Bluetooth®

Bluetooth Mesh Models

Technical Overview

Bluetooth® technology is a wireless standard with agreed, formal specifications that support global interoperability between devices from different manufacturers. The same thinking went into Bluetooth mesh. Luminaires, sensors, switches, and other types of devices *just work* when installed in a state-of-the-art smart building, with interoperability assured.

Interoperability is a benefit of standardization across every layer of the entire communications stack — from the physical layer, dealing with the analogue world of radio at the bottom, to user level behaviors that products may exhibit at the top. The Bluetooth mesh specifications define those product behaviors in terms of granular, standard building blocks called *models*. This paper provides a guided tour of Bluetooth mesh models.

Author: Martin Woolley

Version: 1.0

Revision Date: 27 March 2019

Revision History

Version	Date	Author	Changes
1.0	27 March 2019	Martin Woolley	Initial Version

table of contents

table of contents (cont.)

1.0 What is a Mesh Model?

As noted in the Bluetooth mesh glossary of terms, a model:

定义一组状态，状态转换，状态绑定，消息和其他相关行为。 节点中的元素必须支持一个或多个模型，并且它是定义元素具有的功能的模型。 有许多模型由蓝牙信号定义，其中许多模型被故意定位为“ 通用 ”模型，在各种设备类型中具有潜在的效用。

> **“...defines a set of States, State Transitions, State Bindings, Messages, and other associated behaviors. An Element within a Node must support one or more models, and it is the model or models that define the functionality that an Element has. There are a number of models that are defined by the Bluetooth SIG, and many of them are deliberately positioned as “generic” models, having potential utility within a wide range of device types.”**

The glossary and the Bluetooth Mesh Technology Overview are recommended reading if any of these terms are new to you.

Essentially, models are specifications for *standard software components* that, when included in a product, determines what it can do as a mesh device. Models are self-contained components and products will incorporate several of them. Collectively, from a network’s point of view, models make the device what it is. 从本质上讲，模型是标准软件组件的规范，当包含在产品中时，它决定了作为网状设备它可以做什么。模型是独立的组件，产品将包含其中的几个组件。总的来说，从网络的角度来看，模型使设备成为现实。

1.1 State

Models contain states. States are data items that indicate the condition of the device, such as on/off or high/low. States may be simple, containing only a single value, or composite, containing multiple fields, similar to a struct in programming languages like C.

In some cases, there are relationships defined between state items. These relationships are called *state bindings*. A state binding indicates that if one of the states in the relationship changes, then the other one needs to have its value recalculated. Sometimes state bindings are conditional and may be enabled or disabled by some other state. Developers must implement the required logic for any state bindings that are defined for the models they are using and ensure that logic is executed whenever required.

Conversely, where state bindings are not explicitly defined in the Bluetooth Mesh Model Specification, states must act independently. For example, if the generic on/off state indicates that a device is currently off, increasing the generic level state should have no user-discernible effect. Switching the device on by setting the generic on/off state to 1 should not only switch the device on, but it should begin functioning at the level that has been set. This can be readily understood if you consider a rotary dimmer switch that is rotated to change the level of the lights in the room but can also be pressed to switch them on or off. You can rotate the control when the lights are off and nothing will appear to happen, but if you then press the switch, with it in the same rotated position, the lights will come on at the selected level of brightness.

1.2 Categories of Model

Models are classified as being either *clients*, which do not contain state, or *servers*, which do. State is the term used for a data item which represents the condition that some aspect of a device is in, such as whether it is on or off or what level it is turned up to.

Some server models are associated with another server model with a name that is similar but includes "SetUp" in it. For example, the *Sensor Server* model has an associated *Sensor Setup Server* model. SetUp server models are technically no different to other server models in that they contain a state and produce and consume particular types of messages. Their purpose is to allow the separation of a model's configuration settings from the main model state items so that distinct access control policies can be applied. It is common to allow a network administrator to configure a model's associated settings via its SetUp Server model but not allow standard users to do this.

1.3 Model Communication and Behaviors

Models talk to each other by sending and receiving messages. There are numerous types of message, and these are defined as part of the specification for each model so that it is clear what types of message a model can produce and what types of message it can receive and understand.

Messages either communicate a state value to other devices or they change a state value, eliciting a response, often visible, from a device.

Models defined by the Bluetooth Special Interest Group (SIG) in the Bluetooth Mesh Model Specification are known as Bluetooth SIG models. Vendors may define their own models too, and these are known as vendor models. Vendor models should be used with caution and only when there is no possible way to use Bluetooth SIG models to meet the requirements.

Models can have specified dependencies on other models. A model may extend another model, a process whereby the first model adds states to the second model. A model may also require that a model which extends it be present. Models that do not extend other models are known as *root models*.

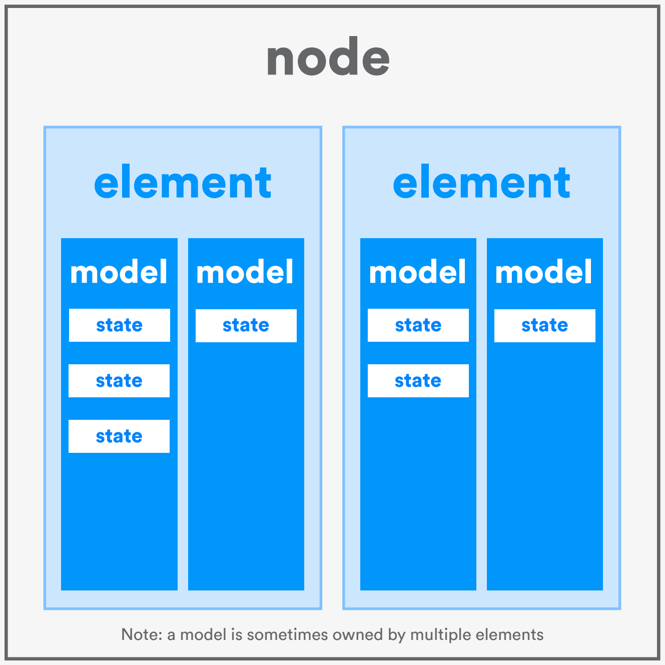


Figure 1 — Node Composition

1.4 Software Developers and Bluetooth Mesh Models

Object Orientation

Software developers should find it easy to imagine model specifications as being akin to classes in the object-oriented (OO) software engineering paradigm and model implementations in code inside a device as an instance of the model or *object*.

The Bluetooth mesh specifications do not stipulate any particular approach to implementing models in code; that's left to the developer and the programming language and APIs in use. But models do lend themselves to an object-oriented approach, and the specification even talks about one model *extending* another, a concept that is also reminiscent of OO.

SDK Variations

There are a number of SDKs (software developer kit) for developing mesh firmware. Some are from Bluetooth module vendors that are specific to creating code for their modules. Others, such as the Zephyr RTOS SDK, are hardware agnostic and allow for the creation of firmware for numerous different target boards. At this time, Zephyr supports 100 different target boards.

Whatever SDK you use, the principals involved in implementing mesh firmware will be the same. In this paper, code created with the Zephyr SDK will be presented as a way of illustrating points from a developer's point of view.

```
// models - an array of specific model definitions
static struct bt_mesh_model sig_models[] = {
        BT_MESH_MODEL_CFG_SRV(&cfg_srv),
        BT_MESH_MODEL_CFG_CLI(&cfg_cli),
        BT_MESH_MODEL_HEALTH_SRV(&health_srv, &health_pub),
        BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_ONOFF_SRV,
generic_onoff_op,
                                                  &generic_onoff_pub, NULL),
        BT_MESH_MODEL(BT_MESH_MODEL_ID_GEN_LEVEL_SRV,
generic_level_op,
                                                  &generic_level_pub, NULL)};

// elements - contains arrays of SIG models and vendor models (none in
this case)
static struct bt_mesh_elem elements[] = {
        BT_MESH_ELEM(0, sig_models, BT_MESH_MODEL_NONE),
};

// node composition - contains an array of elements
static const struct bt_mesh_comp comp = {
        .elem = elements,
        .elem_count = ARRAY_SIZE(elements),
};
```

Node Composition
One of the first key tasks a mesh firmware developer must undertake is to define their product's mesh node composition. This means defining in code how many elements the node has and what models each of the elements contains. Figure 1 on page six shows the relationships between the node, its elements, the models contained within elements, and the items of state that each model contains.

Details will vary across SDKs, but using the Zephyr SDK node composition involves creating a series of arrays, each of which contains structs defined by macros that the SDK provides. It might look something like the example shown in the code block shown above that shows four models belonging to an element, which is the sole element of the node.

Properties
There are two forms that data items can take in a Bluetooth mesh model.

State values are members of particular models and have a value with a meaning that the specification defines. They are not self-describing, and the state a message relates to is inferred from the opcode of the message.

Properties, on the other hand, are instances of *characteristics* to be interpreted in a given context.

Characteristics are also used with *GATT*. A characteristic defines the fields its value consists of, such as permissible values and their meaning and, in the case of GATT, includes an explicit type identifier in the form of a UUID (universally unique identifier). When used in GATT, characteristics are members of *services*, and the service that owns a characteristic provides a context within which to interpret and work with the characteristic. For example, the Alert Level characteristic can be a member of either the Link Loss service or the Immediate Alert service. The meaning of the characteristic varies depending on which service it is a member of, and this is defined in the GATT service specification.

Bluetooth mesh does not use GATT services. Instead, properties provide context for interpreting a related characteristic:

> **"The Temperature 8 Characteristic is a type which represents a temperature measurement, has a format of uint8, and uses units of 0.5 degrees Celsius. Several properties are defined for this characteristic, thus allowing it to be interpreted in various contexts. The Present Indoor Ambient Temperature property indicates that the Temperature 8 characteristic should be interpreted as being a measurement which was taken indoors, whereas the Present Outdoor Ambient Temperature property relates to measurements taken outdoors, and the Present Ambient Temperature property is not specific about the type of location, and this is left to be derived from other location properties."**

back to contents

Properties are explicitly identified by a *Property ID*. In a model where a property is in use, the property ID and property value comprise the value of a *state*. For example, the *sensor data* state contains one or more pairs of property ID and a corresponding sensor value.

Properties allow the same model to be used with a wide range of data types, which, in the case of models like the *sensor server model*, is hugely advantageous since any type of sensor data can be handled and interpreted with respect to any context, provided a suitable property has been defined. Without this approach to describing and encapsulating data, many different types of sensor models would be required, or the sensor server model would need to have a large number of states for each of the different types of sensor data it might need to support.

Client and Server Decoupling

When implementing models, it is important to respect the fact that client models and server models must know nothing about each other's implementation details. For example, a server should not need to know or choose to exploit knowledge of the specific values that a client might be able to send. Each is a black box to the other.

Coding Models

Apart from specifying which models belong to each element in node composition, what else do developers need to do to incorporate the models that have been selected for their product? In some cases, nothing at all. Some models like the *health server model* are mandatory (in the primary element of a node, which all nodes have), and the SDK may provide a complete implementation, which is easily incorporated in the node's composition.

In most other cases, a number of other steps will be necessary:

```
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_GET BT_MESH_MODEL_
OP_2(0x82, 0x01)
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_SET BT_MESH_MODEL_
OP_2(0x82, 0x02)
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_SET_UNACK BT_MESH_MODEL_
OP_2(0x82, 0x03)
#define BT_MESH_MODEL_OP_GENERIC_ONOFF_STATUS BT_MESH_MODEL_
OP_2(0x82, 0x04)

// each array member contains opcode, min msg len, handler function
static const struct bt_mesh_model_op generic_onoff_op[] = {
       {BT_MESH_MODEL_OP_GENERIC_ONOFF_GET, 0, generic_onoff_get},
       {BT_MESH_MODEL_OP_GENERIC_ONOFF_SET, 2, generic_onoff_set},
       {BT_MESH_MODEL_OP_GENERIC_ONOFF_SET_UNACK, 2,
generic_onoff_set_unack},
       BT_MESH_MODEL_OP_END,
};
```

1. RX Message Handler Functions
The opcodes of messages associated with each model and which the node might receive (RX) must be registered and, one or more functions for handling those message types, implemented. Here's what that looks like in Zephyr code above.

Messages received by a model either change a state value (*set*) or request that the current value of a particular state be reported in a status message (*get*). Set messages come in two forms: those that do not require a response (unacknowledged) and those that require the new state value to be sent back in a status message. The term set is sometimes used to mean either of these two variations.

When handling state changes produced by set messages, developers must ensure that any defined and active state bindings are processed, recalculating other dependent state values as required.

2. TX Message Producer Functions
Models almost certainly need to transmit (TX) messages as well as receive them. Functions that formulate mesh messages and use the appropriate API to send messages need to be written and their execution triggered by suitable events or device interactions, such as the user pressing buttons or turning knobs. Developers will be largely concerned with the *access layer* part of messages rather than those fields that are related to lower layers of the stack, though there can be exceptions. It may be necessary to explicitly increment the *SEQ* field to avoid having devices reject messages as forming part of a suspected replay attack, or the software framework may do this automatically.

3. Bind Application Keys to Models
All mesh messages are encrypted and authenticated using AES-CCM. Header fields are also obfuscated to make network-pattern-analysis attacks difficult. Fields from upper layers of the stack are encrypted using an *application key*, and fields from lower in the stack are encrypted using a *network key*. This separates network and application security and allows nodes to perform network functions, such as the relaying of messages without needing or having the ability to decrypt the application payload of the message.

A good mesh software framework automatically secures messages through encryption and obfuscation, using the network and application keys established when the device was *provisioned*. But a node may have several application keys, and each must be associated with specific models through a process known as *key binding*. This ensures that the stack knows which application key to use with which types of message. Developers will almost certainly need to perform explicit application key binding in their code. On Zephyr, application key binding looks like this:

```
/* Bind to generic level server model */
err = bt_mesh_cfg_mod_app_bind(net_idx,
                               addr,
                               addr,
                               app_idx,
                               BT_MESH_MODEL_ID_GEN_LEVEL_SRV,
                               NULL);
```

net_idx and *app_idx* are index values that reference specific keys from the list of one or more network and application keys that a node might have been equipped with when initially provisioned and configured.

Application key binding is the basis for access control in a Bluetooth mesh network. Issuing the network administrator with the application key bound to the *sensor setup* server model gives that user the ability to update that model's state and configure the associated *sensor server* model. Other users, not in possession of this application key, cannot configure the sensor setup server.

2.0 Overview of Mesh Models

The standard Bluetooth SIG models are defined in a dedicated specification called the Bluetooth Mesh Model Specification. In this specification, you will find extensive and rigorous information on each of the 52 standard mesh models.

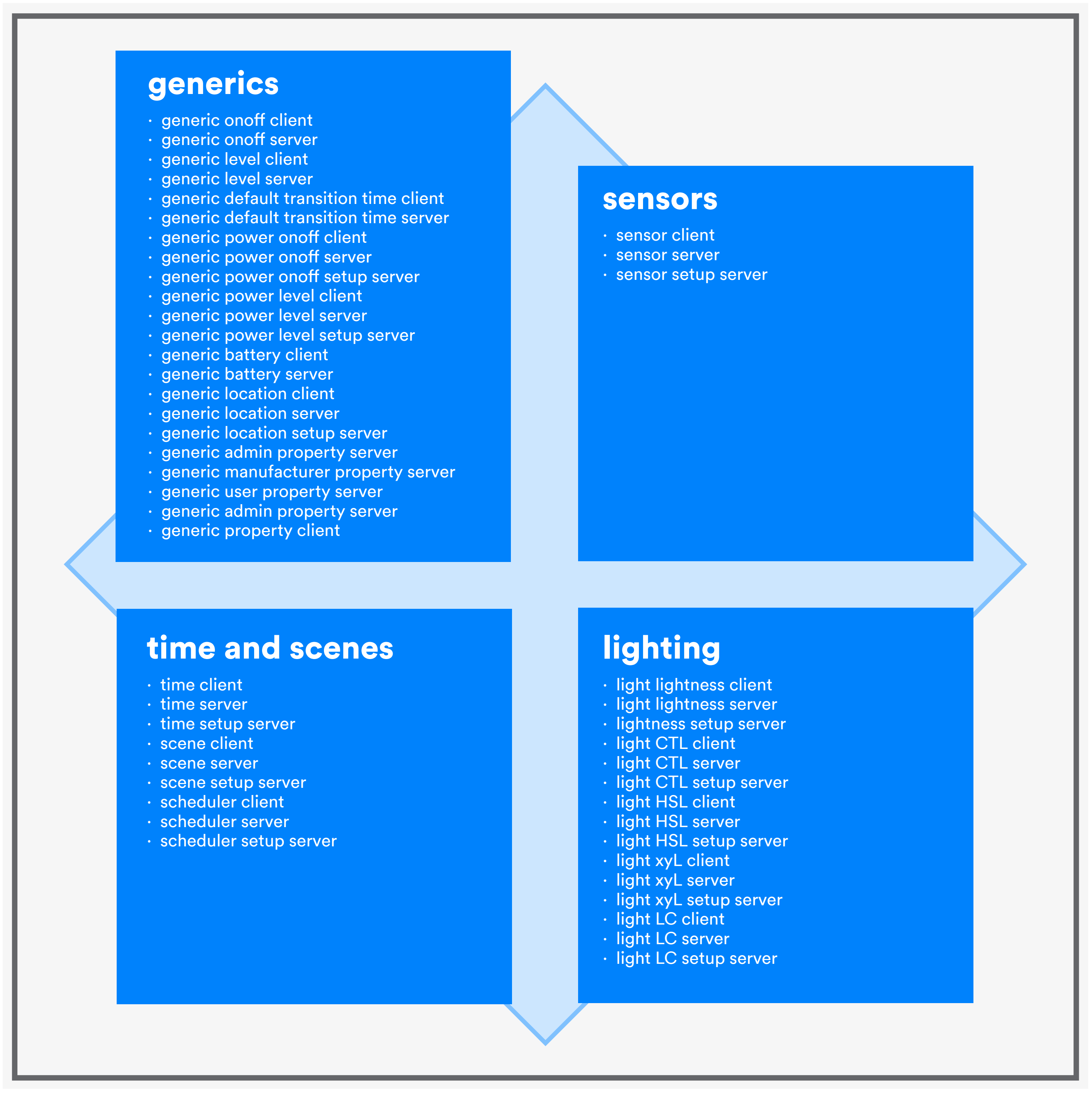


Figure 2 - The Bluetooth Mesh Models

What can we learn about mesh models from Figure 2? First, there are four groups of models: the generics, models for sensors, models for lighting, and models concerned with time and a mesh automation feature called the *scene*. If you review the lists in Figure 2, you will also find that every client model has a corresponding server model and vice versa and that some server models have a corresponding setup server model too.

Generally, models are optional. Developers implement those models that equip their products with the mesh capabilities they need. But there are two models whose inclusion is mandatory and, collectively, these models are the heading of the *foundation models*.

3.0 A Guided Tour of Foundation Models

The foundation models are concerned with enabling the configuration and management of the Bluetooth mesh network and the devices it contains. There are two sets of foundation models and these are described in the Bluetooth Mesh Profile Specification.

3.1 The Configuration Server and Client Models

All devices need to be configurable. Implementing the configuration server model is therefore mandatory and provides the device with the ability to be configured, typically using a smartphone application that will implement the configuration client model.

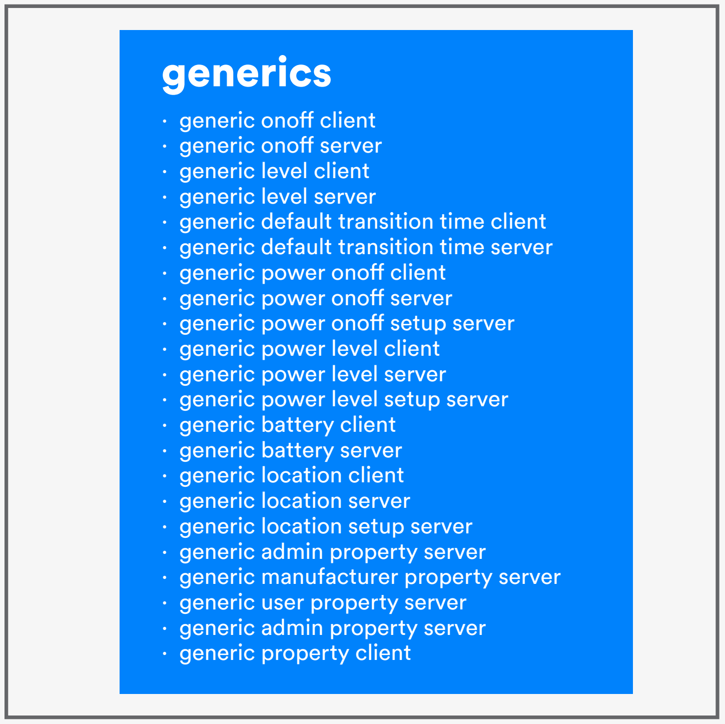
The configuration server model contains a significant number of states that allow various aspects of a device to be configured. The device's overall composition is held within a state called the *Composition Data* state. The destination address to use when publishing messages and other parameters relating to periodic message publication; the addresses subscribed to; and which, if any, of the special relay, friend, low power node, and proxy roles a device may play are all part of the configuration model's data.

Typically, developers only need to ensure the configuration server model is part of their device's firmware. The configuration data comes from a configuration client application, usually at the same time the device is provisioned to equip it with security cases. However, sometimes developers explicitly perform part of the device's configuration from within their code.

3.2 The Health Server and Client Models

The health models are concerned with fault reporting and diagnostics. The primary element of all nodes in a Bluetooth mesh network must include the health server model. Other elements may inform the health server model of faults. A series of fault-related states, such as *current fault*, are defined for the health server model. Faults are represented by single octet codes. Some values in the available range are reserved for Bluetooth SIG use and others are for vendor specific codes. Table 4.2.1 in the Bluetooth Mesh Profile Specification identifies the standard fault codes defined by the Bluetooth SIG.

4.0 A Guided Tour of Generic Models



generics

- generic onoff client
- generic onoff server
- generic level client
- generic level server
- generic default transition time client
- generic default transition time server
- generic power onoff client
- generic power onoff server
- generic power onoff setup server
- generic power level client
- generic power level server
- generic power level setup server
- generic battery client
- generic battery server
- generic location client
- generic location server
- generic location setup server
- generic admin property server
- generic manufacturer property server
- generic user property server
- generic admin property server
- generic property client

Figure 3 - The Bluetooth Mesh Models

The generics collection of Bluetooth mesh models are designed to be used by any kind of device, offering a set of common, generally applicable capabilities. As Figure 3 shows, there are 22 generic models relating to 8 states.

4.1 The Generic OnOff Client and Server Models

At a Glance

The generic onoff models make it possible for one device to switch other devices on or off.

About These Models

The server model contains one state only: the generic onoff state. This is a simple boolean state that indicates whether an element is currently switched on or off. A value of 0x00 means it is off, and a value of 0x01 means it is on. The generic onoff client may send generic onoff get, generic onoff set, or generic onoff set unacknowledged messages. It must be able to receive and handle generic status messages if it is able to send get or set (acknowledged) messages.

4.2 The Generic Level Client and Server Models

At a Glance

Some devices can be turned up or down; lights can be dimmed and the temperate of a room can be increased by turning up the thermostat. The generic level models allow control to be exercised over the level of other devices.

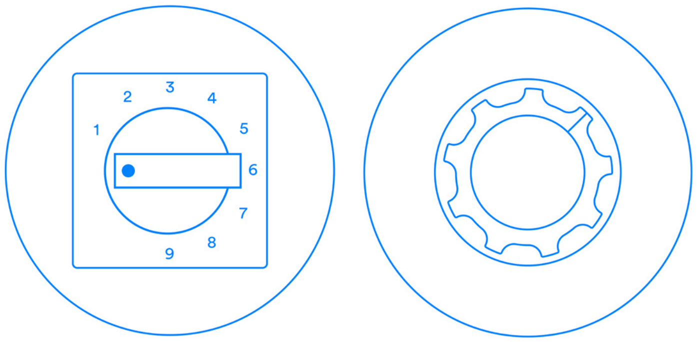


Figure 4a (left) - A 9 x Position Rotary Switch
Figure 4b (right) - Analogue Rotary Control

About These Models

The generic level server model contains a state called *generic level* that can be positive or negative and has a range of -32,767 to +32,767.

Different products may need to approach level control in different ways, such as from a user interface point of view. Imagine a 9-position rotary switch like the one in Figure 4a.

Consider the similarities and differences between this type of level controller, with its 9 fixed choices of position and an analogue rotary control that allows the position to be set anywhere through a continuous range.

In either case, the control needs to implement the generic level client model.

The fixed position control in Figure 4a must divide the positive generic level value range into 9 equally spaced bands, mapping its 9 selectable levels to the generic level state values defined by the Bluetooth Mesh Model Specification. The values delineating the bands (0, 3641, 7282, 10922, 14563, 18204, 21845, 25485, 29126) are the level values sent in generic level set messages from the device.

It might be tempting to think that the rotary control in Figure 4b will not need to perform this kind of value mapping, but it too will deliver values at a certain level of granularity and magnitude to the firmware of the device it is a part of, and they will need mapping to the generic level state's value range in an appropriate way.

Changing Levels

Several ways of changing generic level are supported by the generic level models' set and set unacknowledged messages.

Generic level set - changes the generic level state to an absolute value.

Generic delta set - changes the generic level state by a relative, positive, or negative amount.

Generic move set - initiates changing the generic level state in either a positive or negative direction and at a given speed. The speed with which the transition takes place is calculated from a *delta level* field in the message and a value known as the *transition time*. Transition time must either appear in the *generic move set* message itself, where it is an optional parameter, or be available in a state called the *generic default transition time,* which belongs to the *generic default* transition time model which may or may not be present. If transition time is not available from either of these two sources, the operation will not be executed and *generic level* will not be changed.

The move transition may continue indefinitely. It will stop if a *move set* message with the *delta level* field set to zero is received. When *generic level* reaches its upper or lower limit, during a move transition, the implementation may decide to either terminate the transition at that point or take some other action, such as wrapping around and continuing.

Each of *generic level set*, *generic delta set*, and *generic move set* support the optional fields *delay* and *transition* time.

The *delay* field allows the client to inform the server to defer execution of operation for a period of time after receiving the message. This can be helpful in synchronizing operations that affect multiple receiving devices.

Transition time is used to calculate the speed with which a transition should take place. It encodes two data items, from which an elapsed time for the transition must be calculated. It is one octet in size and its 8 bits are used as follows:

Field	Size (bits)	Definition
Default Transition Number of Steps	6	The Number of Steps
Default Transition Step Resolution	2	The resolution of the Default Transition Number of Steps field

Figure 5 - Transition Time (from the Bluetooth Mesh Model Specification)

The four values which *transition step resolution* may take represent 100 milliseconds (0b00), 1 second (0b01), 10 seconds (0b10), and 10 minutes (0b11), respectively. The transition time represented by this state is calculated by multiplying the *number of steps* and the time value represented by the step resolution. Durations from 0 seconds (immediate) to 10.5 hours can be encoded with the *transition time* state.

The word “steps” might suggest that transitions should take place in a series of discreet increments/ decrements. This is not the case. The steps and step resolution fields are solely there to allow the calculation of the elapsed time of the transition. How the change manifests itself in user-visible ways is a product issue, and how the transition takes place in code is an implementation detail.

Note that some level control requirements cannot be completely met by the simple, generic level models. Lighting is a case in point. Human perception of brightness in lights is not linear, and so more specialized models for controlling the brightness or *level* of lights are provided in a Bluetooth mesh network. We will review the lighting models later in this paper.

4.3 The Generic Power OnOff Client, Server, and Setup Server Models

At a Glance

These models enable the initial state that a device is in immediately after powering up to be configured. For example, it may be preferable that the initial state of a device when powered up is that it is off, as indicated by a value of 0×00 in the generic onoff state. Alternatively, for another product, it may make more sense for the initial state to be on, with generic onoff set to 0×01.

About These Models

The generic power onoff server model has a single state, generic on powerup which has three values defined with meanings shown below in Figure 6 from the Bluetooth Mesh Models Specification.

Value	Description
0x00	Off. After being powered up, the element is in an off state.
0x01	Default. After being powered up. the element is in an on state and uses default state values.
0x02	Restore. If a transition was in progress when powered on, the element restores the target state when powered up. Otherwise, the element restores the state it was in when powered down.
0x03-oxFF	Prohibited.

Figure 6 - Generic OnPowerUp states

This model has several relationships with other models. It extends the generic onoff server model, and it requires the generic power onoff setup model be present. The latter model extends both the generic power onoff server model and the generic default transition time server model. This is depicted in Figure 7.

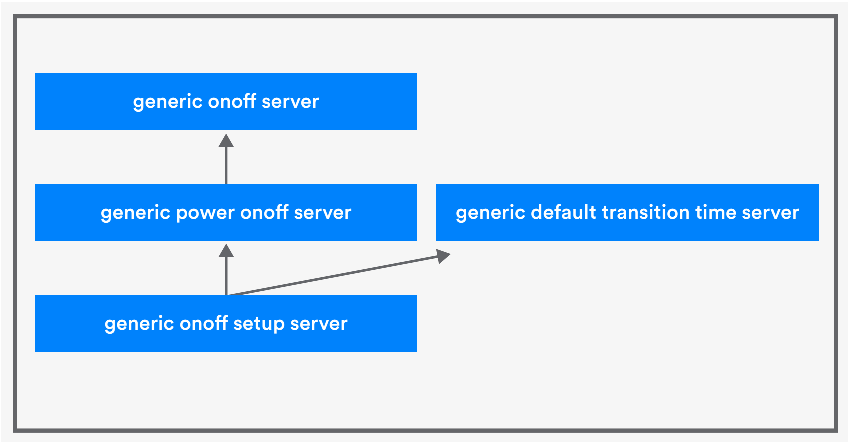

generic onoff server
generic power onoff server
generic default transition time server
generic onoff setup server

Figure 7 - Generic Power OnOff Server and Associated Models

It may not be obvious why the generic default transition time server is part of this picture. The *generic on powerup* state can be used to define what action to take when powering up, if a transition had been in progress when powering down. Therefore, since this behavior is not triggered by the receipt of a message, which could contain the transition time field, the generic *default transition time* state must be available for use in re-establishing transitions on power up.

4.4 The Generic Power Level Client, Server, and Setup Server Models

At a Glance

These models allow control over a device element's power to be exercised. Through relationships with other models, such as the generic onoff server, generic level server, and generic power on server, various state bindings allow specific power levels to be established or re-established when the device is switched on or off or has its generic level state changed.

About These Models

Figure 8 depicts the relationships the *generic power level* server has with other models. It extends any model depicted with an arrow directed from this model to another model, directly or indirectly. It is extended by a model that has an arrow going to the *generic power level server.*

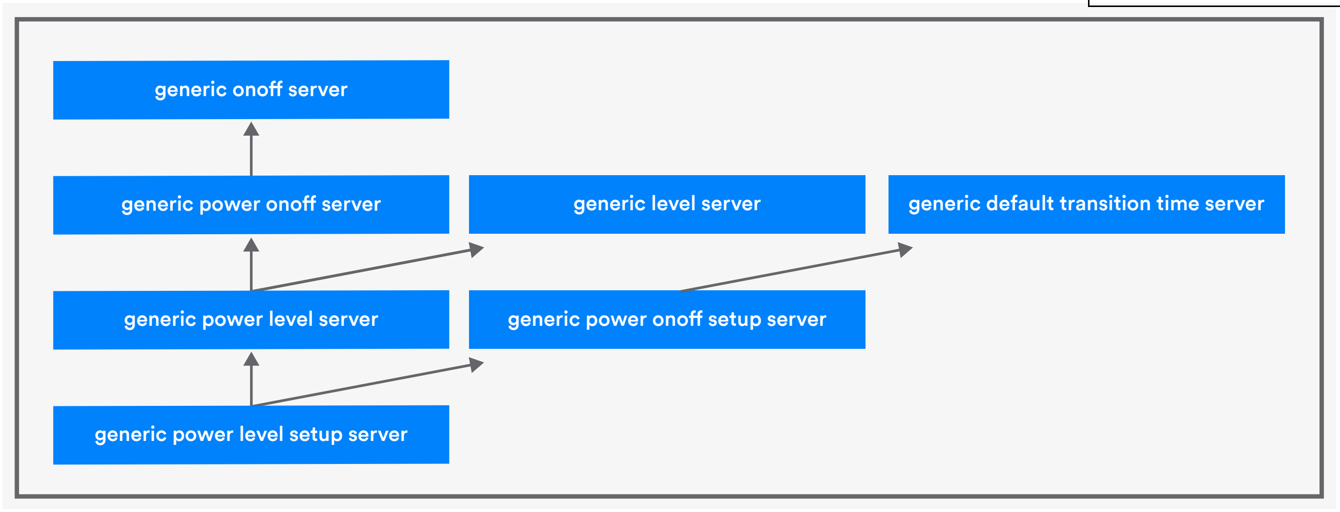

generic onoff server
generic power onoff server
generic level server
generic default transition time server
generic power level server
generic power onoff setup server
generic power level setup server

Figure 8 - Generic Power Level Server and Associated Models

The best way to understand the purpose and behavior of these models, especially the *generic power level server* model, is to understand the states the server contains.

The *generic power level server* model contains one state, the *generic power level* state. It also inherits *generic onoff* and *generic level* from the models it extends.

Generic power level is a composite state, meaning it consists of a number fields, each of which is a state in its own right. These are shown and described in Figure 9 .

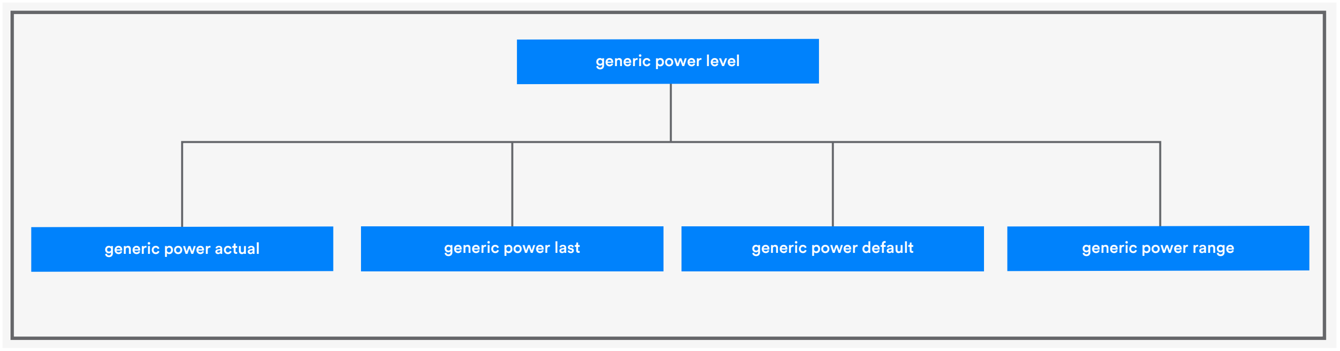


Figure 9 - The Composite Generic Power Level State

State	Description
Generic power actual	Sets element's power level as a linear percentage of the maximum available. Note that with this state set to zero, the device is permitted to continue to be sufficiently powered for wireless communication to remain available. It is like putting the device into standby mode.
Generic power last	Records the last known value of *generic power actual,* so the last power level can be restored when the device is switched on. This behavior is governed by a state binding with the *generic onoff* state and whether or not the *generic power default* state is zero.

State	Description
Generic power default	If this state has a non-zero value when the device is switched on and the generic onoff state changes to 0×01, the power level is restored to the value of this state.
Generic power range	Contains the minimum and maximum power levels the device can be set to as a percentage of the maximum level the device is capable of outputting.

State Bindings
A number of state bindings are defined, and these have a variety of behaviors to be achieved. Readers should review the Bluetooth Mesh Model Specification for full details and definitions of these state bindings.

They can be summarized as follows:

State	Bound to State	Description
Generic power actual	Generic level	Generic power actual = generic level + 32768
Generic level	Generic power actual	Generic level = generic power actual – 32768
Generic power actual	Generic onoff	Determines the value of the generic power actual state depending on combinations of the values of the generic onoff, generic power last, and generic power default states. See specification for details.
Generic power actual	Generic onpowerup	Determines the value of the generic power actual state during the physical powering up of an element. Depends on combinations of the values of the *generic onpowerup*, generic power default, *generic power last*, and *generic power default* states. See specification for details.
Generic power actual	Generic power range	Establishes minimum and maximum values for generic power actual when it is not zero.

4.5 The Generic Battery Client and Server Models

At a Glance
The generic battery server model represents an element that is battery powered. The client model can be used to monitor the state of battery-powered elements.

About These Models
The *generic battery* server model is a root model that contains a single state that messages may act upon: the *generic battery state*. This state contains four values that provide information about a battery's current level, time to charge and discharge, and various other aspects of the battery, such as whether or not it is removable.

The client and server models produce or consume *generic battery get* and *generic battery status* messages. The server must implement support for both message types. For the client, support for the *get* message is optional, and support for the status message is mandatory if *get* is supported, otherwise it is optional.

4.6 The Generic Location Client, Server, and Setup Server Models

At a Glance

Sometimes it is useful to know where a device is in your network. The generic location models allow you to do that. The generic location server model allows a node's location to be encoded in various ways and queried or reported to clients using associated messages. Want to know where a particular device is? Ask it.

About These Models

The generic location models center around the *generic location* state, so that is a good place to start when looking at what these models make possible. The *generic location* state consists of a series of fields, which between them allow the following information about a node's location to be encoded.

Global location - This is expressed as a longitude and latitude, using the WGS84 World Geodetic System and an altitude in meters above the WGS84 coordinates.

Local location - This is expressed as a number of decimeters North and East, relative to some externally defined local coordinate system. A local altitude is also available, and this is a measure of altitude relative to the global altitude, also measured in decimeters.

Floor number - This field contains the floor number that the node is found on in a building. It is encoded in a special way, usually with a +20 delta. So, the encoded floor number value 22 represents the second floor in the building. Some special values are defined too. 0x00 represents floor *-20 or below*. 0xFC represents floor *232 or above*. The ground floor might either be floor 0 or floor 1 according to local conventions and the special values 0xFD and 0xFE represent these two possibilities.

Uncertainty - This field contains 16 bits of information. It can indicate whether the node is stationary or moving. If it is a mobile device, the time since its position was last updated is available. The precision of location measurements is also encoded in this field and ranges from 0.125 meters to 4096 meters.

The *generic location server* model requires the *generic location setup server* model to be present. The generic location setup *server model* allows the generic location state to be updated using generic location global set [unack] and generic location local set [unack] messages. The *generic location server* model supports get and status messages only, and so is effectively a read-only model. Devices that implement the generic location server model can either report their location on demand, when they receive a *generic location local get* message or a *generic location global get* message, or they can report it in a proactive way by publishing *generic location local status* and *generic location global status* messages.

4.7 The Generic Default Transition Time Client and Server Models

At a Glance

State changes can either be instantaneous or they can take place over a specified period of time. There are two ways in which a non-instantaneous state change can be initiated. Many message types support an optional field called *transition time* and, if included in a message, this will determine the time it takes for a state change to be executed. In addition, the *generic default transition state*, which might be available in the optional *generic default transition time server* model, can also be a source of transition time information for state changes.

About These Models

These are simple models with the usual get, set, set unacknowledged, and status messages defined. The *generic default transition time* state that these messages act upon has already been introduced and explained in the section (2) on the generic level models.

4.8 The Generic Property Client and Server Models

At a Glance

As explained earlier, the property models allow lists of arbitrary numbers of properties to be associated with a device. Properties are grouped in different models so that different user groups — the manufacturer, administrator, and standard user — only have access to permitted properties. It is also possible for a property server model to find a client that is capable of consuming and using a particular property type. Collectively, the property models provide a generalized data storage and communication mechanism that can accommodate a wide range of data values and types without models themselves needing to be changed.

About These Models

The specific models that deal with Bluetooth mesh properties are as follows:

Generic manufacturer property server

Generic admin property server

Generic user property server

Generic property client

Generic client property server

The manufacturer, admin, and user property servers hold those properties to which manufacturer, administrator, and standard users should have some level of access. Access to each of these three server models is controlled by the user's client device needing to possess the application key bound to the model that the user wishes to access. Access to specific properties in a model is controlled by a field in the property state that determines whether read only, write only, or read-write access is granted by the property represented by the state.

The manufacturer, admin, and user property servers contain similar states called *generic manufacturer property*, *generic admin property*, and *generic user property*. Each has three fields containing the property ID, access flags (read, write, read-write), and the property value.

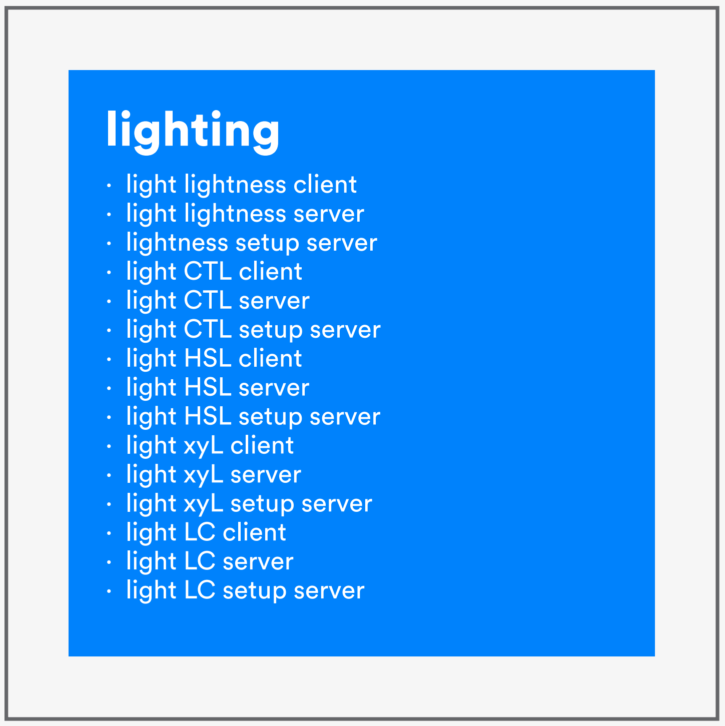
back to contents

The fourth server model, the *generic client property server* model, allows applications, such as the provisioning and configuration application, to find clients that are capable of consuming and processing particular properties. For example, it might be desirable to find a device with a user interface that can display a particular temperature property. The *generic client property server* contains a list of one or more *generic client property* states, each of which contains the ID of a property supported by the client.

5.0 A Guided Tour of Lighting Models

Lighting can be surprisingly sophisticated and therefore needs specialized Bluetooth mesh models to meet its sometimes complex requirements. The Bluetooth mesh lighting models allow control over the on/off state of lights, their lightness, color temperature, and their color (using various color spaces). Importantly, they also provide a highly sophisticated software-based lighting controller that can enable smart lighting automation scenarios. As Figure 10 shows, there are 16 lighting models related to 5 distinct aspects of lighting.

Before beginning the guided tour of the models, consider the nature of lights and the various ways they can be controlled.



lighting

- light lightness client
- light lightness server
- lightness setup server
- light CTL client
- light CTL server
- light CTL setup server
- light HSL client
- light HSL server
- light HSL setup server
- light xyL client
- light xyL server
- light xyL setup server
- light LC client
- light LC server
- light LC setup server

Figure 10 -The Lighting Models

5.1 Lighting Overview

Controlling Lights

Lights are often controlled manually by pressing buttons, turning knobs, or pushing sliders. But they can also be controlled by sensors, indicating to the lights that there is someone in the room or that the ambient light level has become low because it is getting later in the day or because a cloud has obscured the sun. Lights can be controlled by timers too.

The generic onoff and generic level models detailed in section 4 could be used to control some of the basic attributes of a light, but people perceive lighting conditions in more complex ways, with brightness or *lightness* perceived according to a non-linear scale.

Lights have more attributes than their on/off state or their lightness that we might wish to control. Some lights can have their color controlled, and there are a number of ways of modelling color in lights.

Smart Lighting

Smart buildings require smart lighting. Smart lighting can be controlled by manual actions taken by building occupants, but, more importantly, a smart lighting system is informed by sensors and uses control algorithms to achieve self-optimizing behaviors that make the system efficient, cost effective, and pleasing to the people that use the building. The Bluetooth mesh lighting models include a particularly special set of models, such as the Light LC models that provide sophisticated, automated control of lights.

5.2 Lighting Concepts

To appreciate the lighting models, it helps to understand certain concepts from the world of lighting. The key ones are as follows:

Color Temperature

The color temperature of a light source is what leads people to describe colors as either *cool* or *warm*. It has a more scientific definition that relates to the temperature of the light radiated by the object, measured in Kelvin. Surprisingly, lower color temperatures are those we describe as *warm* and higher temperatures as *cool*. In commercial lighting applications, warmer color temperatures are often used to promote relaxation and cooler temperatures to enhance the concentration of occupants working in the room.

Color-Tunable Light

Color-tunable light (CTL) is a capability of some lights that allows color temperature to be controlled via two dimensions: lightness and color temperature.

Hue

Colored light has a number of properties, of which hue is one of the main ones. Typically, hue measures the angular position of a color in a color wheel.

Lightness

Lightness is the term used to refer to the perception of brightness.

Saturation

Saturation is another property of colored light and measures the ratio of an object's color to its lightness. A given color with a high lightness is said to be less saturated than the same color with low lightness.

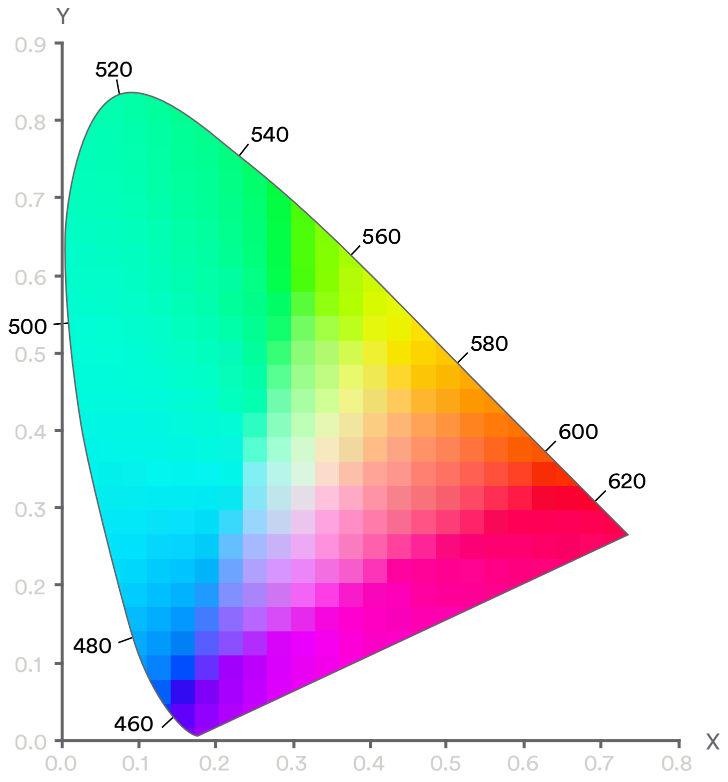


Figure 11 - CIE1931 is especially popular in professional lighting applications

Color Models

A color model, not to be confused with a Bluetooth mesh model, is a mathematical way of representing colors. There are several color models in popular use, each with its own strengths and weaknesses.

HSL (hue, saturation, lightness) represents colors using a cylindrical representation. The angular position in the circular cross section of the cylinder represents the hue, the distance from the center of this circle represents the saturation, and the distance from one end of the cylinder represents the lightness, with one end representing black and the other white.

The **RGB** (red, green, blue) color model is an additive model where given levels of red, green, and blue light are mixed to produce a color that people can perceive.

The **CIE1931** color space defines the mathematical relationships between wavelengths of light and perceived colors in vision. Just like RGB and HSL, colors in this model are defined by three values: x, y, and Y. x and y are coordinates of the color on a color chart, and Y measures the luminous intensity. CIE1931 is especially popular in professional lighting applications.

Each color model has an associated color space that is a set of colors that the model allows to be reproduced.

5.3 The Light Lightness Client, Server, and Setup Models

At a Glance

These models allow the lightness of a lamp to be controlled by mesh messages and events, such as powering up the device.

About These Models

Figure 12 depicts the relationships the light lightness server model has with other models. It extends any model depicted with an arrow pointing to it from this model directly or indirectly. It is extended by a model which has an arrow going to the light lightness server.

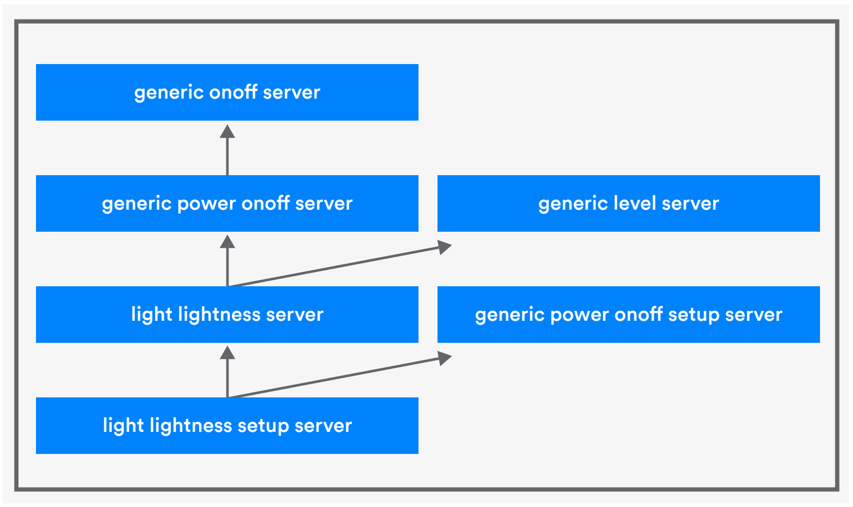

generic onoff server
generic power onoff server
generic level server
light lightness server
generic power onoff setup server
light lightness setup server

Figure 12 - Light Lightness Server and Associated Models

A number of states are involved in the control of lightness and contained within these models. Understanding these states will provide a good start to understanding the models.

The *light lightness* state is a composite state, consisting of the *light lightness linear*, *light lightness actual*, *light lightness last*, and *light lightness default* states.

There are two distinct ways that lightness may be changed using these models. The *light lightness linear* state provides control along a linear scale, but which people will perceive as non-linear lightness changes. Conversely, the *light lightness actual* state provides control along a non-linear scale that produces lightness changes perceived by people as being linear.

A range of supported lightness levels, from a minimum level to a maximum level, may be set for the server using its setup model, which contains the *light lightness range* state, a composite state that includes the light *lightness range min* state, and the light *lightness range max* state. The configured range is used in lightness state transitions to ensure only valid, supported values are used by the model.

In addition to states concerned with controlling lightness on a given scale, there are states concerned with restoring the lightness level when the device is switched back on or powered up. These are the *light lightness last* state and light *lightness default state*, both of which are involved in the functioning of the *generic power onoff server* model.

State Bindings
Light lightness actual and *light lightness linear* are related by two-way bindings. If one changes then the other must be recalculated.

Light lightness actual is also bound to the *generic level*, *generic onoff*, *generic onpowerup*, and *light lightness range* states. The precise details of these bindings are defined in the specification, but the general nature of these bindings should be intuitive enough. For example, changing the generic level in a light that has the *light lightness server* model will change its lightness states as well.

5.4 The LC Client, Server, and Setup Models

At a Glance
Collectively, the lighting control (LC) models form a lighting controller: a software component that allows sophisticated, sensor and user-driven lighting control to be set up. Occupancy and ambient light sensors are catered for so that techniques like daylight harvesting can be employed. As the state of the lighting controller changes, the *light lightness* state of the light under control progresses through a series of levels, with the transition from one to another governed by configurable timing parameters so that changes are not abrupt and feel natural to building users.

Decentralized Control
Legacy lighting control requires the installation of dedicated, physical devices, called controllers, sitting in between sensors and lights. This is called a centralized lighting control architecture. See Figure 13.

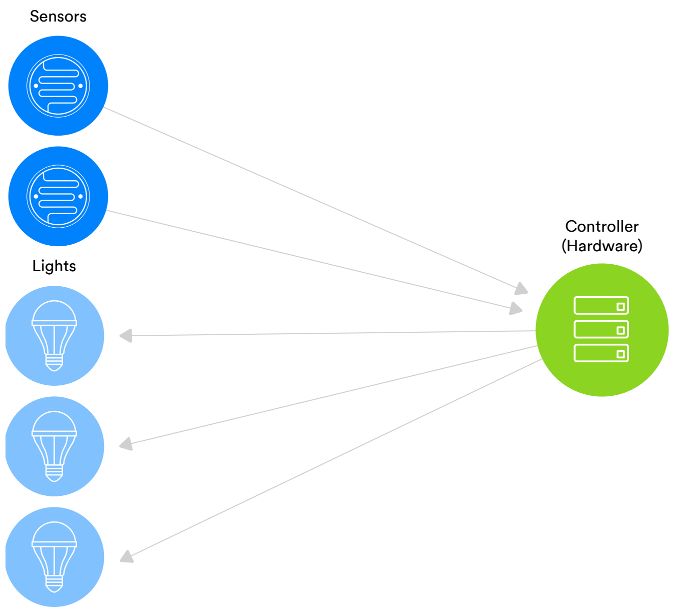


Figure 13 -The Lighting Models Legacy, Centralized Lighting Control

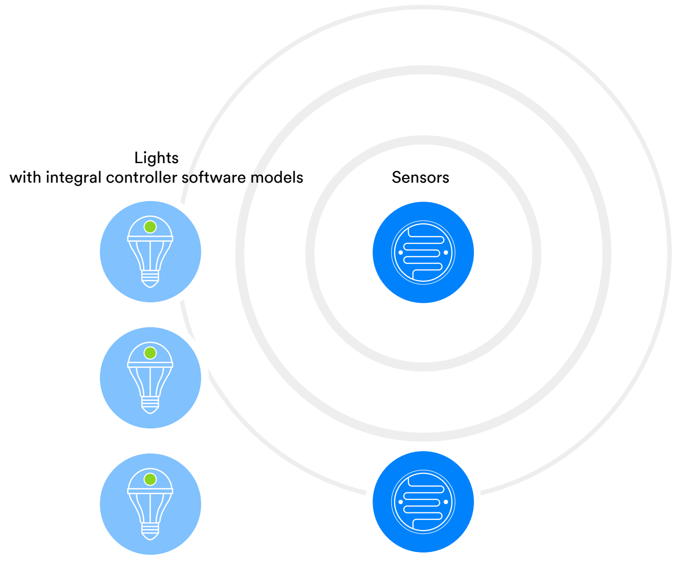


Figure 14 -The Bluetooth Mesh Decentralized Controller Architecture

Bluetooth mesh lighting control is entirely software based and supports a superior, decentralized controller architecture with the controller embedded in the lights rather than in physically separate hardware devices. There are cost advantages and, as described in an article on Bluetooth mesh and scalability, significant performance advantages to this approach. Figure 14 illustrates the Bluetooth mesh decentralized controller architecture.

Of States and States Machines

The terms *controller* and *lighting controller* are used in this paper as an informal shorthand for an element that has the *light LC server* and *light LC setup* server models. The aggregate capability given by these models is known as a lighting controller. The *light LC client* model is used by elements that are able to configure a *light LC setup server* model on a remote device.

The *light LC* server is unusual in that it consumes messages from a model that is not part of the same family, namely the *sensor server* model. This is so that sensors, such as ambient light and occupancy sensors, can provide input to the controller's operation.

The concept of a *finite state machine* is important to understanding the way the Light LC models work to form a lighting controller. Indeed, the Bluetooth Mesh Model Specification approaches the definition of lighting control in a different way to that of the other collections of models. A finite state machine for lighting control is presented and much of the specification refers to that state machine. The state machine defined in the specification is an abstraction that defines how the overall lighting controller works. Sitting underneath this are the mesh models and their mesh states, and it is these mesh states that the finite state machine acts upon and is informed by. The use of the word *state* in these two contexts, that of the overall lighting controller and that of a mesh state data item inside a mesh model, can be a little confusing at first, but it makes perfect sense if you keep the context in mind when reading this section. In this paper, the term *mesh state* refers to a state that is part of a mesh model, and the term *controller state* refers to a state that is part of the lighting controller finite state machine.

The following controller states are defined as part of the lighting controller's finite state machine. Note that this information is a summary of section 6.2.5.1 of the Bluetooth Mesh Model Specification:

State Machine State	Meaning
Off	The lighting controller is disabled and light lightness is not controlled.
Standby	The lighting controller is enabled, but occupancy state changes reported by sensors are ignored.
Fade on	Occupancy has been detected, and the lightness level of lights are in the process of transitioning to the level defined in the light *LC lightness* on mesh state.

State Machine State	Meaning
Run	Lights are now at the lightness level defined by the light *LC lightness on* mesh state, and lightness stays at this level until a timer expires and causes a transition to the Fade controller state to take place. Occupancy events reset the timer. The controller transitions to and stays in the Run controller state when a room is occupied, and it will stay in that state as long as the room continues to be occupied.
Fade	The room is regarded as no longer occupied, so the lightness level starts to transition to the level defined by the *light lightness prolong* mesh state.
Prolong	The Prolong controller state can be thought of as an intermediate state, with a corresponding, intermediate lightness level to which lights fade after occupancy has ceased to be detected. On entering the Prolong controller state, a timer is started. When the timer expires, the controller will start to transition into the next controller state. One example that illustrates the purpose of the Prolong controller state is to avoid abruptly plunging an area of an open-plan office into complete darkness when there are still people working at the other end of the office, which is monitored by different occupancy sensors.
Fade standby auto	After the Prolong controller state's timer expires, the controller switches into the *fade standby auto* state and transitions the lightness level to that defined by the *light lightness standby* mesh state over some transition period.
Fade standby manual	In this state, the controller also transitions the lightness level to the level defined by *light lightness standby*, but switches into this state in response to a manual event, such as receipt of a mesh message like *light LC light onoff set*, which switches the lights off.

Figure 6.7 of the Bluetooth Mesh Model Specification provides a diagrammatic reference to the controller's finite state machine, showing the set of controller states, the valid transitions between states, and the events that trigger them.

Figure 6.4 from the specification shows an example of the controller states being transitioned through and the effect this has on lightness levels at each stage. It is repeated here in Figure 15 for convenience.

Transition Times

Each of the four fade states are transitional in that the system is in the process of transitioning to another state. For example, the Fade On controller state is a state the controller will be in whilst transitioning from the Standby controller state to the Run controller state and corresponding lightness level. How long it takes to transition from the current lightness level to the target lightness level, defined for the next state, can be specified in the *transition time* optional field in relevant mesh

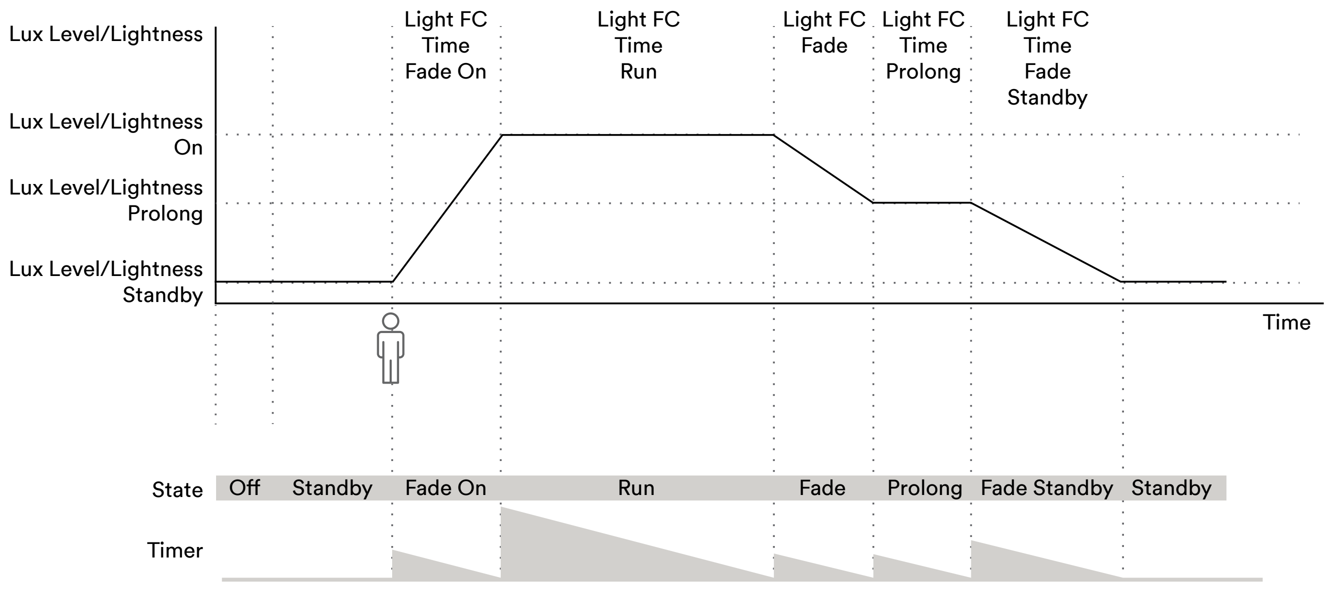


Figure 15 -Lighting controller state transitions

messages, or it can be taken from mesh states, such as *light LC time fade on*.

See section 6.2.5.13.1 of the Bluetooth Mesh Models Specification for details of mesh states that define lighting controller state transition times.

The Details

The LC client, LC server, and LC setup server models form the most sophisticated and, in some ways, complex family of models defined for Bluetooth mesh. They sit at the heart of the support Bluetooth mesh has for advanced commercial lighting systems. This paper has reviewed the concepts governing the operation of these models and how they form a lighting controller, but not looked closely at the underlying mesh states or even the models themselves. There are a significant number of mesh states and properties, some of which allow a lighting controller to be configured to behave in a number of ways. If you are happy with the introduction to the Bluetooth mesh lighting controllers that this section has provided, your next step should be to drill down to the detail provided in the Bluetooth Mesh Model Specification.

5.5 Light CTL Client, Light CTL Server, Light CTL Temperature Server, and Light CTL Setup

At a Glance

These models allow the control of a tunable, white light source. Tunable white lights offer control over the color temperature of a white light and leverage the most recent research into human biological and cognitive responses to light.

About These Models

Figure 16 depicts the relationships the *light CTL server model* has with other models.

Figure 17 shows the simpler *light CTL temperature server* model that extends the *generic level server* model only.

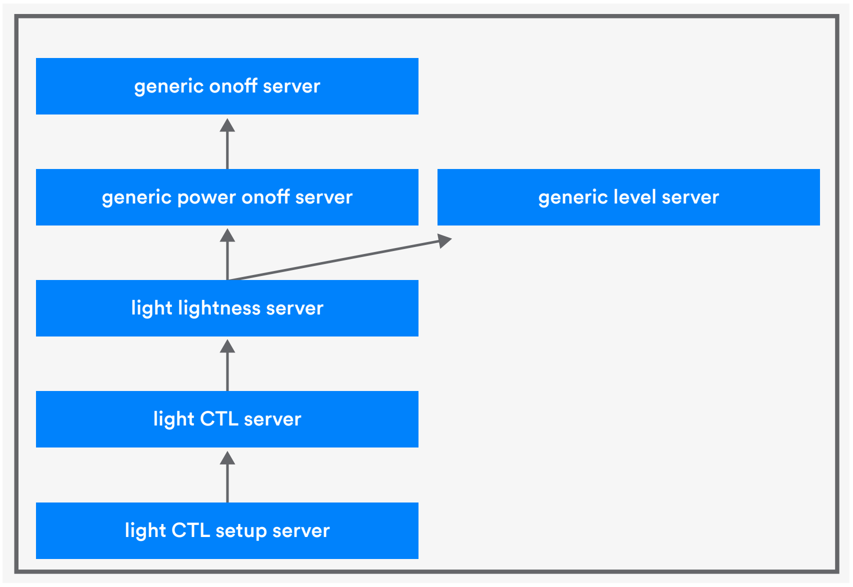


Figure 16 - Light CTL Server and Associated Models

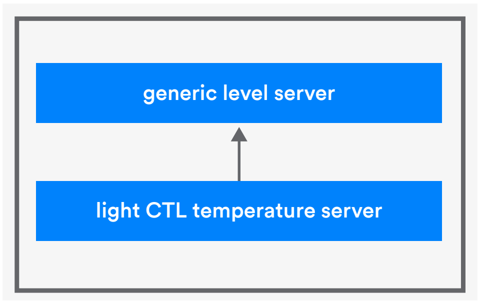


Figure 17 - Light CTL Temperature Server

Central to the *light CTL* models is a composite state called the Light CTL state that belongs to the light CTL server model. It consists of the 6 sub-states shown in the table below.

State	**Description**
Light CTL temperature	Sets the color temperature level.
Standby Light CTL temperature range	Sets the maximum and minimum color temperature that an element is capable of supporting.
Light CTL temperature default	A default color temperature level for use when powering up in a way determined by the *generic onpowerup* state.
Light CTL delta UV	Some lights allow the color temperature to be varied by some delta away from the usual curve that color temperatures are measured from (known as the *black body locus*). This technique allows certain colors, such as pinks, to accentuated. This state allows a delta value to be set for this purpose.
Light CTL delta UV default	A default *delta UV* value for use when powering up in a way determined by the *generic onpowerup* state.
Light CTL lightness	Controls the lightness of a tunable white light source. Comparable to the *light lightness* state, but for tunable white lights whose color temperature can by definition be varied.

The *light CTL temperature* server model contains only the *light CTL temperature* state plus the *generic level* state due to its extension of the *generic level server* model. It is simpler than the *light CTL server* model, but it may not be obvious why the *light CTL temperature* state appears in both these models.

The answer is that color-tunable light can be changed by manipulating two dimensions: lightness and temperature. It was a requirement that each of these be controllable by making changes to the *generic level* state or, in other words, through a state binding with that state. This implies there must be two distinct instances of the generic level server model to support the two distinct bindings with the *generic level* state, and the only way to accomplish this is to have two elements in the node's composition; the first allows *light CTL lightness* to be modified via the *generic level* state, and the second allows the *light CTL temperature* state to be controlled via *generic level* state changes. The specification designates one element as the *main* element and the other as the *temperature* element. Developers must ensure their node composition reflects this dual-element approach when implementing if independent control via level changes is needed for the two dimensions of CTL.

The *Light CTL client* model provides access to the states in both the *light CTL server* and *light CTL temperature server* models and includes support for the usual set, get, and status message types for each state. Check the section 6.6.2 of the [specification] for details.

State Bindings

Various state bindings are defined for the states involved in these models, and some of the more interesting ones have been mentioned already. Generic level can be used to control the two dimensions of CTL; power-up events can be used to restore CTL states via bindings with the *generic onpowerup* state. CTL temperature values are restricted by a binding with the *CTL temperature range state*, which is involved in various state binding calculations to ensure values do not fall outside the permitted range.

5.6 The Light HSL Client, Server, and Setup Models

At a Glance

These models provide control over color-changing *lights*, using the hue/saturation/lightness (HSL) model of color representation.

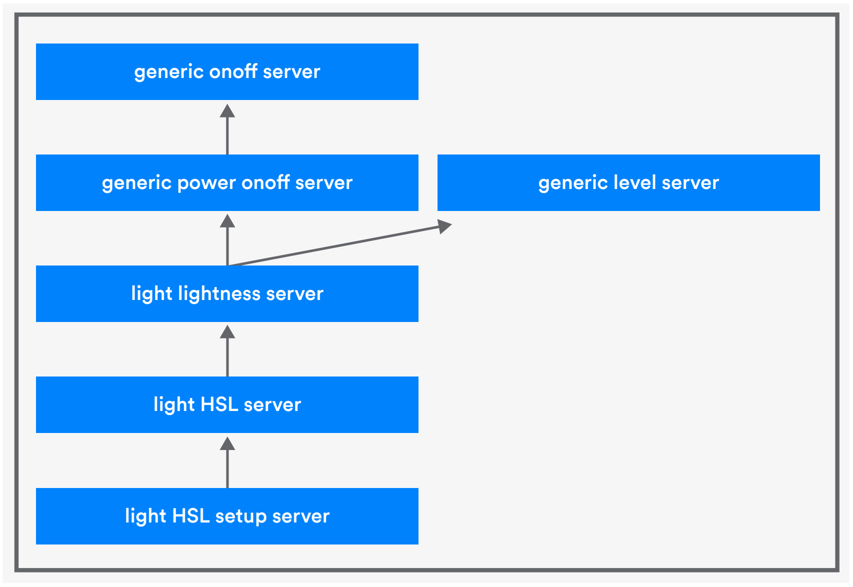


Figure 18 - Light HSL Server and Associated Models

About These Models

Figure 18 depicts the relationships the light HSL server model has with other models.

The most important Bluetooth mesh state involved in these models is the light *HSL state*. It is a composite state consisting of sub-states *light HSL hue*, *light HSL hue default*, *light HSL saturation*, *light HSL saturation default*, and *light HSL lightness*.

Light HSL hue represents the hue as a 0-360-degree angle around a color wheel.

The Perceived lightness of a light (*L*) is approximately the square root of the measured light intensity (*Y*):

$$L = 65535\sqrt{\frac{Y}{65535}}$$

Where *L* is the perceived lightness and *Y* is the measured light intensity (from 0 to 65535).

Figure 19 - Relationship Between Perceived Lightness and Measured Light Intensity

Light HSL saturation represents saturation as a 16-bit value with 0×0000 representing the lowest perceived saturation level and 0xFFFF the highest perceived level.

Light HSL lightness measures lightness on a perceptually uniform scale (see Figure 19).

These states within the *light HSL server* model can be controlled by messages from the corresponding client model in the usual way. Additional messages, *light HSL target get*, and *light HSL target status* allow all three of *light HSL lightness*, *light HSL hue*, and *light HSL saturation* to be queried and reported on by a single message type. If a transition of any of these states is in progress at the time the status message is to be produced, a *remaining time* field is included in the message to indicate how long it will be before the transition to the target state has been completed.

State Bindings

Bindings are defined such that HSL color can be controlled via the generic level of an element and so that the color can be restored to some state when the element is powered up. In addition, there is a relationship between *light lightness actual* and *light HSL lightness,* which makes sense given HSL has lightness as one of its dimensions. In brief:

> *Light HSL hue is bidirectionally bound to generic level, to generic onpowerup, and to light HSL hue range.*
>
> *Light HSL saturation is bidirectionally bound to generic level, generic onpowerup, and light HSL saturation range.*
>
> *Light HSL lightness is bidirectionally bound to light lightness actual.*

5.7 The Light xyL Client, Server, and Setup Models

At a Glance

These models provide control over color changing lights, using the CIE1931 model of color representation.

About These Models

Figure 20 depicts the relationships the *light xyL* server model has with other models, and it is similar to the relationship that the *light HSL server* model has with other models, as shown in Figure 18.

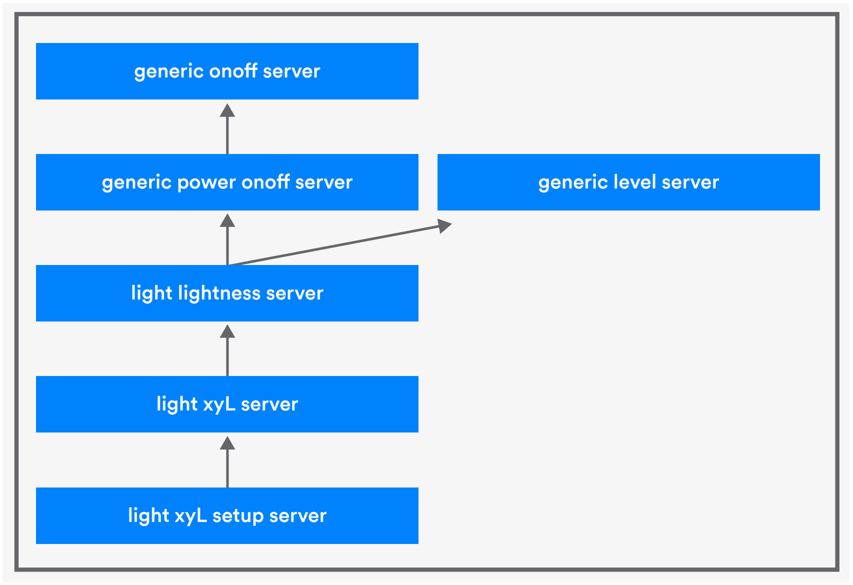


Figure 20 -Light XYL Sserver and Associated Models

Reviewing the *light xyL* state reveals much of what you need to know about these models. It is a composite state that consists of states *light xyL x*, *light xyL x default, light xyL y*, *light xyL y default*, and *light xyL lightness.* These states and the messages provided by the client and server model allow the coordinates of color, according to the CIE1931 color space chart, to be manipulated and defaults to be used when powering up the device to be set.

The *light xyL x* and *light xyL y* states represent coordinates in the range 0 to 1 and are transformed to a 16-bit state value by the formulae:

CIE1931_x = (Light xyL x) / 65535

CIE1931_y = (Light xyL y) / 65535

The special state values 0×0000 and 0xFFFF represent the CIE1931 coordinate values of 0 and 1, respectively.

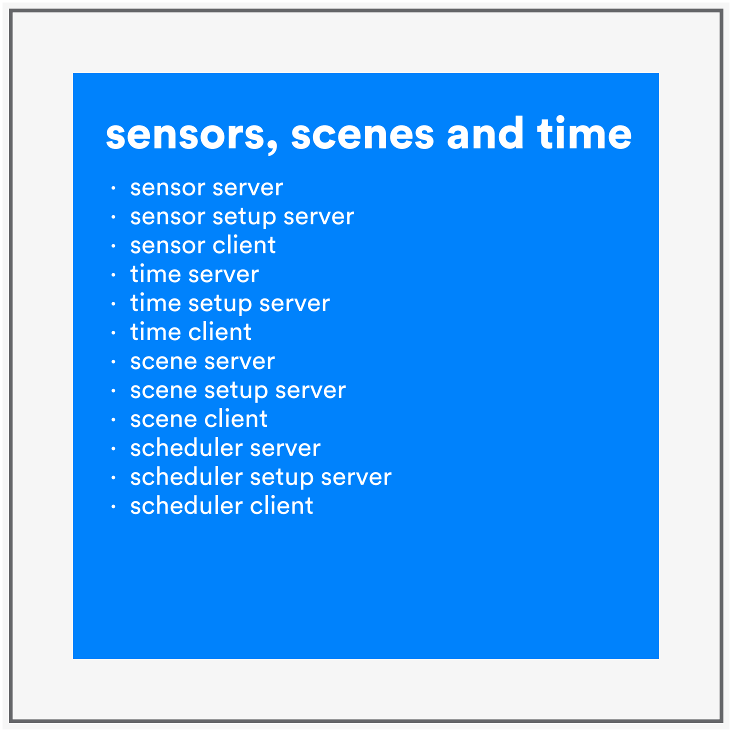
State Bindings

Light xyL x is bound to *generic onpowerup* and *light xyL x range*. This means the x coordinate can be restored when powering up the device and state binding calculations will keep coordinate values within the valid range for this device. *Light xyL y* has similar bindings.

Lights may implement the server models for both HSL color control and CIE1931 (i.e. *light HSL server* and *light xyL server*). When this is the case, indirect state bindings will exist between the *light xyL* state and the *light HSL* state. This means that lights can be controlled by clients of either type of model.

6.0 A Guided Tour of Sensor, Scene, and Time Models

Bluetooth mesh scenes define entire collections of settings for an environment, optimizing it for a particular purpose. For example, you could choose to define a scene that puts a room into the perfect state for a presentation. Switching to a particular scene can be triggered by sensors or a time schedule.



sensors, scenes and time

- sensor server
- sensor setup server
- sensor client
- time server
- time setup server
- time client
- scene server
- scene setup server
- scene client
- scheduler server
- scheduler setup server
- scheduler client

Figure 21 - The Sensor, Scene, and Time-Related Models

Sensors play a critical role in many mesh networking applications, including, but not limited to, that of the smart building. They detect and report events like the changing occupancy status of rooms, and they measure attributes of the environment, sharing this data with other devices.

Sensor data can be used to influence or control the operation of one particular type of device, or it can be used to change the state of many devices of many different types, all in one go.

As an example of the first of these cases, the lightness of lights in a room can be dynamically adjusted in response to changing ambient light levels, as reported by ambient light sensors.

As an example of the second case, consider what we might want to happen when a person walks into a previously empty room. We might want the lights to switch on, the heat to be turned up slightly, and the blinds to open. A Bluetooth mesh network makes this scenario possible through the use of *scenes*. Scenes are collections of memorized model states that are identified by a scene number. Devices can be instructed by a Bluetooth mesh message to switch to the states that belong to a specific scene. This is how mass changes, affecting many different types of devices, can be orchestrated in response to an event like an occupancy change.

Some state changes, including scene switches, can be executed according to a time schedule. A Bluetooth mesh network includes a *scheduler* that is responsible for this behavior. To work though, nodes must have access to a common, accurate system time. Consequently, there are time models, states, and messages, as well as some special roles nodes may play regarding the propagation of time across the network.

6.1 The Sensor Client, Server, and Setup Models

At a Glance

These models provide a generalized approach to sensor operation in a Bluetooth mesh network and allow any type of sensor to communicate sensor readings to other nodes in the network. The sensor setup server allows the sensor and format of its data to be configured.

About These Models

The sensor models make extensive use of properties within a relatively small number of *states*.

Properties differ from states in that they contain both an identifier and a value. The identifier tells us what type of data the property contains so that it is self-describing. States, on the other hand, have no explicit type identifier, and it is the model or message the state is contained within that tells us the data's state.

Leveraging properties has allowed the three sensor models to accommodate any type of sensor and sensor data, rather than requiring different models, messages, and states for each conceivable type of sensor that might be part of a network.

An element that implements the sensor server model must also have the sensor setup server model, which extends it. The sensor client model is not related to other models and can be used standalone. Figure 22 illustrates the relationship between the three sensor models:

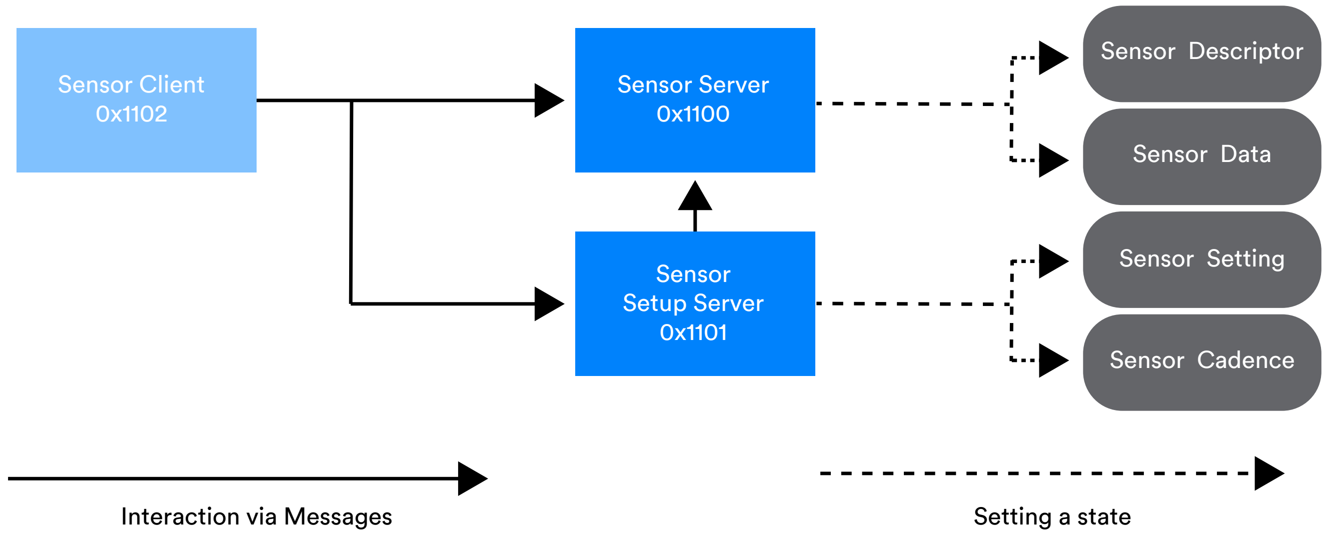


Figure 22 - The Sensor, Scene, and Time-Related Models

Sensor States

The sensor models are defined around a single composite state called the sensor state. This is a fairly complex state whose primary parts are distributed across two models, the *sensor server* model and the *sensor setup server* model, as shown in Figure 22.

The complete breakdown of the sensor state is shown in Figure 23 on the next page.

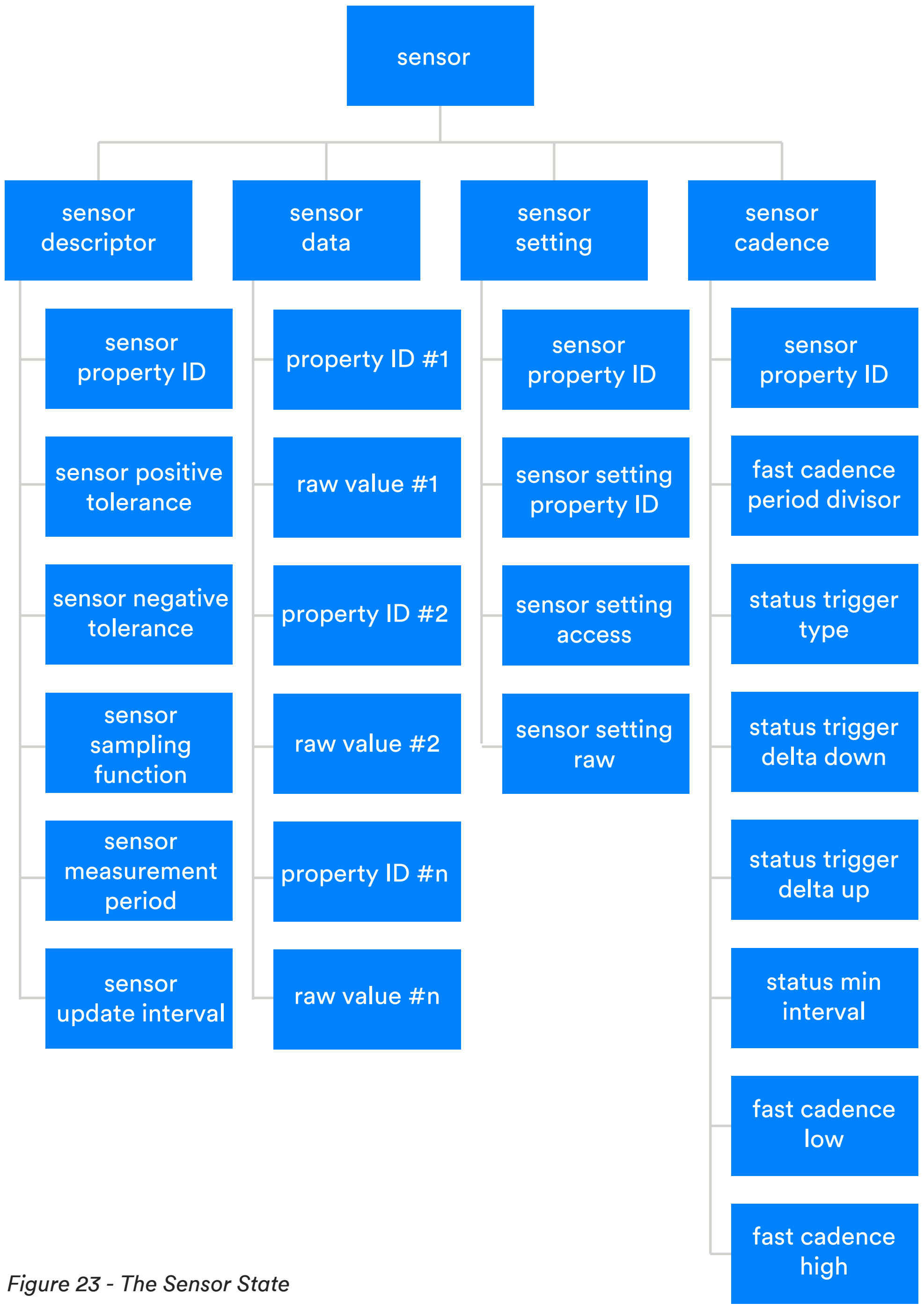


Figure 23 - The Sensor State

The *sensor data* state contains an array of property ID / raw value pairs. The Bluetooth mesh device properties specification defines properties and the characteristics to which they relate. In some cases, a referenced property has a simple value that may be acted upon using sensor model messages, such as *sensor get*, which return the sensor data state value in a *sensor status* message. Some properties define arrays of data, suited to creating histograms, and individual columns from within this tabular data can be accessed with messages like *sensor column get*, which returns a *sensor column status message*.

The *sensor descriptor* state contains information that describes the sensor data available from this sensor. It is not expected to change over the entire lifetime of the sensor.

The tolerance fields provide an indication of the magnitude of possible errors in measurements reported by the sensor. The *sensor sampling* function field indicates the type of function applied to measured sensor values. For example, some sensor data values are instantaneous snap shots of the measured phenomena. Or, perhaps, an averaging function, such as the arithmetic mean, is being applied to measured values and it is this that is contained within the sensor data state. Where a function, such as an averaging function, is being applied, the *sensor measurement period* field indicates the time period over which measurements are being averaged, and the sensor *update interval* indicates the frequency with which each measurement is made by the sensor.

Sensors often have configurable settings, such as sensitivity thresholds. The *sensor setting* state contains a list of such settings and their values. Each member of the list consists of the ID of the property to which the setting applies, the ID of a property that identifies the setting itself, an indication of whether the setting is read only or may also be written to, and the raw setting value itself. For example, occupancy sensors often have motion-sensitivity settings that allow the sensor to be configured so that false alarms, triggered perhaps by small furry animals, are not created. Property 0×0043 Motion Threshold allows the configuration of the required sensitivity level in this case.

The *sensor cadence* state allows the frequency with which a sensor publishes status reports relating to each sensor data type (identified by property ID) to be configured. The rate of publication can be configured to vary according to various conditions. When the value falls within a configured range, the publication rate can be increased. If particularly large increases or decreases in the sensor data value are measured, the reporting rate can also be increased. In each case, the *fast cadence period* divisor indicates by how much the rate of publication should be increased when any of these circumstances arise.

Sensors and Other Models
The *light LC server* model can consume *sensor status* messages. This allows sensors, such as ambient light sensors and occupancy sensors, to be used with important lighting control scenarios. The lighting models are explored in a previous section of this paper.

6.2 Time, Scenes, and Scheduling

At a Glance
Bluetooth mesh *scenes* allow collections of devices of various types to be instructed to load specific settings simultaneously. This allows changes that affect many types of devices to be orchestrated all in one action. Scene selection can be triggered by a Bluetooth mesh message or via a time schedule. In support of scheduled operations, Bluetooth mesh makes it possible for an accurate system time to be propagated to nodes across the network.

Scenes and Scene Registers
A scene is a uniquely numbered list of states with associated state values that is split up and distributed across a number of elements within nodes in the network. Each element that uses scenes has a *scene register*, which is a state contained within the scene server model. The scene register is a table with each row identified by a scene number. Each row in the table also contains an object that acts as a container for all of the states and values that need to be memorized as part of that scene. The specific structural details for this container object are not specified and are left to the implementor.

The aggregate of all rows with the same scene number from all nodes in the network is a unique scene.

Example scene registers for elements within two types of node are shown in Figure 24 and Figure 25.

scene number	state container
10	generic onoff=0,light lightness actual=0,light HSL hue=0x42f4f4
Standby13	generic onoff=1,light lightness actual =65535,light HSL hue=0x42f4f4

Figure 24 - Example Scene Register for an Element of a Light

scene number	state container
10	generic level=100
Standby13	generic level=0

Figure 25 - Example Scene Register for the Blinds

The scene models define messages that the scene client model can publish to store, recall, or delete scenes from within receiving elements' scene registers.

In an example building and network, the requirement might be that when a room is occupied, lights are switched on and set to a given lightness level and hue, and the blinds are opened. An occupancy sensor could publish a scene recall message which specifies that scene 13 be activated when the room becomes occupied, and similarly, publish a scene recall message, activating scene 10 when the room becomes unoccupied. The example *scene register* in Figures 24 and 25 should illustrate how, in the first case, scene 13 would switch the lights on, set their lightness to full brightness, their color to a subtle blue color, and cause the blinds to open (level 0). In the second case, switching everything to scene 10 results in the lights being switched off and the blinds closing (level 100). All of these changes happen simultaneously in response to the scene client model in the sensor publishing a single scene recall message.

A Bluetooth mesh network can have up to 65,535 distinct scenes defined for it. Individual elements can store state values for up to 16 distinct scenes in their scene register, which should be more than enough for any type of device.

Time and Time Propagation

Times in a Bluetooth mesh network are based on the International Atomic Time (TAI) standard. Three models, the *time client*, *time server*, and *time setup server* are defined. When the time server model is present in an element, the time setup server model must also be present, per the usual usage pattern.

The time server model contains a single state called *time*. It contains a TAI time, information about uncertainty, and the degree to which the time can be trusted, plus information about time zone offsets. The *time setup server* adds a further state called *time role*. Time roles define whether or not an element participates in the propagation of time state values across the network and, if so, how. Four distinct roles are defined and represented by the time role state. They are listed in Table 5.2 of the Bluetooth Mesh Model Specification, which is repeated here for convenience:

Value	Role	Description
0x00	None	The element does not participate in propagation of time information.
0x01	Mesh Time Authority	The element publishes Time Status messages but does not process received Time Status messages.
0x02	Mesh Time Relay	The element processes received and published Time Status messages.
0x03	Mesh Time Client	The element does not publish but processes received Time Status messages.
0x04-0xFF	Prohibited	

The *time setup server* model defines messages that allow time role to be maintained in an element.

Time server models respond to messages relating to their time state in the usual way. But to propagate messages across the network, *time servers* with the *mesh time authority* role publish the time periodically in accordance with the publish period state, which is part of the *configuration server* model. Elements with the role mesh *time relay* also publish unsolicited time status messages, but they do so only when one is received from a *mesh time authority*. *Mesh time clients* are end points in the time propagation process, receiving and storing time data from *time status* messages, but not publishing or relaying them. It is in this way that time is distributed across the network.

Scheduling

Certain types of state changes can be scheduled to take place at a specific time every day, on a specific day of the week. This is made possible by a set of models, the *scheduler server*, *scheduler setup server*, and the *scheduler client*. The *scheduler server* extends the *scene server* model, and the *scheduler client* model extends the *scene server* model so the ability to schedule actions that change state is dependent on the scene models being implemented.

The *scheduler server* model contains a tabular state called the *scheduler register*. This state allows up to 16 sets of scheduling data to be stored, each consisting of scheduling time and frequency information, an action to take, and a scene number (optional). It therefore allows up to 16 state changing actions to be scheduled. Actions allowed switch the element on, switch it off, or recall the specified scene.

The scheduler offers a great deal of flexibility in how actions are scheduled, and every action can have an associated transition time specified for it as well.

7.0 Summary

This ends the technical overview of the Bluetooth mesh models. You should have a good understanding of how the *generics* support the use of fundamental capabilities that many device types possess, how commercial lighting requirements are met by the lighting models, and how sensors can be used to inform other devices of environmental data, perhaps initiating scene changes as a result. Finally, you should know how time plays a role in a Bluetooth mesh network, with a scheduler available to trigger state changes on a scheduled basis.

The Bluetooth mesh models are the building blocks for interoperable mesh products and the means by which diverse requirements can be met in smart buildings and elsewhere.

7.1 Additional Resources

The following resources are recommended to help you learn more about Bluetooth mesh:

Mesh Technology Overview	A short technical introduction to Bluetooth mesh technology.
Mesh Glossary of Terms	A glossary of Bluetooth mesh terminology.
Mesh Developer Study Guide	Self-study material with hands-on programming exercises for developers.