Phone App Specifications

Introduction

MiOS is a lightweight home automation system. The 'brain' of the MiOS software is the back-end, the engine, which runs stand-alone on a variety of internet-connected devices, such as PC's, MAC's, Wi-Fi access points, and dedicated home automation gateways. MiOS also includes a portal at mios.com, which acts as a secure relay to MiOS systems that may be behind firewalls. Users can register for an account at mios.com, and that account can be linked to one or more MiOS systems to provide the user remote access to their Mios system with a simple http get request. The URL you will open is generally data_request?id=xxx, where xxx is some sort of request or control command.

This document describes how to create a simple user interface to control a MiOS system using the simplified 'lu_sdata' (LuaUPnP Simple Data) request. This document describes a simple control-only user interface, meaning it lets the user run scenes and control devices, but does not provide any means to change configuration or do advanced tasks. Whatever device is running the user interface (cell phone, web page, television, etc) will be referred to as the 'controller', and the MiOS engine, or system, that is being controlled by the controller is the 'engine'.

You can test out all the commands in a normal web browser. For example, if your engine is on the same local network as your web browser, and your engine has the IP address 192.168.2.150, you can view the status of all scenes and devices by opening http://192.168.2.150:3480/data_request?id=lu_sdata in your browser. If you want to turn on device #5, then you can open http://192.168.2.150:3480/data_request?id=lu_action&DeviceNum=5&serviceId=urn:upnp-org:serviceId:SwitchPower1&action=SetTarget&newTargetValue=1

The user can control the system in one of two ways: 1) Directly using the IP of the MiOS system if the phone is connected to the same LAN by Wi-FI, or 2) if the phone is not on the same network (say it is using a 4G network instead), it can control it through remote forward server that acts as a relay.

Data to store on the phone

The following data must be stored on the phone in persistent storage, such as in a file or in a registry. When the application loads, the contents of this should be cached in memory variables that are shared and accessible by all the threads in the system.

Username
PK_Device (our serial number)
HWKey (hardware key
LAST_IP (the IP of the phone the last time it was run)
LAST_WIFI (y/n flag to indicate if the phone was connected to Wi-Fi last time)
LAST_ENGINE_CONNECT (L=Local, R=Remote, N=No engine)
LAST_PK_Device (the last serial number of the gateway we connected to).

Network Retries

If a URL has a server with a 1 or 2 on the subdomain, and you cannot get a response on server 1, then try server 2 and vice-versa. We are going to add more servers as we need to scale our solution, but we will always stick to a pair where the odd number is the lower one. For example, if you get a URL to autha8.mios.com and it fails, try autha7.MiOS.com. If you get a URL to provision3.mios.com, and it fails, try provision4.mios.com. Always try both URLs twice unless otherwise noted, and if they fail all 4 times, display an error to the user: “Cannot reach the MiOS server. Please confirm your phone has a valid internet connection.” with an OK button that exits the app. Note that sometimes you will pick the server to connect to, and sometimes it will be given to you by another server. For example, to locate the list of MiOS systems, you use either account1 or account2. It really does not matter which you choose first. But that locator may contain a pointer to the relay server and explicitly specify relay2.mios.com. In this case, when a specific server is specified, always use that one first (try relay2.mios.com twice). If that fails, try relay1.

Predefined Variables

There are a number of predefined variables in each phone app:

- **Server_Autha** e.g. us-autha11.mios.com
- **Server_Autha_Alt** e.g. us-autha12.mios.com
- **Server_Authd** e.g. us-authd11.mios.com
- **Server_Authad_Alt** e.g. us-authd12.mios.com
- **AppKey** e.g. “oochoh2thu3feZei” (MMS application key, used to identify the app with MMS servers)
- **PK_Oem** e.g. 7 (numerical value with the id of the app oem)

Application Flow

1. Authentication

1.1 Login

Login is done with the MMS call **autha/auth/username/x GET**:
https/[Server_Autha]/autha/auth/username/x?SHA1Password=y&PK_Oem=z&AppKey=w

with:

- x is the username (urlencoded)
- y is the encrypted password per MMS spec - SHA1(username + password + salt),
- z is the PK_Oem (hardcoded for each app instance),
- w is the app key assigned to the application (also hardcoded for each app instance).

MMS call will return http status 200 in case of success, or a different status in case of failure. If login is successful, the http response of the request will contain the identity token used for further authentication on MMS cloud servers.

1.2 User Registration

User registration is done with MMS API call **autha/auth/username POST**
https/[Server_Autha]/autha/auth/username POST passing an argument:

```
json = {
        Username: x,
        Email: y,
        Password: z,
        PK_Oem: w
}
```

You should get back a http status of 200 for success or a different status for failure. In case of success, response content will contain a Json string with identity tokens.

1.3 Identity validation with existing tokens

When tokens already exist on the phone, you can checked if they are still valid by calling MMS API call **info/session/token GET**

https/[Server_Autha]/info/session/token passing identity tokens as headers.

It will return http status 200 if successful and response content with a string which is the session key for autha server.

1.4. Get the list of devices

MMS call: /account/account/account/x/devices GET

Description: call https://[Server_Account]/account/account/account/x/devices , with x being PK_Account returned in the Identity token. For each device returned in output Json there is a Server_Device node which gives the host of the device MMS server and a PK_Device which is the unique identifier of the device.

2. Select gateways list (MMS locator service)

The list of devices to which you are able to connect is retrieved with the MMS API call **locator/locator/locator GET**

https://[Server_Authd]/locator/locator/locator GET passing as argument PK_Account which is read from the identity token.

This URL returns a list of all MiOS systems to which you can connect in this format:

```
{
        “Devices”: [

        {
```

“MacAddress”: “00:26:b8:71:ac:5e”,

“PK_DeviceType”: 1,

“PK_DeviceSubType”: 2,

“Server_Device”: “us-device11.mios.com”,

“Server_Event”: “us-event11.mios.com”,

“PK_Account”: 2945,

“Server_Account”: “us-account11.mios.com”,

“InternalIP”: “192.168.8.60”

“LastAliveReported”: “date time format”

}]
}

3. Connect to a gateway

Once a gateway is selected, set the **LAST_PK_Device** variable to the PK_Device of that gateway.

3.1 Connect Locally

If the selected unit has a non-empty locator service in the InternalIP field, set the base URL for all data commands which are sent to the unit as **http://[InternalIP]/port_3480/**
This will be referred to as the **CommandUrl**.

If the direct connection fails, you should automatically switch over to the remote connection. If the network settings change, for example - a mobile phone switches from 4G to Wi-Fi, try the direct connection again.

3.2 Connect remotely

If the selected unit has an empty InternalIP, you have to connect to the unit using relay (remote access) servers. To find out the servers for the selected unit, make the MMS call **device/device/device/x GET:**
https://[Server_Device]/device/device/device/x , where Server_Device is defined in the locator output, and x is PK_Device of the selected device.

The success response is in this format:

{
“PK_Device”: “1234”,
“ExternalIP”: “xx.xx.xx.xx”,
“AccessiblePort”: “21238”,
“InternalIP”: “192.168.8.10”,
“AliveDate”: “2013-07-19 10:45:35”,
“FirmwareVersion”: “1.6.204”,
“UpgradeDate”: “2013-06-13 13:50:17”,
“Uptime”: “4days5_04”,
“Server_Device”: “us-device11.mios.com”,
“Server_Event”: “us-event12.mios.com”,
“Server_Relay”: “us-relay11.mios.com”,
“Server_Support”: “us-ts11.mios.com”,
“Server_Storage”: “us-storage11.mios.com”,
“Server_Device”: “us-device11.mios.com”,
“Timezone”: “PST8PDT,M3.2.0,M11.1.0”,
“LocalPort”: “80”,
“ZWaveVersion”: “3.20-I1_r0_h7039_n1_s0_is0_o0_si0_rp1_su”,
“FK_Branding”: “1”,
“Platform”: “MiCasaVerde%20VeraLite”,
“UILanguage”: “en”,
“UISkin”: “smartrg”,
“HasWifi”: “0”,
“HasAlarmPanel”: “0”,
“UI”: “6”,
“EngineStatus”: “0”
}

Set the Server_Relay variable to the field with the same name from the Json output, then call **info/session/token GET** on the relay server to get a session key for that server

https:/[Server_Relay]/info/session/token passing identity tokens as headers.

Once you receive a session key on a relay server, set **CommandURL** to **https://[Server_Relay]/relay/relay/relay/device/[LAST_PK_Device]/session/[Server_Relay_MMSSession]/port_3480/**

3.3 Connect to a device through a relay to get data

MMS call: /relay/relay/relay/device/x/session/y GET

Description:

https://[Server_Relay]/relay/relay/relay/device/x/session/y/port_3480/data_request?id=sdata with x being PK_Device selected on #3 and y MMS session for relay server retrieved on #6.

4. Polling Loop

The software must be written so there is a background poll loop that continuously polls the MiOS engine for changes and which updates an object-oriented class structure containing the list of sections, rooms, scenes and devices, as well as the state of the engine (ip address, running state, etc). This must run in a separate thread that is completely independent of the front end. This thread may block for extended periods of time while it is waiting for data from the MiOS engine. This thread must never wait for user input, it should always be running continuously. And the front-end code which is handling the user interface must never call a function in the back-end polling thread that blocks. In other words, the front-end user interface should always be responsive to the user and should never be waiting for the back-end to do something. When the front-end needs to change something in the back-end, such as specifying the IP address the back-end should use for polling, the front-end should simply update variables in classes or objects that cause the back-end polling loop to do something.

So the bottom line is the back-end and the front-end must be separate threads and neither must wait for the other one.

If the user has not yet selected the MiOS engine to which they want to connect, then the back-end polling loop will remain idle until a MiOS engine is selected. The thread will start up once an engine has been selected. Since the http request that the polling group are making may block for up to 60 seconds, the front-end needs to have a way to terminate the back-end polling loop and abort any pending, blocking requests, so that if the user chooses to quit the application, the application must be able to terminate immediately and not wait for the back-end polling loop to finish a pending request.

The back-end will retrieve any changes made to the system - for example, if a light is turned on and off, or the temperature changes on a thermostat. The back-end must update variables in shared objects and have a way to notify the front-end that those objects have been updated. This causes the front-end to wake-up and immediately re-render the devices which have

changed. For example, as soon as a light turns on, the icon on the front-end should immediately change to indicate this.

These should be separated as much as possible with separate classes. That way it is impossible to reskin the user interface at some point without having to rewrite the poll loop. The poll loop has no user interface and simply runs as a separate thread, continuously polling the engine. The user interface should be separate from the poll loop so it can be replaced easily.

This back-end polling loop will retrieve from the MiOS engine a list of rooms, scense, devices and categories. It will also retrieve various status flags. All this data from the lu_sdate request. Everything that is retrieved from this request must be stored in shared variables that will be used by both the front and back-ends.

Both the back-end and the front-end need to access the same object-oriented class structure and need to be able to operate completely independently, so separate threads and mutexes are required.

The engine has the same 4 possible states as devices and scenes. These states are ‘none’, ‘pending’, ‘success’, ‘error’.

As mentioned, the back-end (polling loop) creates a class structure which the front end renders, and one of the things in the class structure is the state of the engine. This should be stored as a signed integer with an initial value of -2 (which means ‘not connected’). When the front-end UI starts up and sees the engine state is -2, it should display a ‘Please wait, connecting…’ message.

The back-end is simply a polling loop that continuously requests the ‘lu_sdata’ data request from the engine. This is done by fetching the following URL:
[CommandURL]data_request?id=lu_sdata

The polling loop continues to repeat indefinitely as long as the application is running. Each time it passes the load time and data version of the prior request. Additionally, you should add a timeout argument to the URL. This is the number of seconds that the URL will block waiting for a change. For example, a URL like this…

[CommandURL]data_request?id=lu_sdata&loadtime=1282441735&dataversion=441736333&timeout=60

…will block for a maximum of 60 seconds if no devices or scenes have changed since the loadtime of 1282441735 and dataversion of 44173633. The timeout should be as large as is reasonable to minimize the amount of traffic between the controller and the engine, but small enough to ensure the socket libraries in the controller do not time out and exit with an error before the ‘timeout’ number of seconds has passed.

Additionally, you should add a minimum delay to the argument which is the number of milliseconds that the engine will block the request even if there are changes. For example:

[CommandURL]data_request?id=lu_sdata&loadtime=1282441735&dataversion=441736333&timeout=60&minimumdelay=2000

The above means the request will block for up to 60 seconds and if a change occurs after the request has been blocking for 2 seconds, the request will return instantly. But if the change has already occurred or occurs in less than 2 seconds, the engine will make sure the request blocks at least 2 seconds before returning. The reason for this is that often there is a flurry of changes in rapid succession. For example, if a user runs a scene that turns on 20 lights without the minimum delay, 'lu_sdata' would constantly increment the dataversion and return immediately for several seconds while the scene is running. This would constantly poll and max out the CPU of both the engine and the controller.

If the last 'lu_sdata' request failed to return anything, then you must introduce your own delay to prevent from bogging down the controller or engine in a constant polling loop.

It is harmless to pass in 0's in the initial request for the loadtime and dataversion. This is the same as omitting them, and means you will receive the full set of data. Also, the loadtime value will change when users make changes to their configuration, like adding or renaming a device/scene/room. This means you'll also get a full set of data (full=1). When the engine is running, status changes like a light going on or off will only increment the dataversion. So when you pass in a value for loadtime, if it still matches the current loadtime timestamp of the engine's configuration, then 'lu_sdata' will not return sections, rooms or categories, and it will only return the scenes and device that have changed since the dataversion you passed in. It also will not return the name of the scene or device, since changing the name results in a new loadtime.

Thus the back-end poll loop may want to set a 'full' variable to 1 whenever it receives a full set of data, in which case the front-end user interface knows to re-render the entire UI. When the poll loop receives partial or diff data, it can simply update the internal object or variable that stores the device or scene and set a 'dirty' or 'updated' flag for that screen or device. This is how the front-end UI knows to update that scene or device on its screen. So, when a device is turned on or off, the poll loop's pending 'lu_sdata' will immediately return with the new state of the device. If that device is shown on the UI, the UI should immediately change to reflect the new status of the device.

Some sample pseudo code illustrating how to do the back-end polling loop is available on this page: http://wiki.micasaverde.com/index.php/UI_Simple#Software_architecture:_Two_threads_with_a_polling_loop . The page also contains a C++ version of the polling loop.

The C++ version and the pseudo code are meant to be examples, not complete applications. The assumption with the pseudo code is that there is other initialization code which set variables

like IpAddress, based on the information from Mode 1/Setup, namely the locating of the engine. While (Quite==false) means this is a loop that keeps going until the user wants to quit the application. In reality, when the user chooses to quit, you'll need to kill the thread since the FetchURL could be blocking while waiting for changes.

On the first pass, the URL will have loadtime=0 and dataversion=0, meaning we always get the full list of sections, rooms etc. So, when ParseResponse parses the Json data, it will create all the objects and variables that the front-end needs to present this information to the user.

If the URL fetching fails, you should set the engine status back to the initial value of '2'. This means the user's UI will be interrupted so they cannot continue. They are shown a 'Please wait. Connecting…' modal in this case. It is normal for the engine to become unavailable for up to 45 seconds or so. For example, if the user just installed some plugins, or saved a bunch of changes, the engine will go offline and stop responding while it resets. Therefore, when URL fetching fails for the first time, you should display the 'Please wait…' message but don't treat it as an error. That's why we have an if(NumFailures>MAX_FAILURES), and presumably, since we have a delay of 2 seconds for each request, MAX_FAILURES may be 30 if we want to go to an error condition after 60 seconds of failures. In reality, if FetchURL blocks for a while, having a fixed number of failures, like MAX_FAILURES, may not be good. If FetchURL is blocked for 60 seconds, you wouldn't want the user to see a 'Please Wait…' message for up to 30 minutes. Also, the 'Please wait. Connecting…' modal dialog needs to have 'Quit' and 'Setup' options so the user can exit the app or access their settings to change their connection.

CheckConnection is a function that should take care of a problem by reporting it to the user and letting them check their network settings. In reality, if FetchURL fails then you should run a process that checks whether the local connection is ok then switches to a remote connection, like 4G, if it is not. For example, like https://relay2.mios.com. Similarly, if the network settings have changed from a remote connection to a local connection, you should check the local connection is ok. This is assuming the user chose the default 'Automatic Connection' on the setup connection page (see 'And also offer choices asking the user if they want 'Automatic connection', 'Local connection only', or 'Remote connection only').

If the connection fails, you can reset the minimumdelay on the next request to 0. There's no reason for the engine to wait a couple of seconds before returning on a lu_sdata request for initial data as you don't want to slow down the user's experience. You do, however, need to introduce your own sleep of a couple of seconds between requests so you don't max out the CPU on the engine or the controller.

In this code, it's assumed that ParseResponse is the function that parses the Json data and creates all the objects and variables in memory for the front-end to render. If ParseResponse sees the Json tag "full":1, then it should purge its list of devices/scenes/rooms/categories and rebuild them from scratch. It should also set a flag so the UI can re-render the new data. ParseResponse also needs to set the dataversion and loadtime that are passed in the next request to whatever values just came in on this request. ParseResponse should also set the

variable with the state of the engine (EngineState in this example) to the ‘state’ tag that comes back in the Json data. This way the UI can show the actual state, or the ‘Please wait. Connecting…’ message if the state is -2 (i.e. the controller is not connected to the engine).

If we got a valid response, then the back-end should not introduce its own delay. It should simply set the minimumdelay and let the engine handle the delay (CurrentMinimumDelay=DefaultMinimumDelay). This way you will not introduce any unnecessary latency to the user interface.

5. Data Structure

5.1 Top level tags

Note that the top-object contains these tags:

full=”1” loadtime=”1282441735” dataversion=”441736333” state=”-1” comment=””

‘Full’ is either 0 or 1:

- 1 - ‘lu_sdata’ has returned the complete list of sections, rooms, devices and scenes.
- 0 - ‘lu_sdata’ is only returning the devices and scenes which have changed since the last request.

On subsequent calls to ‘lu_sdata’ you will get the loadtime and dataversion in the URL, like this:

[CommandURL]/data_request?id=lu_sdata&loadtime=1282441735&dataversion=441736333

The state is the state of the engine. It is the same as the state of the devices and scenes. See the list of states below to find out what the numbers mean. If the state is not ‘none’, the main menu should display at the top a red, green or blue icon (for error, success or pending states) and show the user what is in the comment tag.

5.2 Categories

Each category has a name and an id tag like this: name=”Thermostat” id=”5”. You will only see the categories for devices that exist in the MiOS installation. Always use the id for internal reference. The ‘name’ of the category will change according to the user’s language.

5.3 Sections

Each section has a name and an id tag like this: name=”My Home” id=”1”. Again, in most homes, there is only 1 section. If there is only 1 section then you do not need to show the section to the user.

Note. Sections are currently not used, so MiOS installations only have one section.

5.4 Rooms

Each room has a name and an id, and also states the section to which the room belongs.

5.5 Scenes

Scenes also have a name and an id. They also have an ‘active’ status of either 0 or 1. If the active status is ‘1’ then the scene is currently active - all devices are in the status dictated by the scene. So if the scene ‘Go to Bed’ turns the lights off, then, assuming all the lights are actually off, the ‘active’ status of the scene is ‘1’. This is whether or not the user recently ran the scene. Scenes also have a state and comment (see the list of states below). The back-end (polling loop) actually doesn’t do anything with the active status, state or comment other than store it in shared objects or variables that the front-end UI uses to render the controls to the user.

5.6 Devices

Devices also have a category in addition to a name, id, state and comment. They also have additional tags which depend on the category of device. Light switches, for example, will have a ‘status’ tag and a value of 0 or 1, indicating whether the light is on or off. Thermostats also have a ‘temperature’ tag. See the device categories section below for more.

6. The front-end UI

As mentioned earlier, the front-end of the controller should be kept separate from the back-end as much as possible so the UI can be reskinned. It will, however, need to access the object and variable created by the back-end in order to render the UI. When the back-end receives a full set of data from ‘lu_sdata’ (i.e. full=1) then a flag should be set which causes the front-end to re-render the entire UI. You should not, however, lose the user’s place. If the user had selected “Living Room” and “Overhead Chandelier’ then the UI should retain this selection even if it runs

a full re-render. This may not be possible if the device or room has since been deleted. So the front-end needs to take into account that everything may change during a full re-render.

When the back-end updates the current state variables for devices and scenes, the front-end should update those devices or scenes if they are visible so the user will see a change. As mentioned before, the icon for the device as well as what control button is highlighted should change to reflect the current state.

If there is no physical exit button, then the top-level menu needs to have an exit option.

When rendering scenes and devices, what you show the user is the ‘name’. However, what is used internally is the ‘id’, which is not shown to the user.

There are three things you need to show to users:

1) The state of every device, scene and engine (meaning whether the device is busy or has an error condition.

2) The status of the device or scene (meaning whether the device is on or off, or what temperature it is).

3) The buttons that the user needs to control the device or scene.

2) and 3) are really the same. The control buttons for a light switch (on and off) also show the status of the light. If it’s ‘On’ then the ‘On’ button is highlighted. There may be additional icons and readouts that indicate device status, which control buttons it has, and so forth. How these are displayed to users depends on the device category. These are explained in the following 2 sections.

There is flexibility in the button to make it look like an app should for that platform, so long as the functionality is the same. For example, in the mockups you’ll see separate on and off buttons for lights. But, on the Apple iPhone, on-off buttons are typically shown as a single object with a slider that shifts left/right. Since that is the de-facto standard for iPhone user interfaces, that system should be used instead of the separate on/off buttons. The functionality is identical.

Throughout the user interface there’s the help button on all screens. This button should go to a generic help page and pass the originating screen as an argument so the help is context sensitive.

State of engine, device or scene

The engine, as well as every device and scene, has a ‘state’ tag which indicates if everything is ok. ‘lu_sdata’ returns a number from -1 to 6 to indicate the state. However, in the user interface,

some state numbers mean the same thing - so you actually have only 4 states to contend with. The -2 state only applies to the engine, and means that you can't talk to the engine.

The four states returned by 'lu_sdata' are:

NONE. State value **-1**. This means there is nothing to report to the user. The device/scene/engine is working ok and is not doing anything at the moment.

PENDING. State values **0, 1, 5, 6.** All these mean a state of 'pending'. A pending state is always shown in blue. So for devices and scenes there should be a state indicator. When the engine itself is in a pending state, you should display a prominent 'busy' indicator at the top of the UI. Note - some of the commands might return error when the state is busy depending on which device is being controlled and what the engine is currently doing. The value of the 'comment' for the engine, scene or device should be available to the user.

ERROR. State values **2, 3**. In either case, a red indicator should be shown for the engine, scene or device to communicate the failure. You should also display a comment.

SUCCESS. State value **4**. Display a green indicator and the comment. In general, the success state is temporary. If you turn on a light, for example, the engine will report the state of the device as SUCCESS for about 30 seconds. This lets the user confirm that whatever they did went through ok. The state should revert to 'NONE' after a short time.

Status of device, scene and control buttons

This depends on what category of device you are showing. The status tags returned by 'lu_sdata' are different per device type. For example, a thermostat has status tags like mode, heatsp, coolsp, etc., whereas a light switch has tags such as level and status. Normally a control button, like the button that turns on a light, is bound to a status so the user can tell whether the button is highlighted or not if the command is active. For example, a light switch has a 'status' tag that is 0 or 1 depending on whether it's on or off.

There are also 2 command buttons: on and off. Those 2 command buttons are bound to the status, meaning that if the status is 0, then the 'off' button should appear highlighted so the user can tell the light is already off. Each control button is tied to a command even if the command button is already highlighted. In other words, even if the status of a light is 0 (off), and the off button is already highlighted, when the user selects the off button, send the 'off' command regardless.

You send commands by opening a URL on the engine, just like the pol loop does. The difference is that to run a command, the data_request is generally 'lu_action' instead of 'lu_sdata'. The engine is multi-threaded and can handle many requests at once. So you should not interrupt the background polling loop when you want to send a command. The polling loop

will continue to block on the ‘lu_sdata’ request and, in parallel, you will do a ‘lu_action’ request when the user selects a command button.

When the user selects a control button for a device or scene, you will use lu_action to tell the engine what to do. In all cases you will get back either an ‘Ok’ or a ‘JobID’ inside a response tag. If you don’t, you will get back an error message which you should pass to the user. If you don’t get anything back, retry the lu_action once every 2 seconds for up to 30 seconds, displaying an hourglass or busy indicator while you do.

If you still cannot get a response to ‘lu_action’ after 30 seconds, report to the user that you have lost contact with the engine. It is normal for ‘lu_action’ to fail if the engine is reloading. So, for example, if a user clicks ‘Lock’ and you get the error ‘Lock Operation not available’, then show a popup message in the UI: ‘Lock Operation Not Available’, along with an ‘ERROR’ icon and ‘Ok’ button. On the other hand, if you do not receive any response at all to lu_action, or get a html error, just keep retrying over and over until you either get a response or a timeout.

You will always pass a ‘service’, an ‘action’ and optionally arguments on ‘lu_action’. For this reason, you should have a common function, like SendAction (string service, string action, string arguments), which has the logic of sending the command to the engine.

When the user selects a control button, do not automatically show that button highlighted. For example, if a light is ‘off’ and the user selects ‘on’, just give the user feedback with the ‘on’ button so they know their selection was recognized. Do not automatically switch which button is highlighted until the command has been processed. The background poll loop will provide a new ‘status’ of the device when the engine has completed turning off the light.

As soon as you send the command with ‘lu_action’, the polling loop should immediately return and the state of the device or scene will change to ‘PENDING’. This means the user should see the ‘blue’ busy indicator so they know the engine is interacting with the device in the background. When the engine has succeeded in setting the device, the state will change to ‘SUCCESS’ and the status variable(s) will also change, causing the UI to re-render the control and display the new status.

The following sections outline device categories, status codes and control buttons:

Scene

There is simply a ‘Run’ button for the control. The status is active or not active (active=1 or active=0). Scenes only have a ‘run’ button as the action, and if the scene is active, simply show the ‘run’ button in a highlighted state. When selected, use service “urn:micasaverde-com:serviceId:HomeAutomationGateway1” and action “RunScene” and pass the id of the scene as the SceneNum argument. For example, to run scene id 5, open this request:

http://192.168.2.117/port_3480/data_request?id=lu_action&serviceId=urn:micasaverde-com:serviceId:HomeAutomationGateway1&action=RunScene&SceneNum=5

Devices

For devices, refer to the 'category' tag for the device:

CATEGORY #2 - Dimmable Light

The field 'status' is either 0 or 1 for off or on. If status is 1, the 'level' is a value from 0-100 to indicate the brightness. Buttons 'on' and 'off' should be bound to the 'status' of the device and, if selected, should send the service "urn:upnp-org:serviceId:SwitchPower1", action "SetTarget" with the argument "newTargetValue" is a value of 0 for off or 1 for on. There should also be a slider which represents the 'level' for the device from 0-100. If the user changes the level, send the service "urn:upnp-org:serviceId:Dimming1", action "SetLoadLevelTarget" and argument "newLoadlevelTarget" is 0-100.

CATEGORY #3 - Switch

The same as dimmable lights except switches only have a 'status'.

CATEGORY #4 - Security Sensor

The field 'tripped' is either 0 or 1 to indicate whether or not the sensor was tripped (1= tripped, 0= not tripped). 'armed' is also either 0 or 1 to indicate arming state. There is a single control button, 'ARM'. If the 'armed' status is 1, display the 'ARM' button in a highlighted state. If it is 0, show it in a normal state. When the 'ARM' button is selected, this is one case where you do not use 'lu_action' - use 'lu_variableset' instead. ARM is simply a flag in the engine, so use this command:
[CommandURL]data_request?id=lu_variableset&DeviceNum=x&serviceId=urn:micasaverde-com:serviceId:SecuritySensor1&Variable=Armed&Value=y

Replace 'x' with the device id. Replace 'y' with 0 if the armed status is 1, or replace 'y' with 1 if the armed status is 0 - so the ARM button is a toggle.

CATEGORY #5 - Thermostat

The 'fan' field is 'Auto' if the mode is off/normal, or 'ContinuousOn' if the fan is always on.

‘mode’ indicates the mode of the thermostat, with options being ‘Off’, ‘HeatOn’, ‘CoolOn’ and ‘AutoChangeOver’.

‘hvacstate’ indicates the current state of the thermostat. For example, if it’s in cool mode, it indicates if the compressor is actively cooling or the set point has been reached. Its values are ‘Idle’, ‘Heating’, ‘Cooling’, ‘FanOnly’, ‘PendingHeat’, ‘PendingCool’ and ‘Vent’.

‘temperature’ is the current ambient temperature. ‘heatsp’ and ‘coolsp’ are the current heat/cool setpoints. The user indicates in their system location preference whether they want English or metric, so just display the values as shown and don’t worry about converting them.

There are 4 mode buttons - ‘Off’, ‘Heat’, ‘Cool’ and ‘Auto’ - which should be bound to the ‘mode’ value. When these buttons are selected they should send the service “urn:upnp-org:serviceId:HAVAC_UserOperatingMode1” and action “SetModeTarget” with the argument

“NewModeTarget” has the values “Off”, “HeatOn”, “CoolOn” or “AutoChangeOver”. Depending on the space on the UI, these can either be 4 buttons with the active mode button selected, or a drop-down that shows the active mode and lets the user change it.

The fan is simply on or off, so just display a ‘fan’ button which is highlighted if the ‘fan’ variable is 1. Be sure it’s separate from the 4 modes so it doesn't look like a 5th mode. If the user selects it, send the service “urn:upnp-org:serviceId:HAVAC_FanOperatingMode1” and action “SetMode” with the argument “NewMode” has the values “Auto” for off or “ContinuousOn” for on. In other words, if the fan was off and the fan button is not highlighted, send the value “ContinuousOn”. And if it was on, send the value “Auto” so it is a toggle.

Display the values for ‘heatsp’ and ‘coolsp’, but if the user selects either one, display a drop-down or a pop-up or keypad to let the user set a new temperature. Then send the service “urn:upnp-org:serviceId:TemperatureSetPoint1_Heat”, or “urn:upnp-org:serviceId:TemperatureSetPoint1_Cool” with the action “SetCurrentSetpoint” and the argument “NewCurrentSetpoint’ is whatever temperature the user chose. Display the value of the ‘temperature’ to the user as read-only, since it’s the ambient temperature.

CATEGORY #6 - Camera

These don’t have a status. The control button is marked ‘VIEW’, but it doesn’t send an action. It just displays an image viewer on the controller.

The viewer should have a large JPEG image which you get by calling: **[CommandURL]/data_request?id=cam_image&Device_Num=x** , where x is the device ID. This request returns a JPEG file. If the camera also has a status ‘streaming’ and the status is

not empty, then the image viewer popup should display a ‘LIVE VIDEO’ button, which displays the streaming video instead. See the ‘Remote_Camera_Streaming to learn how.

Both the JPEG image viewer and the motion image viewer must have buttons for the user to control the pan/tilt/zoom of the camera. The buttons are L, R, U, D, +, - (for left, right, up, down and zoom in/out). When those buttons are pressed, use the ‘lu_action’ with the service “urn:micasaverde-com:serviceId:PanTiltZoom1”, and use the action “MoveLeft”, “MoveRight”, “MoveUp”, “MoveDown”, “ZoomIn” or “ZoomOut”. There are no arguments for those actions.

If the camera supports streaming, attempt to connect to it using the following flow:

- Detect if camera is accessible on local IP returned in sdata object

- If yes, try to start local streaming on address **http://[Camera.ip][Camera.streaming]**

- If streaming fails, switch to emulated mode, which gets frames from the camera every second. You can access emulated mode with the URL **http://[Camera.ip][Camera.url]**

- If the camera is not accessible on an internal ip, make the call **[CgiURL]relay.sh?device=[Camera.id]&ip=[Camera.ip]** to the gateway, to open remote access tunnels. This will return a Json like this:
{“server”:”us-relay12.mios.com”,”port”:22804}

- Set CameraRelayURL to **https://[relay.server]:[relay.port][Camera.streaming]**

- If streaming fails, revert to the frame-by-frame emulated mode as described above. Display the images retrieved with the request
[CommandURL]/data_request?id=cam_image&Device_Num=x

CATEGORY #7 - Door Lock

The ‘status’ field is 0 for unlocked and 1 for locked. Display a single ‘lock’ and ‘unlock’ button which is bound to the ‘status’ variable. If selected, use the service “urn:micasaverde-com:serviceId:DoorLock1”, action = “SetTarget”, and value newTargetValue (which is 0 for unlocked and 1 for locked).

CATEGORY #8 - Window covering

Window covering is the same as Dimmable Light.

CATEGORY #16 - Humidity sensor

The ‘humidity’ field is the relative humidity from 0-100. Just display the ‘humidity’ - there are no control buttons.

CATEGORY #17 - Temperature sensor

The ‘temperature’ field is the same as with a thermostat. Just display the ‘temperature’ - there are no control buttons.

CATEGORY #18 - Light sensor

The ‘light’ field is a measurement of the ambient light in the room. There is no universal scale, so just display the number for the user. Just display the ‘light’ - there are no control buttons.

CATEGORY #21 - Power Meter

The ‘watts’ field is the current consumption in watts. Just display the ‘watts’ - there are no control buttons.

Note - All devices, regardless of category, will have a ‘watts’ tag if we can measure the current consumption.

The User Interface screen mockups

The attached mockups.pdf file shows what all of the screens look like. The pdf only shows basic content. You are expected to make it look professional and conform to the design principles of similar apps on your target platform (iPhone, Android, etc).

There are five buttons at the bottom of the application that should always be available to let the user jump to the main features. Within each of those five sections, there may be extra buttons along the top which either show functions or are tabs to select the content with the page. For example, in the ‘Scenes’ section, the buttons at the top are to add and delete scenes. In the ‘Lights and Climate’ section, the buttons at the top should look like tabs. This is because ‘lights’, ‘climate’, ‘windows’ and ‘sensors’ are simply filters that determine which types of device are shown on the page.

Normally, the front-end should always run without calling any blocking functions. This ensures it is always responsive regardless of the activities of the back-end polling loop. There are a couple of exceptions to this where the front-end may display a ‘busy’ modal (explained below). When the system is busy like this, it must be made clear to the user by making use of the standard mechanism of the given phone. For example, an iPhone screen should darken with the little busy icon in the middle and a ‘Working…’ message (or similar). Other than those occasional

moments where there is a busy message, the front-end must always be responsive to the user, and the user should always be able to quit very quickly.

The following are the screens and explanation of what happens when the user touches each of the buttons Each title refers to a page in the pdf. Be sure to turn on bookmarks in the pdf because each page is given a name that can only be seen when bookmarks are enabled. Each name corresponds to a section below:

FIRSTRUN

---

1. Verify PK_Device and HWKey

If you do not have a PK_Device and HWKey stored locally, that means the app was started for the first time and these items have not yet been provisioned.

Call MMS request to provision the device: https://[Server_Authd]/authd/provision PUT

The arguments of the request are:

MacAddress = phone device identifier,
PK_DeviceTyep=3, PK_DeviceSubType (301 for Android, 302 for iPhone, 303 BlackBerry, 304 Windows Phone), InternalIP (optional), EK_Oem (integer, id current app OEM).

{ “PK_Device”: 4392, “HWKey”: ”e8fa138af5d3d8f19485275382a8c2e7” }

A ‘busy’ modal is shown to the user while they wait for the URL to return. Display a message to the user during this modal: “Please wait. Contacting MiOS servers…’. Provide a ‘cancel’ button that aborts the request. If you receive a valid response, store both values on the phone.

Continue to step 2 once a valid PK_Device and HWKey are stored.

2. Check that a valid username is stored on the phone

If not, display the FirstRun screen described above.

If you already have a Username and PK_Device stored locally, skip to screen Main_1. There are basically two sections on the **FirstRun** screen for the user depending on whether they have a MiOS username or not. Highlight the section that the user is currently in as shown in the mockup.

If the user chooses the first option, you should attempt the login. If it isn't successful, display a popup message that says "Login not valid" with an 'Ok' button that returns to the previous screen. If login is successful, read the content of the response which contains identity tokens, per MMS API, then store the tokens along with the Username and Password on the phone. Then continue to Main_1.

If the user chooses the second option, you should attempt to create the user. If the request fails, display a popup message that says "Unable to add user" along with an 'Ok' button that returns to the previous screen. If the http status is 200 then store the identity, Username and Password on the phone and continue to Main_1.

3. Check if valid identity tokens are stored on the phone

Validate any identity tokens stored on the phone. If they are valid, continue to Main_1. If not, continue to FirstRun_B.

Note - The username, password, PK_Device and HW_Key should be stored in a memory variable in the application so you do not have to retrieve them from the phone's storage each time.

FIRSTRUN_B

Let the user enter their password. Display the keyboard by default and have the Ok button grayed out until the user stops typing. If the user chooses 'Cancel' then just exit the application. If the user chooses 'Different User' then erase the Username store on the phone and go back to the FirstRun screen.

If the user enters a password and chooses 'Ok' then call the login MMS service. If you receive any http status other than 200 then show a popup message that says "Login Not Valid". Provide an 'Ok' button that just returns the user to the previous screen. If you receive a "SUCCESS" response then read the content of the response as it contains identity tokens per the MMS API. Store the token along with the Username and Password on the phone then continue to Main_1

Note - The username, password, PK_Device and HW_Key should be stored in a memory variable in the application so you do not have to retrieve them from the phone's storage each time. When the user enters their password here, do not store it on the phone - just keep it in memory.

MAIN_1

1. Fetch the current IP address of the phone and see if you are connected to the internet over Wi-Fi. If the current IP address matches the value stored in LAST_IP, and the current Wi-Fi connection matches the value stored in LAST_WIFI, then skip this paragraph and start the background polling thread. Call the locator service if either does not match, or if LAST_IP or LAST_WIFI are empty (meaning this is run for the first time).

Display a list of the MiOS Systems (i.e. the items in the ‘units’ tag).

Display the icon for ‘Fast connection over Wi-FI’ next to the unit If a system is indeed available on the local network (the Json tag ‘InternalIP’ is not blank). This will only happen if you are connected over Wi-Fi.

Automatically connect to the first available unit on the local network. If none are available then connect to the first available unit that is not on the local network. Store an ‘L’, ‘R’ or ‘N’ in the ‘LAST_ENGINE_CONNECT’ in the phone’s storage depending on whether the connection is local, remote, or there is no MiOS system to which you can connect.

If you are connecting to the MiOS system, store its serial number in LAST_PK_Device_Gateway. This means that the next time you will not call locator service again. Start the background polling thread if you have a PK_Device_Gateway.

Note - Here are the categories and subcategories for the accessories:

Category 1 = Phone
  Subcategory 1 = iPhone
  Subcategory 2 = Android
  Subcategory 3 = BlackBerry
  Subcategory 4 = Windows Smartphone

Category 2 = Camera
  Subcategory 1 = Trick Camera
  Subcategory 2 = Panasonic

Category 3 = TV
  Subcategory 1 = Samsung

Note - Once the background polling thread has started, the status of the MiOS system (the top level ‘status’ tag of the ‘lu_sdata’ request) will be the blue/green/red/blank status icon under the engine. The status text will be to the right of the icon, where it says ‘Downloading Data’ right now.

Note - An empty PK_Device_Gateway field means the locator service didn't return a valid list of MiOS systems. Display Main_2 instead. You won't be rescanning every time the application starts - only when the user chooses to scan for new devices.

Note - The 'scenes', 'lights & climate' and 'cameras & security' tabs at the bottom relate to devices in scenes from the MiOS engine to which you are connected. So if the user is not connected to the MiOS engine, or if the background polling loop is not working, then almost everything throughout the UI will be unavailable.

2. Show the 'Remember My Password' option as checked if there is a password stored on the phone. If the user unchecks the box then delete the pass route on the phone. If the user checks the box then write the password currently stored in a memory variable into the phone's storage. If the user chooses to log-out then go back to FirstRun.

3. If the user chooses a different MiOS system, just update the memory variable with its PK_Device_Gateway and update LAST_PK_Device_Gateway in the phone's storage.

4. If the user chooses to scan for new devices then go to #1 above and run the locator service.

5. If the user chooses 'Setup Rooms' then go to the 'Rooms' screen.

MAIN_2

If the user chooses 'Scan for new devices' then run 'locator_json' from #1 in MAIN_1.

SCENE_1

This lists all the scenes from 'lu_sdata'. If you're associated with a MiOS engine, just display a message in the middle of the screen - "You need a gateway to create scenes and automation".

If the polling loop shows a 'disconnected' status then display a message in the middle of the screen - "Please wait. Contacting your gateway…". The 'disconnected' status means you are able to see the gateway, but not connect to it. The user should be able to navigate to other tabs during the reconnection attempt.

If you are connected to the MiOS engine but there are no scenes in 'lu_data', just display "No Scenes" in the middle of the screen.

Display the scenes without a room first, then display all rooms in alphabetical order. Show each room's scenes as a list under the room's name. The 'go' button should be highlighted if the scene has the 'active'=1 tag in 'lu_sdata' (as 'Come home from work' is shown). The far-left icon shows the state of the scene.

'New Scene' goes to SCENE_2.

'Edit/Delete Scene' means that 'Rename' and 'Delete' buttons are presented next to the scene name instead of the 'Go' button. 'Delete' shows a confirmation window with 'Are you sure you want to delete this scene?', along with 'Yes' and 'No' buttons. If the user chooses 'Yes' then call **data_request?id=lu_scene&action=delete&scene=x**. If they choose 'Edit', display Scene_1B. If they choose 'Ok' then call **data_request?id=lu_scene&action=rename&scene=x&name=y&room=z**

SCENE_2

Call **data_request?id=lu_scene&action=record** when the screen first loads. Confirm that you receive an 'Ok' in response.

If the user chooses a pause, show a popup window that offers a list of times - 1 second, 3 seconds, 5 seconds, 10 seconds, 20 seconds, 40 seconds, 1 minute, 2 minutes, 5 minutes and 15 minutes. Show an 'Ok' button that lets the user confirm their choice. Call **data_request?id=lu_scene&action=pause&seconds=y** with the number of seconds. If the user choose to stop recording, call **data_request?id=lu_scene&action=stoprecord.** If you receive an 'Ok' response then continue to SCENE_4. If you receive an empty response then continue to SCENE_3.

SCENE_3

'Ok' just goes back to the normal, SCENE_1, screen. 'Help - I don't understand!' shows the help texts.

SCENE_4

Call **data_request?id=lu_scene&action=listrecord** to retrieve a list of devices and actions in Json format, with an ID for each. Show these in a scrollable list if there are more than will fit on a screen. Place a delete button to the right of each action. If the user selects 'Delete' then call **data_request?id=lu_scene&action=deleterecord&id=x** for that action, then call the listrecord action again and refresh the list. 'Cancel' returns the scene to SCENE_1. 'Save' advances the scene to SCENE_5

SCENE_5

Let the user type in a name and take a room. If they choose ‘Add Room’ then display a popup message - ‘What do you want to call the room?’. Provide an input box and ‘Ok’ and ‘Cancel’ buttons. If the user provides a name, call **data_request?id=lu_scene&action=create&name=whatever**.

Note - The background polling loop should immediately return a full set of data in ‘lu_sdata’ once you do the call above. Be sure the page updates accordingly.
‘Cancel’ returns the scene to SCENE_1. ‘Save’ calls **data_request?id=lu_scene&action=saverecord&name=whatever&room=x**.

LIGHTS & CLIMATE + CAMERAS AND SECURITY

Like the scenes, this displays data you get back from the polling loop in the background. Display the same message as in the scenes tab if there is no MiOS engine or if you cannot reach it. The one exception is if you’re displaying cameras since those work without a MiOS engine (explained later).

At the top you will have the sub-options for each of the device categories. For the sake of illustrating the controls, I’ve listed all of the control options on the mockup screen. This wouldn’t be the case in the real world. For example, when you select the climate option at the top, you would only see the main thermostat. You’d have to select the other tabs to see the other devices.

When displaying cameras, display each camera’s still image in the UI. This is one case where the front-end will spawn a separate thread - to retrieve all still images from all cameras. This needs to be done in a separate thread so that the front-end UI remains active and responsive. The image boxes are simple placeholders with a camera icon until the still images load. In other words, it’s kind of like a web page on first load or full refresh. You see placeholders for graphics until the graphics have loaded, but the page remains responsive throughout. Users can choose a camera image to view that camera, or change tabs to do something else. If they change tabs then this should kill the thread that is fetching the still images.

If the user touches an individual image, display the image full screen along with a ‘Close’ button. Make sure the background retrieval of images does not stop users from selecting an image, even if the image is still processing.

If the camera supports video streaming then there will be a ‘streaming’ data tag in the ‘lu_sdata’ or locator_json. You should make the ‘Video’ button selectable.

Note - On all tabs, if the user does not have a MiOS system, display the message “You need to connect to MiOS to use this application”.