Radio Thermostat Company of America

Wi-Fi USNAP Module API

Version 1.0

March 29, 2011

Date	Author	Revision	Changelog
23 Mar 2011	Dan Goodman	V1.0	• Initial Release

LEGAL ADVISORY: Use of the Radio Thermostat Company of America Wi-Fi USNAP Module API ("Local API") is subject to the restrictions set forth in the Terms of Use set forth on the following webpage: http://radiothermostat.com/documents/RTCOA%20API%20Terms%20of%20Use%20033011.pdf . Without limiting the generality of the foregoing, you may not use the Local API for commercial purposes or modify, copy, distribute, transmit, display, perform, reproduce, publish, license, create derivative works from, transfer, or sell the Local API or any information, software, products or services related thereto unless expressly permitted in writing by Radio Thermostat Company of America. Use of the Local API beyond the scope of authorized access granted pursuant to said Terms of Use immediately terminates your right to use the Local API, and may render you liable for damages and injunctive relief in a court of law.

Table of Contents

1 Introduction

The Radio Thermostat API is an HTTP API that enables client programs to query the thermostat state, and to manage/control the thermostat operation. In the HTTP terminology, the thermostat will be a server that will respond to HTTP requests from client devices. The API is designed with a RESTful architecture in mind.

The API is expected to be used by a variety of client applications such as graphical interface applications running on Windows, MAC, or iOS devices. It can also be used by programs such as curl to facilitate control via non-interactive client applications. Further, the HTTP GET APIs can be exercised using a browser.

The Radio Thermostat API provides a set of operations on a typical thermostat device and has been designed to be independent of the underlying implementation of the thermostat. This document is intended for use with the Radio Thermostat Application Developer's Guide.

Note

The API described in this document is for firmware version 1.04.64 (and above) and System API version 113 and Thermostat API version 100. This firmware can be used with Radio Thermostat's CT-80 v2.18, CT-30e v1.75 and 3M50 v1.09 thermostats. Please upgrade to firmware version 1.04.64, or later, before using this API.

1.1 Resource Model and Data Format

The API is built around a resource model for the thermostat following the RESTful architecture paradigm. The general idea is the following:

- Each resource is represented by a URI
- The HTTP operation GET retrieves a (JSON) representation of the resource.
- The HTTP operation POST updates the resource and assumes a JSON representation of the resource as the values to be updated.
- HTTP 1.1 is supported except for features explicitly mentioned as not supported.

1.2 Guidelines for using the API

Radio Thermostat Company of America anticipates that over time, client applications will be required to support an installed base of devices that consist of multiple device models, where not all devices will support the most recent API versions. This API provides methods to query detailed version information of a device that includes the device model, the firmware version, and the wireless software version in addition to the API version. Client applications should use the detailed version information to dynamically determine the appropriate API to use while communicating with a particular device.

Any client using this API should follow these guidelines for using the API. These guidelines help in ensuring that the client application continues to work with later versions of the API.

- The API will continue to evolve as enhanced functionality is included with the API. These revisions to the API will be identified by the API version specification. Changes to the API include (but are not restricted to) the following:
    - New resources (URIs) could be added
    - New elements could be added to responses of GET and POST methods
    - Error codes are likely to be added and refined. Applications should be prepared to handle new error codes.
- Incremental API releases within a major API release will maintain backward compatibility. Care will be taken that applications using the older versions of the API do not break, provided they adhere to these guidelines. What this implies is:
    - The published resources (URIs) will continue to be accessible in all versions of the API for incremental API releases.
    - Various fields within the representation of these resources will carry the same meaning in all versions of the API.
    - No fields will be deleted from the HTTP GET response of a resource, although new fields may be introduced.
    - A POST with a published set of fields on a URI will always result in the same state transitions (effects) at the server end.
- Applications should not expect strict ordering of individual fields within a response. For example,

  ```
  { "x": 1, "y": 2 } is equivalent to { "y": 2, "x": 1 }
  ```

  The applications using this API should consider both the above representations to be equivalent.

- Applications should ignore any other fields in the response that it does not understand. For example, in a hypothetical representation with versions, *APIv1* and *APIv2*:

  ```
  APIv1:
  { "x": 1, "y":2 }

  APIv2:
  { "x": 1, "y": 2, "z":3 }
  ```

  An application that is implemented for APIv1 should work properly with APIv2 as well. Since the application is not aware of the field *z* and its semantics, it should silently ignore that field.

- If parsing arrays, the applications should ignore any additional indices in the array that they are not aware of. For example, in a hypothetical representation with versions, *APIv1* and *APIv2*:

  ```
  APIv1:
  [ [ "Sunday", "27" ],
    [ "Monday", "28" ],
    [ "Tuesday", "29" ],
  ]
  APIv2:
  [ [ "Sunday", "27", "Cloudy" ],
    [ "Monday", "28", "Rainy" ],
    [ "Tuesday", "29", "Sunny" ],
  ]
  ```

  An application that is implemented for APIv1 should work properly with APIv2. Since the application is not aware of the third index in the array (weather prediction, in this case), it should ignore that.

- If parsing arrays with header, applications should read the header to understand the relative positions of elements within the array. Applications should ignore any columns that they do not understand. For example, in a hypothetical representation with versions, *APIv1* and *APIv2*:

```
APIv1:
[ [ "date", "day" ],
  [ "21-10-2010", "sunday" ],
  [ "24-11-2010", "monday" ],
]

APIv2:
[ [ "time", "day", "date" ],
  [ "12:00", "sunday", "21-10-2010" ],
  [ "15:30", "monday", "24-11-2010" ],
]
```

  An application that is implemented for APIv1 should work properly with APIv2 as well. Since the application is not aware of the field *time* in the headers, it should ignore that column. The application should also infer the indices of the *days* and *date* based on the headers mentioned.

2 Thermostat API

2.1 Introduction

The thermostat API consists of three types of base resources, a) thermostat, b) system and c) cloud.

2.2 Thermostat Resource

The thermostat resource is a top-level composite resource that is used to retrieve current state of the thermostat and update its state (i.e., control the thermostat operation.)

The location of the thermostat resource is at **http://<ip-address>/tstat**. This location is returned as part of the service discovery protocol.

The thermostat resource provides a representation for most of the common attributes of a thermostat including the room temperature, operating mode, temperature setpoint (target). A client device typically polls the current state of the thermostat using the HTTP GET operation on the thermostat resource, and can use up to date thermostat state information to maintain its own internal representation of the current state of the thermostat. Likewise, a POST on the thermostat resource can be used to change the operating mode, establish new temperature setpoints, etc.

2.2.1 Thermostat API Version

The thermostat API version defines the API version that this device will use. This document is for Thermostat API version 100.

Table 1: Data Representation of Thermostat Version

Attribute	Description	GET	POST	Data Format
version	Thermostat API version	X		Integer representing the version number of the thermostat API supported by the device.

2.2.2 Thermostat Resource Data Representation

The thermostat resource encapsulates the following attributes.

Table 2: Data Representation for Thermostat Resource

Attribute	Description	GET	POST	Data Format
temp	Current temperature	X		Floating point representing value in degrees Fahrenheit.
tmode	Thermostat operating mode	X	X	Integer value: 0: OFF 1: HEAT 2: COOL 3: AUTO
fmode	Fan operating mode	X	X	Integer value: 0: AUTO

Attribute	Description	GET	POST	Data Format
				1: AUTO/CIRCULATE 2: ON
override	Target temperature temporary override status	X		Integer value: 0: Override is disabled 1: Override is enabled. Note: Firmware versions prior to 1.04 can return any non-zero value if override is enabled.
hold	Target temperature Hold status	X	X	Integer value: 0: Hold is disabled 1: Hold is enabled
t_heat	Temporary Target Heat Setpoint	X	X	Floating point representing temperature value in degree Fahrenheit.[#]
t_cool	Temporary Target Cool Setpoint	X	X	Floating point representing temperature value in degree Fahrenheit.[#]
a_heat	Absolute Target Heat Setpoint	X	X	Floating point representing temperature value in degree Fahrenheit.
a_cool	Absolute Target Cool Setpoint	X	X	Floating point representing temperature value in degree Fahrenheit.
a_mode	Absolute Target Temperature Mode		X	Integer representing the absolute target temperature mode. 0 – Disable Absolute Target Temperature Mode 1– Enable Absolute Target Temperature Mode
t_type_post	Target Temperature POST type	X		Integer value that indicates whether a POST on t_heat/t_cool will result in temporary or absolute temperature change. 0: Temporary Target Temperature 1: Absolute Target Temperature 2: Unknown This attribute is deprecated and will be obsoleted in future versions of the API.[#]
tstate	HVAC Operating State	X		Integer value: 0: OFF 1: HEAT 2: COOL

Attribute	Description	GET	POST	Data Format
				Note: This functionality may not be available in all models of the thermostat.
fstate	Fan Operating State	X		Integer value: 0: OFF 1: ON Note: Only available with CT-30
time	Thermostat's internal representation of time	X	X	JSON object with the following fields: day, hours, minutes. **day**: Integer value representing the day of the week, with day 0 being Monday. **hour**: Integer value representing number of hours elapsed since midnight. **minutes**: Integer value representing number of minutes since start of the hour.

- The default behavior of POST on *t_heat* and *t_cool* is to update the *temporary* target temperature. Some custom flavors of the firmware update the *absolute* target temperature when *t_heat* or *t_cool* values are updated. This distinction can be made by referring to the *t_type_post* attribute. This differing behavior is deprecated and will be obsoleted in future versions of the API. No comment can be made about whether the t_heat/t_cool data returned in a GET /tstat/ response indicates temporary or absolute target temperatures.

2.2.3 Thermostat Program Resource

The thermostat maintains two programs – a **heat** program and a **cool** program. Every program entry consists of *time* and the corresponding *temperature setpoint*. Every day of the week can have a set of *time-setpoint* pair programmed in the thermostat.

2.2.3.1 Thermostat Program for a Day

The thermostat program for a single day (for either the heat or cool) mode can be accessed directly using a URI as follows:

- **http://<ip-address>/tstat/program/<mode>/<day>**

where <*mode*> is either *heat* or *cool*. The <*day*> is one of *mon, tue, wed, thu, fri, sat*, or *sun*.

Table 3: Data Representation for program for a day resource

Data Format	Explanation
"d":[a1, b1, a2, b2, a3, b3, …]	Where: value of **d** is one of 0, 1, 2, 3, 4, 5, 6 representing *mon, tue, wed, thu, fri, sat* and *sun* respectively. value of **a<i>** is the time expressed as minutes

Data Format	Explanation
	from the start of the day (Integer) value of **b<i>** is the temperature at time **a<i>** expressed in degree Fahrenheit (Floating point)

2.2.3.2 Thermostat Program for a week

The programs for the entire week are also available at the following URI's:

- **http://<ip-address>/tstat/program/heat**
- **http://<ip-address>/tstat/program/cool**

Each of the heat or cool programs consists of one program for each day of the week. The data format of the response consists of all the attributes corresponding to the days of the week, from 0 to 6. The rest of the format is as explained in Section 2.2.2.1.

2.2.4 Thermostat Model

The thermostat model resource is a string that provides the model number and firmware version of the thermostat. The thermostat model resource is available at:

- **http://<ip-address>/tstat/model**

Table 4: Data Representation of Thermostat Model

Attribute	Description	GET	POST	Data Format
Model	Thermostat model and version	X		String representing the model and version number.

2.2.5 Thermostat LED

The thermostat LED resource provides control over the LEDs available on the thermostat. The thermostat led resource is available at:

- **http://<ip-address>/tstat/led**

Table 5: Data Representation of Thermostat LED

Attribute	Description	GET	POST	Data Format
energy_led	Energy LED Status Code		X	Integer that represents: 0 – Off 1 – Green 2 – Yellow 4 – Red

Note: The LED state is volatile. The state of the LED may change when the WiFi module or the thermostat reboots, or when the thermostat modifies it to show energy status.

2.2.6 Thermostat Messaging Areas

The thermostats typically have one of two types of messaging areas, Price Messaging Area (PMA) and User Messaging Area (UMA). Some models may not have a messaging area. The thermostat messaging areas resource allows control of these user areas. The thermostat messaging area resources are available at:

- **http://<ip-address>/tstat/pma**
- **http://<ip-address>/tstat/uma** (Only available on CT-80)

Table 6: Data Representation of Thermostat User Messaging Area (uma)

Attribute	Description	GET	POST	Data Format
line	The line no. of the messaging area		X	Integer representing the line number to write to. Valid Values: 0, 1
message	The message to be displayed		X	String containing the desired message to display.

Table 7: Data Representation of Thermostat Price Messaging Area (pma)

Attribute	Description	GET	POST	Data Format
line	The line no. of the messaging area		X	Integer representing the line number to write to. Valid Values: 0, 1, 2, 3
message	The message to be displayed		X	String containing the desired message to display. **Note:** The PMA can only display numbers

Note: The PMA/UMA state is volatile. Their state may change when the WiFi module or the thermostat reboots, or when the thermostat modifies it to show updates.

2.3 System Resource

2.3.1 Basic system information

Table 8: Data Representation of /sys

URI	GET	POST	Data Format
/sys	X		The data returned has the following information: **UUID**: unique identifier for the device (String) **api_version**: HTTP API version. This specification is for api_version=113. (Integer) **fw_version**: Firmware version (String) **wlan_fw_version**: Underlying WLAN firmware version (referred from WiFi-certification documentation)(String)

URI	GET	POST	Data Format
			Example: { "uuid": "0021e82d978f", "api_version":113, "fw_version":"1.04.64", "wlan_fw_version":"v10.105576" }

2.3.2 System Service

Table 9: Data Representation of /sys/services

URI	GET	POST	Data Format

/sys/services	X		The data returned has the following information:
			services_names: A list of service names available on the device. This is the same as that announced on the SSDP protocol by this device. This is returned as a JSON array of strings. Each member of the array identifies one service.
			httpd_handlers: A list of all the URIs that are available along with the kind of operations (GET/POST) available on them. This is returned as JSON attribute-value pairs of the form “u1”:[a1, b1] where,
			ui: the URI
			ai: 1-GET is allowed, 0-GET is not allowed
			bi: 1-POST-allowed, 0-POST is not allowed
			Note that while all the URIs on the /sys/ resource are listed, only the top-level URIs of other resources are listed.
			
			Example:
			{
			"service_names": [
			"com.rtcoa.tstat:1.0",
			"devices.controller.tstat:1.0"
			],
			"httpd_handlers":{
			"/tstat":[1,1],
			“/cloud”: [1,1],
			"/sys/network":[1,1],
			“/sys/updater":[0,1],
			"/sys/filesystem":[0,1],
			"/sys/firmware":[0,1],
			"/sys/fs-image":[0,1],

URI	GET	POST	Data Format
			"/sys/fw-image":[0,1], "/sys/command":[0,1], "/sys/services":[1,0], "/sys/mode":[1,1], "/sys/name":[1,1], “/sys/reboot”:[1,1], } }

2.3.3 System Name

Table 10: Data Representation of /sys/name

URI	GET	POST	Data Format
/sys/name	X	X	**name**: Descriptive system name for easy identification(String) Example: {"name":"thermostat-2D-97-8F"}

2.3.4 System Command Handler

Table 11: Data Representation of /sys/command

URI	GET	POST	Data Format
/sys/command		X	**command**: Command to be issued to command handler. Currently only “reboot” command is supported. (String)

2.3.5 System Operating Mode

Table 12: Data Representation of /sys/mode

URI	GET	POST	Data Format
/sys/mode	X	X	**mode**: Indicates system operating mode. 0 – provisioning, 1 – normal (Integer) Example: {"mode": 0} • A POST with the value of mode as 0, resets the device back into provisioning mode.

2.3.6 Network Configuration

Table 15: Data Representation of /sys/network

URI	GET	POST	Data Format
/sys/network	X		For GET: The network information consists of the following attributes in JSON format **ssid** (String): SSID of the configured network **bssid** (String): BSSID of the configured network **channel** (Integer): WLAN Radio Channel **security** (Integer): Security mode as defined in /sys/scan **ip** (Integer): Indicates whether IP address is configured via DHCP or statically assigned. 0 – static, 1 – dhcp. **ipaddr** (String): IP address **ipmask** (String): Subnet mask **ipgw** (String): Gateway **ipdns1** (String): Primary DNS Server# **ipdns2** (String): Secondary DNS Server# **rssi** (String): Signal Strength

(**Note:** *API versions before v112 included the passphrase member in the GET response. This has been discontinued for later versions of the API*.)

2.4 Deprecated APIs

The following APIs have been deprecated from earlier versions:

- /sys/info
- /cloud/url
- /cloud/authkey
- the *t_type_post* field in the /tstat/ resource
- the behavior that sets absolute target temperatures in response to *t_heat* and *t_cool* fields in the /tstat/ resource

2.5 Using the API with Curl

The easiest way to explore the API is using curl command-line program that can be used to do both GET and POST operations. The following examples illustrate some common use-cases.

2.5.1 Get Thermostat State

The most common operation is to retrieve the current dynamic state of the thermostat resource.

```
$ curl http://192.168.1.101/tstat
```

```
{"temp":76.50,"tmode":2,"fmode":0,"override":0,"hold":0,"t_cool":85.00,"time":{"day":6,"hour":12,"minute":54}}
```

2.5.2 Set target heat temperature to 72 in HEAT mode

The following curl command will set the thermostat operating mode to HEAT and the target heat temperature to 72.

```
$ curl -d '{"tmode":1,"t_heat":72}' http://192.168.1.101/tstat
{"success": 0}
```

2.5.3 Set target heat temperature to 74 and enable hold

The following curl command switches the thermostat to HEAT mode, enables hold state, and sets the target temperature to 74.

```
$ curl -d '{"tmode":1,"t_heat":74,"hold":1}' http://192.168.1.101/tstat
{"success": 0}
```

2.5.4 Set target cool temperature to 80

The following curl command sets the thermostat operating mode to COOL, enables temporary override, and sets the target temperature to 80.

```
$ curl -d '{"tmode":2,"t_cool":80}'  http://192.168.1.101/tstat
{"success": 0}
```

2.5.5 Set fan mode to ON

The following curl command sets the fan mode to ON.

```
$ curl -d '{"fmode":2}' http://192.168.1.101/tstat
{"success": 0}
```

2.5.6 Get Heat Program Information for Monday

The following command retrieves the heat program for Monday (day 0). The values in the program are: 70 degrees at 12:50 AM (50 minutes into the day), 71 degrees at 1:40 AM (100 minutes into the day), and so on.

```
$ curl http://192.168.1.101/tstat/program/heat/mon
{"0":[50,70,100,71,150,72,200,73]}
```

2.5.7 Get Heat Program Information for the Whole Week

```
$ curl http://192.168.1.101/tstat/program/cool
{"0":[450,80,550,90,650,100,750,95],"1":[360,78,480,85,1080,78,1320,82],"2":[360,78,480,85,1080,78,1320,82],"3":[360,78,480,85,1080,78,1320,82],"4":[360,78,480,85,1080,78,1320,82],"5":[360,78,480,85,1080,78,1320,82],"6":[360,78,480,85,1080,78,1320,82]}
```

2.5.8 Update Cool Program for Tuesday

```
$ curl -d '{"1":[120,80,240,81,360,82,480,83]}'
http://192.168.1.101/tstat/program/cool/tue
{"success": 0}
```

In the above POST message, index "1" corresponds to the day Tuesday. Note that the index and the day specified (Tuesday) in the URL must match.

This POST message programs the cool settings for Tuesday as follows.

2am: 80 degrees
4am: 81 degrees
6am: 82 degrees
8am: 83 degrees

2.5.9 Get the Thermostat Model and Firmware Version

$ curl http://192.168.1.101/tstat/model

{"model":"CT80 V2.14T"}

The above shows that the thermostat is a CT-80 running thermostat firmware version v2.14T. Note that the thermostat firmware version is different from the firmware version running on the USNAP Wi-Fi module. Together, the thermostat model and thermostat firmware version enable the application developer to correctly utilize model specific features for a particular device. In general, an application developer should use all of the following version indicators to program a particular device correctly:

1. Thermostat model and Thermostat firmware version information as returned by the */tstat/model* command. These provide an insight into the capabilities of the hardware.
2. Thermostat API version information as returned by the */tstat/version* command. This provides information on the thermostat operations supported by a particular version of the USNAP WiFi firmware.
3. The thermostat System API version information as returned by the */sys* command in the *api_version* field. This provides insight into the system level capabilities supported by a particular version of the USNAP WiFi firmware.

2.5.10 Get basic system information

```
$ curl http://192.168.1.101/sys
{"uuid":"0021e82d94e4","api_version":113,"fw_version":"1.04.64","wlan_fw_version"
:"v10.105576"}
```

The above command fetches the basic state of the system

2.5.11 Set system name

The following command names the device as "tstat-livingroom".

```
$ curl -d '{"name":"tstat-livingroom "}' http://192.168.1.101/sys/name
{"success": 0}
```

The default value for name is "thermostat-xy-za-bc". Where xy-za-bc represent the last 6 hexadecimal digits of the MAC address of the thermostat.

2.5.12 Get system mode

```
$ curl http://192.168.1.101/sys/mode
{"mode":1}
```

The above response indicates that the system is in “normal” mode.

2.5.13 Reboot system

The following command reboots the system.

```
$ curl -d '{"command":"reboot"}' http://192.168.1.101/sys/command
{"success": 0}
```

2.5.14 Get services list

The following command retrieves the list of services available on the device:

```
$ curl http://192.168.1.101/sys/services
{"service_names":["com.rtcoa.tstat:1.0","devices.controller.tstat:1.0"],"
httpd_handlers":{"/tstat":[1,1], "/cloud": [1,1],
"/sys/network":[1,1],"/sys/updater":[0,1],"/sys/fs-image":[0,1],"/sys/fw-
image":[0,1], "/sys/command":[0,1], "/sys/services":[1,0],
"/sys/mode":[1,1], "/sys/name":[1,1]}}
```

2.6 Limitations and Caveats

2.6.1 Concurrent Connections

The web server is single threaded and connection requests are processed serially. Thus a new connection request is processed only after an existing connection has been terminated.

2.6.2 Chunked encoding

Chunked encoding is NOT supported by the web server.

2.6.3 Interaction between t_heat, t_cool and t_mode

The thermostat resource has two temperature setpoints: t_heat and t_cool (for heat and cool setpoints). Depending on the current operating mode, either the heat setpoint or the cool setpoint (or none if the mode is OFF) is returned.

Additionally, t_heat can be set only t_mode is HEAT and t_cool can only be set when t_mode is COOL. If t_mode is not specified correctly, then the mode will be switched automatically.

2.6.4 Differences between thermostats

Depending on the thermostat model, some advanced modes may not be available.

Table 1: Differences between Thermostats

Feature	CT-30	CT-80
`Auto/Circulate Fan Operating Mode (fmode is AUTO/CIRCULATE)`	Not Available	Available
`Automatic Thermostat Operating Mode (tmode is AUTO)`	Not Available	Available
`Number of Program Settings per day (per mode – heat or cool)`	4	7
`Override status after setting a temporary target using the HTTP API (e.g. command in Sec 2.5.2 and Sec 2.5.4)`	When override status is read, it is reported as 0. This is a known issue in thermostat firmware.	Override is reported back correctly as 1.

2.6.5 Differences in behavior with firmware version

This document is compatible with firmware version 1.04.64, or later. Please upgrade to this firmware version.

Apart from differences listed in the preceding sections of the document, there is one difference in the way the firmware responds to any HTTP request. All the HTTP interactions (apart from upgrade) operate on JSON encoded data. All the data being sent and received from the device will be in JSON format. For example, for firmware versions prior to 1.04, a successful POST response is a string: `"Tstat Command Processed".` For firmware versions 1.04 and later, a successful POST response is also JSON encoded as follows: `"{"success": 0}".`

2.7 Success and Error Codes

2.7.1 Invalid URI

In response to Invalid URI requests a 404 HTML page redirecting back to the home page is returned.

2.7.2 Common Scenarios

All the operations return the following common success/error messages.

- For success
  ```
  { "success" : 0 }
  ```

Note1: In the firmware 1.04.64 or lower,

- a success is reported even when an invalid field is included in the POST request.
- a success is reported if multiple fields are POSTed and only a few of them are invalid.

Note2: In future releases of the firmware,

- if no valid field is specified an error would be returned.
- if multiple fields are POSTed, the return will be {"success":n} where n indicates the number of valid fields in the request that were processed.

- For error
  ```
  { "error": -1 }
  ```
  (Most likely cause, if some field is missing in the POST data)
- For invalid URLs
  ```
  { "error_msg" : "Invalid HTTP API"}
  ```

2.7.3 Thermostat API Errors

There may be some transient errors when communicating with the thermostat firmware. When these errors occur, the thermostat specific values are reported as -1. For example:

- { “tmode”: -1, “fmode”: 2, “temp”: -1, “hold”: 1}

In the above JSON string, there was an error while retrieving the values of *tmode* and *temp*.

3 Marvell Service Discovery Protocol

3.1 Introduction

The Marvell Service Discovery Protocol is a simple protocol designed to enable client programs to discover (web/http) services offered by devices using Marvell Wireless Micro-controller solution. The protocol itself is modeled after the Simple Service Discovery Protocol (SSDP), which was the basis for UPNP Service Discovery.

Note **The description in this document is for service discovery protocol version 1.**

3.2 Specification

3.2.1 Message Types:

Message Type	Description
WM-DISCOVER	The WM-DISCOVER request is sent by a client to find services. The request is sent on UDP multicast address 239.255.255.250 at port 1900. The message includes the services that the client is interested in.
WM-NOTIFY	The WM-NOTIFY is a response to the WM-DISCOVER request when a device offers a matching service. It's a unicast UDP message to the sender of the WM-DISCOVER request. The response message includes a list of the matching services that the device offers along with the Base URI for each service.
WM-PRESENCE	The WM-PRESENCE message is a multicast of the services offered by a device. ***This is not yet implemented***.

3.2.2 Protocol Version

The current service discovery protocol version is 1.0. This is included in all messages, and will enable protocol upgrade in the future.

3.2.3 Services

Conceptually, a service is identified by a **name** and a **location**. The goal of the service discovery protocol is to find the location of a service given its name. More specifically, a service definition includes the following fields.

Field	Description	Mandatory/Optional
Location	The service location is the URI at which the service is located. The primary focus is on HTTP based services, so the Location would be something like: `http://<IP-Address>/<Service-Prefix>` that has the IP-address of the device and the Service-Prefix for the URI's for the service.	Mandatory
Manufacturer Qualified Name	A service is identified by a name (string). In order to make the services unique, we use the Reverse-DNS naming convention to identify services – a convention that has been used in Java (See: http://en.wikipedia.org/wiki/Reverse-DNS).	Optional, if device qualified name is present. Otherwise mandatory.
Device Qualified Name	This is an alternate service name (string) that can be used to define a generic service that is not tied to a specific manufacturer.	Optional, if manufacturer qualified name is present. Otherwise mandatory.
Manufacturer Fingerprint (ID)	The service can carry a manufacturer fingerprint (or ID) that can be used by manufacturers to ensure a level of protection and security.	Optional

3.2.4 Service Matching

Service discovery requests send a service name string that the client is interested in. The query may include a "*" wild-card at the end of the string. A service match occurs:

- If the wildcard is used, then the service name prefix in the discovery request exactly matches with the initial part of one of the service names.
- If the wildcard is not used, then the service name in the discovery request should match exactly with one of the service names.

If the service request contains only the wildcard, without a prefix string, all the services available on the device match.

3.2.5 Message Flow

- A client that wants to discover WM services will send a discovery request to 239.255.255.250:1900 (UDP).
- If a device with matching services is present, it sends a discovery response (unicast UDP message)
- Every thermostat device announces the availability of services on boot-up (start of services). (not implemented yet)
- Every thermostat device should also announce availability of services on starting network services. Network services have to be stopped before entering deep sleep and restarted, if required, on waking up from deep sleep. (not implemented yet)

3.2.6 Message Formats

Message Type	Message Format	Notes
WM-DISCOVER	TYPE: WM-DISCOVER VERSION: 1.0 SERVICES: <service-request-string>	The service request string may consist of one or more service requests. Multiple requests are treated as a request to match any one of the services. Multiple requests are separated by a comma (","). Any service request may have a "*" at the end.
WM-NOTIFY	TYPE: WM-NOTIFY VERSION: 1.0 SERVICE: <service-name> LOCATION: <location-uri>	There may be one or more SERVICE/LOCATION pairs. One for each of the matching services. If a service is described by both a manufacturer and a device qualified name, the two names will be separated by a semicolon (";").

3.3 Limitations and Caveats

The protocol does not currently have any expiration information or announcements of termination of services.

3.4 Example Program to discover a service

3.4.1 Source Code

```
/*
 *  Copyright (C) 2009-2010, Marvell International Ltd.
 *  All Rights Reserved.
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>

#define MAX_BUF 4096
#define LOCATION_HDR "Location:"
#define SSDP_ADDR "239.255.255.250"
#define SSDP_PORT 1900

unsigned int quiet = 0;
char ip[16];

void parse_options(int argc, char *argv[])
{
  int i = 1;
  memset(ip, 0, sizeof(ip));

  while (i < argc) {
        if (!strcmp(argv[i], "--quiet"))
              quiet = 1;
        else if (!strcmp(argv[i], "--ip")) {
              snprintf(ip, sizeof(ip), "%s", argv[i + 1]);
              i++;
        } else
              printf("Ignoring unknown option %s\n", argv[i]);

        i++;
  }
}

int main(int argc, char *argv[])
{
  int sock, ret, one = 1, len, ttl = 3;
  struct sockaddr_in cliaddr, destaddr;
  struct timeval tv;
  char buffer[MAX_BUF] = "TYPE: WM-DISCOVER\r\nVERSION: 1.0\r\n\r\nservices:
com.marvell.wm.system*\r\n\r\n";
  char *token;
  int count = 0;
  struct ip_mreq mc_req;
```

```
  parse_options(argc, argv);

  /* Create socket */
  sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if (sock < 0) {
        printf("%s: Cannot open socket \n", argv[0]);
        exit(1);
  }

  /* Allow socket resue */
  ret = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, (char *)&one,
                sizeof(one));
  if (ret < 0) {
        printf("%s: Cannot prepare socket for reusing\n", argv[0]);
        exit(1);
  }

  /* Set receive timeout */
  tv.tv_sec = 3;             /* 3 second timeout */
  ret = setsockopt(sock, SOL_SOCKET, SO_RCVTIMEO, (struct timeval *)&tv,
                       sizeof(tv));
        if (ret < 0) {
                printf("%s: Cannot set receive timeout to the socket\n",
argv[0]);
                exit(1);
        }

  ret = setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, (void *)&ttl,
               sizeof(ttl));
  if (ret < 0) {
        printf("%s: Cannot set ttl to the socket\n", argv[0]);
        exit(1);
  }

  /* construct a socket bind address structure */
  cliaddr.sin_family = AF_INET;
  if (strlen(ip) > 0)
        cliaddr.sin_addr.s_addr = inet_addr(ip);
  else
        cliaddr.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
	cliaddr.sin_port = htons(0);

	ret = bind(sock, (struct sockaddr *)&cliaddr, sizeof(cliaddr));
	if (ret < 0) {
		printf("%s: Cannot bind port\n", argv[0]);
		exit(1);
	}

	/* construct an IGMP join request structure */
	mc_req.imr_multiaddr.s_addr = inet_addr(SSDP_ADDR);
	mc_req.imr_interface.s_addr = htonl(INADDR_ANY);

	/* send an ADD MEMBERSHIP message via setsockopt */
	if ((setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
			(void*) &mc_req, sizeof(mc_req))) < 0) {
		perror("setsockopt() failed");
		exit(1);
	}

	/* Set destination for multicast address */
	destaddr.sin_family = AF_INET;
	destaddr.sin_addr.s_addr = inet_addr(SSDP_ADDR);
	destaddr.sin_port = htons(SSDP_PORT);

	/* Send the multicast packet */
	len = strlen(buffer);
	ret = sendto(sock, buffer, len, 0, (struct sockaddr *)&destaddr,
			sizeof(destaddr));
	if (ret < 0) {
		printf("%s: Cannot send data\n", argv[0]);
		exit(1);
	}

	/* quiet the noise */
	if (quiet == 0)
		printf
		    ("Sent the SSDP multicast request and now waiting for a
response..\n");

	while (1) {
```

```
            /* Wait for response */
            len = sizeof(destaddr);
            ret =
                recvfrom(sock, buffer, MAX_BUF, 0,
                        (struct sockaddr *)&destaddr, &len);

            if (ret == -1)      /* time out */
                  break;
            count++;      /* Valid response */

            /* Parse the response */
            token = strtok(buffer, "\r\n");
            while (token != NULL) {
                  if (!strncasecmp(token, LOCATION_HDR, strlen(LOCATION_HDR))) {
                        printf("Found a wireless microcontroller, base URI: %s\n",
                              token + strlen(LOCATION_HDR));
                        break;
                  }
                  token = strtok(NULL, "\r\n");
            }
      }

      /* send a DROP MEMBERSHIP message via setsockopt */
      if ((setsockopt(sock, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                        (void*) &mc_req, sizeof(mc_req))) < 0) {
            perror("setsockopt() failed");
            exit(1);
      }

      if (quiet == 0)
            printf("Found %d device(s)\n", count);

      return 0;
}
```

3.4.2 Building and executing the sample program

Use wm_demo_discover.c to discover wireless microcontrollers on a network. This utility has the capability to report multiple devices, if present on the network. Compile the above code using the following command:

```
# gcc -o wm_demo_discover wm_demo_discover.c
```

wm_demo_discover can take options:

1. --quiet : Reports scriptable output. Without this, a verbose output is presented
2. --ip <ip-addr>: If you laptop has multiple interfaces, you can choose which network to multicast this packet over using this option. The IP address that is specified should be the address of the interface (wired or wireless) that is connected to the same wireless access point as the thermostat that you are attempting to discover.
3. --search <search str>: The service to search for.

The thermostat devices can now be found using:

./wm_demo_discover --search "com.rtcoa.tstat*" --ip 192.168.1.103

Found a device, base URI: http://192.168.1.104/tstat

Found a device, base URI: http://192.168.1.107/tstat