H145 Mission System Documentation

This documentation is early and subject to change.

Last Update: 2022/6/23

Basic mission details

A mission json file is referred to as a Mission Descriptor. It can be loaded into H145 and then operate alone while the user conducts the mission.

title	Title used when displaying your mission in a list
aircraft	Must be H145 (array of supported aircraft)
applicable	Array of variants. If omitted, all variants will apply. Inapplicable missions will be hidden in the mission catalog. EMS FIREFIGHTER
api_version	Must be 0.1
start_info	The start location or start locations can be specified. This will prevent showing the mission in the Library as it has a natural start point on the map. If you do not specify a start_info, then you will use the library to begin your mission.

	`location`
	Specify [lat, lon] for the fixed starting location.
	
	`icon_src`
	Specify an HTTPS or data URI. This icon will be shown on the map. Suggested size 32x32px.
	
	`query`
	A data query in the same format as used below in missions

Loading missions from a server

To load missions from a server, do not provide locations/objects/threads/objectives, instead provide a URL which is a websocket server. When the user selects the mission your server will be contacted and at that point you will be able to manage the mission system indefinitely until the user selects another mission manually.

url	"localhost:40510"

Authoring mission packs

Missions can be added to any other Community package or be authored alone, the only thing to do is create an `hpgmission` folder within your package, and place a folder hierarchy below with your mission json files. All contents (folders and json files) below `hpgmission` across all Community packages will be merged into the catalog list. Feel free to create a folder structure for regions or otherwise create organization.

Mission sections

locations	Table of locations referenced throughout the mission file. These are locations like “accident_site” or “hospital_helipad” that mark the coordinates. You can easily copy/paste a location from Bing maps or Google maps by right clicking and selecting the coordinates from the menu.
objects	Table of dynamic objects created when the mission starts. The objects have a title which is what identifies them in MSFS (like an airplane), and they have a default location you may place them at.
threads	Table of background execution threads which occur regardless of the current objective. This allows parallel processing of logic. You may wait for a specific variable to be true, enter some processing, and then quit forever or start the process again. This can be used to design triggers and add other logic to your mission, like enabling a sequence of events only when the user enters an area, regardless of where they are in the mission objective list.

objectives	List of Sequential tasks the user will work through. Every mission has at least one objective and when the list of objectives is complete, the user has finished the mission. Each objective itself is a set of commands which execute sequentially. You can direct the user to an area and then proceed to the next objective only when they have arrived at the area of interest.
userActions	TODO - Not yet documented

OBJECT

Objects are created when the mission starts and manipulated throughout the mission. The VAR 1 variable is commonly used to configure the visual state of the object.

title	string	Title from an aircraft.cfg, registered in MSFS. See the section **Creating Dynamic Objects**.
location	LOCATIONREF	Location to create the object. **Optional**: objects without a location will be created at Null Island [0, 0] and may be later moved by using move_object.

Special object variables

These variables are interpreted by the system in a special way.

Name	**Function**
VAR 1 VAR 2	Mapped to simulation vars unique to the object: VAR 1: (A:GENERAL ENG THROTTLE LEVER POSITION:1, percent) VAR 2: (A:SPOILERS LEFT POSITION, percent) These variables are unique for every object and will be available in the model behaviors XML. This allows each object to have independent visual states and behaviors.
COUPLED	Object user coupling mode. When an object is coupled it will be modified automatically based on the coupling state. 0: No coupling 1: Couple to hoist position - Object will be continually snapped to the position below the hoist 2: Couple to external cargo position - Object will be continually snapped to the position below the cargo hook 3: External cargo position auto-couple armed - Object will switch to coupling mode 2 automatically when within range. 4: Firefighting target (fire)

	- The user may use the Bambi bucket to reduce VAR 1 (quantity of fire) for this target. VAR 2 is set to the most recent quantity reduction by the user bucket dump. 5. Firefighting pool - VAR 1: Radius of pool (METERS). VAR2: Depth of pool (FEET, negative)
MODE	Object mode. The mode is used to control the physics and behavior of the object. 0: Hold position on ground 1: Repositioning mode - Use LAT/LON to configure the next location, and then set mode to 0 to switch back to ground hold. 2: 3-axis Velocity control - Use VELOCITY X, VELOCITY Y and VELOCITY Z to control the object physics over time 3: MSFS default Physics
WP INDEX	Activation navigation index. Set index 1 to activate the waypoint engine and cause the object to rotate on the yaw axis to orient such that velocity z will drive the object to the waypoint. 0: not active 1-5: navigation to waypoint 1-5. The waypoint engine will set the WP INDEX to 0 upon reaching a situation where the next waypoint (WP INDEX + 1) is a waypoint at location 0,0. The waypoint engine will also set VELOCITY Z to 0 at this time.
VELOCITY X VELOCITY Y VELOCITY Z	Object velocities. Only applicable when MODE=2. These velocities will be sent directly to MSFS to instruct the object movement.

THREAD

Threads are background command lists which execute independently of the currently active objective. Threads may be used to schedule activities regardless of where the user is in the objective list

interval	milliseconds	Update interval (higher is better for performance)
commands	COMMANDLIST	List of commands, execute in order.

OBJECTIVE

Your mission must have at least one objective or it will complete immediately after starting. The objectives each have a list of commands and when one objective is completed the first

command in the next objective will be started. When the last command in the last objective finishes, the mission is complete and will end.

title	string	Text to display to the user for this objective.
commands	COMMANDLIST	List of commands, execute in order.

Commands

Commands are executed one at a time from a command list, and each command may execute nearly instantly or take some time to finish.

Control Flow

if

Choose between one set of commands or another set of commands based on a condition.

if: QUERY, then: COMMANDLIST, else: COMMANDLIST	The IF statement will execute the subsequent command list (either the then or else list) and then return processing to the next command after the if. Nesting of multiple IF statements is supported.

while

Continue executing the commands while the condition is true

while: QUERY, TEST, do: COMMANDLIST	If the query+test evaluates to true then the command list will be run one time, then the query+test evaluated again. **Note** your command list needs to have sleep or other time-consuming commands or you risk a busy loop or fresh data being unavailable.

sleep

Delay for some time until executing the next command.

sleep QUERY	Sleep for some amount of time in **seconds**.
sleep “forever”	Sleep indefinitely

General

set

Set a variable in the global scope or on an object.

set {var: [“var_name”, “var_type”], value: QUERY}	Set simulator variables. - L:Vars, A:Vars - H:Events, K:Events
set {object:”object_name”, var: “var_name”, value: QUERY}	Set object-specific variables. **object_name** is from the objects table. **var_name** may be: VAR 1 VAR 2 COUPLED MODE WP INDEX VELOCITY X VELOCITY Y VELOCITY Z
set {object:”object_name”, var: “var_name”, value: {location: LOCATIONREF}}	Set location to an object-specific variable. **var_name** must be empty (“”) to set LAT/LON or: WP 1 WP 2 WP 3 WP 4

wait_for

Do not proceed until a specific condition is true.

wait_for: QUERY, TEST	Compare QUERY to TEST and keep checking until it is true, only then proceed.

Object & Thread Manipulation

create_object

Create an object immediately and insert it into the objects table

create_obejct: {name: “my_object”, title: string, fallback_title: string, location: LOCATIONREF}	Create a new object. The parameters have the same meaning as creating an object directly at mission start.

destroy_object

Remove an existing object

destroy_object: “my_object_name”	Remove an object from the world and also remove its name from the objects table. After this call, It is safe to immediately create a new object with the same name.

move_object

Move an object to a new location.

move_object: “object_name”, to:LOCATIONREF	Move the specified object to the new location, returning it to mode 0.

create_thread

Create a new background thread execution queue and start it immediately.

create_thread: {commands: COMMANDLIST, interval: 100}	Create a new thread and insert its name into the threads table. The parameters are the same as starting the thread at the beginning of the mission.

Cockpit Presentation

set_route

Set the map route (magenta line).

set_route LOCATIONREF	Set the map route line to a single target

set_hover_display

Set the hover display (crosshairs) target and distance.

set_hover_display {range: QUERY, target: LOCATIONREF}	Set hover display (crosshairs) target and range. Range units: Nautical Miles
set_hover_display {range: 0}	Clear the hover display

set_df

Set the Direction Finder information. The DF displays a bearing pointer on the MFD in the cockpit.

set_df: {location: LOCATIONREF, freq: FREQ}	FREQ is a decimal value which is displayed on the MFD.

set_message

Display a text message on the mission map

set_message: {text: “my message here”}	Message will be displayed until the end of the mission or until another set_message replaces it.
set_message: {text: “my message here”, timeout: 9000}	Timeout is in milliseconds, in the future it will change to seconds.
set_message: {text: “”}	Clear message.

set_ui

Set the bottom footer data in the mission app.

set_ui: {footer_left_label: string, footer_left_value: string}	Set the label and value of the left footer cell.
set_ui: {footer_right_label: string, footer_right_value: string}	Set the label and value of the right footer cell.

set_map

Draw icons and lines on the mission map.

set_map: {add: {line: {points: [LOCATIONREF1, LOCATIONREF2, …], stroke: { color: '#FF33FF', width: 3 }, id: 'my_route'}}}	Draw a line between two points.
set_map: {remove: {line: {id: 'my_route'}}}	Remove previously drawn line.
set_map {add: {point: {location: LOCATIONREF, id: “my_icon”, icon: URL}}}	Draw an icon at a point. URL may be HTTPS or a data-uri.
set_map {update: {point: {id: “my_icon”, icon: URL}}}	Update the icon URL for an existing point.
set_map: {remove: {point: {id: “my_icon”}}}	Remove previously drawn point.

set_modal

Present a message to the user where they can pick a button to dismiss it.

set_modal {title: string, text: string, options: [MODALBUTTON1, MODALBUTTON2, …]}	Title: title of message Text: body of message Options: Represents the buttons at the bottom of the message. You must have at least one for the user to be able to dismiss the message, and you may have a maximum of 4.

Set_modal MODALBUTTON

The options in the modal dialog.

{ text: string, commands: COMMANDLIST, style: MODALSTYLE}	**MODALSTYLE** (blank) theme default “primary” Green “Secondary” Gray “Danger” Red

Other

create_location

Create a location based on a set of input configurations (zones).

create_location: “location_name”, zones: [ZONE1, ZONE2, …]	Choose a random zone from the zones and create a location based on its configuration.

create_location ZONE

zone: {zone_type: ZONETYPE, location: LOCATIONREF, radius: RADIUS, minRadius: MINRADIUS, commands: COMMANDLIST, query: DATAQUERY	Choose a random zone from the zones and create a location based on its configuration. **ZONETYPE:** random_point or query_clsoest_result or query_random_result **random_point**: Generate a point anywhere within the circle between MINRADIUS and RADIUS. **query_closest_result** Query expanding outwards in concentric circles until the query has suitable results. **query_random_result** Query using MINRADIUS and RADIUS to discover ALL results within the circle and then pick one at random.

	RADIUS and MINRADIUS are in meters.

create_fire

Create a random cluster of fires based on the configuration

create_fire: LOCATIONREF, title: “object title”, showIcon:true|false, size: FIRESIZE	FIRESIZE is the number of fires to spawn. Title is the actual fire object to be spawned, usually **Airbus H145 Fire** but can be an alternate. fn:all_fires_extinguished is paired with create_fire and can be used to check for the user having fought all of the fires.

remote_notify

Notify the remote mission server with some information from the simulator. Not applicable for missions without a mission server.

remote_notify: “tag_name”, params: [QUERY, QUERY, …]	Send a set of values to the remote mission server. **Only applicable when a mission server is used.**

Predicates

QUERY

Look up a variable, generate a number, do math, or provide a hard-coded value. Do not confuse a QUERY with a DATAQUERY.

{var: [“simvar_name”, “simvar_units”]}	Get simulator variables: - A:Vars, L:Vars
{rand: [QUERY(min), QUERY(max)]}	Generate a random number between QUERY and QUERY (min and max)
{clamp: [QUERY(value), QUERY(min), QUERY(max)]}	Limit a value between min and max
{scale: [QUERY(a_value), QUERY(a_min), QUERY(a_max), QUERY(b_min), QUERY(b_max)]}	Convert a value (a_value) from one scale (a_min to a_max) to another scale (b_min to b_max). Values outside of the range (a_min to a_max) will

	be clamped.
{round: QUERY, to: 1}	Round a value to a specified number of digits.
{floor: QUERY}	Floor a value (round down).
{add: [QUERY, QUERY]} {subtract: [QUERY, QUERY]} {multiply: [QUERY, QUERY]} {divide: [QUERY, QUERY]}	Arithmetic operations.
{object:"object_name", var: "distance"}	Query a variable specific to an object. **var** may be **distance** to query in nautical miles. **var** may also be any other object variable.
{fn: "HOIST_SEND_TO_GROUND", params:[]}	Call special built-in functions: HOIST_SEND_TO_GROUND HOIST_REEL_UP_AND_STOW
{has_location: "location_name"}	Test if a location exists or not. This is used for example to allow optional base return legs in default templates.
{location:LOCATIONREF}	Only applies when a LAT/LON is expected (rare)
any number	A number like 0 or -5 or 100.5 is a valid query (and very common).

TEST

Test provides the logical operators to compare one query to another.

eq QUERY	Equal to QUERY
ne QUERY	Not Equal to QUERY
gt QUERY	Greater than QUERY
gte QUERY	Greater than or equal to QUERY
lt QUERY	Less than QUERY
lte QUERY	Less than or equal to QUERY

LOCATIONREF

Specify a specific location, or derive a location relative to another location.

[43.7927201, -122.734048, 359]	[Latitude, Longitude, Heading]

“location_name”	Lookup in locations table
{bearing: 0, dist: 0}	Bearing and distance to user aircraft
{bearing: 0, dist: 0, object: “object_name”}	Bearing and distance to object
{closest: [LOCATIONREF], to:LOCATIONREF}	Calculate the closest location from a list of locations, relative to the TO location. The result is a location which can be set as a waypoint. This can be used to “choose the closest of three targets” and build a dynamic waypoint plan.

DATAQUERY

A data query is an OSM Overpass API query

“[out:json];node({{bbox}})[man_made=silo];out center;”	Basic string query. Test at Overpass Turbo: 1. https://maps.mail.ru/osm/tools/overpass/ 2. https://overpass-turbo.eu/

query: {
query:
"[out:json];(area({{bbox}})[amenity=hospital];
area({{bbox}})[aeroway=helipad];); out center;",
"groups": [
{amenity: "hospital"},
{aeroway: "helipad"}
],
logic: {"intersection": 0.2}
}
	Advanced intersection query.
	
	Locate hospitals with a helipad 200 meters or closer, otherwise exclude the result as not applicable.

Dynamic Object Library

H145 Crew

The H145 Crew object contains the crew, pilots and stretcher. The visual states below may be configured for the various standing/walking/waving states.

title	$TITLE Crew

	Airbus H145 ADAC Crew Airbus H145 DRF Crew Airbus H145 CMH Crew Airbus H145 HeliOtago Crew Airbus H145 Norsk Luftambulanse Crew Airbus H145 Bundeswehr Crew Airbus H145 CAL FIRE Crew Airbus H145 San Diego Gas Electric Crew

NOTE: Livery authors should add their title to allow **$TITLE Crew** to work, which is automatically replaced based on the livery name, and with a check for the livery author to have provided a crew title replacement within their livery json file. See the main user guide livery authors section for more information on this.

Visual states

VAR 1	-1: Hidden 0: HEMS standing 1: HEMS standing with (backpack) 2: HEMS walking 3: HEMS walking with (backpack) 4: HEMS crouching on ground 5: HEMS crouching on ground with (backpack) 6: HEMS crouching on ground with (backpack on ground) 7: HEMS waiting 8: stretcher no-patient 9: stretcher patient 10: stretcher walking no-patient 11: stretcher walking with patient 12: stretcher standing1 no-patient 13: stretcher standing1 with patient 14: pilot standing 15: pilot waving 16: pilot walking
VAR 2	**Only applicable to VAR 1 values of 14-17.** 0: Black pilot with headset 1: Black pilot with helmet 2: White pilot with headset 3: White pilot with helmet

H145 Injured Human

The injured human object is a human laying on the ground waiting for medical attention.

title	Airbus H145 Injured Human

Visual states

VAR 1	-1: Hidden 0: Injured human in pain 1: Injured human packed into hoistable stretcher

H145 Waving Civilian

The waving civilian is a human standing waving, attempting to get help for his fallen friend.

title	Airbus H145 Waving Civilian

Visual states

VAR 1	-1: Hidden 0: Civilian waving **NOTE:** Use L:WAVING_CIVILIAN_STOP to 1 to stop waving

H145 Flare

This is a marine flare with orange smoke.

title	Airbus H145 Flare

Visual states

VAR 1	-1: Hidden 0: Smoke auto (ON for high visibility setting, OFF for realism) 1: Smoke on (ON fo both setting positions)

Creating Custom Dynamic Objects

You may create your own dynamic mission objects that H145 can spawn. They can use the same COUPED and MODE flags as the built-in objects.

Unpack the **Mission Object Sample** from **Tools**. Included in the sample is a blender asset which has already been exported for you into the MSFSPackage, which is an SDK project which you load in MSFS to compile the asset and produce a package for redistribution.

The procedure is as follows:

- Prepare an asset. Follow `Blender Asset\Ambulance.blend` as an example.

- Export your asset into `MSFSPackage\PackageSources\SimObjects\Airplanes\sample-ambulance\model\H145_GenericVehicle`
- In MSFS, enable developer mode and load the project `MSFSPackage\MSFS_DynamicObjectSample.xml`
- Copy the output package `hype-mission-dynamicobjectsample` from `MSFSPackage\Packages` to your Community folder.

Now the object is registered with the simulator and available for creation. Using Scenario Editor, use the More Objects toolbar item and find **Sample Ambulance** in the list. The object can be placed and used in H145 missions now.

In order to package multiple objects you will need to change the name. To change the name of your object you will need to edit these locations under `MSFSPackage\PackageSources`:

File	**Text to change**
ExtraFiles\hpgmission\packageObjects.objmeta	Airbus H145 Ambulance Sample
SimObjects\Airplanes\sample-ambulance\aircraft.cfg	Airbus H145 Ambulance Sample **Tip:** isUserSelectable=1 will allow you to see the object directly, and isUserSelectable=0 will ensure that your distributed package doesn't have extra stuff showing up in the aircraft selector menu for the end user.

To combine multiple assets into one package, use MSFSLayoutGenerator.exe to update the layout.json after combining all of the output folders.

Mission Server

A mission server may dynamically generate and apply mission descriptors as well as send other commands and observe status. The server is essentially just a websocket server which listens for the simulator to connect and then speaks a JSON RPC type protocol.

A very simple **Mission Server Sample** in node.js is included in the **Tools** folder.

Commands sent from the H145 to the mission server

{control_msg: "hello"}	After connecting the H145 will alert you that it is ready for you to send a mission

{control_msg: “canceled_by_user”}	The H145 is alerting you that the user has selected another mission and you are no longer active. The connection will disconnect after this message
{remote_notify: “tag_name”, params:[QUERY1, QUERY2, …]}	Sent from the active H145 mission. This is data that you would like to be advised about. Remote_notify can be used within objectives or configured in a background thread to provide notifications for specific conditions and data.

Commands sent from the server to H145

{load_mission: MISSION_DESCRIPTOR}	Request the H145 to clear the current mission and then load your new mission immediately.
{exec_commands: [COMMAND1, COMMAND2, …]	Request the H145 to execute a free-standing command list. This list executes in parallel with the current objective and all background threads.