/lights/taxi_lights_on, sim/lights/nav_lights_on	Execute command or set of commands delimited by comma ","
commands_to_exec_when_left	str	sim/lights/taxi_lights_off, sim/lights/nav_lights_on	Execute command or set of commands delimited by comma ","
Datarefs - breaking change since v3.304.12 - attribute string format change			
dataref_to_modify_when_fired	str	sim/cockpit/electrical/taxi_light_on=1|sim/cockpit/electrical/nav_lights_on=0|sim/cockpit2/switches/landing_lights_switch[1]=0.5|sim/flightmodel2/lights/landing_lights_brightness_ratio=0.5	Modify dataref or dataref array. Supports integer/float/double/array int and array floats. Has special formatting: {dref_name1}={value}| {dref_name2}={value}|....
dataref_to_modify_when_left	str	sim/cockpit/electrical/taxi_light_on=0|sim/cockpit/electrical/nav_lights_on=1|sim/cockpit2/switches/landing_lights_switch[1]=0|sim/flightmodel2/lights/landing_lights_brightness_ratio=0	Modify dataref or dataref array. Supports integer/float/double/array int and array floats. Has special formatting: {dref_name1}={value}| {dref_name2}={value}|....
Set other trigger state - v25.02.1 - TBT - to be test ;-)			
set_other_triggers_as_success	str	"trig_1, trig_2, ..."	**Used Only when trigger is fired.**

			List of trigger names you would like to set their status as “success”. It won’t fire any event or script and it should not fire in the next flight callback. Internally we are setting the state to: “*never_trigger_again*”. Post scripts will be called though.
Set other tasks state - v25.02.1			
`set_other_tasks_as_success`	str	“task1, task2…”	**Used Only when trigger is fired.** List of task names that you would like to force flag as success
`reset_other_tasks_state`	str	“task1, task2…”	List of task names that you would like to force flag them as ready to be evaluated.

- *These outcomes will make it easier to do some low level manipulation without writing scripts. For more complex logic you will have to use external scripts.*
- *You can refer to custom commands or datarefs too.*
- *Use comma delimited to modify multi commands*
- *You can’t set a specific dataref array value, with the new implementation.*
  *The syntax should be: "**sim/cockpit2/engine/indicators/prop_speed_rpm**=690.0,691.0" for 2 engines.*

`message_name_when_entering_physical_area` - One time message
This new attribute was designed with specific cases in mind, where you want to send a “one time message” immediately on entering the physical area of a trigger.
This can help the simmer, they can understand what are the steps they need to conduct in order to successfully finish the "task" at hand.

Examples for useful messages:
"Stop the plane and push the park break."
"Turn off the engine and pull the park break."

!!! Do not use this attribute as part of a successful trigger. Use the other message options !!!

<set_datarefs /> and <set_datarefs_on_exit />

(v3.304.12)

```
<set_datarefs>

</set_datarefs>
```

The <set_dataref> and <set_datarefs_on_exit> are new implementations of the "`dataref_to_modify_when_fired`" and "`dataref_to_modify_when_left`" attributes. They are a sub element of the <outcome> element. You can use both or none of them.
The new sub-elements allow you to manually define datarefs values using a simple mechanism of "*key=value*" where the key represents the full dataref path and the value represents the dataref scalar or array value (arrays separated with comma ",").

You can read all about this in the "<weather>" "flight leg" element since they share the same properties and behavior, the difference is in the element *name* and *when* it will be called.
For the trigger element, we call the <set_datarefs> when trigger is fired and when we leave the trigger area, on-exit, we will call the <set_dataref_on_exit>.

<loc_and_elev_data>

If you define a trigger based on physical location: "rad", "poly" ~~or "slope"~~, then you have to also define the boundaries of that area. The boundaries are a set of "coordination" and "elevation" information.

Examples:

Polygonal area

```
<loc_and_elev_data>
  <point lat="-18.7510338" long="146.578293"/>
  <point lat="-18.7591095" long="146.584763"/>
  <point lat="-18.7591953" long="146.584625"/>
  <point lat="-18.7511101" long="146.578003"/>
</loc_and_elev_data>
```

Radius area

```
<loc_and_elev_data>
  <radius length_mt="20"/>
  <point lat="-19.2536716" long="146.770935"/>
</loc_and_elev_data>
```

Rectangle Polygonal area (v3.0.254.5)

```
<loc_and_elev_data>
  <rectangle dimensions="1500|60" heading_psi="0.12" first_point_is_center="false"/>
  <point lat="47.43806230" long="-122.31162020"/>
</loc_and_elev_data>
```

Will calculate the "rectangle" area based on the values you define

Here are all options in location and elevation. These are just examples:

```
<loc_and_elev_data>
  <radius length_mt="" />
  <point lat="" long="" /> <!-- One point for radius based triggers -->
  <point lat="" long="" /> <!-- Three or more points for poly based triggers -->
  <point lat="" long="" /> <!-- Three or more points for poly based triggers -->
  <reference_point lat="" long=""/> <!-- optionally, manually define center of area -->
  <elevation_volume  elev_lower_upper_ft="100" /> <!-- NOT VALID -->
  <elevation_volume  elev_lower_upper_ft=""/> <!-- valid, plugin will ignore elevation
                                                  test -->
  <elevation_volume  elev_lower_upper_ft="-100|-4000"/> <!-- plugin will fix -->
  <elevation_volume  elev_lower_upper_ft="100|4000"  /> <!-- valid -->
  <elevation_volume  elev_lower_upper_ft="--100"  /> <!-- valid, below 100ft -->
  <elevation_volume  elev_lower_upper_ft="++100"  /> <!-- valid, above 100ft -->
```

```
  <elevation_volume  elev_lower_upper_ft="+++100"  /> <!-- valid, 100ft above ground level and on -->
  <elevation_volume  elev_lower_upper_ft="---100"  /> <!-- valid, 100ft above ground level and down -->
</loc_and_elev_data>
```

When defining elevation, you should define min/max values since we are creating a volume where planes need to fly into. See next page for more explanation.

<radius />

One liner element. Relevant only if we define trigger type "rad".

"Radius" attributes

Attribute Name	Type	Value	Meaning
length_mt	int	50, 1000..N	Length of radius in **meters**.

<rectangle />

One liner element. Relevant only if we define trigger type "poly" and we want the plugin to calculate the rectangular area.

"rectangle" attributes

Attribute Name	Type	Value	Meaning
dimensions	str	"1500|60"	Dimensions in ***meters***. The pipe("|") divides between the 2 **mandatory** values
heading_psi	decimal	[0..360], 12.5, 25.45	Heading to the top left point of the rectangle area. Use true north heading.
first_point_is_center [v3.0.301]	bool	[true,**false**]	If false, then the first point is "bottomLeft" of the rectangle, but if it is "true" then the first point is the center of the rectangle and two vectors will be created in length dimension/2

The <point> in "poly + rectangle" trigger represents the bottom left of the area. The first dimension value will calculate the top Left and the second value will calculate the bottom Right and then the top Right of the trigger area.
The vector bearing will be calculated relative to the "heading_psi".

<point />

One liner element. Holds coordination information for our trigger area.

"point" attributes

Attribute Name	Type	Value	Meaning
lat	decimal	-34.342564	Latitude value
long	decimal	+20.543543	Longitude value

```
<point lat="" long="" />
```

<**reference_point** /> - optional

One liner element.
This is an optional element and should be used for "poly" or "slope" trigger types. It is relevant if you define more than 2 points and you prefer to provide the center of area manually, instead of letting the plugin calculate by itself. The "reference_point" element holds exactly one point info.

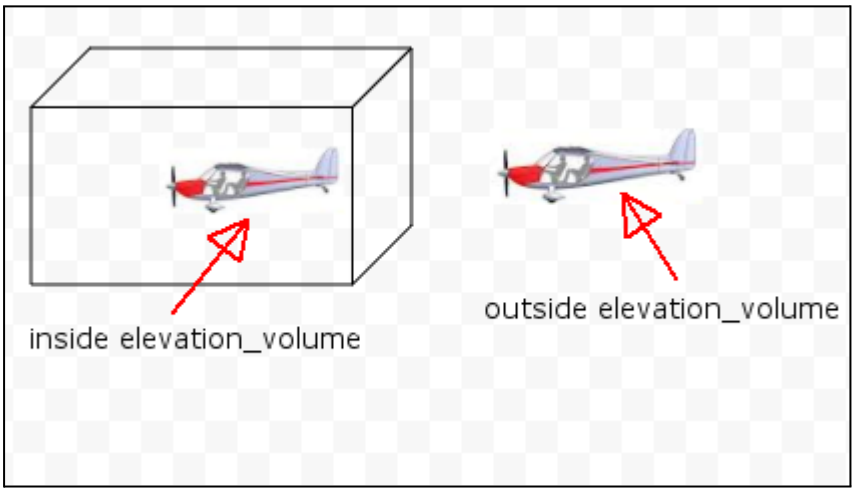
"reference_point" attributes

Attribute Name	Type	Value	Meaning
lat	decimal	-34.342564	Center of latitude value
long	decimal	+20.543543	Center of Longitude value

```
<reference_point lat="" long="" />
```

<elevation_volume>

One liner element.
In Mission-X v3.0.214 and up if you just want to check if the plane is **airborne**, you can just set plane_on_ground="no" or plane_on_ground="" **(leave empty)** and the plugin will be fired once plane enters the physical area.
If you want to catch the plane in a certain elevation volume, then you should define the "elevation_volume" element.



The elevation volume should have a lower/upper values so plane can "pass through" it and thus trigger the event.
For Mission-X v3, there is a simple syntax to describe volume in one attribute. The idea was to make it simpler and shorter.

Understanding the signs:

Assume X=100ft and Terrain elevation=50ft

symbols:

"|" = separate between low and high values.

"++"= from elevation X and above. (X=100ft).

"--"= from elevation X and below. (X=100ft).

"---"= volume = X feet relative to terrain elevation and below.(X=150ft)

"+++"= volume = X feet relative to terrain elevation and above.(X=150ft)

```
<elevation_volume  elev_lower_upper_ft=""/> <!-- valid: plugin ignores elevation test -->
<elevation_volume  elev_lower_upper_ft="100|4000" /> <!-- valid -->
<elevation_volume  elev_lower_upper_ft="--100"  /> <!-- valid, below 100ft -->
<elevation_volume  elev_lower_upper_ft="++100"  /> <!-- valid, above 100ft -->
<elevation_volume  elev_lower_upper_ft="---100" /> <!-- valid, 100ft above terrain and down -->
<elevation_volume  elev_lower_upper_ft="+++100" /> <!-- valid, 100ft above terrain and up -->
<elevation_volume  elev_lower_upper_ft="+100"  /> <!-- not valid -->
```

The "+++/---" add the terrain elevation into the elevation while "++/--" do not. Used in generating random missions.

Interpretation:

elev_lower_upper_ft="100|4000"	Volume is between 100 and 4000ft
elev_lower_upper_ft="--100"	Volume area includes all volume below 100 ft
elev_lower_upper_ft="++100"	Volume area includes all volume above 100 ft
elev_lower_upper_ft="---100"	100 ft above terrain and lower.
elev_lower_upper_ft="+++100"	100 ft above terrain and higher

"elevation_volume" attribute

Attribute Name	Type	Value	Meaning
elev_lower_upper_ft	str - complex	See examples above	Holds volume information. You should define lower and upper boundaries. If not set, then the elevation test is ignored and only the "inside/outside" area test is being done.

```
<reference_point lat="" long="" />
```

Important

If you define a trigger condition "plane_on_ground="true"" then it will ignore "elevation_volume" settings.

Troubleshoot

Trigger does not fire any event even though the plane is in the area.
Check the "elevation_volume" definition. It might not have lower/upper elevation values defined *correctly*.

"plane_on_ground" and "elevation_volume" matrix

plane_on_ground	elevation_volume	Outcome
"" - Empty (not set)	Not defined	Plugin tests only physical location
"yes" or "true"	Not defined / defined	Plugin ignores elevation definitions. It only tests if a plane is on the ground.
"no"	Not defined	Plugin tests only physical location.
"no"	Defined	Plugin tests if plane in elevation volume

This matrix should assist when and how to set your trigger. If you just want the event to occur, you can leave the "plane_on_ground" empty.

Triggers and scripts

Trigger Properties

fn_set_trigger_property ("trigger_name", "property_name", "property_value")

Property name	String value to	Value Example
elev_lower_upper_ft	string	Trigger elevation volume
message_name_when_fired	string	
message_name_when_left	string	
post_script	string	"scriptfile.functionName"
all_conditions_met_b	bool	"true", "false"
enabled	bool	"true", "false"
script_conditions_met_b	bool	"true", "false"

fn_get_trigger_info("trigger_name") -
*Will seed **all*** the following arguments:

Seeded Variable Name	Type	Expected Values
mx_state	string	entered, inside_trigger_zone, left, wait_to_be_triggered_again, never_trigger_again, never_triggered
mx_trig_elev_type	string	above, below, min_max, on_ground
mx_elev_lower_upper_ft	string	[--N|++N] "1000| 3400" "--2500" below 2500ft
mx_message_name_when_fired	string	"Some message name"
mx_message_name_when_left	string	"Some message name"

mx_post_script	string	Script name to call after trigger finish evaluation
mx_all_conditions_met_b	bool	true, false Trigger might have few test conditions: "On_ground", "cond_script" etc...
mx_script_conditions_met_b	bool	true, false Only script should alter this flag, so we know cond_script was successful or not.
mx_enabled	bool	true, false
mx_is_linked	bool	true, false
mx_plane_in_physical_area_b [v3.304.14]	bool	true, false
mx_plane_in_elev_volume_b [v3.304.14]	bool	true, false

Functions to get/set Trigger properties

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
Trigger related functions	
fn_get_trigger_info ("trigger name")	See above for detail explanation.
fn_set_trigger_property ("trigger name", "properrty_ name", "property value") fn_set_trigger_property (mxCurrentTrigger, "properrty_ name", "property value")	Property value is a string representing a real type. Check table above for more information.
fn_reset_trigger_timer	When a trigger evaluates: "eval_success_for_n_sec" it starts an internal timer. You can decide to reset it depending on the need.

Example:

```
' disable trigger so it only run once.
if not fn_set_trigger_property ("trig_start_mxpad01", "enabled", "false") then
  print mxError;
endif
```

Trigger summary

To define a trigger, you need to decide what you want to achieve by it:

- Does the trigger test the success of a task ?
- Do you want to test something while the plane is on ground or airborne ?

- Do you need to read planes DataRefs and fire the trigger according to their values (park break or speed etc) ?
- Does the trigger act like an instructor or is it a simple messaging means.
- You want the plane to be airborne but do you care about elevation tests?

Remember: We fire a trigger by implementing either "`script_name_when_fired`" or "`message_name_when_fired`" or by implementing both.

Remember: We also fire a trigger when we leave it, by implementing either "`script_name_when_left`" or "`message_name_when_left`" or by implementing both.

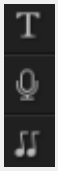
If you do not want to use scripts, then you will have to settle with what triggers allow you out of the box, but that will greatly limit your mission designs options.

<message_templates>

There are few ways to define messages with different complexities:

1. Easiest: predefined messages that we will call from triggers or scripts.
2. Script modified message, it can then be called from script or by a trigger.
3. Create a message dynamically from a script/scriptlet and call it.
4. Send a one time message from script.
5. You can create a dialog by calling a message from another message (see below).
6. [new]Since v3.306.1 you can have a "story mode" type message.

A message template is combined from 2 parts, the message header and body which is actually represented by the sub element - mixture. A mixture can be one of the following types: "*text*", "*comm*" (communication) and "*back*" (background).
Understanding the attributes and dependencies between the different mixtures, will allow for some flexibility in sound management.
Here is an example of a message:



```
<message name="LandedKOAJ" mode="" add_minutes="" timelapse_to_local_hours="" set_day_hours=""
         post_script="" next_msg="" >
  <mix track_type="text" mute_xplane_narrator="" hide_text=""
       override_seconds_to_display_text=""  override_seconds_calc_per_line="" >
       <![CDATA[some message.]]></mix>
  <mix track_type="comm" sound_file="file1.mp3" sound_vol="30"
       override_seconds_to_play="8" />
  <mix track_type="back" sound_file="file2.mp3" sound_vol="20"
       override_seconds_to_play="" />
</message>
```

You can remove elements and attributes that are not in use or with default values:

```
<message name="LandedKOAJ" >
  <mix track_type="text" ><![CDATA[some message.]]></mix>
  <mix track_type="comm" sound_file="file1.mp3" sound_vol="30" override_seconds_to_play="8" />
  <mix track_type="back" sound_file="file2.mp3" sound_vol="20"/>
</message>
```

It is the same elements with the same rules but without the attributes we do not need or their default values are good enough.

Dialogs

The <message> element has new extensions that will allow you to create a dialog like conversation, manipulate time and even inject metar files. The "*add_minutes*" and "*timelapse_to_local_hours*" attributes allow you to manipulate time until 24 hours from the current time. This will occur at the end of the broadcast message.
The "*next_msg*" attribute allows you to call another message to continue the current one and the "*inject_metar_file*" will inject a custom metar file you created or want to use.
For better message precision use the "`override_seconds_to_play`" attribute.

Caveats: The plugin's message manager won't override broadcasted messages. It will wait for the message timer to be completed and only then it will broadcast the next message in the queue even though it might already be ready.
This also means that triggers might be fired but since another message is broadcasting we won't hear immediately the trigger's message.

The background mixture sound file won't run on the same message timer as the *text* and *communication* mixtures. That will lead to a shorter message queue and better message response.

Another way to manage messages timing is to use the:
`override_seconds_to_display_text` or ***`override_seconds_calc_per_line`***
The "***`override_seconds_to_display_text`***" is probably the most predictable one, since you define the number of seconds the message should be broadcasted before the "message manager" can move on to the next one.

Other ways is to use: `fn_end_current_message`, `fn_end_current_message_and_background` and `fn_abort_current_message` to manage programmatically playing messages.

<message>

"message" attributes

Attribute Name	Type	Value	Meaning
`name *`	str	`{some message name}`	Unique message name
`next_msg`	str	`"{message name}"`	At the end of the current broadcast message the plugin will call a message with the name: "{message name}. This allows you to easily create conversation-like scenarios and make sure the content of the message will be displayed correctly in the mxpad window.
Story Mode v3.306.1			
`mode`	str	`""`|`"story"`	Empty means default message behavior. "story" will force open the main plugin window and will display the message/s in it until it reaches the end of the message text.
`ignore_punctuations_b`	bool	`"[true|false]"`	During text reveal, the plugin will delay in certain punctuation to emphasize the sentence. Example: ",.!?"
Post action attributes			
`add_minutes`	int	`0..240`	At the end of the message add minutes to the time. You can't add more than 4 hours or the plugin will ignore the attribute.
`timelapse_to_local_hours`	str	`"h24:mm:cycles"`	Provide the local hour you wish to transition the clock, the minutes and how many cycles to reach it (max 24). The plugin will transition the hours as timelapse in Nnnn seconds every cycle. This will mimic time being passed.
`set_day_hours`	str	`"123:23:12"` `"-1:23:12"`	Set day to the 123th day of the year and to 23:12 local time. If you just want to ignore part of the timestamp you can write: "-1" in any part. Example: "-1:23:12" will set the time to 23:12 and will skip the day.

inject_metar_file Xp11 only	str	"{metar file name}"	**XP11 only:** Place the metar file at your mission root folder or the custom metar file subfolder (see *global_settings*).
open_choice	str	"Choice name"	Open choice window at the end of the text message.
post_script	str	Script name or scriptlet name	Will execute a script code at the end of the message broadcasting and from v3.306.3 before processing the other attributes, like "next_msg". That way you could branch and tweak the mission according to user decisions. It will also seed the "*mxCurrentLeg*", before calling the script
Read the topic "embedded script" to learn how to send "predefined parameters" if you need.			
fade_bg_channel (v3.306.3)	str	"name*,[seconds]" "start_message,8" "%self%,8"	Provide a string and a number separated with a comma. The string is the *name of the "message"* where you started the "background <mix>, and the *number* represents the *seconds* to fade out the channel. Can use "***%self%***" if the background started in the current message. The "name" is mandatory, but the "seconds" is optional. You can have the same effect in scripts using "**fn_fade_out_bg_channel**"
Edge case - if message is last one in the flight leg (not the whole mission)			
fallthrough_b (v3.306.1b)	bool	[true|**false**]	Default "false". If we want a quick transition to the next leg, and this message is the last one, we should set this attribute to "true" and the plugin won't wait for the message timer to complete until it will transition to the next leg. The transition will be immediate and the message will broadcast itself fully, so we don't lose anything.

*There are some limitations in calling "**post_script**" from a "message". The big one is that it is aware only of "mxCurrentLeg" but not "current task/trigger" so you have to write explicit code to handle such elements from your script.*

Tip:
To assess the time it takes the message to be displayed from start to end, use the "**debug**" binary of the plugin and let the "story message" auto run on the screen. When the message will end, you will see in the "Log.txt" file the time it took the message to run and then you will have a good **estimation** of how much time a background music file should run.

<mix /> - "text" track type

One liner element. "mix" attributes for "text" only:

Attribute Name	Type	Value	Meaning
track_type *	str	"text"	This table explain only text based mixture
enabled	bool	"**true**|false" "**yes**|no"	
Attribute related to default message type			
mute_xplane_narrator	bool	"true|**false**" "yes|**no**"	Instead of using X-Plane synthesizer, display message in text box
hide_text	bool	"true|**false**" "yes|**no**"	Usually combined with mute narrator. Do not display text, play a sound file instead.
override_seconds_calc_per_line	float	**0**, 5, 8	Replace plugins default "seconds" per line calculation, with this value.
override_seconds_to_display_text	float	**0**, 5, 8	Decide the time to display the text message before allowing another message to be broadcast.
Attribute related to story mode			
characters v3.306.1	str	"n|narrator|#color,p|pilot|#color" Example: n|Narrator|#ffff00	The "characters" attribute defines the "people" that interact in the message text. Each character should have: 1. Alias to the character name. 2. Full Name to display. 3. Color, in hex format (#ffff00). More in Appendix B.
Attributes control how to display in MX-PAD			
label	str	"Jane", "Tower", "Inst."	Labels represent the source of message. Should be less than 6 characters. ~~If Label is empty, message text will fill all row~~
label_placement	str	"**L**|R" ~~"**LEFT**|RIGHT"~~	**"L"**: is the default value. Where to place the label in the mx-pad row. Left, right to the text. L = Left, R= Right
label_color	str	"**white**", **""**, "blue"	[**white**|yellow|green|red|orange|purple|blue]. Empty value, means: "white"
Message text - CDATA special xml element			
<![CDATA[]]>*	str	Message text	Message free text

Only the "text type track" can have the MX-PAD related attributes.

<mix /> - "sound file" track type

One liner element. "mix" attributes for sound files:

Attribute Name	Type	Value	Meaning
track_type *	str	"comm"|"back"	"comm": use a file to playback text message "back": sound file is used as background music
sound_file *	str	{filename}.mp3	Enter sound file. mp3 or wav format.
sound_vol	int	0..100	Default 0
override_seconds_to_play[1]	int	0..N	Seconds to play the sound file
Specific Background <mix>			
track_instructions v3.304.6	str	"sec|cmd|[vol]|[trans_steps],..." "10|-|1|6.0|s"	This attribute is **only** for "back" mixture type. It allows you to define sets of volume actions on the playing sound file. "sec": define when to start the command. "cmd": is either "+/-/!/s/r". +/- increase/decrease volume respectively. "!/s" means to stop the playing file. "vol": What is the new volume value (0..100). "trans_sec": optional, define gradual increase/decrease of volume until we reach the target volume, like fade in and out behavior. You can define one or more sets, divided by the comma **","**.

[1] *If the track is communication type ("comm") and "override_seconds_to_play" equal or smaller than "0", then we use "text" mixture time.*

track_instructions attribute

The main enhancements of the "**back**ground" mixture are:

- The ability to control the sound volume during its play through.
- Pick between "xp" and "os" timer. Default is "xp", meaning the pause state affects the timer.

Instruction format: "{when}|{command}|[{new vol} or {repeat}]|[{transition steps}]

Sub command	Meaning
when	When to start applying the command relative to the start of the streamed file (The value is in seconds).
command	Which command to execute (see their meaning below).
[new volume] or [repeat no]	Optional, what is the new volume
[transition steps]	Optional, How many cycles until we reach that volume level. Think of this similar to fade in/fade out

Command signs table

command	What it does
+	Increase volume
-	Decrease volume
{s|S|!}	Stop playing (also discards the rest of the commands that follow it).
r	Repeat stream from start - repeat only once.
{l|L}	Loop. Endless loop from the "seconds" defined. You can control how many times to loop.

Track instructions examples:

Instruction	What it does
"20|+|35|5"	At the ***20***th second, increase volume to "35" in "5" transition steps.
"90|-|0|20"	At the "90th second, decrease volume to "0" in "20" transition steps.
"5|r|25"	Repeat after 5 seconds and immediately change the volume to "25", there is no transition time ("r" does not support transition time).
"5|r"	Repeat after 5 seconds (no volume change), maybe you want to keep the last volume active.
"120|L" "60|L|2"	Loop after 120 seconds indefinitely. Loop after 60 seconds for two times. **Be careful. If you don't define a repeat number the instruction will repeat endlessly.**
"0|!" or "110|s" **other variations:** "0|S" or "110|!" "{time in sec}|{s|S|!}"	Stop playing and release the background sound channel. The first example means: "*stop immediately the stream*". The second example means: "*At the 110th second of the stream, stop it*". In both cases the background channel will be released.

*If no transition steps are given, then the volume change is **immediate**.*

Here is a set of instructions:
track_instructions="10|+|35,90|-|20|15,200|-|0|10,0|!"
Parse explanation:

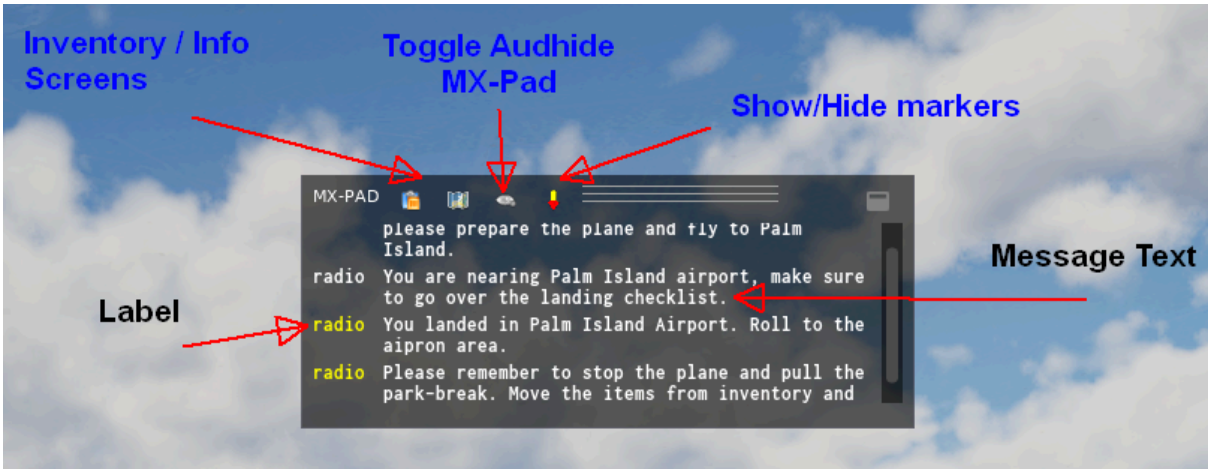
1. At the 10th second from the start of the stream, immediately change sound volume to "35" (no fade in effect).
2. At the 90th second from the start of the stream, gradually decrease the volume to "20" in "15" steps (decrease volume).
3. At the 200th second from the start of the stream, gradually decrease the volume to "0" in "10" steps (fade out effect).
4. Stop the stream (!).
   The "0" means I don't care where we are on the stream, I just want to stop it.

The benefit of using "loop" command:
"120|r,120|r,120|r" equals "120|L|3"

When counting repeat times, be aware that the "instruction" command "r" or "l" will be applied after we played a chunk of the track. So when we repeat, it is equal to "1 + Repeat times". So if we define "*three*" loops, it is equal to "4" times the same track.

Here is an example of a message that is being displayed in the mx-pad:

```
<message name="start_message" >
  <mix track_type="text" label="radio" label_color="yellow" ><![CDATA[some message.]]></mix>
</message>
```



The following message will be displayed in the mx-pad, the label "radio" will be displayed on the left side, while the color will be yellow. The text will be narrated by X-Plane or OS synthesizer and without any sound channel.
This is the easiest way to create a formatted mx-pad message without using sound channels or script commands.

In most cases a designer will probably use mixture type "text" since it is the simplest configuration and it uses OS synthesizer, but if you would like to make a more immersive message then you should use "comm" or "back" sound mixtures with the message.

Story Mode

A message can have a "story" mode too. This is a new feature in v3.306.1.
The idea is borrowed from applications like "renpy" where you build a heavily narrative story with images to better transfer the story line.
In Mission-X you can have something similar, although on a smaller scale, where the flying takes place between the conversations, which should add to the overall experience.

Here is an example for a <message> based story mode.
It basically allows you to create a conversation with images to share with the simmer.

```
<message name="starting_message" mode="story">
<mix track_type="text" characters="n|Narrator|#ffff00,p|Pilot|#ff0000,r|Radio|#00ffff">
  <![CDATA[[i] bg "bg_image"
   p "Good morning."
   [i] l "leftImage"
   [i] c "centerImage"
   [i] r "rightImage"
   r "Good morning pilot. Goods were loaded into the plane, please prepare the plane and
fly to Palm Island..."
   [p] 2
   [i] c ""  <= clear center image
   [i] l ""
   [i] lm "left_med_img"
   p "Thanks, will do"
   [i] rm "right_med_img"
   n "Make sure to go over the flight plan and checklist before you depart."
   [h] <= hide briefer window
  ]]>
</mix>
<mix track_type="back" sound_file=""  sound_vol="" />
</message>
```

A message based story will reveal the *text* line by line and not as one big message.
A message based story ignores <comm> mixture, the plugin will ignore it.
Images will be cleaned at the end of the message. YOu will have to load them again in the next message.

In the example above, you can see that in one message text, you basically create a conversation while having special actions to tell the plugin what to do.

Supported Actions

<table>
<tr><th>Action</th><th>Format</th><th>Explanation</th></tr>
<tr><td>[i]</td><td>{location} {file}

<code>[i] L "left"</code></td><td>Load image file. The extension is not mandatory and the plugin will search for "file", "file.png" and "file.jpg" file names (in that order)

The {location} defines were to display the image in the plugin screen. Locations can be:<table><tr><td>bg</td><td>790x300</td><td>Background image</td></tr><tr><td>l,r,c</td><td>240x300</td><td>Left, right, center - Small Image</td></tr><tr><td>lm, rm, cm</td><td>360x300</td><td>Left, right, center - Medium image</td></tr></table>
An empty file string will remove the image on the screen.</td></tr>
<tr><td>[p]</td><td>{time in seconds}

<code>[p] 5</code></td><td>How much time to add to the default pause at the end of the message. Default pause for each line is 3 seconds, so the time will be added to it.</td></tr>
<tr><td>[m]
{alias}</td><td>{alias} "text"
[m]{alias} "text"

>{alias} "text"
<{alias} "text"</td><td>The [m] action is optional and it represents shared message text with the "mx-pad".
For a more streamlined approach any line that does not start with an action brackets "[ ] " will be dealt as a "message text" action.
This is an implicit action. But, if you want to share the message text with the "mx-pad" window, you will have to add the action: "[m]" at the beginning of the line.
There must be no spaces between the punctuation, the action and the character alias.</td></tr>
<tr><td></td><td colspan="2">Special characters, at the beginning of the line, that will work only with text messages and affect the punctuation rule.
There must be no spaces between the punctuation, the action and the character alias.</td></tr>
<tr><td></td><td>></td><td>Ignore punctuation rules.
Override "ignore_punctuations_b" for the current line only.</td></tr>
<tr><td></td><td><</td><td>Force punctuation rules.
Override "ignore_punctuations_b" for the current line only.</td></tr>
<tr><td></td><td>%pilot%</td><td>Pilot name.
Since v3.306.1, the simmer can add their pilot nick name in the "setup" screen. This can be dynamically replaced in the message where it finds "%pilot% in the text line.</td></tr>
<tr><td>[h]</td><td>[h]</td><td>Hide screen. Must be the last line of the message.</td></tr>
</table>

The message text is a special case since you don't have to use an action to display the text line, it is the "default" action.
The [m] action is optional and can be used to share the message line with the mx-pad screen. Useful when you want the simmer to see key lines after the conversation ends.

There must be no spaces between the punctuation, the action and the character alias.

Example to a story based screen, displays background image and three small images:

The medium sized image will overlap each other, unless only left and right images will be displayed at the same time.

Functions to handle messages

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
Message related functions	
fn_get_message_info ("message name")	Seed basic message properties into the script. See table below for supported list.
fn_set_message_property ("message name", "property_name", "property value")	Property value is a string representing a real type. Check the table below for more information.
fn_set_message_label_properties ("message name*"(s), "label*"(s), "label placement*" (s), "color" (s))	**All arguments should be filled up.** Function sets message label properties related to the MXPAD presentation. You should fill in the necessary message characteristics so it will look right. Attributes: **Message name***: unique name from XML file or dynamically created message. **Label**: "short string" describing who is sending a message: "pilot", "tower" etc… Default: empty string. **Label placement**: "**left**|right" **Color**: [**white**|yellow|red|green|blue]
fn_send_comm_message("message name"*, "track name", one_time(b)) Send message name from mission XML file to broadcast (mandatory). *fn_send_comm_message ("hello_msg")* *This message will repeat if the plugin will continue to execute this command.* *fn_send_comm_message ("hello_msg", "hay1", **true**)* If you want just **1** time message, then you need to provide a tracking name and send ***true*** as one_time value.	Send a pre-defined or newly created message, using its unique name. "**Message name**" is mandatory. * [optional] "Track Name": if we track a message, and we send text with the same "track name", it will replace the original message with the new one. *[optional] "one_time" **boolean** value. Default is ***false***. [true|**false**]
fn_send_text_message ("text message"*, "track name", one_time(b)) To send a text message every time: *fn_send_text_message ("Hello World")* To send a message only one time, we need to track it: *fn_send_text_message ("Hello World", "hay2", **true**)* Pay attention that the last attribute is boolean not a string. Also the track name is mandatory in this case.	Send a text as a "communication" message. * Text messages may not be empty. * [optional] "Track Name": if we track a message, and we send text with the same "track name", it will replace the original message with the new one. *[optional] "one_time" **boolean** value. Default is ***false***. You must handle repeated calls or messages will be sent over and over. Replaces: fn_send_onetime_text_message

Functions that should be called from the <message> itself from a post_script or while they are being broadcasted. You need to time them correctly.	
fn_end_current_message ()	Set the message timer to end. Does not stop the background sound though.
fn_abort_current_message ()	Stop the current message and its background sound too (if it plays in the background). Useful when you have a long running background channel and you want to make sure you stop it when the message ends. This will also skip post message steps too.
fn_abort_bg_channel ("name") [v3.306.1b]	End the background channel that is being played. You should pass the message name that started the background channel. Relevant only if the background is being played and will be ignored if it is not in the playing queue.
fn_abort_all_channels () [v3.306.1c]	Stop all background and communication channels in the memory pool. Good to make sure that before we add messages with sound we make sure that nothing is running in the background.
fn_fade_out_bg_channel ("name", [sec_fade(r)]) [v3.306.1b]	End the background channel. Will clear all future commands and will add a "fade" command that will finish in "sec_fade" seconds. "sec_fade" is an optional value. How many seconds to assign the fading timer. Default is "6.0" seconds.

Example:

Send text as a communication message. Will be repeated if the same code is called.

```
fn_send_text_message("Hey this is a text as communication message")
```

Same as the previous command, only this time the plugin won't send the same "tracked" message twice.

```
fn_send_text_message("Hey this is a text as communication message", "message1", true)
```

Call message by the name "TakeOff" from mission XML file

```
if not fn_send_comm_message("TakeOff") then
    print "Error sending Message."; 'debug
    print mxError;
endif
```

Continued… (s)=string, (b)=bool, (r)=number

Continue...	
fn_create_new_comm_message ("message name", "message text", "track name", "mute_xplane_narrator(b)", "hide_text(b)", "seconds_to_display_text(i)"	Create a new comm message, no mixture channels defined. You need to "set" them manually.
fn_set_comm_channel ("message_name", "sound file",	Add communication channel to a

seconds_to_play (r), sound_volume (r))	message
fn_set_background_channel ("message_name", "sound file", seconds_to_play (r), sound_volume (r))	Add background channel to a message

Creating on the fly new message example:

```
if not fn_create_new_comm_message ("MyMessage", "write message") then
  print mxError; 'debug
endif
```

```
' Change message text from script using their properties
'                       {message name},      {property}, {property value}
fn_set_message_property("EnteredGAParkArea", "message", "You entered parking area")
' change triggers message from script
'                       {trigger name},      {property},                {property value}
fn_set_trigger_property("trig_park_in_KOAJ", "message_name_when_fired", "EnteredGAParkArea")
```

Message Properties

You are allowed to modify the following messages properties during mission runtime:

fn_set_message_property ("message_name", "property_name", "property_value")

Property name	Expects string as	Value Example
message	string	The message itself. Modify original message
override_seconds_to_display_text	int	Number to replace original time in seconds
mute_xplane_narrator	bool	true, false
hide_text	bool	true, false
enabled	bool	true, false
is_pad_message	bool	true, false
New since v3.306.3		
next_msg	string	Enter message name
add_minutes	string	A string number between 1..240
timelapse_to_local_hours	string	A string in the format: **"h24:mm:cycles"**
set_day_hours	string	A string in the format: "day in year:hour:min" "123:23:12" "-1" at the start, means current day. "-1:23:12"

post_script	string	Name of script / scriptlet
open_choice	string	Name of pre-defined <choice> element.
fade_bg_channel	string	A "complex" string that holds: "message name, seconds". Define which active bg channel to fade out and in how much time. Default is 6 seconds.

fn_get_message_info("message_name").
*Will seed **all*** the following arguments:

Seeded Variable Name	Type	Expected Values
mx_name	string	Name of the message
mx_message	string	Current Message text
mx_next_msg v3.306.3	string	Next Message Name
mx_open_choice v3.306.3	string	Choice Name
mx_mode v3.306.3	string	Message Mode ("" or "story")
mx_enabled	bool	true, false
mx_override_seconds_to_display_text	int	Timer number in seconds

GPS

The <gps> element receives a set of <point>s with only "lat/lon" coordinates, and will build an automatic flight plan in the FMS based on latitude/longitude. Meaning, you can't alter them, but you can replace them or add to them.
This is an optional feature.

```
<gps>
   <point lat="47.439491" long="-122.315522"/>
   <point lat="47.437840" long="-122.462416"/>
   <point lat="47.643279" long="-122.521499"/>
</gps>
```

Tip: you can use your mandatory tasks based triggers as part of the points. Just copy one of their points into that element to build a plausible route.

Functions to handle GPS/FMS

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
Message related functions	
fn_add_gps_xy_entry (lat, lon, elev_ft, entry)	[v3.0.242.10b3] adds a lat/long in FMS/GPS. "Lat" and "lon" are mandatory. Default behavior: will add coordinate to the end of flight plan. It is quite safe to always define elevation as 0. Entries can be between 0..99. All values are numbers, not strings.

<inventories>

Plane storage and external storage can add to the immersion of the mission. The "inventory" feature is an optional implementation of storage-handling in X-Plane. We have 2 types of inventories:

1. **Location based** - needs X/Y coordinates in the X-Plane world. It will be available once the plane enters its area.
2. **Plane storage** - This is your plane, so no coordinate is needed.

Inventory limits:

	New Limit	Old Limit
Plane	30 items	15 items
External	100 items	15 items

As of v24.12.2 there are two types of "plane" storage layout.
Since X-Plane 12, there are "station" storage areas that directly affect the CG of the plane. This means that storage placement has an impact. The "older" plane storage implementation, directly modified the "payload" which evenly distributed the weight for best CG.
The plugin allows you to define "XP11 compatible storage" or "not", which will display the newer layout.

But before, I'll need to explain a few things regarding the "new" storage layout.

1. It is highly dependent on the aircraft creator. They have to define "stations" in their planes, which will be reflected in their ".acf" file.
2. The plugin reads the ".acf" file, and collects the "station" names and their index from there.
3. Not all functionality was implemented for the newer XP12 layout.
4. Based on that information, you can then test how you would like to approach the inventory implementation and "force" the "compatibile" layout or not.

Do remember, the fact that I use the "compatibility" and "xp11" jargon, does not imply inferiority of that approach, after all it serves us well. It is just a way to handle the "storage" which was enhanced in X-Plane 12, and now takes shape in the plugin.

Inventory behavior in the XP11 compatible mode

The items in the plane inventory will be added directly to the plane's payload weight and will be evenly distributed. We calculate the weight as follows: "Plane Empty Weight" + "Pilot Weight" + "Passengers Weight" + "Storage Weight" + "Inventory Items Weight" (See "General settings" for more information).

- To represent an external inventory, we use: "<*inventory*>" + "<*item*>" list + <*loc_and_elev_data*> - elements.
- To represent the plane inventory, we just use the "<*plane*>" + "<*item*>" list.

Here are some examples:

```
<inventories>
  <!-- blueprints: items that might or might not be in an inventory. -->

  <plane>
    <item barcode="goods_for_palm" name="goods for PALM"  weight_kg="58" quantity="1" />
  </plane>

  <inventory name="YPAM Office" type="rad" >
    <item name="YBTL_mail" barcode="YBTL_mail" quantity="1"  weight_kg="20"/>
    <item name="YBTL_cargo" barcode="YBTL_cargo" quantity="1"  weight_kg="120"/>
    <loc_and_elev_data>
      <radius length_mt="20" />
      <point lat="-18.751736" long="146.579529"/>
    </loc_and_elev_data>
  </inventory>

</inventories>
```

Inventory behavior in XPlane-12 with stations

The items in the plane inventory will be added to the plane's "station" array and will affect the CG due to their location in the plane.
We calculate the weight as follows: " "Pilot Weight" + "Storage Weight".
The pilot weight will be added behind the scenes if the "pilot station" is less than 40kg. This means that you will see the "pilots" weight and not the "items" in the "pilots" station dataref (not the inventory screen), unless you reach the 40kg mark, in which the plugin will use the "items" weight.

- To represent an external inventory, we use: "<*inventory*>" + "<*item*>" list + <*loc_and_elev_data*> - elements.
- To represent the plane inventory, we just use the "<*plane*>" element, but **no** item list will take effect for now. If you want to place items in the plane, the simmer will have to move it from an external inventory.

<inventory>

An external storage you can interact with

Attribute Name	Type	Value	Meaning
name *	str	"Transit", "Hangar"	Unique inventory name
type *	str	"rad","poly"	Each inventory must have a physical area that allows the plane to interact with it. The area should be: "radius" based or "polygonal" based.

<item>

One liner element. Describe something that can be placed in an inventory.

Attribute Name	Type	Value	Meaning
name	str	"Parcels to palm"	Item description
barcode *	str	"parcels"palm"	Unique barcodes can be the same as the "name" or any, just remember that in order to refer to an item, you should only use the "barcode" string, not the name string.
weight_kg	float	15, 10, **0**	Weight of 1 item in KG.
quantity	int	1,20,**0**	How many items are in your inventory.
image_file_name	str	"item01.png"	An image that represents the inventory item at hand. ~~Thumbnail size: {45x65}px~~ ~~Max suggested image size: ~{223x151}px~~ [v24.06.1] Thumbnail size: {50x40}px Max suggested image size: ~{250x150}px ~~Max Inventory area size: !{352x178}px~~ Clicking on the item image will zoom in and out
New since v24.12.2			
mandatory	bool	[**false**|true]	Is this item a mandatory item ? You can restrict the target inventory for mandatory items.
target_inventory	str	"inv name1, inv name2 ..."	Provide zero or more valid inventories to restrict for which inventories you can move the mandatory item from the plane. ***Empty list*** means that you can move the item to any inventory and you will test the validity using "embedded script".

<loc_and_elev_data>

For inventory, you only define the physical area, meaning: "points" and "radius" if the inventory type is "rad".
The elevation is always: "on ground", so no need to define it.
Please see the trigger explanation for the "loc_and_elev_data".

<plane>

Plane element only needs the item list in it.

<station>

Only for the newer inventory layout.
Its attributes are automatically generated by the plugin from the ".acf" file. Do not provide a name, but you should be able to set an "id" for ad-hoc item placement.

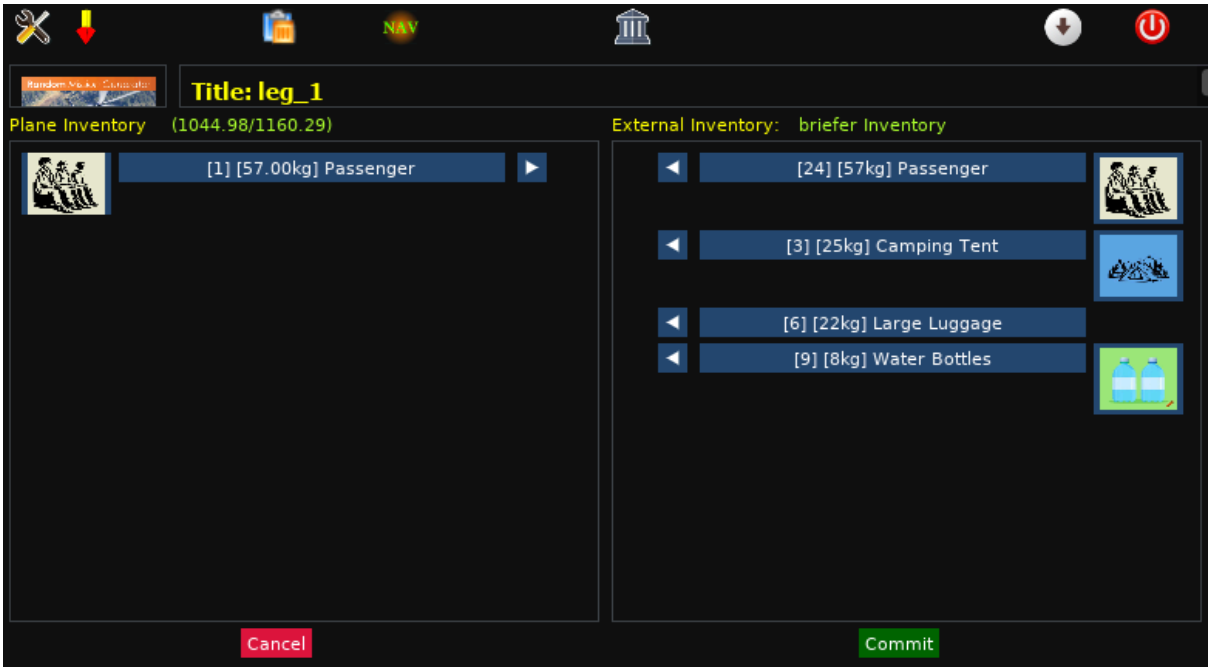
id*	int	"0..9"	The station index. Example: "0" = Pilot.
name (used internally)	str	"station"	Station name. Should be set by the plugin from the ".acf" file, and not the designer.

<item>

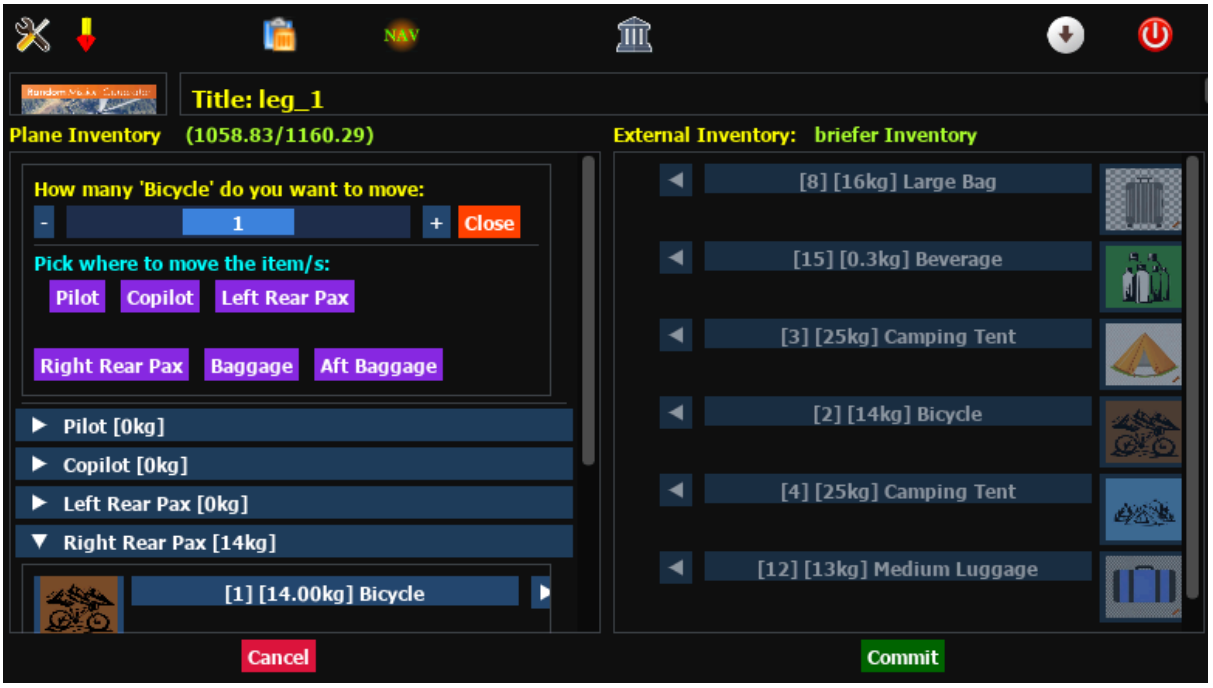
Attribute Name	Type	Value	Meaning
Check the <item> definition in the <inventory> element above.			

When you enter an inventory area, an icon will be displayed on your MX-PAD title bar.

See below for the XP11 compatible inventory layout example:



The newer layout, since X-Plane 12, supports the "station" location.



When a simmer picks an item to move to the plane, they have the option to define "how many" and where to place the item/s. This will directly affect the CG of the plane.

The station names and weight suggestions are dependent on the ".acf" file settings. This means that some planes might not have a "good" station distribution.

Functions to handle inventory items

These functions were not tested with the new plane inventory layout, yet.

(s)=string, (b)=bool, (r)=number

Function	Layout Comp.	Comment / Example
Inventory related functions		
fn_get_inv_layout_type v24.12.2	xp11/12	Returns the "inventory layout" type the mission is running at. "**11**" = XP11 compatible (no station). "**12**|[any number]" = XP12 compatible = "stations".
fn_move_item_to_plane ("from inventory"*(s), "barcode"*(s), quantity(r), station(r)) v24.12.2	xp11/12	Move an item to the plane. - The first two attributes are mandatory but the last two are optional. "quantity" is a number. Default is "1". "station" is a number. If you do not set it, the plugin will use "0" as default.
fn_is_item_exists_in_plane("barcode")	xp11/12	Returns the number of items in the plane. 0 = None
fn_is_item_exists_in_ext_inventory ("external inventory name", "barcode")	xp11/12	Returns number of items in external inventory. 0 = None
fn_move_item_from_inv ("from inventory"*, "barcode"*, "quantity", "to inventory", "station_id" v24.12.2) v24.12.2: Added "station_id" ALL VALUES ARE STRINGS	xp11/12	Move items from one inventory to another. - The first two attributes are mandatory but the last two are optional. - Default value for quantity is 1. - Default value for "to inventory" is "empty string", this will discard the items from source inventory. - Plane inventory name is "**plane**". Can work with "xp12 station" inventory layout. - (v24.12.2) "station_id" was added to be compatible with XP12 inventory layout. Default is "0" but if the plugin won't find the item in the default station ("0"), it will search the first "item" in the other stations.

Example:

```
' Check if parcel has been delivered
quantity = fn_is_item_exists_in_ext_inventory ("YPAM Office", "goods_for_palm")

if ground_speed < 2 and parkbrake = 1 and quantity > 0 then
  fn_set_trigger_property ("trig_park_in_YPAM", "script_conditions_met_b", "true")
endif
```

<choices>

The choice screen is displayed in its own window. The designer can use it to make the simmer pick an option in order to:

- Branch the mission narrative.
- Do a special action.
- Interact with the simmer and make decisions based on their picks.

Limitations:

- You are limited to 6 options - per <choice> sub element.
- The size of the choice screen is not dynamic, so try to keep the option text short and to the point.
- The designer controls the "choice window".
  You decide when to display or hide the window, so make sure it won't get in the way of the simmer.
  In 2D mode, the choice window is displayed in the middle of the screen, and in VR it is part of the messages page.

The <choices> element is a container that holds all the <choice> elements.
You can define the <choices> element in the <flight_plan> or in the <MISSION> element.
You can make as many as you want but make sure that the sub elements <choice> are uniquely named.

<choice>

```
<choice name="choice1" next_choice="choice1" text="title text" />
  <option .../>
  <option .../>
  ...
</choice>
```

The choice element holds the pool of options that a simmer has to pick from.
A choice must have a unique name and it can have branching "choices" using the "next_choice" attribute either at the "choice" level or the "option" level.
The "text" attribute is optional and represents the title text (if you want).

You can envision the "**choice**" element as a user based trigger, any option the simmer picks can fire any one of the outcomes in the "option" attributes.

The default behavior of a "choice" window is to hide itself after picking an option.
What affects its visibility is the "`next_choice`" attribute that you can find in the "**option**" element or the "**choice**" element.
If you want to continuously see the same choice until a "close option" will be picked, then define the "`next_choice`" at the "**choice**" level to have the same name as itself, and leave the <option>s "`next_choice`" attributes empty.
Caveat - in this scenario the window will never hide itself without "hide" code.

On the next page, you will see the <choice> attributes.

Choice attributes

Attribute Name	Type	Value	Meaning
name *	str	"{Name}" "Pick teleport destination"	Unique choice name
text	str	**"Some title text"**	Optional: Serve as a title text.
next_choice	str	"_" | {name}	Default "" (empty string) - will hide after one pick. If next_choice is defined then picking an option can progress to the next choice. If "next_choice" is empty and we need to progress, then it will hide the choice window.

- *If you want to continue and pop the same Choice, then fill in the same choice name.*
- *If you won't define any continuous choice name the choice name will hide after the first pick.*

<option>

```
<option  text="{valid XML free text}"
         message_name_when_fired="" script_name_when_fired=""
         commands_to_exec_when_fired="" dataref_to_modify_when_fired=""
         onetime_option_b=""  next_choice="" hide="" />
```

Option attributes and their meanings

Attribute Name	Type	Value	Meaning
name *	str	"option Name"	Unique option name, I suggest to just use numbers
text *	str	"valid text"	Short description of the choice at hand. No more than 60 characters. Please be advised that some characters in XML will invalidate your file, so make sure not to use them directly, like: "&".
onetime_option_b	bool	"true|**false**" "yes|no"	Default: ***false*** Only fire once. Once you click it, the outcome will fire and the option will hide itself (the internal "hide" attribute will be set to true).
next_choice	str	{set name}	Which set to display. If not defined then the plugin will check <choice next_choice="?"> attribute value. If the options "next_choice" and the <choice> "next_choice" attribute are empty or not set, then it will hide the "choice window".
Outcome when an option is picked - will fire in the following order			

script_name_when_fired	str	{see trigger outcome example}	Call script or scriptlet from <embedded_scripts> element. [v3.0.304.7] Plugin will seed "mxCurrentLeg" before running the script
message_name_when_fired	str	{see trigger outcome example}	Call message in "message_templates" element
dataref_to_modify_when_fired	str	{see trigger outcome example}	Modify dataref or dataref array. Supports integer/float/double/array int and array floats. Has special formatting: {dref_name1}:{value}, {dref_name2}:{value}...
commands_to_exec_when_fired	str	{see trigger outcome example}	Execute command or set of commands delimited by comma ","
Internal attribute			
hide	bool	"true|**false**" "yes|**no**"	**Internal Attribute** Depends on onetime_option_b, **False** = Display option True = Hide Option

Some highlights:

- Option element can be hidden automatically after it has been picked, if "onetime_option_b" is set to "true".
- Option can define a branching choice using "next_choice".

Choice related Functions

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
<choice> related functions	
fn_set_choice_name ("choice name")	Set a choice name to display its options. Think of it as *arming* the options to display. It won't automatically display the choice window, you will have to use the fn_display_choice_window() function.
fn_display_choice_window()	Display the choice window.
fn_hide_choice_window()	Hide the choice window.
fn_get_active_choice_name() [v3.306.3]	Get active choice name.

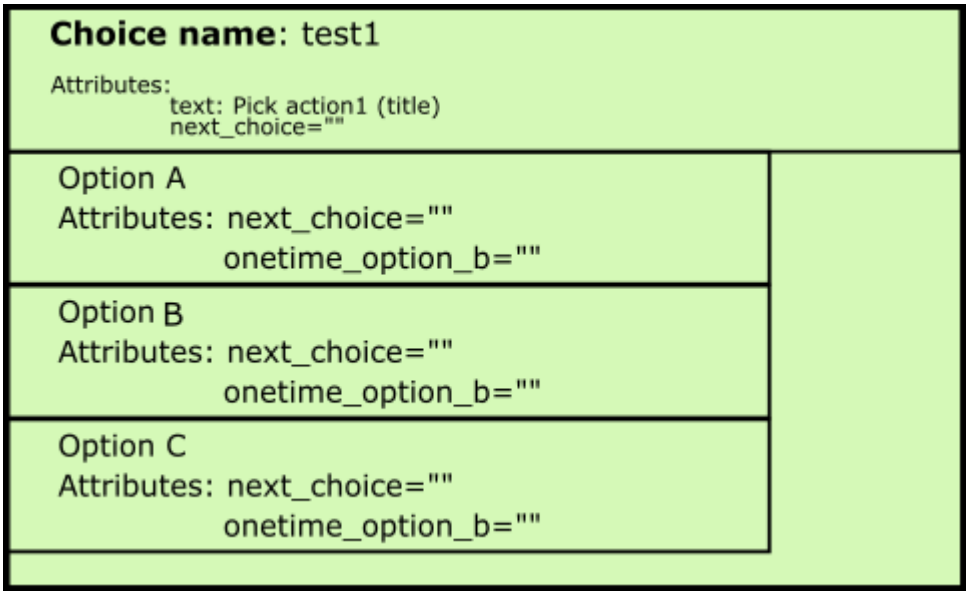
```
<scriptlet name="test_choice_scriptlet">
  <![CDATA[
```

```
      fn_set_choice_name("leg_1_choices_1")
      FN_DISPLAY_CHOICE_WINDOW()
  ]]>
</scriptlet>
```

Visual explanation of <choice> handling

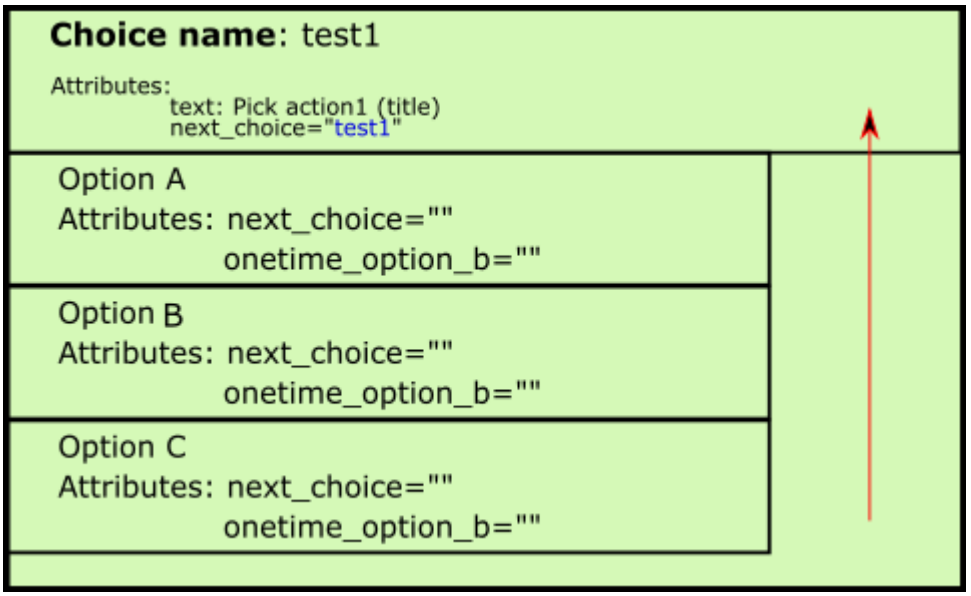
Let's assume that we have a <choice> element with three options.

Example 1: The choice name is: "test1".
No "next_choice" attribute was set.



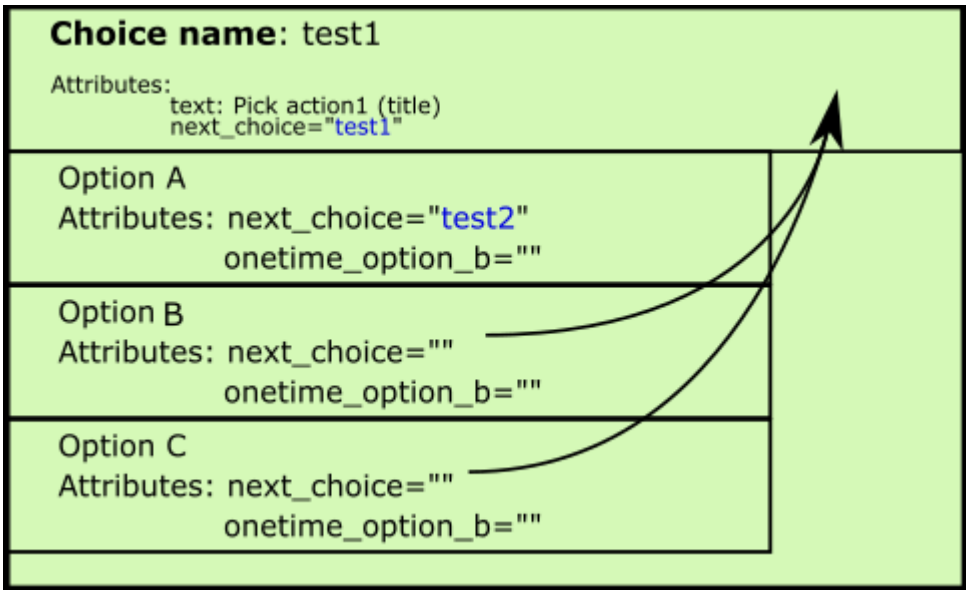
Outcome:
The choice window will ***hide*** itself after the first option was picked.

Example 2: We define "next_choice" to be the same name as current choice at the choice element.



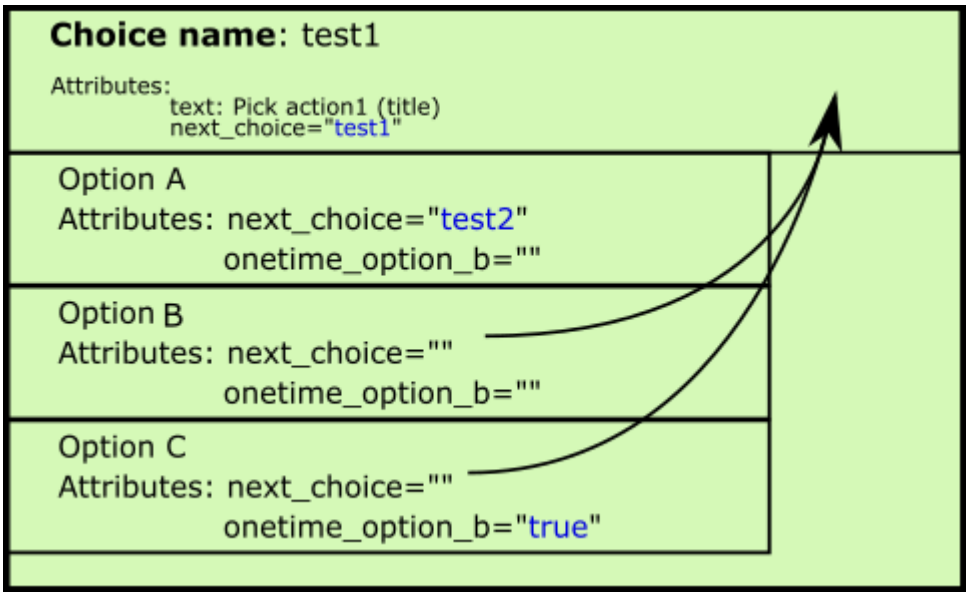
Outcome:
When we pick one of the <option> there is no "next_choice" attribute defined, therefore the plugin will fallback to the parent element, "<choice>" and use its "next_choice" attribute value. Since we use the same choice name, the window will **not** hide itself (it is the same as calling oneself).

Example 3: The *choice* and one of the *options* elements define different "next_choice" values.



Outcome:
Pressing "Option A" will display a new "choice" dialog (in the same window).
Pressing "Option B or C" will fallback to the parent "next_choice" value, which will keep the same window open.

Example 4: Option "C" attribute "`onetime_option_b`" is set to "***true***"



Outcome:

Once a user picks "Option C" the *option line* will hide itself, leaving options A and B available in the "choice window". The window itself will stay on the same choice since it will fallback to the "*next_choice*" attribute of the parent element.

3D Objects

The 3D Object element allows the designer to place a 3D Object in the X-Plane world.
It is not a replacement to "Custom Scenery", it should assist in driving the mission from a visual perspective.
The 3D Object can be a ***static*** or a ***moving*** element.

In order to display a 3D Object, you will need:

1. "OBJ8" file available to the plugin.
2. Location: where to position it.
3. Condition: When to display the object:
    a. * In which "flight leg"
    b. [optional] Until "flight leg"
    c. [optional] Script condition - w.i.p (do not use yet)
    d. [optional] Distance to display in nm.
    e. [optional] Link to Task ("{objective}.{task} name")

    Except the first condition, all the rest are optional and will allow you to better fine tune when and where to display the 3D Object.

Each 3D Object is a template, and therefore should have a unique name so you could refer to it from other elements or scripts.
Since X-Plane 11.10 Laminar Research prefers to use "instancing" technique over 3D drawing. For the sake of compatibility the plugin will create an instance of the 3D Object template when you use the: "<display_object>" element in a <leg>. The instance name must be unique and it will "copy" the "<obj3d> template definitions while allowing you to replace certain attributes using a special "replace_" prefix to the same template attribute names.

Benefits: we can use the same "obj3d" template and create many instances with different names and to also alter each instance specific: location and even conditions, if we so choose. To do that use the "replace_" directive before the list of supported attributes. The format is: "replace_{attrib}". See "display_object" element.

Here is an example of a static 3D Object template:

```
<object_templates>
  <!-- Green Marker -->
  <obj3d  name="marker01" file_name="green_marker01.obj" >
      <conditions distance_to_display_nm="10" keep_until_leg="" cond_script="" />
      <!-- location: 0 = ground level.  -->
      <location lat="-19.463989" long="23.373879" elev_ft="[+/-nnn|0], 1450" />
      <tilt heading="0" pitch="0" roll="0"/>
  </obj3d>
<object_templates>
```

You can remove elements and attributes that are not in use:

```
<object_templates>
   <obj3d  name="marker01" file_name="green_marker01.obj" is_virtual_b="no">
      <conditions distance_to_display_nm="10" />
      <location lat="-19.463989" long="23.373879" elev_ft="[+/-nnn|0], 1450" />
      <tilt heading="0" pitch="0" roll="0"/>
  </obj3d>
<object_templates>
```

It is the same elements with the same rules but without the attributes we do not need or their default value is good enough.

<obj3d> - 3D Object definition

Attribute Name	Type	Value	Meaning
name*	str	"green_arrow"	Unique name to the object template
file_name*	str	{filename}.obj	The object file name (obj8 type), relative physical path or virtual path

<conditions />

One liner element. Help determine when to display/hide 3D object

Attribute Name	Type	Value	Meaning
distance_to_display_nm	float	10, 5...N	Distance in nautical miles to display the 3D Object.
keep_until_leg	str	"Leg 2"	Display 3D Object until "leg X" starts
cond_script	str	"script.function"	A function that will evaluate "something" and will update the Instance profile: "script_conditions_met_b" property. The instance name is defined in the "display_object" element.

<location /> - Only for static 3D Objects

One liner element. Location element is *only mandatory* for static 3D Objects
It allows you to place the 3D Object

Attribute Name	Type	Value	Meaning
lat / lon	float	10.123650 -92.325641	Position coordination. For precise location use decimal location with 6 positions after the decimal sign.
elev_ft	float	**0**, 1450	Plane elevation. 0: ground level
elev_above_ground_ft	float	**0.0**, +140	If set, it will **override** elev_ft and will reset it to Zero. Plugin will probe for terrain elevation and will add the "elev_above_ground" value to it.

<tilt />

One liner element. Define how to place the 3D Object

Attribute Name	Type	Value	Meaning
heading_psi	float	10.123650 -92.325641	Heading PSI value
pitch	float	-90..**0.0**..+90	Plane Pitch should be between -90 and + 90 Default: 0.0 (zero)
roll	float	-359..**0.0**..+359	Plane roll between 0.0 and 359

<path> Part of moving object

Path element is mandatory to define a moving 3D Object.

The path includes points which represent targets to reach. Each point also includes information regarding the location, speed, heading, tilt and roll properties the Object should take during transition.

The starting coordinate is always the ***first*** point. It, implicitly, pushed to the beginning of the path. Therefore its "heading_psi" defines the starting heading for the object.

The rest of the points (second point onward) should only use: "***adjust_heading***", since the plugin calculates the relative transition between them.

The plugin positions the objects heading using the first point "*heading_psi*" attribute and "transitions" the object to the "next point" using the "next point" adjust_heading/roll/tilt and pitch to make the transition through time.

Calculation example:

```
displayCoordinate.setHeading(pointTo.adjust_heading * time + pointFrom.getHeading() );
// Where time = fragment of time passed between two frames.
```

Example:

```
<path cycle="no">
  <point lat="-19.253647" long="146.770935" elev_ft="100" heading_psi="80" />
  <point lat="-19.253647" long="146.770735" elev_ft="100" adjust_heading="-20" />
</path>
```

The plugin will see two points in the path, the first and second one. The first is the starting location while the second is the target to reach for. Pay attention that the second point uses "`adjust_heading`" !

<path> attributes:

Attribute Name	Type	Value	Meaning
cycle	bool	"true|**false**" "yes|**no**"	Should 3D Object continue movement to the first location once it reached the end of the path.

<point />

One liner element. Holds coordination information of our 3D Object target locations, in PATH

"point" attributes

Attribute Name	Type	Value	Meaning
Lat *	decimal	-34.342564	Latitude value
Long *	decimal	+20.543543	Longitude value
elev_ft	float	0.0, 1254	If "**elev_ft**" is zero, then the plugin will place the object on ground.
elev_above_ground_ft	float	**0**, 150, 50	If defined will override elev_ft definition. Plugin will calculate elevation by probing terrain at the point location, and will add the elevation above ground to it..
heading_psi	float	0..356	Use **only for the first point** in a path.
adjust_heading	float	-359..+359	Adjust the heading relative to the current point heading. You can adjust in small numbers to create a turn like effect between the 2 points, but then you will need more points to create the illusion. **Default: 0** = no change in headings
speed_kmh	float	**0**, 10, 25 etc...	Speed in kilometers per hour. Default: **0kmh** (v3.0.254.7).
pitch	float	-89..**0**..+89	Values should be between -89 and 89
roll	float	-359..**0**..359	

Static 3D Object does not need <path> element, only the <location> one:

```
<obj3d  name="static_giraffe" file_name="xp_giraffe_y.obj" obj_file_when_inactive="" >
    <conditions distance_to_display_nm="10" keep_until_goal="LandInChitabe" cond_script="" />
    <location lat="-19.463989" long="23.373879" elev_ft="0" />
    <tilt heading_psi="65" pitch="0" roll="0"/>
</obj3d>
```

While moving objects needs the "path" element and more than one point, the location element is basically ignored:

```
<obj3d  name="moving_giraffe" file_name="xp_giraffe_y.obj" obj_file_when_inactive="" >
    <conditions distance_to_display_nm="10" keep_until_leg="" cond_script="" />
    <location lat="-19.463989" long="23.373879" elev_ft="0" />
    <tilt heading_psi="65" pitch="0" roll="0"/>
    <path cycle="no" >
      <point lat="-19.463989" long="23.373879" elev_ft="0.0" speed_kmh="20"  heading_psi="65" />
      <point lat="-19.463638" long="23.374584" elev_ft="0.0" speed_kmh="4"   adjust_heading="0" />
      <point lat="-19.463327" long="23.374590" elev_ft="0.0" speed_kmh="8"   adjust_heading="-15" />
      <point lat="-19.462733" long="23.374417" elev_ft="0.0" speed_kmh="0"   adjust_heading="0"/>
    </path>
</obj3d>
```

Here we are moving a 3D Object starting in point: "lat="-19.463989" long="23.373879"", elevation is "0" so it is placed on the ground, movement speed is 20 km h and starting heading is "65". When an object moves, it moves to the second point: "lat="-19.463638" long="23.374584"" and its heading does not change "adjust_heading="0"". When an object reaches point 2, it will continue to point 3: "lat="-19.463327" long="23.374590"", this time heading will be adjusted gradually by "-15" degrees until reaching point 3, and so on.

How to display a 3D Object

When you want to display a 3D Object, you need to define it at the "<leg>" level. This will put the "object" in the plugins "display pool". Once it is there, it will be evaluated every flight loopback if to display or to hide.

```
<leg name="" >
  <link_to_objective name="start" />
  <display_object name="marker01" instance_name="unique name in mission"
                  target_marker_b="" [link_to_task=""] [replace_{attrib_name}=""] />
</leg>
```

<display_object> - element to link to 3D Object in a <leg> scope.

One liner element. Instance attributes:Example for replacing template attributes:

```
<display_object name="marker01" instance_name="marker01_2" target_marker_b="true"
link_task="Land and Park in YPAM.LandInYPA"  replace_lat="-18.751736"
replace_long="146.579529" replace_elev_above_ground_ft="20" />
```

"Target_marker_b" allows the designer to flag specific instances as marker helpers. This will also allow the plugin to hide the markers if the simmer wishes too.

Relevant only to static objects.

The <display_object> element will actually create an instance from the "3d object template". Therefore you should:

1. Have a unique instance_name in mission *.
2. Flag the instance as a "target marker" or not, using the "***target_marker_b***" attribute. This will be affected by the "setup" screen.
3. Instance objects can be displayed across a few legs in the "flight plan".
4. The conditions to display/hide an instance are:
    a. Obj3d template conditions.
    b. "Leg" to display the instance (where <display_object> was defined)
    c. [optional] link_to_task: If task is active then show, if completed or inactive then hide.
5. Use **replace**_xxx attributes to modify the ***static*** "obj3d" template to create a unique instance with its own attributes, different from the template.
   Example: you might want to place a marker (same object file) in 4 different locations.
   The template is limited to defining only 1 location. Therefore your options are:
    a. Create an "obj3d template" for each marker or
    b. Create 1 "obj3d" template and create 4 instances for the same object only replacing the lat/lon/elev etc.. at the instance level.

Last:
The sum of all conditions decides if an "instanced" object will be "displayed" or "not".

> *You can leave all conditions empty and so the object will be displayed immediately when the "flight leg"becomes active.*

Instance related Functions

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
3D Object instance related functions	
fn_get_distance_to_instance_in_meters ("instance name")	[v3.0.242.10b3] return the distance in **meters** to an active 3D instanced object. Return "-1" if the instance is not active. Return "Nnn" if the instance exists.
fn_get_elev_to_instance_in_feet ("instance name")	[v3.0.242.10b3] returns the elevation difference between 3D instanced object and plane. Return "-1" if the instance is not active. Return "Nnn" if the instance exists. Example: when you want to test if a plane is near a 3D object but not just in distance but also in elevation.
fn_get_bearing_to_instance ("instance name")	[v3.0.242.10b3] returns the bearings in degrees between plane and instanced 3D object. Will return "-1" if the instance is not active.

```
<scriptlet name="leg2_calc_bearing_to_raft_scriptlet">
<![CDATA[

FN_ADD_GPS_XY_ENTRY (23.814365, -75.793940) ' add lat/lon coordinate to
the end of entries

bearing_to_raft = fn_get_bearing_to_instance ( "moving_life_raft_01" ) '
store bearing to instance
if bearing_to_raft >= 0 then
  message = "We evaluated the bearing to the raft as: " +
STR(bearing_to_raft) + " degrees. Fly lower than 2000 feet to identify
them. You added the coordinate to your 'flight plan'."; ' prepare msg
  print message; ' debug
  fn_send_text_message (message);
endif
]]>
</scriptlet>
```

<xpdata> - data from X-Plane

By "xpdata" we refer to "DataRefs", which represent information that X-Plane read/write too and "share" with external programs like plugins and other third party applications.
To read/write from an embedded script you need to pre-define the DataRef you want to read from. This is done to:

1. Make the designer focus on what he/she needs.
2. Prepare the "pointers" and decide if this is a scalar or array parameter by the plugin. All this should be done ahead of time to decrease useless overhead during execution time.

The element to point to "DataRefs" is "<dataref>"

```
<xpdata>
  <dataref name="retract_gear" key="sim/aircraft/gear/acf_gear_retract"/>
  <dataref name="prop_speed_rpm_arr[0]" key="sim/cockpit2/engine/indicators/prop_speed_rpm"/>
  <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="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"/>
</xpdata>
```

<dataref />

One liner element.
Holds information on X-Plane dataref or custom dataref so we can read or set their values from scripts.

"<dataref>" attributes:

Attribute Name	Type	Value	Meaning
`name *`	str	`"retract_gear"`	Unique name that will identify the dataref in your scripts.
`key *`	str	`"sim/aircraft/gear/..."`	Full dataref path. You should use the "DataRefs.txt" in "/Resources/plugin" folder or the dataref page.

If your "key" represents an array dataref, then your "name" attribute should have an element indicator like: "{name}[element]" where "element" is equal to 0..n-1 of the elements in the array.
Example: if you want the first element you will use: "{name}[0]", if you need the third element you will use: "{name}[2]".

```
<dataref name="ignition_arr[0]" key="sim/cockpit2/engine/actuators/ignition_key"/>
```

Dataref format in triggers, start cold and dark and weather:
As of **v3.304.12** you can provide more than one variable from the array offset but it must be a continuous data without holes in it. There is no more support to modify all arrays by just sending one value to the name of the array.

Example:
Dataref: *sim/weather/region/cloud_base_msl_m* has 3 cells in its array.

0	1	2

Remember, arrays start with cell "0" until "n-1".
So array of 8 members is equal to "0..7"

Valid Usage:
You want to set all three cells:
sim/weather/region/cloud_base_msl_m[0]=1111,2222,3333
You define the offset "0" - where you want to start, and the sequence of values.
The offset "0" is optional in this case.

Wrong Usage:
You want to just fill in cell "0" and cell "2"
sim/weather/region/cloud_base_msl_m[0]=1111,,3333
You can't skip part of the cells. The plugin will ignore the empty cell definition and will write to cells "0" and "1" instead of "0" and "2".

Valid Usage:
You want to set the second and third cells:
sim/weather/region/cloud_base_msl_m[1]=2222,3333
We define our array offset to start from cell "1" onward.

Valid Usage:
You want to set only the second cell:
sim/weather/region/cloud_base_msl_m[1]=2222
We define our array offset to start from cell "1" and since there is only one value the plugin will only set it accordingly.

<end_mission>

To every mission there should be one end mission element.
The <end_mission> element should have *success* and *fail* elements that control the images and messages that will be displayed.
Example:

```
<end_mission>
  <success_image file_name="success.png"/>
  <success_msg>
    <![CDATA[well done pilot. Have a nice day and see you tomorrow.]]>
  </success_msg>

  <fail_image file_name="fail.png"/>
  <fail_msg>
    <![CDATA[You failed !!!]]>
  </fail_msg>
</end_mission>
```

<success_image /> <fail_image />

One liner element.

Attribute Name	Type	Value	Meaning
file_name *	str	"success.png"	Enter the file name to display

<success_msg /> <fail_msg />

One liner element.

Sub element	Type	Value	Meaning
<![CDATA[ ]]	str	Free text	Enter the

End Mission Related Functions	
fn_abort_mission ("reason")	Abort mission function. You must send a text string to explain the reason why the mission has been aborted. This should be displayed in the Briefer.
fn_update_end ("tag name", "attrib name", "value") [v24.02.5] Example: fn_update_end ("fail_image", "file_name", "fail02.png")	This function will drill down from <end_mission> root element, and will update its sub element - "attrib name" with the provided "value". All parameters are strings.

<embedded_scripts>

Conditions, logic, complex logic, modifying X-Plane DataRefs on the fly can be done through scripting (well almost everything).
Mission-X plugin exposes a set of functions that allows you to interact with the plugin or directly with X-Plane, it is up to you to devise a way to leverage it to your benefit.

<embedded_scripts> element allows us to define the name of *script* files or *scriptlets* we would like to use during the mission.

As of version 3.306.3, you can send a predefined parameters to any script using the pipe symbol "|" and then a set of "in{Name1}=value,in{Name2}=value" string.

- The parameter ***must*** start with the prefix "***in***" followed with the name and then the value.
- ***Supports***: numbers, bool and text.
- ***No special characters*** are allowed, so quotes should *not* be used.

Example:

```
<outcome
script_name_when_fired="disable_trigger_generic|inTrigName=trigleg01" />
```

In this example, we call the scriptlet "`disable_trigger_generic`" and we send a seeded parameter "`inTrigName`" with the value: "`trigleg01`".

In the scriptlet code we will write something as the follow:

```
<scriptlet name="disable_trigger_generic">
<![CDATA[
print "Generic trigger disable script.";

' This is to make sure we have a local value with the same name if we did not send an ad hoc value.
inTrigName=STR(inTrigName)
if inTrigName = "0" then
  print "using mxCurrentTrigger: ",mxCurrentTrigger; ' debug
  inTrigName = mxCurrentTrigger
endif

print "Disabling trigger: ",inTrigName;

fn_set_trigger_property (inTrigName, "enabled", "false"); ' Disable target trigger
fn_set_trigger_property (mxCurrentTrigger, "enabled", "false"); ' Disable myself

 ]]>
</scriptlet>
```

"script": set of commands reside in a file. Each line in the file is interpreted every *flight loop*. We should be careful with what we write inside the script file.
A script can also include other scripts.

"scriptlet": A short script, mostly one to two lines of code, we embed in the XML file instead externally. You can define it at the <embedded_script> element or inside each

<flight_plan> as a sub element, so the code will be "near" the task at hand.
A scriptlet can only include one file.
One or two lines of code does not mean hardcoded limiting commands. You can write as many rows as you want, but it was not created to replace external script files.

```
<embedded_scripts>
  <file name="myscripts.bas" />
  <file name="include01.bas" />
  <file name="mxpad01.bas" >
    <include_file name="include01" />
  </file>

  <scriptlet name="scriplet_dummy_message1" ><![CDATA[fn_send_comm_message
            ("{message name}", "{track name}", "{one time?}")]]></scriptlet>
  <shared_variables>
    <var name="step1" type="number" init_val="0" />
  </shared_variables>
</embedded_scripts>
```

"embedded_scripts" sub elements

<file />

One liner element. Holds information regarding external script files.
"<file>" attributes:

Attribute Name	Type	Value	Meaning
name *	str	{file name}.bas	Filename that holds the script
<include_files>			
name *	str	{script name}	No need for extension. The name of the script to add at the top of the script.

A <file> represents an external script.

The external file might have an extension, usually ".bas", but the plugin will strip off the extension, so you should just refer to the name of the file without the extension when defining "base_source" attribute and such.

```
    <file name="include01.bas" >
    <file name="runup_test.bas" >
     <include_file name="include01" />
    <file>
```

In the following example we included the file "include01". This file should have been defined first also as a <file>. The plugin strips the extension ".bas" and stores only the name "include01", this is why "include" elements do not need the whole prefix.

When we link"tasks" or "triggers" to scripts the script name should include the file+function name as follow: "ScriptName.function", no extensions are allowed.

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

<scriptlet > </scriptlet>

Holds a short script body, instead of managing the same in an external file.

"<scriptlet>" attributes:

Attribute Name	Type	Value	Meaning
name *	str	{file name}.bas	Filename that holds the script
include_file	str	{script name}	Appends the code to the beginning of the scriptlet code
![CDATA[ ]] *	str	{Short script code}	Short script. Useful to send a one time message or set "script_conditions_met_b" property value.

```
<scriptlet name="on_ground" include_file="include01" ><![CDATA[some code]]></scriptlet>
```

The <scriptlet> element was devised to simplify short code management. Instead of creating a file or a function in a long or external source file, you can just write the short code in the XML (inline code).
Suggested usages:

- Modify a property or DataRef every flight loop.
- Send a one time message once an objective becomes active.

<shared_variables>

The "Shared variables" container represents predefined global variables that we can use in our "embedded_scripts". This should be simpler than handling their creation inside the script itself.
We use the sub element "<var>" to define one or more variables:

<var />

One liner element. Holds information of "global variables" a script can use.

Attributes:

Attribute Name	Type	Value	Meaning
name *	str	iStep, iSelection	Global variable name.
Type *	str	[bool|number|string]	Type of variable
init_val	str	"0", "1", "hello"	The plugin will try to convert the value to the respective data type.

```
<shared_variables>
  <var name="Leg1_progress" type="number" init_val="0" />
  <var name="bWasInVnkt"    type="bool"   init_val="false" />
</shared_variables>
```

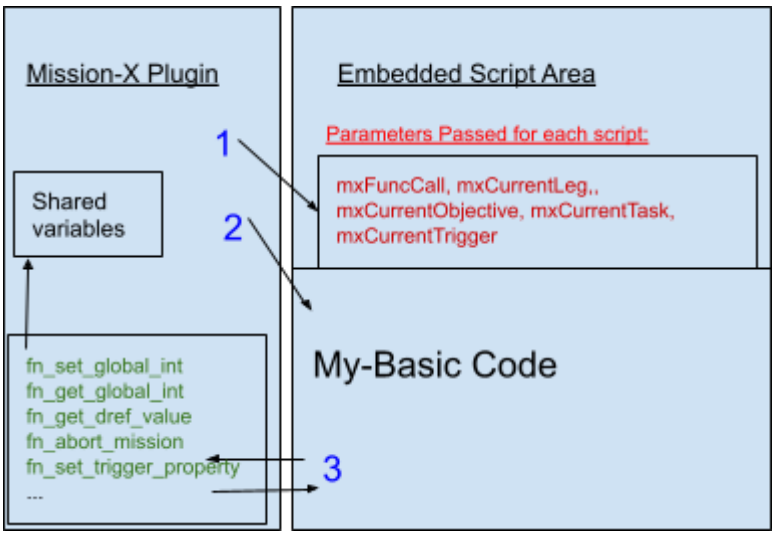
I strongly advise creating those elements as part of the flight leg they are related to, for example if you call "[pre/post]_leg_script", create the "scriptlet" inside the "flight_leg" element, especially if it is a short script.
That can make it simpler to follow the mission flaw especially in longer missions.

This is also true for triggers, choices and so on.

Understanding the Embedded Script Implementation

When embedding a scripting language, we need to define how and when to interact with the external script, what we send to the script engine and what we expose to it too.
Let's take a look at a high level flow of instruction calls between Mission-X plugin and the engine itself:

```
Assume a script is being called from "task1" in "Leg1 and objective1".
Script being called: "townsville_and_back.func1"
```



When the plugin calls a script it does the following:

1. Plugin fetch the script name "`townsville_and_back`" and before executing it, it inject "*seeded parameters*", some of them might be empty, depends on the stage when it was called,
   example:
   `"mxCurrentLeg" will store "Leg1" as a value.`
   `"mxFuncCall" will store the "func1" function name.`
   `"mxCurrentTask" will store "task1" as value.`
   `"mxCurrentTrigger" will be empty since the script was called from a "task" level not a "trigger".`
   These parameters indicate from where the script originated.

2. Once parameters are seeded, the plugin "executes" the "embedded script".

3. The embedded script can use the "seeded" parameters (in red) to locate the needed code and execute it. If we use the format: "*script.function*" then we can easily manage our script by writing:
   if (mxFuncCall = "func1") then
   call func1()
   endif
   The script can also call specialized functions from the plugin (in Green) to fetch or set information regarding plane/datarefs/MXPAD/messages, it should also

set global variables for later use.
Parameters defined in the script will lose scope (erased) after the script finishes execution.

Script seeded arguments name convention:

- Seeded arguments start with "mx{Name}".
- Seeded properties start with "mx_{name}" (used in "*fn_get_xxx_info*" functions). Some properties are shared with other components, therefore they do not start with mx_{name}.
  Use the tables below as reference.
- External functions start with: "fn_{name}".

In any script you can refer to the following seeded arguments (depends from where you called the script):

mxFuncCall - holds a lookup function name to use in the script.

mxCurrentTask - Current task Name

mxCurrentLeg - Current flight leg Name

mxCurrentTrigger - Trigger Name

mxCurrentObjective - Current objective Name.

mxError - Error message. Empty means OK.

mxCurrent3dObject - Current 3D Object Template Being used.
mxCurrent3dInstance - Instance name we gave to the 3D Object template.

Example:

```
' Main branching code
if (mxFuncCall = "check_ypam_park") then
  call func_check_ypam_park();
```

```
' Read DataRef and check if exists using mxError
ground_speed = fn_get_dref_value ("ground_speed");
if (mxError = "") then
  print "ground_speed: ", ground_speed," "; ' debug
else
  print "Error: ", mxError; ' debug
endif
```

List of Components you can modify through script:

Check each major topic for the "script" implementation. Some of the functionality is generic and not part of specific topics like Global or Dataref functions.

Setting Properties Using Scripts During Runtime

Some mission components can be modified using scripts to alter the course of the mission or to have more precise information for simmer.
The script components are listed below, and they are exposed to the scripts.
Most property values are sent as "string" values, pay attention to which variable has (b) or ® next to it, or it does not have "" around it.
Failing conversion will:

a. Fail the command.
b. Return an error message (*mxError*), but won't abort the script engine.

Check the Log.txt or mission.log for these errors, and try to solve them.

Task Properties

See Task topic.

Trigger Properties

See trigger topic.

<Message> Properties

See message topic.

External Functions To Use In My-Basic Scripts

When we use Mission-X functions, in embedded scripts, there are two ways to send values between the script and the plugin. The options are:

1. Send the native value, like number or boolean (I'll use "**s**tring,**r**eal,**b**oolean" as a hint in the tables for the correct value type).
2. Send a string that represents the value while the plugin will automatically try to translate the string to the correct value type.
   Example, we send "true" to the plugin. If a plugin expects a boolean value, it checks if this string represents a valid boolean value (both "true" and "yes" are considered boolean valid).

How to read the tables
The following table lists Mission-X special functions that allow external scripts to communicate with the plugin. The "Function" column lists all the function names and expected values to send. Pay attention to the (hint).
If the value is a string - then you will see quotation marks.
If it is a native value, then no quotation marks will be specified.

Example:
`get_task_info ("task name")`

Last: In most cases every function returns a boolean as success/fail (1/0) and seed a *mxError* parameter. If *mxError* is empty then all is well. In some cases the function will return the actual requested value.

Functions to manage script "global variables"

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
Global functions	
~~`fn_store_global_bool ("name", value(b) )`~~ `fn_set_global_bool ("name", value(b) )`	Deprecated ~~`fn_store_global_bool`~~
`fn_get_global_bool ("name", [init value(b)])`	Get "name" bool variable, if it does not exist then create it and initialize it with the default init. Plugin default value is "false"
~~`fn_store_global_int ("name", value(r) )`~~ `fn_set_global_int ("name", value(r) )`	Deprecated ~~`fn_store_global_int`~~
`fn_get_global_int ("name", [init value])`	Get "name" integer variable, if it does not exist then create it and initialize it with the default init. Plugin default value is "0"
~~`fn_store_global_decimal ("name", value(r))`~~ `fn_set_global_decimal ("name", value(r))`	Decimal number Deprecated: ~~`fn_store_global_decimal`~~
`fn_get_global_decimal ("name", [init value])`	Get "name" decimal variable, if it does not exist then create it and initialize it with the default init. Plugin default value is "0.0"
~~`fn_store_global_string ("name", "value")`~~ `fn_set_global_string ("name", "value")`	Deprecated ~~`fn_store_global_string`~~
`fn_get_global_string ("name", ["init value"])`	Get "name" string variable, if it does not exist then create it and initialize it with the default init. Plugin default value is "" (empty string)
Check existence of global variables	
`fn_is_global_bool_exists ("name")`	Search global value, returns 1 or 0
`fn_is_global_number_exists ("name")`	Search global value, returns 1 or 0
`fn_is_global_string_exists ("name")`	Search global value, returns 1 or 0
Free global variables	
`fn_remove_global_bool("name")`	Return boolean if success or fail
`fn_remove_global_number("name")`	For both integer and decimal numbers. Return boolean if success or fail
`fn_remove_global_string("name")`	Return boolean if success or fail

All "[init value]" are optional.

The "three stopper" Functions

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
[v3.304.13] **Timer names are predefined as 1,2 or 3**	
fn_start_timer (1..3 (r), seconds (r))	Start timer, provide a timer number between 1..3 (as a number) and the time to run (as a number). Default time is 24 hours (86400 seconds), if not provided. Calling the "start" function, when the timer is running, will reset it and then restart it.
fn_stop_timer (1..3 (r))	Stop and reset the timer. Provide only its timer number between 1..3
fn_get_timer_ended (1..3 (r))	Provide the timer number between 1 and 3. Returns **1** if the timer ends and **0** if not.
fn_get_timer_time_passed(1..3 (r)) Returning a number represents seconds passed.	The function **returns** the time passed for a specific timer number. If it returns **0** then it might not be active. If the returned number does not change then the timer might finish. Remember the timer will never reach the number you entered, so make sure you test against "**n-1**".

Examples:

```
In one script/scriptlet define the timer
fn_start_timer(1, 1200.0) ' We start timer number "1" for ~20 minute

In other script/scriptlet check the time passed timer
time_passed_f = fn_get_timer_time_passed (1) ' We get timers "1" passed time
if time_passed_f >= 1199.0 then
  fn_stop_timer (1)
  fn_set_datarefs(...)
  fn_set_task_property_in_objective (...)
endif
```

Functions to get/set DataRef data

(s)=string, (b)=bool, (r)=number

Function	Comment / Example
Get/Set X-Plane DataRefs	
fn_get_dref_value ("DataRef name")	"sim/flightmodel/position/latitude" DataRef return values as numbers
fn_get_dref_val_as_str ("DataRef name")	You can request to convert the DataRef value to string.
fn_set_dref_value ("DataRef name", value(r))	I suggest you prefer the "fn_set_datarefs" function over this one.

fn_set_datarefs("s")	You send one string in the format *"key=value|key2=value|key3=value1,value2..."* and the plugin will parse and assign their values to their respective datarefs
fn_set_datarefs_interpolation(seconds (r), cycles (r), dataref list "s")	Set the time in seconds and how many cycles to run the "stopper" until we set the final values in "dataref list" (linear delta values per cycle). **Provide:** Seconds = number Cycles = number "dataref list" - string of "key=value|key=value..." (*just like in the fn_set_datarefs() function*).
fn_remove_dref_from_interpolation_list(dataref list "s") [v3.306.3]	List of dataref names separated by a comma. Example: "**sim/weather/region/cloud_base_msl,sim/weather/region/cloud_tops_msl_m**"
fn_reset_interpolation_list() [v3.306.3]	Clear all datarefs from the interpolation list. Useful when you want to start a new interpolation and you want to make sure there are conflicting datarefs from previous commands.

Examples:
In mission file:

```
<dataref name="ground_speed" key="sim/flightmodel/position/groundspeed"/>
```

In script:

```
groundSpeed = fn_get_dref_value("ground_speed")
```

Other examples:

```
fn_set_datarefs("sim/weather/region/cloud_base_msl_m=6307,9010,13000|sim/weather/region/update_immediately=1")
fn_set_datarefs_interpolation(70, 3, "sim/weather/region/cloud_base_msl_m=6607,9310,13300|sim/weather/region/cloud_tops_msl_m=8250,10310,18100")
fn_set_datarefs_interpolation(1, 30, "sim/cockpit/autopilot/heading=170.0")

fn_remove_dref_from_interpolation_list("sim/weather/region/cloud_base_msl_m")
```

Generic Function List

(s)=string, (b)=bool, (r)=number

Generic Functions	
Navigation based functions	
fn_get_distance_to_coordinate (latitude , longitude)	Returns decimal value if success, returns "-1" if failed.
fn_get_distance_between_two_points_mt (lat1, lon1, lat2, lon2)	Returns the distance between two points in **meters.**
fn_get_distance_to_reference_point ("task name")	Works only for tasks that are linked to triggers. If the trigger is location based, the function tries to estimate distance. Returns decimal distance to a trigger based location (poly/rad/slope). If task is not valid function returns "-1"

fn_get_distance_to_instance_in_meters ("instance name")	Returns the distance in **meters** to an active 3D instanced object. Return "-1" if the instance is not active. Return "Nnn" if the instance exists.
fn_get_elev_to_instance_in_feet ("instance_name")	Returns the elevation difference between 3D instanced object and plane. Return "-1" if the instance is not active. Return "Nnn" if the instance exists. Example: when you want to test if a plane is near a 3D object but not just in distance but also in elevation.
fn_get_bearing_to_instance ("instance_name")	Returns the bearings in degrees between plane and instanced 3D object.
fn_add_gps_xy_entry (lat, lon, elev_ft, entry)	Adds a lat/long in FMS/GPS. "Lat" and "lon" are mandatory. Default behavior: will add coordinate to the end of flight plan. It is quite safe to always define elevation as 0. Entries can be between 0..99. All values are numbers, not strings.
fn_get_nav_info ("xplm_nav_name", "name fragment", "nav_ID_fragment", "get plane coordinate(b)" "lat(r)", "lon(r)", "frequency(r)", "navtype (r)")	Get information regarding navaid. Based on XPLMFindNavAid() More explanation below.
fn_execute_commands ("command1,command2…")	Execute x plane command or multiple commands using comma delimited. Will init mx_error_message if any command fails so you can debug your code. Failure of this function does not mean all commands fail, but maybe one of the commands could not be found.
fn_get_xp_version()	Returns X-Plane version as a number: 11 or 12..
End Mission Related Functions	
fn_abort_mission ("reason")	Abort mission function. You must send a text string to explain the reason why the mission has been aborted. This should be displayed in the Briefer.
fn_update_end ("tag name", "attrib name", "value") [v24.02.5] Example: fn_update_end ("fail_image", "file_name", "fail02.png")	This function will drill down from <end_mission> root element, and will update its sub element - "attrib name" with the provided "value". All parameters are strings.
Weather Related functions	
fn_set_predefine_weather_code(r) [v3.304.13]	Provide weather preset code: xp12 (0..8), xp11 (0..7)
fn_inject_metar_file ("metar file name") [xp11 only]	Inject metar file. Please make sure that this command won't be called each flight loop back or it will endlessly inject the same metar file until the mission will end.
Plane/Camera related function	

fn_position_camera (lat(r), lon(r), elev_mt(r), heading(r), pitch(r), roll(r))	Position camera in X-Plane world. All values are decimal numbers, no strings. Best used in 2D mode, VR can work but the Mission-X window won't be available outside of the plane.
fn_position_plane (lat(r), lon(r), elev_mt(r), heading(r), speed(r))	Position plane in X-Plane world. All values are decimal numbers, no strings.
fn_get_aircraft_model(**0**..9)	Returns the ***file name string*** of the airplane index. Default is **0** = your plane (not sending an index number will default to 0)
UI Related Functions	
fn_open_inventory_screen	Opens the inventory screen
fn_open_image_screen	Opens the map/image screen
fn_hide_3d_markers	Hide 3D markers
fn_show_3d_markers	Show 3D markers
fn_load_image_to_leg (file name(s)*, [leg-name(s)]) [v3.304.13]	Load an image into the "map/image" screen for a specific leg name. The file name is mandatory. If you won't provide the leg name, the plugin will use the current active leg.

On the next page we will list the Navigation function.

fn_get_nav_info() function example

The function mimics the LR SDK function, so you need to provide a value for each parameter or "empty string" if there is none ("").
The plugin, according to the info you will provide, will seed "script variables" in return for you to use.
Let's go over the function variables in detail:

fn_get_nav_info (...)

Attribute name	Expected Value
nav_name	Unique name (free string name)
name_fragment	Search navaid using name string, name can be partial
nav_id_fragment	Search navaid using ID string, ID can be partial
get plane coordinate	"true"|"false" Will be used as lat/lon when searching nearest NAV info.
Lat lon	Search navaid nearest to the coordination provided. If plane location were flagged as true, then it will use their value.
frequency	Filter navaids based on this frequency
navtype	A **number** representing the type of navaid to search/filter. You need to send a **number** based on the following list: Nav_Unknown = 0 Nav_Airport = 1 Nav_NDB = 2 Nav_VOR = 4 Nav_ILS = 8 Nav_Localizer = 16 Nav_GlideSlope = 32 Nav_OuterMarker = 64 Nav_MiddleMarker = 128 Nav_InnerMarker = 256 Nav_Fix = 512 Nav_DME = 1024 Nav_LatLon = 2048

The combination of the different nav attributes will filter and return the result as seeded script parameters

Example:

```
' search nearest NDB/VOR
' Use the table above for reference. We use type: "6" which is equal to NDB+VOR
if not fn_get_nav_info ( "nearest_VOR_to_plane", "", "", "true", "", "", "", "6" ) then
  print mxError;
else
  print mxNavType, ", ", mxNavLat, ", ", mxNavLon , ", ", mxNavHeight , ", ", mxNavFreq
, ", ", mxNavHeading , ", ", mxNavID , ", ", mxNavName , ", ", mxNavRegion ' debug
endif
```

List of parameters that will seed from *fn_get_nav_info*() function:

Seeded parameter name	type
mxNavType, mxNavFreq, mxNavRegion	Integer number
mxNavLat, mxNavLon, mxNavHeight, mxNavHeading	Decimal number
mxNavID, mxNavName	String

Functions to get/set Task properties

See <task> topic.

Functions to get/set Trigger properties

See <trigger> topic.

Functions to handle Inventory items

See <inventory> topic.

Functions to handle messages

See <message> topic.

Choice related Functions

see <choice> topic.

Flight Leg related Functions

See <leg> topic.

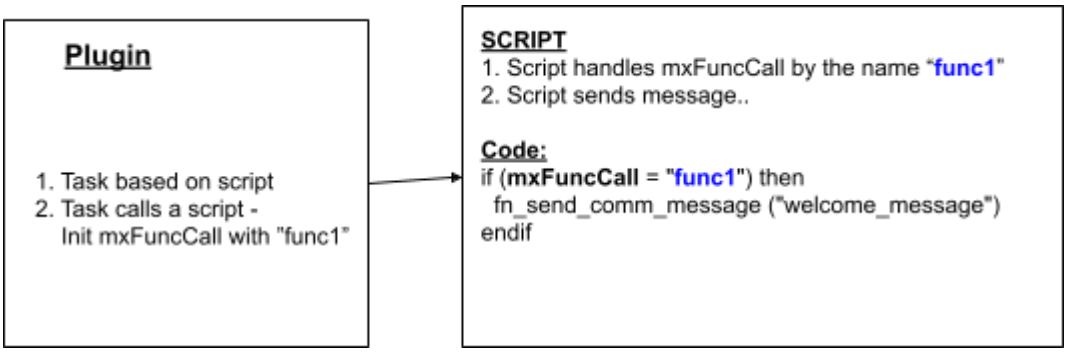
Instance related Functions

See <3d Object> topic (at the end of it)

Tips and Pitfalls in scripting

Repeated function call pitfall

Scripting can be a useful tool to create a simple or complex logic for your mission, but it is important to understand that with flexibility comes also complexity while designing the mission. Let's look at the flow of script again:



We have a task that is based on a script, and the script needs to send a "welcome" message. The plugin calls "*script.func1*", the "*mxFuncCall*" argument is being seeded with "*func1*" string and the script will call "*welcome_message*"..... ***Every flight loop***.

The problem with this code is that the plugin will always call the tasks script with the same function name - "func1". The script will always send the same message over and over until an objective will be achieved.

When writing such a script we need to "imagine" the flow of code and decide if repeated calls to "*script.func1"* is contributing to the mission to move forward or if it is a waste or a faulty design. We also need to think if all code in the script is handled correctly when it is repeated.

If we just want a "one time" message, we can use:

- *fn_send_comm_message("messageName", "trackName", true)* or *fn_send_text_message("textMessage", "trackName", true)*.
  This won't solve the repeated call to the "script.func1" but it will skip the message call after one execution.
- Other option is:
  Create a global variable and change its value once the message is being sent.
  Enclose the "message call" with an "if" statement and make sure it is called only when the global variable is "0" or "false" (for example).
  Please do remember to read the "global variable" parameter before the conditional statement and create it if it is not present.
- Maybe the best option for non mandatory tasks/triggers is:
  Set the task as complete or disable it so plugin will skip it in the next iteration:
  fn_set_task_property ("task1", "script_conditions_met_b", "true") **or**
  fn_set_task_property ("task1", "enabled", "false")

Using the "debug" build will show the different messages being sent from script in Log.txt file. This will assist in solving the branching and flow of the mission.

Run once is better than repeated execution
Whenever you can skip a script or part of a script code, then do so.
Use global variables to mark progress of a script or to branch to other parts of it. This way you can decide which part to execute or to ignore.

Before writing your own logic code, try to use built-in features instead
Example: if you want to check if a plane is on the ground, just flag the attribute instead of writing the same code in a script.
Scripts should compensate on the features the generic mission xml does not cover or support, for example - ground speed test.

Erase any global variable you won't need
Using global variables is quite handy especially for branching code, or to store data between different "legs" of the "flight_plan". Please do remember that for each type you should use a unique name.
Once the global variable is not needed any more, it is a good practice to release them.
Use "`fn_remove_global_xxx`" to release the correct parameter type.

If you have any additional "lesson learned" experiences, please share with me and I'll add it to the document.

MX-PAD

MX-Pad is your window that holds: a quick toolbar and your latest messages/conversations.
The MX-PAD is different in VR, it is embedded in the *flight information* screen.

To better understand MX-PAD, let's take a look what it is all about:

All the <message>s since v3.0.224.x are mxpad messages. So you can display a colored label for the message, or the plugin will default to "mx" or "radio".

Usage of MX-PAD

- The MX-PAD displays the last messages broadcasted to the user. Think of it as a history of messages. It can hold up to 20 messages.

- It has a quickbar that allows the simmer to open the Information and inventory screen when they are available.

- It allows the designer to create some sort of interaction with the "pilot" so they will feel as if they are really communicating with someone.
  Since v3.0.224.x, *use the enhanced "<message>" attributes to create simple dialogs without interaction or <choices>.*

"Mxpad" options were deprecated, use choices or a simple message to achieve the same.

Where to go from here...

After going over all the rules and short examples, it is time to:

1. Create a simple mission by yourself (maybe use the demo mission as reference).
2. Review the other documentations:
    a. Hand on mission creation.
    b. Scripting guide.

I strongly suggest just going over both documents and only then trying to create your first mission. Start from a simple mission (only one flight "leg", objective and task), base the task on triggers and do not implement any scripts.
Once it works, add some messages (add some triggers to send messages).
For one time messages you can:

1. Create a task based on a trigger.
2. Set the trigger area of effect.
3. Create a message.
4. Call the message from step 3 when the plane "enters" the trigger area.
5. **Make sure** the task attribute: "*force_evaluation"* is set to *"no"*. This way it will only fire once without repetition.

Once you will feel more comfortable with simple mission creation, and you better understand how to combine the different components, you should start and venture into the scripting world. That was a little dramatic, but if you want to achieve a higher granularity on simmer actions, scripting is the way to go.

Advance Topics:

Dynamic Messages

Full chapter can be read in the "Design Templates" document. Check Advance Topics.
Some designers might prefer to send a message relative to a trigger/task position.
Messages such as:

- Communicating a specific message relative to a location.
- Nearing a target/inventory area.

The "<*dynamic_message*>" allows us to use existing messages or define a new one relative to a certain location. The location is represented by a trigger name or a task name that is mapped to a trigger.
We can construct a "<*dynamic_message*>" in two ways:

Option 1: Define a message relative to a location, either trigger or task, and the message text you want to send.
Its format should look as follows:

```
<dynamic_message [relative_to_trigger="trig_name" | relative_to_task="task
name"]  length_mt= "" >{Message Text}</dynamic_message>
```

Attribute	Comment
relative_to_trigger	Message is coupled with a trigger.
relative_to_task	Message is coupled with a task
length_mt	The radius of the message area, used with triggers
message_name_to_call	Name of a predefined message
override_task_name	Custom task name, forces the plugin to use this name for the mandatory <task> element
override_trigger_name	Custom trigger name, forces the plugin to use this name for the trigger used with the dynamic message
message_name_when_entering_physical_area	Which message to call if we entered the physical area even if not all conditions were met. See designer guide for more information.
script_name_when_fired [v3.306.3]	Override script_name_when_fired
script_name_when_left [v3.306.3]	Override script_name_when_left
Message text if we are not basing it on a predefined message	
{Message Text}	Free text, make sure not to use problematic characters that might break the XML file.
Used with the	
sound_file_name	Provide a sound file name f

Explanation: The plugin needs to create a new "radius based trigger" relative to a known

target location. The best way to pick a target location is to provide its trigger name.

When calling a script, you can send a predefined parameter with the "%self%" keyword, that way a dynamic trigger name can be sent as a parameter or you could just use "mxCurrentTrigger" in your code instead. Depends on the "script code"

It is preferable to use "mxCurrentTrigger" wherever you can in your code.

Example:

```
script_name_when_fired="disable_trigger_generic|inTrigName=%self%"
```

Incorrect:

```
script_name_when_fired="disable_trigger_generic|inTrigName=mxCurrentTrig
ger"
```

mxCurrentTrigger can only be used inside a script.

All this "Hocus Pocus" is being built only when the simmer loads the mission (not while the mission is being generated), and you should be able to see it in the "missionx.log" file.

APPENDIX A - Designer Tools

Getting Coordinates and Weather from X-Plane

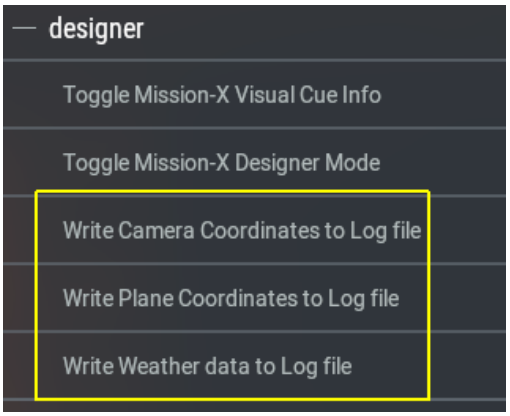
One of the actions that any designer needs to do when they are planning a mission is

1. Define specific locations for "tasks", "triggers" and inventories.
2. Define Weather.

There are few options to achieve that:

- *X-Plane "data output"* option: Problem, it only displays 2-3 decimal values, while we need 5-6 for better precision, and you can't copy from it.
- *DataRef Tool*: Great tool to test X-Plane "datarefs" and even "commands". Highly recommended, and you should be able to also copy from its screen. But it is not that great for bulk copy (as of this writing)
- *Mission-X own commands*: writes to the *missionx.log* file.

You have a few useful commands to pick coordinates or weather from X-Plane.
"mission/designer/write coordinate to Log file" - write plane coordinate.
"mission/designer/write Camera coordinates to Log file" - writes camera location.
"mission/designer/write Weather coordinates to Log file" - writes current weather datarefs based on predefined datarefs and write it like a script command.

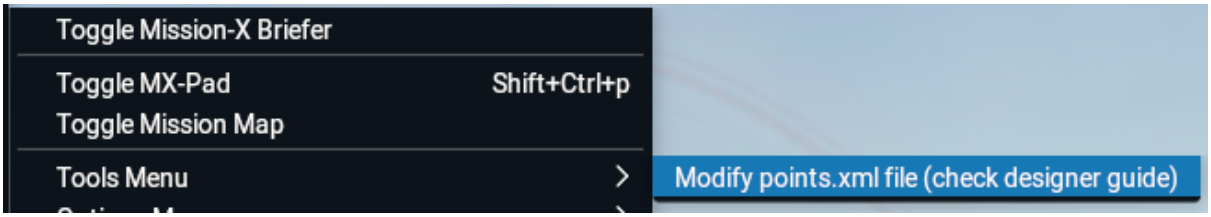


Map a key or a joystick button to them, and everytime you activate the command it will write to X-Planes missionx.log file information that you could copy and paste to your mission file.

```
lat="-19.253646" long="146.770929" elev_ft="18.01" heading_true_psi="20.79" groundElev="13.765" ft
```

If you use a Mac or Linux, you can just use "$ *tail -f missionx.log*" and copy the relevant output from the terminal.

Automatically modifying your XML template points



This tool was written specifically for template mission files. The idea is to auto test the elevation and "wetness" of any <point> element in the file that do not have a "template" attribute set and its "lat"/"lon" attributes are already defined.

Example,
if you have a <point> element with the following settings then it will be skipped, since "template" is set.

```
<point lat="47.196034" long="8.52531" template="land" radius_mt="50"
loc_desc=""/>
```

If you have a point with the following settings then the plugin will try to probe the coordinates defined in lat/long.

```
<point lat="47.196034" long="8.52531" template="" radius_mt="50" loc_desc=""/>
```

The plugin will check if the point coordinate is "**wet**" or has a "**slope**".
According to the result the plugin will decide whether to change the template to "hover" or "land" according to slope and terrain type values.
For "wet" and "slope" terrain (6 degrees) the template will be updated to "hover".

How to use it

1. You should place your file in "{xplane}/Custom Scenery/missionx/random" folder and rename it to "points.xml".
2. Log to X-Plane and position your plane in an airport near the locations of the <point>s in the file.
   For example, if the locations are around Seattle then pick KSEA (or othe icao in that area) .
3. *Pick "Mission-X => Tools => Modify points.xml" sub menu.*
4. Check the folder for the file: "points_mod.xml" which should have all the "template" attributes modified.

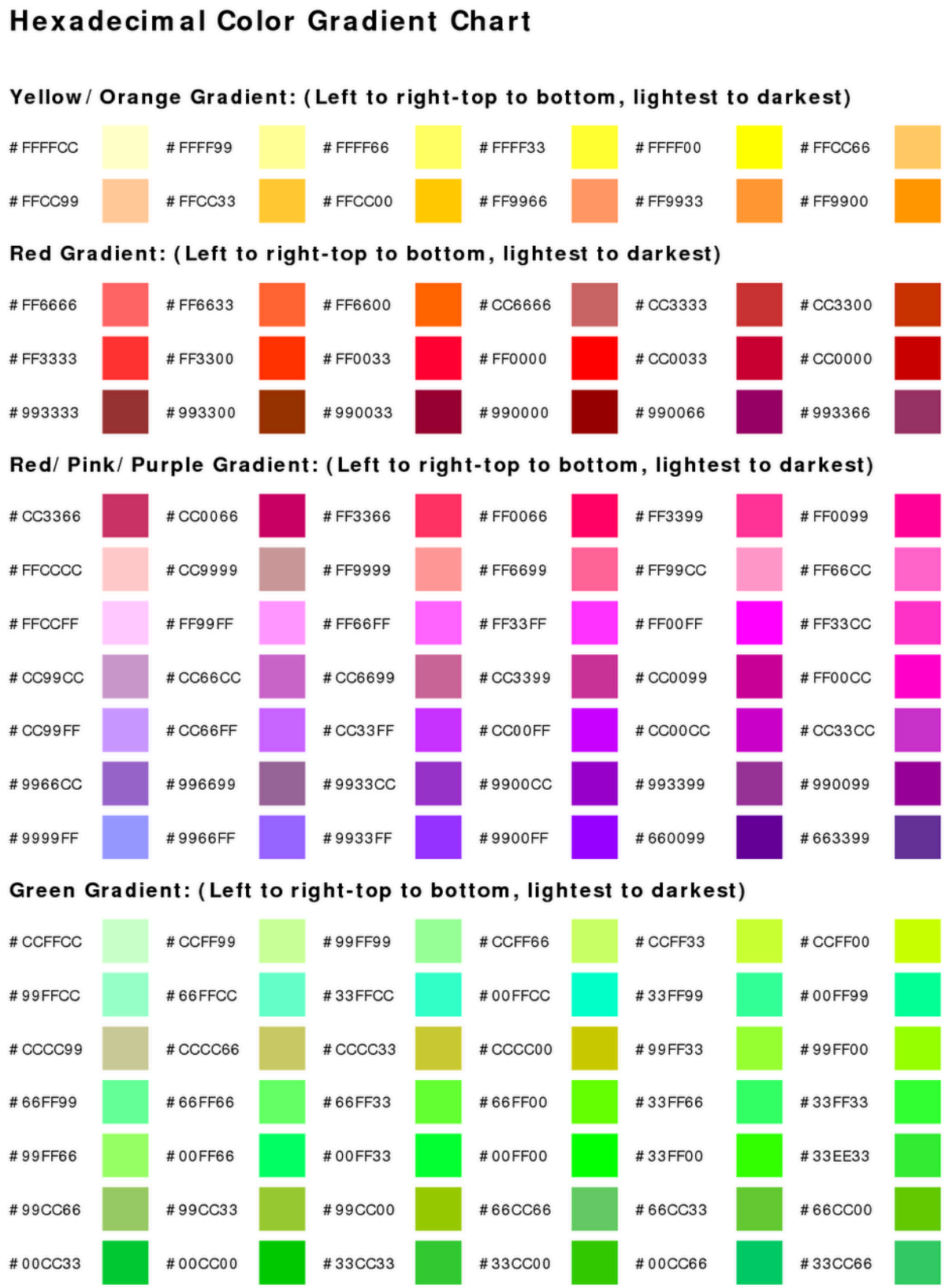
Caveat: To have good probing results you should move the plane to the region where they are located.
If the coordinates you have are not in the "DSF" files that were loaded by X-Plane, I'm not sure the results to some or all <points> would be representative of the real elevation.

APPENDIX B - SQLite custom functions

The Mission-X plugin added a few custom functions to the "sqlite" language.

Function Name	What it does
mx_bearing (lat1, lon1, lat2, lon2)	Find the azimuth to a point relative to a specific position.
mx_calc_distance (lat1, lon1, lat2, lon2)	Calculate the distance in "nautical miles" between two coordinates.
mx_is_plane_in_rw_area (pLat, pLon, rwcLat1, rwcLon1, rwcLat2, rwcLon2, rwWidth)	We provide the runway lat/lon cubic points and we return *true* if the plane point is in it.
mx_get_shortest_distance_to_rw_vector (pLat, pLon, rwcLat1, rwcLon1, rwcLat2, rwcLon2)	Basically we search for the "height" vector to the center of the runway. We use this to determine the plane touchdown relative to the center of the runway.
mx_is_plane_in_airport_boundary (lat1, lon1, 'lat,lon|lat,lon|...')	We provide a plane position and a set of "lat,lon|lat,lon" string text that represents the nodes of the boundary. The function tests if the plane position is inside the bounded nodes.
mx_get_center_between_2_points (lat1, lon1, lat2, lon2)	Returns a string coordinate in the format "x,y" that represents the center of a vector. You need to provide the lat/lon of two points.
mx_get_point_based_on_bearing_and_length_in_meters (lat1, lon1, bearing, distance in meter)	Find a point in space with a given azimuth and distance relative to a coordinate. Example: place an object relative to current plane position.

APPENDIX C - Hex Color Table



https://htmlcolorcodes.com/color-chart/web-safe-color-chart/

APPENDIX D - Mission Generator

Check "User Guide" and the "generating Random Missions" document.

APPENDIX E - Pilot's Alphabet

A Alpha	**N** November
B Bravo	**O** Oscar
C Charlie	**P** Papa
D Delta	**Q** Quebec
E Echo	**R** Romeo
F Foxtrot	**S** Sierra
G Golf	**T** Tango
H Hotel	**U** Uniform
I India	**V** Victor
J Juliet	**W** Whiskey
K Kilo	**X** X ray
L Lima	**Y** Yankee
M Mike	**Z** Zulu

Revision Table

Revision No.	Last Modified	What was changes
0.35	16-feb-2025	* Added <outcome> attributes: "set_other_tasks_as_success" and "reset_other_tasks_state" to simplify the "land_hover" generated medevac missions, only for helos. * Added <outcome> attribute: "set_other_triggers_as_success", not tested yet.
0.34	22-dec-2024	* Added <compatibility> element to GlobalSettings. * New Inventory layout was added, this affects "embedded scripts" * Added Inventory functions: "fn_get_inv_layout_type" and "fn_move_item_to_plane". * Extended function "fn_move_item_from_inv" to support inventory layout that supports "station". * Added "mandatory" and "target_inventory" attributes in the "<item>" element.
0.33	01-nov-2024	* Updated the inventory image thumbnail sizes to 50x40.
0.32c	17-may-2024	* Added more explanations and some syntax fixes.
0.32b	11-mar-2024	* Added sqlite plugins custom function names, added to the SQLite db.
0.32a	24-feb-2024	* Fixed attribute name in document: "mute_xplane_narator" to "mute_xplane_narrator".
0.32	19-feb-2024	* Added <end_message> element explanation. * Added explanation of "fn_update_end()" function.
0.31	10-feb-2024	* Extended "fade_bg_channel", supports the "%self%" keyword. * Extended "track_instructions" commands. Added loop ("l") command.