Accessory Configuration

Jump to bottom

José Antonio Jiménez Campos edited this page last month · 124 revisions

This section is mandatory. It contains an array of HomeKit accessories that are available on your device.

Each accessory has a HomeKit service with a number of keys defined to describe it and how it should behave. The list of keys that can be used in any service are contained below. Also check each specific service for details of other keys available for that specific service type.

Section	Key	Description
Service Type	`"t"`	The type of the main HomeKit service of accessory
Extra Services	`"es"`	Array of extra HomeKit services of same accessory
Binary Inputs	`"b"`	I/O inputs available on the service
Inching Time	`"i"`	Time period before returning service to previous state
ICMP Ping Inputs	`"l"`	Ping inputs available on the service
Initial State	`"s"`	Service initial state options on boot
Actions on Boot	`"xa"`	Enable / Disable execution of service actions on boot
Delay after creation	`"cd"`	Set delay after service creation
State Inputs	`"f[n]"`	Inputs that set state and/or trigger an action
Status Inputs	`"g[n]"`	Inputs that set a fixed status but no actions
ICMP Ping State	`"p[n]"`	Change the state on a ping result
ICMP Ping Status	`"q[n]"`	Change the status on ping result
Wildcard Actions	`"y[n]"`	Perform an action when a service reaches a target value
HomeKit Visibility	`"h"`	Determines whether service is visible to HomeKit
Serial Number Prefix	`"sn"`	Customise serial number provided to HomeKit accessory

Each element of the array is made of a service type definition, a set of actions to perform when binary inputs change state and the binary inputs that cause the actions to occur. Multiple accessories can be defined for the same physical device. A configuration file should have the following layout e.g.

```
{
  "c": { ... },         <-- Configuration definition
  "a": [
    {                   <-- First accessory
      "t": 2,           <-- Service Type definition
      "0": { ... },     <-- Action definition
      "1": { ... },     <-- Action definition
      "b": [{ ... }]    <-- Binary I/O input definitions
      "l": [{ ... }]    <-- ICMP Ping input definitions
      "s": 5            <-- Initial State definition
      "f0": [{ ... }]   <-- State Inputs definitions
      "g0": [{ ... }]   <-- Status Inputs definitions
      "q0": [{ ... }]   <-- ICMP Ping status definitions
      "p0": [{ ... }]   <-- ICMP Ping state definitions
      "y0": [{ ... }]   <-- Wildcard action definitions
    },
    { ... },            <-- Second accessory
    { ... }             <-- Third accessory
  ]
}
```

The above outline example shows the layout for the configuration settings. The first accessory has been expanded to show the different objects that can be defined for an accessory. The documentation below details the generic information. Refer to each specific device type to understand the actions, binary inputs and notifications available for it.

Service Type

Every accessory needs to have the type `"t"` option set. This defines the type of HomeKit service that the hardware supports e.g. switch, sensor, thermostat etc.

Refer to the list of service types to determine the value to use for the `"t"` option.

Extra Services

Optionally, each accessory can have extra services, adding more features to same accessory. Some advantages declaring more services into an accessory instead creating a different one is to save RAM, to group services in Apple Home App, to create some special accessories that work with group of services (Power strips, irrigation systems...).

It is possible to create any combination of any HomeKit Service into same accessory. Each extra service will can have same options that accessory main service, but Serial Number Prefix.

Since extra services are related only to HomeKit accessories, those services declared with `"h":0` to not add them to HomeKit can not be into `"es":` array and must be declared as stand-alone accessories.

Main recommendation is to add all services into same accessory that stays at same HomeKit room.

This is an example of a Power Strip, created with 3 outlets into same accessory (2 of them are extra services):

```
{
  "c": { ... },         <-- Configuration definition
  "a": [
    {                   <-- First accessory
      "t": 2,           <-- Service Type definition
      "es": [           <-- Array of extra services
        {               <-- Second service
          "t": 2        <-- Service Type definition
        },
        {               <-- Third service
          "t": 2        <-- Service Type definition
        }
      ]
    }
  ]
}
```

Binary Inputs

Binary inputs array e.g. buttons, are defined by the `"b"` key within an service object.

This is an array of objects defining the GPIOs the inputs use, configuration of the GPIO e.g. whether or not it needs a pull up resistor, and the state of the input e.g. normal or inverted signal.

A binary input is a GPIO pin (or pins) that causes service action `"0"` then service action `"1"` to be triggered each time the defined state in key `"b"` is met e.g. the first time the input state occurs service action `"0"` will be triggered. The second time the input state occurs service action `"1"` will triggered. The third time the input state occurs service action `"0"` will be triggered etc.

Binary input key `"b"` is an array object. Multiple GPIO inputs can be defined to cause service actions `"0"` & `"1"` to be triggered.

Key	Default	Type	Description
"g"	none	integer	Binary Input GPIO
"p"	1	integer	Internal Pull-Up Resistor
"i"	0	integer	Inverted Binary Input
"f"	5	integer	Binary Input Filter
"t"	1	integer	Press Type
"m"	0	integer	Mode

```
{
  "c": {},
  "a": [{
    "t": 1,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "b": [{ "g": 0, "p": 1, "i": 0,  "t": 1 }]
  }]
}
```

In the above example a button is attached to GPIO 0 (`"g": 0`), the internal GPIO pull-up resistor is enabled (`"p": 1`), the GPIO logic is not inverted (`"i": 0`) and the button is activated by a single press (`"t": 1`).

NOTE: The example above has entries for the default values. It is recommended these are omitted in an actual configuration to reduce memory usage.

These options are explained in more detail below.

Binary Input GPIO

Key	Default	Type	Description
"g"	none	integer	GPIO connected to the binary input source

This option is mandatory for each binary input definition and must contain the GPIO number the input is connected to. If you want to use ESP8266 ADC as binary input, declare it as GPIO 17.

Internal Pull-Up Resistor

Key	Value	Description
`"p"`	`0`	No internal pull-up resistor used
	`1`	Enable internal pull-up resistor on GPIO input (***default***)

Depending on the hardware attached to a GPIO there may be a need for enabling an internal pull-up resistor. The `"p"` option can be used to enable / disable the pull-up.

Inverted Binary Input

Key	Value	Description
`"i"`	`0`	**Normal input (*default*)**
	`1`	Inverted input

With some peripherals and hardware the actual GPIO signal may need to be inverted to ensure the correct action is performed when the GPIO changes state. This is normal for binary inputs that are active *low*. The `"i"` option can be used to invert / leave the input signal.

Binary Input Filter

Key	Value	Description
`"f"`	`4`	**Soft debounce (*default*)**

The binary input filter `"f"` can be set to any integer value between `1` (soft) and `255` (hard) to avoid interference such as debounce from the input when a button pressed.

NOTE: The binary input filter value overrides the general option `"f"` when defined for a service.

Press Type

Key	Value	Description
`"t"`	`0`	Single press, opposite to 1
	`1`	**Single press (*default*)**
	`2`	Double press

Key	Value	Description
	3	Long press
	4	Very long press
	5	Hold press during 8 seconds

This option defines the type of press required on an input to initiate an action.

You can have the same binary input / GPIO with different press types attached to different services or functions.

Press Type Example

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 1,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "b": [ { "g": 14, "p": 1, "t": 1 }, { "g": 14, "t": 0 }]
  }]
}
```

In this example we have created a toggle switch by declaring two binary input types with the same GPIO `"g": 14` one of press type `1` and one of press type `0`. When you press the button *down* type `"t": 0` is triggered and on button *up* type `"t": 1` is triggered (***the default type***)

Mode

Key	Value	Description
`"m"`	`0`	**Continuous (*default*)**
	`1`	Pulses

This option defines the type of input signal.

ICMP Ping Inputs

ICMP Ping inputs were introduced in firmware version `1.4.0`

Key	Default	Type	Description
`"l"`	none	object	Ping Binary Input
`"p[n]"`	none	object	Ping State Input
`"q[n]"`	none	object	Ping Status Input

ICMP ping inputs can be *action* or *state* oriented, depending on whether you are trying to update the state of your service or perform an action, based on the ping response.

The `"l"` and `"p[n]"` objects are action objects and follow the same rules as Binary Inputs and State Inputs. With an `"l"` key replacing the binary input key `"b"` and `"p[n]"` objects replacing `"f[n]"` objects.

The `"q[n]"` key is a status key and is used to update the service status based on the result of the ping, but not perform the action.

Specific states are defined using the `"p[n]"` & `"q[n]"` options (where `[n]` is an action number e.g. `"p0"`, `"q0"`). Each device has its own set of actions and these are mapped to the state inputs. Check the device page for details of the specific states.

Ping inputs are triggered only on a response change. If you have set up to listen for a specific state `"p0": [{ "h":"iMac-de-Jose","r":0 }]` then this state is triggered on the first occurrence of the target host not responding to the ping. If the host remains unresponsive then no further triggers occur until the host becomes responsive and then unresponsive again.

The ping interval is fixed at 5 seconds and 3 failed responses need to occur consecutively before a response type of `0` occurs.

The following options are available as part of the `"p[n]"` and `"q[n]"` objects:

Key	Default	Type	Description
`"i"`	`0`	integer	Ping State Action
`"h"`	none	string	Target Host
`"r"`	`1`	integer	Response Type

Ping State Action

Key	Value	Description
`"i"`	`0`	**Ping state action is relative (*default*)**
	`1`	Ping state action is absolute

This option defines how the ping state action should behave based on the target response. The default value of `0` indicates that the action should be relative to the ping response. A ping response changing from non-response to response will cause the action to occur only when change occurs (assuming `"r":1`) e.g. If the action is to turn the lights off when the TV responds to a ping then the lights will go off the first time the TV ping responds. If the lights are subsequently turned back on while the TV is still on then no further actions will occur unless the TV ping response status changes.

A value of `1` indicates the action should be absolute to the ping response. A ping response changing from non-response to response will always cause the action to occur (assuming `"r":1`) e.g. If the action is to turn the lights off when the TV responds to a ping then the lights will go out the first time the TV ping responds. If the lights are subsequently turned back on while the TV is still on then the next ping response from the TV will cause the action to re-occur and the lights to go back off.

Target Host

Key	Value	Description
`"h"`	`string`	Target host, with DNS resolution

This option is mandatory and must contain the IP address or host name of the device to be targeted with a ping.

NOTE: When using a DNS name ensure that the DNS server allocated in the DHCP server response to the device can resolve the name. There is no mDNS support.

Response Type

Key	Value	Description
`"r"`	`0`	Target host does not respond to ICMP ping
	`1`	**Target host responds to the ICMP ping (*default*)**

This option defines the expected response in order to trigger the defined action.

Action on Lost WiFi Connection

Key	Value	Description
`"d"`	`0`	**Disable the ping action when WiFi connectivity is lost (*default*)**
	`1`	Enable the ping action even when WiFi connectivity is lost

This option defines if the ping action should occur if there is no WiFi connectivity. By default the ping action is disabled and no action will take place if the WiFi connection is lost.

ICMP Ping Example 1

In this example we want a power switch to turn on whenever a laptop appears on the local network in order to ensure it gets charged while we are using it.

The switch can be configured to ping the laptop DNS name and when a successful response occurs the power is turned on. Similarly, when the laptop is powered down the ping no longer gets a response and the switch can turn off the power.

```
{
  "c":{ "l":13 },
  "a":[{
    "t": 1,
    "0": { "r": [{ "g": 12 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "p0": [{ "h":"iMac-de-Jose","r":0 }],
    "p1": [{ "h":"iMac-de-Jose" }]
  }]
}
```

ICMP Ping Example 2

In this example we have a Sonoff S20 that uses ping to turn on/off the lounge light when ever the TV is turned on and responds to a ping request. The lights will always follow the inverse state of the TV even if changed by an external action (`"i":1`).

```
{
  "c":{ "l":13, "b":[{ "g":0,"t":5 }] },
  "a":[{
    "t": 2,
    "0": { "r": [{ "g":12 }] },
    "1": { "r": [{ "g":12, "v":1 } ]},
    "b": [{ "g": 0 }],
    "p0": [{ "h":"my-tv", "i":1 }],
    "p1": [{ "h":"my-tv", "i":1, "r":0 }]
  }]
}
```

ICMP Ping Example 3

In this example we use ping actions to reflect the current on/off state of the TV e.g. If the TV responds to a ping the HomeKit service shows as *on* in the HomeKit application.

```
{
  "c":{ "l":13, "b":[{ "g":0,"t":5 }] },
  "a":[{
    "t": 2,
    "q0": [{ "h":"my-tv" }],
    "q1": [{ "h":"my-tv", "r":0 }]
  }]
}
```

Inching Time

The Inching Time is an accessory option that defines a time period for a service to return to the previous state e.g. If the inching time is set to `10` seconds a switch would turn off 10 seconds after turning on.

Key	Default	Type	Description
`"i"`	`0`	float	Time period before returning service to previous state

Inching time is started when action `"1"` of the service is triggered. After the inching time has elapsed action `"0"` will be triggered. **NOTE: the lock mechanism is the opposite of this.**

Inching time is a float, so fractions of seconds can be specified e.g. `"i":3.5`

A typical example of using inching time would be to reset a lock after it has been unlocked.

NOTE: Inching Time is not supported by all services, so check before using it.

Initial State

Initial state is defined by the `"s"` key contained within the service object.

Key	State	Description
`"s"`	`0`	**OFF (*default*)**
	`1`	ON
	`4`	Defined by fixed state inputs
	`5`	Last state before restart
	`6`	Opposite to last state before restart

The initial state that a service enters on boot can be set using the `"s"` option.

Check each service for details on whether Initial State is supported.

Execution of Actions on Boot

This option was implemented in firmware version `1.10.0`

The Execution of Actions on Boot is defined by the `"xa"` key within the service object.

Key	State	Description
`"xa"`	`0`	Disabled
	`1`	**Enabled (*default*)**

By default actions are enabled on boot, but some circumstances require no actions to be performed during the boot of an accessory. When this is the case the actions can be disabled by setting the `"xa"` key to 0.

See the TV Service device for an example on why disabling actions is necessary.

Delay after creation

This option was implemented in firmware version `2.4.5`

The Delay after creation option is defined by the `"cd"` key within the service object.

Key	Default	Type	Description
`"cd"`	disabled	float	Time in seconds to delay after creation

By default a service is created within HAA and then the next service is immediately created. There are times when a delay is required after a service has been created in order to exec and free memory before the next service is created.

When this is required the `"cd"` option can be used. Set it to a positive value in seconds and a delay of that time will occur once the service has been created.

The option only needs to be added to a service if a delay is required. If it is not present then no delay will be executed.

~~Kill Switch~~

REMOVED. See: https://github.com/RavenSystem/esp-homekit-devices/releases/tag/HAA_2.5.0

Wildcard Actions

Wildcard actions are defined by the `"y[n]"` key within a service object.

Wildcard actions were introduced in firmware version `1.7.0`

Option	Key	Default	Type	Description
Target Value	`"v"`	none	float	Minimum target value to trigger action
Repeat	`"r"`	`0`	integer	Repeat each time value is assessed.
Action	`"0"`	none	array	Array of actions to perform

Wildcard actions are used to trigger state changes when a minimum value criteria has been met e.g. when a window cover is opened 68%, when the brightness of a lightbulb is set to 20%, when the fan speed reaches 40% or step 3 etc.

Wildcard actions are defined by `"y[n]"`. Where `"y[n]"` is normally `"y0"`, but in the case of a device that has a humidity sensor service (e.g. `"t":24` & `"t":25`) then `"y1"` refers to the humidity sensor.

The Target Value `"v"` is a mandatory key for the wildcard action. It defines the minimum target value required in order to trigger the defined action.

If multiple minimum target values are defined then the action associated with the highest minimum value that satisfies the condition will be performed e.g. in the example below two minimum values are defined; 35ºC & 20ºC. At 36ºC, only the action defined by the 35ºC object will be performed.

The Repeat `"r"` is used to cause the associated action to be performed each time the target value is assessed e.g. In a temperature sensor, if `"r":1` then the action will be performed each time the temperature is read and the target value is met.

The default (`"r":0`) triggers the action only the first time the target value is met.

The Action `"0"` is an array and contains the list of actions to perform when the target value is met. Any, or multiple service actions can be contained within the array.

Wildcard Action Example 1

In this temperature sensor example wildcard actions have been defined to perform actions when the temperature sensor reaches a temperature of 35ºC or 20ºC. Switching GPIO `12` on or off.

Each time the temperature sensor returns a reading above 35ºC the action will be repeated.

NOTE: In the case where the temperature sensor jumps from 18ºC to 35ºC between readings then only the 35ºC action will be performed.

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 22,
    "g": 14,
    "n": 4,
    "j": 30,
    "y0": [{
      "v": 35,
      "r": 1,
      "0": { "r": [{ "g": 12, "v": 1 }] }
    }, {
      "v": 20,
      "0": { "r": [{ "g": 12, "v": 0 }] }
    }]
  }]
}
```

State Inputs

State Inputs manage the state of a service. They monitor GPIO inputs and set the state of the service when conditions are met e.g. In a lock mechanism two separate GPIO inputs could be linked to buttons. One button will unlock the mechanism and one will lock it. Using state inputs linked to each of the buttons the mechanism can be locked or unlocked from each button (see below for an example).

State inputs are defined by an `"f[n]"` key contained within the service object. Where `[n]` is the unique identifier for the state input.

Each service can have its own set of state inputs and in most cases these are mapped to some of the Service Actions the service can perform. They can also be mapped to internal states the service monitors e.g. `"f3"` in a thermostat is used to change the target temperature.

Check the device page for details of the specific states.

State Input Example 1

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 4,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "f0": [{ "g": 14, "t": 1 }],
    "f1": [{ "g": 0, "t": 1 }]
```

```
    }]
  }
```

In this example of a lock mechanism input action `"f0"` is defined as a single press on the button connected to GPIO `14`. When the button is pressed service action `"0"` will be triggered and the mechanism is unlocked. When the button connected to GPIO 0 is pressed the mechanism will lock.

State Input Example 2

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 40,
    "d":18,
    "0": { "r": [{ "g": 12, "v": 1, "i": 0.5}]},
    "1": { "a": 0},
    "2": { "r": [{ "g": 12, "i": 1.5},{ "g": 12, "v": 1, "i": 0.5},{ "g": 12, "v": 1
    "3": { "a": 2 },
    "f3": [{ "g": 14, "t": 1 }],
    "f4": [{ "g": 14, "t": 0 }]
  }]
}
```

In this example of a garage door state input `"f3"` is defined for a garage door as being when the button/sensor connected to GPIO `14` is released (goes high) and `"f4"` is defined as being when the button/sensor is pressed (goes low) press. Either a `door is closing and receives opening order` service action is triggered (`"3"`) or a `garage door is closed` service action is triggered `"4"'`.

See Binary Inputs for details on how to define this option.

Status Inputs

Status Inputs were introduced in firmware version `1.8.0`

Status Inputs are very similar to State Inputs, but differ in that they only set the internal state of a service. They do not trigger any actions. e.g. Staying with the lock mechanism example. A Status Input can be used to set the state of the service when a key is used in the mechanism and it is manually locked or unlocked. See below for an example of how this might be done.

Status Inputs are defined by an `"g[n]"` key contained within the service object. Where `[n]` is the unique identifier for the Status Input.

Each service can have its own set of Status Inputs and in most cases these are mapped to some of the Service Actions the service can perform, but can also be mapped to internal states the service monitors.

Additionally a service can have button(s) or binary input(s) associated with a Status Input. When a specific state occurs then the Service Action status is set.

Check the device page for details of the specific states.

Status Inputs Example 1

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 4,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "g0": [{ "g": 14, "t": 1 }],
    "g1": [{ "g": 14, "t": 0 }]
  }]
}
```

In a modification to the State Inputs lock mechanism example. This service has no buttons associated with it, but does have an input to indicate the mechanism is locked or unlocked by a physical key turning the mechanism. When GPIO 14 goes high the mechanism is unlocked and the service status is set to unlocked: `"g0"` . When GPIO 14 goes low the mechanism is locked and the service status is set to locked: `"g1"` .

Actions

Action keys define what the service should do when state changes occur due to changes in inputs, or when specific triggers occur. Actions can take the form of changes to GPIO lines, system actions (reboot), Network requests, ICMP Ping requests and IR transmissions etc. Multiple actions can be defined for each accessory. Actions are specified by using one or more of the action types listed below.

Depending on the service type a range of different actions may be possible. See each service type for the list of actions. Refer to the **Actions** section in each service page.

Available Action Types are:

Action Type	Key	Description
Service Action	`"[n]"`	Actions specific to a service type

Action Type	Key	Description
Copy Action	`"a"`	Shorthand format for copying an action
Binary Output Action	`"r"`	Change state of a GPIO
Service Notification Action	`"m"`	Send notification to a service within same device
System Action	`"s"`	Perform a system action
Network Request Action	`"h"`	Send Network Request
IR Action	`"i"`	Send IR codes
UART Actions	`"u"`	Send commands via serial port
Set Characteristic Actions	`"c"`	Read value from a service characteristic and write it in other

NOTE: Within an action the copy action type `"a"` is mutually exclusive to all other actions and can only be used on its own. All other action types can be combined.

Actions Example 1

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 1,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "b": [{ "g": 0, "t": 1 }]
  }]
}
```

In this example we have defined an service of type `Switch` with two actions; action `"0"` sets GPIO 12 to a value of 0 and action `"1"` sets GPIO 12 to a value of 1. These actions are performed when the button `"b"` attached to GPIO 0 is pressed with a single press.

For the example the device type `"t": 1`, the value `"v": 0` and the button press type `"t": 1` are included for clarity, but as they are the default values they can be omitted from the config to save device memory.

Actions Example 2

```
{
  "c": { "l": 13 },
  "a": [
```

```
      {
        "t": 1,
        "0": { "r": [{ "g": 12, "v": 0 }] },
        "1": { "r": [{ "g": 12, "v": 1 }], "m": [[ 3, 1 ]] },
        "b": [{ "g": 0, "t": 1 }]
      },
      { "t": 1 },
      { "t": 2 }
    ]
  }
```

In this example we have added a couple of extra accessories with switch and outlet services and service notifications to the second action `"1"` of the first service. A notification is sent with value `1` to the third service, causing the switch to turn on.

NOTE: For simplicity of the example the details of the 2nd & 3rd accessories have not been shown

Actions Example 3

```
{
  "c": {},
  "a": [{
    "t": 21,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }] },
    "2": { "a": 0 },
    "b": [{ "g": 0 }]
  }]
}
```

In this example the third action `"2"` is the same as the second action `"0"`. Using the `"a"` shortcut to copy an action reduces the size of the MEPLHAA script.

Actions Example 4

```
{
  "c": {},
  "a": [{
    "t": 21,
    "0": { "r": [{ "g": 12, "v": 0 }] },
    "1": { "r": [{ "g": 12, "v": 1 }], "s": [{ "a": 1}] },
    "b": [{ "g": 0 }]
  }]
}
```

In this example the second action `"1"` sets GPIO 12 high and performs a system action `"s"` to enter the device setup mode (`"a": 1`).

Service Actions

Every service type has a defined set of service actions `"[n]"` . A service action is represented by an integer e.g. `"0"` , `"1"` etc.

A service action is specific to the service e.g. turn the switch or outlet on (`"1"`) or off (`"0"`), turn on a lightbulb (`"1"`), turn on the heating (`"3"`) in a thermostat device.

Almost all services have a minimum of two service actions; `"0"` & `"1"` . Exceptions are service types such as a temperature sensor or a humidity sensor.

Service actions have other actions embedded within them to effect the action state they represent e.g. a switch On service action `"1"` may set a GPIO to cause the *on* condition to occur e.g. `"1": { "r": [{ "g": 12, "v": 0 }] }`

See above examples of service actions.

Copy Actions

The copy action is defined by the `"a"` key array within a service object.

The copy action is mutually exclusive with all other action types i.e. if a copy action is present within an service definition then no other action types can be present.

If you have two or more service actions that do the same thing then a copy action can be used as a shortcut to reduce memory for the MEPLHAA script. The value defined in the key is a reference to another action to copy e.g. `"3": { "a": 0}` is a shortcut for action `"3"` to copy action `"0"` .

See Action Example 3 for an example.

Binary Outputs

The binary output array is defined by the `"r"` key within a service object.

Option	Key	Default	Type	Description
Output GPIO	`"g"`	none	integer	GPIO changed on result of an action
Inching	`"i"`	none	float	Time period before returning pin to previous state
Value	`"v"`	`0`	integer	Binary level to apply to the output GPIO

Option	Key	Default	Type	Description
Initial value	`"n"`	`0`	integer	Initial value when is declared for first time and open-drain options

The binary output array contains a list of GPIO level changes to perform when an action is invoked e.g. Set GPIO 12 low `{"g": 12, "v": 0}` or set GPIO 12 high for 2 seconds `{"g": 12, "v": 1, "i": 2}`

Sometimes multiple actions need to be performed when an action is invoked. In this case the binary output array will have multiple entries in its list e.g. `{"g": 12, "i": 1.5}, "g": 12, "v": 1, "i": 2}`

Each defined action has an array of binary outputs `"r"` e.g. relays, LEDs etc. These outputs have a number of different configuration options defined below.

Initial Value	Value	Description
`"n"`	0	GPIO is LOW (***Default***)
	1	GPIO is HIGH
	2	GPIO is LOW and set as open-drain
	3	GPIO is HIGH and set as open-drain

Output GPIO

Key	Value	Description
`"g"`	integer	GPIO used as the binary output

This option is mandatory for each action defined. The `"g"` specifies a GPIO to use as the binary output for the action performed e.g. `"g": 12` specifies GPIO `12` to be used as the binary output.

Value

Key	Value	Description
`"v"`	`0`	**Drive output low (*default*)**
	`1`	Drive output high

This option specifies the level to drive the GPIO output pin. Driving the pin low normally results in an "*off*" state and a high normally results in an "*on*" state.

Using `"n":1` key is possible to set a HIGH level when GPIO output appears for first time. Default value is LOW.

Inching

Key	Value	Description
`"i"`	`0.02` to very large	The time in seconds to wait

This option can be used to trigger an automatic return of the output to its previous state after a preset period of time has elapsed. The inching `"i"` value is a floating point number specifying the number of seconds e.g. `{ "i": 1.34 }` will result in a period of 1.34 seconds elapsing before the output pin is returned to its previous state. The time can be specified in hundredths of seconds.
NOTE: Positive values only are allowed

Service Notifications

Service notifications are defined by the `"m"` key array contained within an service object.

Option	Key	Default	Type	Description
`[ A, V ]`			array	Each service notification is an array of 1 or 2 elements.
Service	`A`	none	integer	Service index
Value	`V`	`0`	integer	Value of notification to send to service

Service notifications are notifications sent to another service within the same device. Check each device type's **Service Notifications** section to see what notifications can be sent to it and the values that can be sent e.g. assuming a device has two services; a switch (`"t":1`) & a lock mechanism (`"t":4`) respectively. The following can be added to the switch service to request the lock service to unlock when the switch is pressed: `{"m": [[ 2, 0 ]]}`

The service notification object is a list object and can contain multiple accessory notifications.

If more than one notification is contained in the list then the notifications will be sent in the order they appear in the list.

See Actions Example 2 for an example.

Target Service Number

The target service number is the index into the list of services of accessories in the `"a"` object. The target service index number begins with `1` i.e. the first service in the top level accessories key `"a"` .

If *n* services are defined in the `"a"` and `"es"` objects, then a value of `1` specifies the first service and a value of `n` specifies the last service.

Since HAA v7.5.0, relative index can be used:

- `0` : Target Service will be own service.
- `-N` : Target Service will be own service index -N services.
- `7000 + N` : Target Service will be own service index +N services.

Target Service Value

The target service value is the value to send as part of the notification.

In the case of a service notification the value of the `"v"` option is determined by the service type and is defined in the details of each accessory, in the "**Service Notifications**" section. Check this section in each service to determine the valid values that can be used for services.

There are some values common across all services used to control the internal state of a service:

Value	Definition
`-10000`	Disable service
`-10001`	Enable service
`-10002`	Toggle service state
`-20000`	Disable physical controls
`-20001`	Enable physical controls
`-20002`	Toggle physical controls state

These internal values replace a feature present in pre `2.5.0` releases called **Kill Switches**. Each service can be disabled by setting its internal state to `-10000` . In this state no automations will work.

A service can also have its physical controls disabled by setting its internal state to `-20000` . In this state no physical button/switches will work. However, it can always be managed with a HomeKit client and automations will work.

Service Notification Example

```
{
  "c": {},
  "a": [{
    "t": 2,
    "0": { "r": [{ "g": 12, "v": 0 }] },
```

```
      "1": { "r": [{ "g": 12, "v": 1 }] },
      "b": [{ "g": 0 }]
    },{
      "s":1,
      "0":{"m":[[ 1, -10000 ]]},
      "1":{"m":[[ 1, -10001 ]]}
    }]
  }
```

In the above example a HomeKit switch has been added that can change the internal state of the the first service to be enabled `"0"` or disable `"1"`. Thus the outlet switch connected to GPIO `0` can be disabled using the HomeKit client.

~~Kill Switch Notification~~

REMOVED. See: https://github.com/RavenSystem/esp-homekit-devices/releases/tag/HAA_2.5.0

System Actions

System actions are defined by the `"s"` key array within a service object.

Key	Value	Description
`"a"`	`0`	Reboot the device
	`1`	Enter device setup mode
	`2`	Search for OTA firmware update
	`3`	Disconnect and reconnect to Wifi
	`4`	Reconnect to Wifi resetting network interface

See Actions Example 4 for an example.

The system actions array is used to define system actions to perform when a service action is invoked.

Send Network Request Actions

Send Network Request Actions were first introduced in firmware version `1.3.0`

Send Network request actions are defined by the `"h"` key array within a service object.

Option	Key	Default	Type	Description
Host	`"h"`	none	string	Target host

Option	Key	Default	Type	Description
Port	`"p"`	`80`	integer	Port
Method	`"m"`	`0`	integer	Network method (HTTP, TCP, UDP...)
Headers	`"e"`	`"Content-type: text/html\r\n"`	string	Headers of HTTP request
URL	`"u"`	`"/"`	string	URL of HTTP request
Content	`"c"`	none	string	Content to send
Wait for response	`"w"`	`0`	float	Time to wait for a response when a TCP/HTTP request is sent

Send Network request actions can be used to initiate a TCP or UDP request to a specified target host and URL when an action occurs. They might be used to communicate to non-HomeKit devices from a HAA device e.g. turn on a WiFI lightbulb when a HAA device button is pressed, enter a record into a time series database etc.

The Host `"h"` is any valid IP address or a fully qualified domain name e.g. `"192.168.22.45"` or `"lounge-light.lan"`.

NOTE: If you are using a fully qualified domain name then make sure you have a DNS server on your network capable of resolving the name.

The Port `"p"` is any valid TCP port number.

The Method `"m"` is the communication method to use and can be one of:

Key	Value	Description
`"m"`	`0`	HTTP GET
	`1`	HTTP PUT
	`2`	HTTP POST
	`3`	TCP RAW
	`4`	TCP RAW (HEX format)
	`12`	Wake on LAN
	`13`	UDP RAW
	`14`	UDP RAW (HEX format)

When using the `HTTP` methods; 0, 1 or 2. The HTTP request is made up using the specified options to create a request and then sending it via the specified `Method` to the specified `URL` e.g. `http://<host>:<port>/<url>`

When using `TCP RAW` or `UDP RAW` the `Content` is sent as-is, without any header or other information. This method is useful when communicating to websockets or any other protocol outside of HTTP. When using `TCP RAW` method `4` or `UDP RAW` method `14` the `Content` must be in HEX format e.g. `{ "c":"01b364007aff04" }`. When using either of the TCP methods the `URL` is ignored.

The URL `"u"` is any valid URL e.g. `{"u": "cm?cmnd=Power%20OFF"}`

The content `"c"` is a string and can be any valid content.

When using `Wake on LAN` method `12` the `Content` must be a valid MAC address in HEX format e.g. `{ "c":"0a1b2c3d4e5f" }`.

Complete WOL example action:

```
"h":[{"m":12,"h":"255.255.255.255","p":9,"c":"a1b2c3d4e5f6"}]
```

NOTE: Any content returned by a Network Send Request is ignored by the device.

When using `Wait for response` with `"w":N` value, device will wait N seconds for a response from target host, but received data will not be processed unless Free Monitor service is used.

`"w":N` value is a float, but only one decimal can be used. Default value is `"w":0` (disabled) in normal actions, and `"w":1` in Free Monitor Service.

This feature is needed for some devices that needs to send any kind of acknowledgment to work properly (Eg. Philips HUE Bridge).

Content String Enrichment

First introduced in firmware version `2.5.1`

If the string contains the *magic* expression `#HAA@aacc` then the expression will be replaced by the characteristic `cc` value from the service `aa` e.g. `#HAA@0100` will be replaced with the value held by characteristic `0` from service `1`.

If the service value `00` is used then the characteristic value is read from the actual service containing the HTTP Request Action.

The *magic* expression can be used as many times as needed in the same content string.

The actual characteristic value entered in to the content string is dependent on its type:

Type	Content string substitution
bool	`0` and `1` will be used for false and true values
integer	An integer number, without decimals e.g. `87` or `-143`
float	A number with 3 decimals e.g. `24.520`

Send HTTP/TCP Request Example 1

You want to send a power command to a device at IP address 192.168.2.15 using the GET method:
`http://192.168.2.15/cm?cmnd=Power%20ON`

```
{
  "c": { "l": 13 },
  "a": [{
    "t": 1,
    "0": { "h": [{ "h": "192.168.2.15", "u": "cm?cmnd=Power%20OFF"}]},
    "1": { "h": [{ "h": "192.168.2.15", "u": "cm?cmnd=Power%20ON"}]},
    "b": [{ "g": 0, "t": 1 }]
  }]
}
```

In this example we are using the default method of GET `"m":0` and the default port of 80 `"p":80`. No content is required `"c"` as we are using a GET.

Send HTTP/TCP Request Example 2

In this example you want to send a time series event to an influxDB database to record the current temperature and humidity readings every time the sensor values are read e.g.

```
curl -X POST 'http://influxdb.lan:8086/write?db=haadb' --data-binary
'sensor,location=office temperature=20.230,humidity=38.000'
```

```
{
  "c": { "l": 13 },
  "a": [{
    "t":24,
    "g":14,
    "n":4,
    "j":30,
    "y0": [{
      "v": 0,
      "r": 1,
      "0": { "h": [{ "h": "dixnas1.lan", "p": 8086, "m": 2, "u": "write?db=testdb",
    }]
```

```
    }]
}
```

Send HTTP/TCP Request Example 3

In this example you want to send a time series event to ThingSpeak Internet server, to record the current temperature and humidity readings every time the sensor values are read e.g.

```
{
    "a": [{
        "t": 24,
        "g": 5,
        "y0": [{
            "v": -100,
            "r": 1,
            "0": {
                "h": [{
                    "h": "api.thingspeak.com",
                    "m": 2,
                    "e": "X-THINGSPEAKAPIKEY: GHAXXXXXXXXX\r\nContent-type: applica
                    "u": "update",
                    "c": "field1=#HAA@0000"
                }]
            }
        }],
        "y1": [{
            "v": 0,
            "r": 1,
            "0": {
                "h": [{
                    "h": "api.thingspeak.com",
                    "m": 2,
                    "e": "X-THINGSPEAKAPIKEY: GHAXXXXXXXXX\r\nContent-type: applica
                    "u": "update",
                    "c": "field2=#HAA@0001"
                }]
            }
        }]
    }]
}
```

Send IR Code Actions

Send IR Code Actions were introduced in firmware version `1.5.0`

Send IR Code actions are defined by the `"i"` key array within an accessory object.

Option	Key	Default	Type	Description
Frequency	`"x"`	`38`	integer	Frequency of IR marks, in KHz
Protocol	`"p"`	none	string	IR Protocol to use for the action
Repeats	`"r"`	`1`	integer	The number of times the IR command code will be repeated
Inter-Repeat-Delay	`"d"`	`100`	integer	Custom delay between repeats in milliseconds
Command	`"c"`	none	string	The IR command code to transmit
Raw Code	`"w"`	none	string	The raw IR code in HAA format to transmit

The Frequency `"x"` used in an action to override the global setting defined in the General Configuration for this action only.

The Protocol `"p"` defines the IR protocol to use. When used in a service it overrides the General Configuration setting for actions within the service. When used in an action it overrides the setting defined in the General Configuration setting for the action only.

The Repeats `"r"` defines the number of times the IR command will be transmitted. IR command codes are often sent multiple times in a row to ensure the receiving device is able to capture the code e.g. to turn on some Panasonic TVs, the same code must be sent 5 times.

The Inter-Repeat-Delay `"d"` defines the delay between repeats in milliseconds when `Repeats` are greater than 1. Valid values are from 10 to 65535. If none is declared, default value will be 100 milliseconds.

The Command `"c"` defines the actual command code to be transmitted. At least one definition of the Protocol (`"p"`) must exist, either in the General Configuration section, in the accessory (`"a"`) object, in any extra service, or in the action (`"p"`) object.

The Raw Code `"w"` defines the IR code to be transmitted. The `Raw Code` does not use any protocol (it is raw data) and does not require a `"p"` key in either the General Configuration section, accessory object or within the action object. **NOTE:** If `"w"` and `"c"` options are defined for the same action then the `"c"` option will take precedence and the `"w"` definition will be ignored.

Refer to the Using Infra-red in HAA for details on defining the protocol and protocol codes.

IR Code Example 1

Example of a HomeKit Switch that can turn on and off a Panasonic TV using an IR TX LED connected to GPIO 2

```
{
  "c": { "t": 2, "p": "HtDCAQ0?AQCAAK" },
  "a": [{
    "0": { "i": [{ "c": "aAkAiAhAaDbAaDaA" }]},
    "1": { "i": [{ "r": 5, "c": "aAkAiAhAaDbAaDaA" }]},
    "t": 1
  }]
}
```

UART Actions

UART Actions were introduced in firmware version `2.1.1`

UART Actions are defined by the `"u"` key array within an accessory object.

Option	Key	Default	Type	Description
UART Number	`"n"`	`0`	integer	The UART to use and format of command
Command	`"v"`	none	string	The command code to transmit
Inter-command Delay	`"d"`	`0`	integer	Delay in milliseconds between each command

The `UART Number` defines the UART port to use.

Key	Value	Description
`"n"`	`0`	UART0 and HEX format command **default**
	`1`	UART1 and HEX format command
	`10`	UART0 and TEXT format command
	`11`	UART1 and TEXT format command

See UART Configuration for details and settings.

The `Command` string in HEX or TEXT format, defining the command to send to the attached service e.g. `"41424344"`

Key	Value	Description
`"v"`	string	Data to be transmitted as HEX or TEX code string

The `Inter-command Delay` is the time in milliseconds to delay before transmitting subsequent commands to the same UART when multiple commands are contained in the same `"u"` array.

Key	Value	Description
`"d"`	`0 - 65535`	Delay between each transmitted command

UART Example

```
{
  "c":{"l":13,"b":[{"g":0,"t":5}],"r":[{"n":0, "s":9600, "p":0, "b":0}]},
  "a": [{
    "t": 1,
    "0": { "u":[{"n":0, "v": "48656c6c6f20576f726c64" }]},
    "1": { "u":[
      {"n":0, "d":100, "v": "537769746368204f6666" },
      {"n":10, "v": "ACK" }
    ]}
  }]
}
```

This is a general example of using a UART attached service. The external accessory is connected to the switch on UART0 (`"n":0`) and used 9600 baud, no parity bits and 0 stop bits (`"s":9600,"p":0,"b":0`). To turn off the device the string "*48656c6c6f20576f726c64*" must be sent to it. To turn on the device the string "*537769746368204f6666*", followed by "*4e6f77*" 100ms later.

Set Characteristic Actions

Set Characteristic Actions were introduced in firmware version `11.1.0 Peregrine`

Read value from a source service characteristic and write it in a target service characteristic, even if types are different.

Option	Key	Type	Description
`[ Ss, Sc, Ts, Tc ]`		array	Each action is an array of 4 elements.
Source Service	`Ss`	integer	Source Service index
Source Characteristic	`Sc`	integer	Source Characteristic index
Target Service	`Ts`	integer	Target Service index
Target Characteristic	`Tc`	integer	Target Characteristic index

HomeKit Visibility

Key	Value	Description
"h"	0	Service is not added to HomeKit
	1	**Service visible to HomeKit (*default*)**
	2	Service is hidden in Apple Home App, but is showed in **HAA Home Manager** and compatible Apps. It must be used with Services into Extra Services array.

If this option is set to `0` the service will become invisible to HomeKit but will still function as a service. Its hardware will be fully functional, as well as its status. All actions will function as normal too.

This option is applicable to **all** services.

Next

Now you know all about Accessory Configuration it is time to discover Service Types

Releases | Installation | Configuration | Device Types | Devices Database | Examples

▸ Pages 42

Home

Development
Releases
Build Instructions

Home Accessory Architect
Home Accessory
Installation
Setup Mode
HAA Home Manager App

Configuration
About
General
Accessory
| Actions

Service Types
Air Quality
Battery
Data History
Fan
Free Monitor
Garage Door
HAA iAirZoning
Heater Cooler
Humidifier
Light Sensor
Lightbulb
Lock Mechanism
Sensors
Power Measure
Security System
Stateless Button & Doorbell
Switch & Outlet
Temperature & Humidity
TV Service
Water Valve
Window Covering

Other
Devices Database
Infra-red

Examples
RavenCore v1
TODO List

Clone this wiki locally

```
https://github.com/RavenSystem/esp-homekit-devices.wiki.git
```