# x1990 - Illumination Light Control

Version 0

# IlluminationLight

This feature controls various aspects of illumination light devices such as studio lights or video streaming lights.

## Function documentation templates

**get<Control>Info**()
**get<Control>**() → value
**set<Control>**(value)
**set<Control>Levels**(flags, targetLevels, levelValues...)
**get<Control>Levels**(startIndex) → (flags, targetLevels, levelValues...)

## Functions

[0] **getIllumination**() → state
[1] **setIllumination**(state)
[2] **getBrightnessInfo**()
[3] **getBrightness**() → brightness
[4] **setBrightness**(brightness)
[5] **getBrightnessLevels**(startIndex) → flags, targetLevels, levelValues...
[6] **setBrightnessLevels**(flags, targetLevels, levelValues...)
[7] **getColorTemperatureInfo**()
[8] **getColorTemperature**() → colorTemperature
[9] **setColorTemperature**(colorTemperature)
[10] **getColorTemperatureLevels**(startIndex) → flags, targetLevels, levelValues...
[11] **setColorTemperatureLevels**(flags, targetLevels, levelValues...)

## Event documentation templates

**<control>ChangeEvent** → value

## Events

[0] **illuminationChangeEvent** → state
[1] **brightnessChangeEvent** → brightness
[2] **colorTemperatureChangeEvent** → colorTemperature

# Overview

This feature provides control over brightness, color temperature, and pre-defined levels for illumination light devices.

The unit for brightness values throughout this document is Lumen.

The unit for color temperature values throughout this document is Kelvin.

# Function documentation templates

Since this feature provides sets of functions that work identically for brightness and color temperature the documentation strives to avoid duplicating the same text. To this end, the individual functions refer to function documentation *templates*. The concept of these templates should be familiar to everyone who has worked with templated or generic code. For example, the documentation of getBrightnessInfo() is simply the get<Control>Info() documentation with "control" replaced with "brightness".

## get<Control>Info()

Returns information about the device's control capabilities.

### Parameters

**none**

### Returns

*Table 1. get<Control>Info() response packet format*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | **capabilities** | | | | | | | |
| | reserved | | | | | hasNonLinearLevels | hasLinearLevels | hasEvents |
| **1** | minMSB | | | | | | | |
| **2** | minLSB | | | | | | | |
| **3** | maxMSB | | | | | | | |
| **4** | maxLSB | | | | | | | |
| **5** | resMSB | | | | | | | |
| **6** | resLSB | | | | | | | |
| 7 | reserved | | | | maxLevels | | | |
| **8..15** | reserved | | | | | | | |

**hasEvents [flag]**

The device supports change event notification for this particular control.

**hasLinearLevels [flag]**

The device supports linear levels for this particular control.

**hasNonLinearLevels [flag]**

The device supports non-linear levels for this particular control. If this flag is unset (0) `maxLevels` must be set to 0.

**min [16 bits]**

The minimum control value supported by the device.

**max [16 bits]**

The maximum control value supported by the device. If `min == max` only a single control setting is supported, in which case `res` must be zero. In such a case the setControl() function is not supported by the device.

**res [16 bits]**

The resolution (step size) of control values supported by the device. The following conditions must hold:

```
min ≤ max
res > 0
(max - min) % res == 0
```

**maxLevels [4 bits]**

The maximum number of control levels the device supports for non-linear levels. A value of 0 indicates that non-linear levels are not supported for this particular control and `hasNonLinearLevels` must not be set. If levels are not supported the `hasLinearLevels` and `hasNonLinearLevels` must both be unset (0). If levels are not supported the **set<Control>Levels**(flags, targetLevels, levelValues...) and **get<Control>Levels**(startIndex) → (flags, targetLevels, levelValues...) functions are not supported.

The following table summarizes how some of the above fields play together:

| | `hasLinearLevels` | `hasNonLinearLevels` | `maxLevels` |
|---|---|---|---|
| No level support | 0 | 0 | 0 |
| Full level support (linear and non-linear) | 1 | 1 | > 0 |
| Linear level support only | 1 | 0 | 0 |
| Non-linear level support only | 0 | 1 | > 0 |

# get<Control>() → value

Returns the current hardware control value.

## Parameters

**none**

## Returns

*Table 2. get<Control>() response packet format*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | value (MSB) | | | | | | | |
| **1** | value (LSB) | | | | | | | |
| **2..15** | reserved | | | | | | | |

**value [16 bit]**

Current hardware control value, subject to the following conditions:

```
min ≤ value ≤ max
(value - min) % res == 0
```

# set<Control>(value)

Sets the hardware control value.

Note that this function is not supported if the given control's minimum and maximum values are equal as indicated by **get<Control>Info()**.

## Parameters

*Table 3. set<Control>() request packet format*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | value (MSB) | | | | | | | |
| **1** | value (LSB) | | | | | | | |
| **2..15** | reserved | | | | | | | |

**value [16 bit]**

Control value. This value is subject to the same conditions as the return value of getControl().

## Returns

**None**

### Errors

This function may return standard HID++ 2.0 error codes.

Possible errors include:

- NOT_ALLOWED if setting the control is not supported.
- INVALID_ARGUMENT if the value is out of the [`min`, `max`] range or `value - min` is not a multiple of the resolution.

## set<Control>Levels(flags, targetLevels, levelValues...)

Sets the control levels, i.e. the control values that can be selected by the user directly on the device, typically using one or more buttons. Note that these levels are distinct from the smooth ramping that devices may also provide using the same buttons.

This function is only available if at least one of the `hasLinearLevels` or `hasNonLinearLevels` flags is set as indicated by **get<Control>Info()**.

If level support is available the device must contain a valid level configuration out of factory.

Level definitions come in two flavors: Linear and non-linear.

- **Linear levels** are defined by a minimum, a maximum, and a step size.
- **Non-linear levels** are defined by a comprehensive list of values.

At any given time the device either uses linear levels ("linear mode") or non-linear levels ("non-linear mode"). The two modes cannot be combined.

The **level count** is the number of level values currently available for selection by the user. In linear mode the level count is implied by the formula `(max - min) / res`. In non-linear mode the level count is explicity set by the host (within the constraints returned by **get<Control>Info()**).

In linear mode each call affects all levels. In non-linear mode each call where affects a maximum number of seven target levels, namely the levels with indices in the range [`startIndex`, `startIndex` + `validCount` − 1]. (The exception are calls where `reset` is unset, in which case all levels are affected.)

In non-linear mode level values at indices equal to or larger than the current level count are considered undefined.

In non-linear mode the host is responsible for ensuring that the entire set of levels is valid. This means that all level values must adhere to the min/max/res constraints returned by **get<Control>Info()** and level values must be monotonically increasing. The device is not expected to validate the entire set of level values for each call but may do so (see the `INVALID_ARGUMENT`/"LV" error). Hosts that need to set level values in multiple calls but cannot ensure that no temporary inconsistencies occur between calls should therefore set levels starting from index 0 and increase `levelCount` to only include values that have already been set.

Example: A host looking to set 10 levels might first set 7 levels with a call that has `levelCount := 7` and then perform another call with 3 level values and `levelCount := 10`. This ensures that the

device does not reject the first call if the current value at index 7 happens to be less than the newly set value at index 6.

## Parameters

*Table 4. set<Control>Levels() request packet format. Parameters exclusive to non-linear mode are highlighted.*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | **flags** | | | | | | | |
| | validCount | | | reserved | | | reset | linear |
| **1** | **targetLevels** | | | | | | | |
| | startIndex | | | | levelCount | | | |
| **2** | level0Value (MSB) *or* levelMinValue (MSB) | | | | | | | |
| **3** | level0Value (LSB) *or* levelMinValue (LSB) | | | | | | | |
| **4** | level1Value (MSB) *or* levelMaxValue (MSB) | | | | | | | |
| **5** | level1Value (LSB) *or* levelMaxValue (LSB) | | | | | | | |
| **6** | level2Value (MSB) *or* levelStepValue (MSB) | | | | | | | |
| **7** | level2Value (LSB) *or* levelStepValue (LSB) | | | | | | | |
| **8** | level3Value (MSB) *or* reserved | | | | | | | |
| **9** | level3Value (LSB) *or* reserved | | | | | | | |
| **10** | level4Value (MSB) *or* reserved | | | | | | | |
| **11** | level4Value (LSB) *or* reserved | | | | | | | |
| **12** | level5Value (MSB) *or* reserved | | | | | | | |
| **13** | level5Value (LSB) *or* reserved | | | | | | | |
| **14** | level6Value (MSB) *or* reserved | | | | | | | |
| **15** | level6Value (LSB) *or* reserved | | | | | | | |

### Common parameters

**linear [flag]**

If set (1) the level definition contains linear levels. Parameters exclusive to non-linear mode are ignored.
If unset (0) the level definition contains non-linear levels. Parameters exclusive to linear mode are ignored.
Note that not all devices may support both linear and non-linear levels. The `hasNonLinearLevels` and `hasLinearLevels` flags returned by **get<Control>Info()** indicate which level types are supported.

**reset [flag]**

If unset (0) the level value fields describe the complete set (linear) or a subset (non-linear) of

new levels.
If set (1) all other fields are ignored and the complete set of levels for this control is reset to their factory defaults. This includes the level type (non-linear vs. linear), the min/max/step values (linear), and the level count and individual level values (non-linear).

#### Parameters exclusive to linear mode

**levelMinValue [16 bits]**

The value of the first level.

**levelMaxValue [16 bits]**

The value of the last level, subject to the following condition:

```
levelMinValue ≤ levelMaxValue
```

**levelStepValue [16 bits]**

The value steps between levels, subject to the following conditions:

```
levelStepValue > 0
(levelMaxValue − levelMinValue) % levelStepValue == 0
```

Note that the `levelMinValue`, `levelMaxValue`, and `levelStepValue` values are subject to the `min`, `max`, `res` restrictions of the control (see **get<Control>Info()**).

Example: A color temperature level could be defined as { min: 2700 K, max: 6500 K, step: 950 K } which would give the five values [ 2700 K, 3650 K, 4600 K, 5550 K, 6500 K ].

#### Parameters exclusive to non-linear mode

**validCount [3 bits]**

The number of valid level values contained in the data. Valid values for this field are in the range 1 to 7.

**startIndex [4 bits]**

The zero-based index at which to update non-linear levels.

**levelCount [4 bits]**

Sets the level count, that is the number of levels available for selection by the user.
Valid values for this field are in the range 0 to 15 but must not exceed `maxLevels`. If this field is identical to the device's current level count (as returned by **get<Control>Levels**(startIndex) → (flags, targetLevels, levelValues…)) the number of levels does not change. A value of 0 resets the number of available levels to the factory default, though this does not reset the level values themselves.
Once a new `levelCount` value has taken effect all level values with indices ≥ `levelCount` are invalidated.
If `levelCount` is set to include indices whose level values are currently invalid the values of those levels is undefined.

**level0Value .. level6Value [16 bits]**

The level values for indices `startIndex` to `startIndex + validCount - 1`.

Note that the `levelNValue` values are subject to the `min`, `max`, `res` restrictions of the control (see **get<Control>Info()**).

Example: A brightness level could be defined as [ 1%, 30%, 70%, 100% ].

## Returns

**None**

## Errors

This function may return standard HID++ 2.0 error codes.

In order to assist with debugging additional error data may *optionally* be included in the error response's data bytes. Such information appears in the form of short non-zero-terminated ASCII strings. In the case of short reports the additional error data may be truncated to a single byte.

Possible errors include:

- NOT_ALLOWED if levels are not supported for this control.
- INVALID_ARGUMENT with "T" if the level type (linear or non-linear) implied by the `linear` flag is not supported.
- INVALID_ARGUMENT with "D" if the payload data is too short.
- INVALID_ARGUMENT with "MMS" if the conditions on `levelMinValue`, `levelMaxValue`, or `levelStepValue` are violated.
- INVALID_ARGUMENT with "MMR" if the conditions on the control's `min`, `max`, or `res` restrictions are violated.
- INVALID_ARGUMENT with "LV" if the entire set of level values is otherwise inconsistent (e.g. not monotonically increasing). (Optional, see above.)
- INVALID_ARGUMENT with "VC" if `validCount` is out of range (i.e. < 1 or > 7).
- INVALID_ARGUMENT with "SI" if `startIndex` is out of range (i.e. ≥ `maxLevels`).
- INVALID_ARGUMENT with "PC" if `levelCount` is out of range (i.e. > `maxLevels`).

# get<Control>Levels(startIndex) → (flags, targetLevels, levelValues...)

Retrieves the current control levels.

Please refer to the the documentation on **set<Control>Levels**(flags, targetLevels, levelValues...) for generic information about control levels.

## Parameters

*Table 5. get<Control>Levels() request packet format.*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | startIndex | | | | reserved | | | |
| **1..15** | reserved | | | | | | | |

**startIndex [4 bits]**

The zero-based index starting from which to return non-linear levels. If the device currently uses linear levels this field is ignored.

## Returns

*Table 6. get<Control>Levels() response packet format. Parameters exclusive to non-linear mode are highlighted.*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | **flags** | | | | | | | |
| | validCount | | | reserved | | | | linear |
| **1** | **targetLevels** | | | | | | | |
| | startIndex | | | | levelCount | | | |
| **2** | level0Value (MSB) *or* levelMinValue (MSB) | | | | | | | |
| **3** | level0Value (LSB) *or* levelMinValue (LSB) | | | | | | | |
| **4** | level1Value (MSB) *or* levelMaxValue (MSB) | | | | | | | |
| **5** | level1Value (LSB) *or* levelMaxValue (LSB) | | | | | | | |
| **6** | level2Value (MSB) *or* levelStepValue (MSB) | | | | | | | |
| **7** | level2Value (LSB) *or* levelStepValue (LSB) | | | | | | | |
| **8** | level3Value (MSB) *or* reserved | | | | | | | |
| **9** | level3Value (LSB) *or* reserved | | | | | | | |
| **10** | level4Value (MSB) *or* reserved | | | | | | | |
| **11** | level4Value (LSB) *or* reserved | | | | | | | |
| **12** | level5Value (MSB) *or* reserved | | | | | | | |
| **13** | level5Value (LSB) *or* reserved | | | | | | | |
| **14** | level6Value (MSB) *or* reserved | | | | | | | |
| **15** | level6Value (LSB) *or* reserved | | | | | | | |

**Common return values**

**linear [flag]**

If set (1) the returned level definition contains linear levels. Return values exclusive to non-linear mode are undefined.
If unset (0) the returned level definition contains non-linear levels. Return values exclusive to linear mode are undefined.

#### Return values exclusive to linear mode

The contents of the fields below are only defined when the `linear` flag is set (1).

The three return values below adhere to the same conditions described for the corresponding **set<Control>Levels**(flags, targetLevels, levelValues...) parameters.

**levelMinValue [16 bits]**

The value of the first level.

**levelMaxValue [16 bits]**

The value of the last level.

**levelStepValue [16 bits]**

The value steps between levels.

#### Return values exclusive to non-linear mode

The contents of the fields below are only defined when the `linear` flag is unset (0).

**validCount [3 bits]**

The number of valid level values contained in the data. Valid values for this field are in the range 1 to 7.

**startIndex [4 bits]**

Value of the input parameter of the same name.

**levelCount [4 bits]**

Current number of levels available for selection by the user. This value is always non-zero.

**level0Value .. level6Value [16 bits]**

The level values for indices `startIndex` to `startIndex + validCount - 1`.

## Returns

**None**

## Errors

This function may return standard HID++ 2.0 error codes. Possible errors include:

- NOT_ALLOWED if levels are not supported for this control.
- INVALID_ARGUMENT if `startIndex` is out of range (i.e. ≥ `maxLevels`).

# Functions

## [0] getIllumination() → state

Retrieves the current illumination state.

### Parameters

**none**

### Returns

*Table 7. getIllumination() response packet format*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | reserved | | | | | | | state |
| **1..15** | reserved | | | | | | | |

**state [flag]**

Illumination state. 0 = off, 1 = on.

## [1] setIllumination(state)

Turns the illumination on or off.

### Parameters

*Table 8. setIllumination() request packet format*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | reserved | | | | | | | state |
| **1..15** | reserved | | | | | | | |

**state [flag]**

Illumination state. 0 = off, 1 = on.

### Returns

**None**

### Errors

This function may return standard HID++ 2.0 error codes. Possible errors include:

- HW_ERROR if the illumination cannot currently be turned on (e.g. because of insufficient power).

## [2] getBrightnessInfo()

Returns information about the device's brightness capabilities.

See **get<Control>Info()** for the semantics of this function.

## [3] getBrightness() → brightness

Returns the current hardware brightness value.

See **get<Control>**() → value for the semantics of this function.

## [4] setBrightness(brightness)

Sets the hardware brightness value.

See **set<Control>**(value) for the semantics of this function.

## [5] getBrightnessLevels(startIndex) → flags, targetLevels, levelValues...

Returns the device's current brightness level configuration.

See **get<Control>Levels**(startIndex) → (flags, targetLevels, levelValues...) for the semantics of this function.

## [6] setBrightnessLevels(flags, targetLevels, levelValues...)

Sets the device's brightness level configuration.

See **set<Control>Levels**(flags, targetLevels, levelValues...) for the semantics of this function.

## [7] getColorTemperatureInfo()

Returns information about the device's color temperature capabilities.

See **get<Control>Info()** for the semantics of this function.

## [8] getColorTemperature() → colorTemperature

Returns the current hardware color temperature value.

See **get<Control>**() → value for the semantics of this function.

## [9] setColorTemperature(colorTemperature)

Sets the hardware color temperature value.

See **set<Control>**(value) for the semantics of this function.

## [10] getColorTemperatureLevels(startIndex) → flags, targetLevels, levelValues...

Returns the device's current color temperature level configuration.

See **get<Control>Levels**(startIndex) → (flags, targetLevels, levelValues...) for the semantics of this function.

## [11] setColorTemperatureLevels(flags, targetLevels, levelValues...)

Sets the device's color temperature level configuration.

See **set<Control>Levels**(flags, targetLevels, levelValues...) for the semantics of this function.

# Event documentation templates

## <control>ChangeEvent → value

This event is generated only when user directly interacts with the hardware, e.g. by pressing physical buttons to adjust the control. It is *not* generated in response to **set<Control>**(value) requests.

The event may be generated when powering on the device to notify the host of the current/default control value.

### Parameters

*Table 9. <control>ChangeEvent event packet format.*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | value (MSB) | | | | | | | |
| **1** | value (LSB) | | | | | | | |
| **2..15** | reserved | | | | | | | |

**value [16 bits]**

The new hardware control value.

# Events

## [0] illuminationChangeEvent → state

Notifies the host of a user-initiated illumination state change.

This event is generated only when the user directly interacts with the hardware, e.g. by pressing physical buttons to turn the light on or off. It is *not* generated in response to [1] **setIllumination**(state) requests or plain HID requests.

The event may be generated when powering on the device to notify the host of the current/default illumination state.

### Parameters

*Table 10. illuminationChangeEvent event packet format.*

| byte \ bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| **0** | reserved | | | | | | | state |
| **1..15** | reserved | | | | | | | |

**state [flag]**

The new hardware illumination state. 0 = off, 1 = on.

## [1] brightnessChangeEvent → brightness

Notifies the host of a user-initiated brightness control change.

See **<control>ChangeEvent** → value for the semantics of this event.

## [2] colorTemperatureChangeEvent → colorTemperature

Notifies the host of a user-initiated color temperature control change.

See **<control>ChangeEvent** → value for the semantics of this event.

# ChangeLog

- Version 0: Initial version