How To Add HS-WX300+ to SmartThings Hub

Information and code for adding the WX300 to SmartThings

Purpose

This device handler provides extended feature support for the WX300 in SmartThings new app.

To add this Device Handler:

1. Navigate to this URL and log in with your SmartThings account: https://graph-na04-useast2.api.smartthings.com
2. Click on the "My Device Handlers" menu
3. Click "Create New Device Handler" on the right
4. Click "From Code"
5. Copy/Paste the Device Handler Code from the bottom of this page
6. Click "Create" at the bottom
7. Click "Publish" and select "For Me"
8. Now add the WX300 to the new SmartThings app and test the dimmer.

NOTE: If your device is added as a generic switch or a generic dimmer, SmartThings has assigned it the wrong device type. To fix this follow the instructions below.

1. Navigate to this URL and log in with your SmartThings account: https://graph-na04-useast2.api.smartthings.com
2. Click on the "Devices" menu
3. Click the name of the switch/dimmer on the left
4. Click "Edit" at the bottom of the page
5. Open the "Type*" field and select "HomeSeer Wall Dimmer HS-WD200/HS-WX300"
6. Click "Update"
7. Restart the SmartThings app.

Optional Child Devices

Three additional device handlers are available for the WD200+, WS200+, FC200+, and WX300. These device handlers allow you to add additional child devices to the app that will allow you to control the features related to the custom LEDs on the products.

- HomeSeer Normal Mode Child
- HomeSeer Status LED Blink Frequency Child
- HomeSeer Status LED Blinking Color Child

Device Handler Code

```
/*
 *  HomeSeer Wall Dimmer HS-WD200+ & HS-WX300 (v1.2)
 *
 *  Changelog:
 *
 *    1.2 (11/23/2021)
 *      - Added support for model HS-WX300 in Switch Mode
 *
 *    1.1 (10/06/2021)
 *      - Added support for model HS-WX300
 *
 *    1.0 (10/31/2020)
 *      - Initial Release
 *
 *
 *  Copyright 2020 HomeSeer
 *
 *  Licensed under the Apache License, Version 2.0 (the "License");
 *  you may not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS,
 *  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
```

```
 *
*/

import groovy.transform.Field

@Field int defaultCmdDelay = 100

@Field static Map commandClassVersions = [
        0x20: 1,        // Basic
        0x25: 1,        // Switch Binary
        0x26: 3,        // Switch Multilevel (4)
        0x27: 1,        // Switch All
        0x2B: 1,        // Scene Activation
        0x2C: 1,        // Scene Actuator Conf
        0x55: 1,        // Transport Service (2)
        0x59: 1,        // AssociationGrpInfo
        0x5A: 1,        // DeviceResetLocally
        0x5B: 1,        // CentralScene (3)
        0x5E: 2,        // ZwaveplusInfo
        0x6C: 1,        // Supervision
        0x70: 1,        // Configuration (3)
        0x7A: 2,        // FirmwareUpdateMd
        0x72: 2,        // ManufacturerSpecific
        0x73: 1,        // Powerlevel
        0x85: 2,        // Association
        0x86: 1,        // Version (2)
        0x98: 1,        // Security S0
        0x9F: 1                // Security S2
]

@Field static String normalModeChildDTH = "HomeSeer Normal Mode Child"
@Field static String statusBlinkChildDTH = "HomeSeer Status LED Blink Frequency Child"
@Field static String statusLedChildDTH = "HomeSeer Status LED Blinking Color Child"

@Field static String normalModeChildDNI = "NormalMode"
@Field static String statusColorChildDNI = "StatusModeColor"
@Field static String statusBlinkChildDNI = "StatusModeBlink"
@Field static String statusLedChildDNIPrefix = "StatusModeLed"

@Field static String defaultStatusLedColor = "off"

@Field static Map ledCmdDelayOptions = [50:"50ms", 100:"100ms", 150:"150ms", 200:"200ms", 250:"250ms", 300:"
300ms", 350:"350ms", 400:"400ms", 450:"450ms", 500:"500ms [DEFAULT]", 600:"600ms", 700:"700ms", 800:"800ms",
900:"900ms", 1000:"1s", 1100:"1.1s", 1200:"1.2s", 1300:"1.3s", 1400:"1.4s", 1500:"1.5s"]

@Field static Map statusLeds = [1:"One", 2:"Two", 3:"Three", 4:"Four", 5:"Five", 6:"Six", 7:"Seven"]

@Field static List<Integer> statusLedColorParamNums = [21, 22, 23, 24, 25, 26, 27]

@Field static Map normalBottomLedBehaviorOptions = [0:"Bottom LED ON if load is OFF", 1:"Bottom LED OFF if
load is OFF [DEFAULT]"]

@Field static Map paddleLoadOrientationOptions = [0:"Top of Paddle turns load ON [DEFAULT]", 1:"Bottom of
Paddle turns load ON"]

@Field static Map lowestDimmingThresholdOptions = [1:"6.5% [DEFAULT]", 2:"8%", 3:"9%", 4:"10%", 5:"11%", 6:"
12%", 7:"13%", 8:"14%",9:"15%",10:"16%",11:"17%",12:"18%",13:"19%",14:"20%"]

@Field static Map centralSceneEnabledOptions = [0:"Enabled [DEFAULT]", 1:"Disabled"]

@Field static Map dimmerRampRateOptions = [0:"Instant", 1:"1 Second", 2:"2 Seconds", 3:"3 Seconds
[DEFAULT]", 4:"4 Seconds", 5:"5 Seconds", 6:"6 Seconds", 7:"7 Seconds", 8:"8 Seconds", 9:"9 Seconds", 10:"10
Seconds", 15:"15 Seconds", 20:"20 Seconds", 25:"25 Seconds", 30:"30 Seconds"]

@Field static Map ledModeOptions = [0:"normal", 1:"status"]

@Field static Map normalLedColorOptions = [0:"white", 1:"red", 2:"green", 3:"blue", 4:"magenta", 5:"yellow",
6:"cyan"]

@Field static Map statusLedColorOptions = [0:"off", 1:"red", 2:"green", 3:"blue", 4:"magenta", 5:"yellow",
6:"cyan", 7:"white"]
```

```
@Field static Map statusBlinkFrequencyOptions = [0:"Blinking Disabled", 1:"100ms", 2:"200ms", 3:"300ms", 4:"
400ms", 5:"500ms", 6:"600ms", 7:"700ms", 8:"800ms", 9:"900ms", 10:"1s", 11:"1.1s", 12:"1.2s", 13:"1.3s", 14:"
1.4s", 15:"1.5s", 20:"2s", 25:"2.5s", 30:"3s", 35:"3.5s", 40:"4s", 45:"4.5s", 50:"5s", 60:"6s", 70:"7s", 80:"
8s", 90:"9s", 100:"10s", 110:"11s", 120:"12s", 130:"13s", 140:"14s", 150:"15s", 160:"16s", 170:"17s", 180:"
18s", 190:"19s", 200:"20s", 210:"21s", 220:"22s", 230:"23s", 240:"24s", 250:"25s"]

@Field static Map wireModeOptions = [0: "3 Wire Mode", 1: "2 Wire Mode (Line & Load)"]

@Field static Map createChildOptions = [0:"No [DEFAULT]", 1:"Yes"]

@Field static Map debugLoggingOptions = [0:"Disabled", 1:"Enabled [DEFAULT]"]


metadata {
        definition (
                name: "HomeSeer Wall Dimmer HS-WD200/HS-WX300",
                namespace: "HomeSeer",
                author: "Kevin LaFramboise (krlaframboise)",
                ocfDeviceType: "oic.d.switch",
                mnmn: "SmartThingsCommunity",
                vid: "e01f6333-d277-37a4-876a-309df1c09070"
        ) {
                capability "Actuator"
                capability "Sensor"
                capability "Button"
                capability "Configuration"
                capability "Health Check"
                capability "Refresh"
                capability "Switch Level"
                capability "Switch"
                capability "platemusic11009.firmware"
                capability "platemusic11009.hsLedMode"
                capability "platemusic11009.hsNormalLedColor"
                capability "platemusic11009.hsStatusLedBlinkingColor"
                capability "platemusic11009.hsStatusLedBlinkFrequency"
                capability "platemusic11009.hsStatusLedOneColor"
                capability "platemusic11009.hsStatusLedTwoColor"
                capability "platemusic11009.hsStatusLedThreeColor"
                capability "platemusic11009.hsStatusLedFourColor"
                capability "platemusic11009.hsStatusLedFiveColor"
                capability "platemusic11009.hsStatusLedSixColor"
                capability "platemusic11009.hsStatusLedSevenColor"

                attribute "lastCheckIn", "string"

                command "setNormalLedMode"
                command "setStatusLedMode"

                // colorName or color #, led #(optional), blink frequency #(optional)
                command "setStatusLedColorBlinkFrequency", ["string", "number", "number"]

                fingerprint mfr: "000C", prod: "4447", model: "3036", deviceJoinName: "HomeSeer Wall Dimmer
WD200" //WD200

                // zw:Ls2a type:1100 mfr:000C prod:4447 model:4036 ver:1.11 zwv:7.15 lib:03 cc:5E,55,9F,6C
sec:26,70,5B,85,59,5A,7A,87,72,8E,73,86
                fingerprint mfr: "000C", prod: "4447", model: "4036", deviceJoinName: "HomeSeer Wall Dimmer
WX300" //WX300 in Dimmer Mode

                // zw:Ls2a type:1000 mfr:000C prod:4447 model:4037 ver:1.11 zwv:7.15 lib:03 cc:5E,55,9F,6C
sec:25,70,5B,85,59,5A,7A,87,72,8E,73,86
                fingerprint mfr: "000C", prod: "4447", model: "4037", deviceJoinName: "HomeSeer Wall Switch
WX300" //WX300 in Switch Mode
        }

        simulator { }

        preferences {
                settingsConfigParams.each {
                        createEnumInput("configParam${it.num}", "${it.name}:", it.value, it.options)
```

```
                }
                createEnumInput("debugOutput", "Debug Logging", 1, debugLoggingOptions)

                createWorkaroundPreferences()
        }
}

void createEnumInput(String name, String title, Integer defaultVal, Map options) {
        if (defaultVal != null) {
                input name, "enum",
                        title: title,
                        required: false,
                        defaultValue: defaultVal.toString(),
                        options: options
        } else {
                input name, "enum",
                        title: title,
                        required: false,
                        options: options
        }
}


def installed() {
        logDebug "installed()..."

        initialize()

        return []
}


def updated() {
        if (!isDuplicateCommand(state.lastUpdated, 2000)) {
                state.lastUpdated = new Date().time

                logDebug "updated()..."

                initialize()

                if (state.isConfigured) {
                        executeConfigureCmds()
                }
        }
        return []
}

void initialize() {
        state.debugLoggingEnabled = (safeToInt(settings?.debugOutput, 1) != 0)
        state.ledCmdDelay = safeToInt(settings?.ledCmdDelay, 500)

        if (!device.currentValue("checkInterval")) {
                int checkInterval = ((60 * 60 * 3) + (5 * 60))
                sendEvent(name: "checkInterval", value: checkInterval, displayed: false, data: [protocol:
"zwave", hubHardwareId: device.hub.hardwareID, offlinePingable: "1"])
        }

        if (!device.currentValue("ledMode")) {
                sendEvent(name: "ledMode", value: "normal")
        }

        if (!device.currentValue("normalLedColor")) {
                sendEvent(name: "normalLedColor", value: "white")
        }

        if (!device.currentValue("statusLedColor")) {
                sendEvent(name: "statusLedColor", value: defaultStatusLedColor)
        }

        if (device.currentValue("statusLedBlinkFrequency") == null) {
                sendEvent(name: "statusLedBlinkFrequency", value: 0)
```

```
	}

	statusLeds.each { num, name ->
		if (!device.currentValue("statusLed${name}Color")) {
			sendEvent(name: "statusLed${name}Color", value: defaultStatusLedColor)
		}
	}

	if (!device.currentValue("supportedButtonValues")) {
		sendEvent(name:"supportedButtonValues", value: ["down","down_hold","down_2x","down_3x","
down_4x","down_5x","up","up_hold","up_2x","up_3x","up_4x","up_5x"].encodeAsJSON(), displayed:false)
	}

	if (!device.currentValue("numberOfButtons")) {
		sendEvent(name:"numberOfButtons", value:1, displayed:false)
	}

	if (!device.currentValue("button")) {
		sendButtonEvent("up")
	}

	runIn(3, initializeChildDevices)
}


def configure() {
	logDebug "configure()..."

	if (!state.isConfigured) {
		runIn(8, executeConfigureCmds, [overwrite: true])
	}
	else {
		executeConfigureCmds()
	}
	return []
}

void executeConfigureCmds() {
	List<String> cmds = []

	int changes = pendingChanges
	if (changes) {
		log.warn "Syncing ${changes} Change(s)"
	}

	if (!state.isConfigured || !device.currentValue("switch")) {
		cmds << switchMultilevelGetCmd()
	}

	if (!device.currentValue("firmwareVersion")) {
		cmds << versionGetCmd()
	}

	if (state.lifelineAssoc != true) {
		cmds << lifelineAssociationSetCmd()
		cmds << lifelineAssociationGetCmd()
	}

	if (!state.isConfigured) {
		allConfigParams.each { param ->
			if (param == statusLedBlinkFrequencyParam) {
				cmds << configSetCmd(param, param.value)
			}
			cmds << configGetCmd(param)
		}
	}
	else {
		settingsConfigParams.each { param ->
			Integer storedVal = getParamStoredValue(param.num)
			if ((storedVal != param.value) && (param.value != null)) {
```

```
                                logDebug "Changing ${param.name}(#${param.num}) from ${storedVal} to ${param.
value}"
                                cmds << configSetCmd(param, param.value)
                                cmds << configGetCmd(param)
                        }
                }
        }

        state.isConfigured = true
        sendCommands(cmds, 500)
}

int getPendingChanges() {
        int configChanges = safeToInt(settingsConfigParams.count { it.value != getParamStoredValue(it.num) })
        return (configChanges + (state.lifelineAssoc != true ? 1 : 0))
}


def ping() {
        logDebug "ping()..."
        sendCommands([ switchMultilevelGetCmd() ])
}


def setNormalLedMode() {
        setLedMode("normal")
}

def setStatusLedMode() {
        setLedMode("status")
}

def setLedMode(mode) {
        logDebug "setLedMode(${mode})..."

        List<String> cmds = []

        Integer value = findIntKeyByValue(ledModeOptions, mode)
        if (value != null) {
                cmds += [
                        configSetCmd(ledModeParam, value),
                        configGetCmd(ledModeParam)
                ]
        }
        else {
                log.warn "${mode} is not a valid LED Mode"
        }
        sendCommands(cmds)
}


def setStatusLedColorBlinkFrequency(colorName, ledNumber=null, blinkFrequency=null) {
        logDebug "setStatusLedColorBlinkFrequency($colorName, $ledNumber, $blinkFrequency)"

        if (blinkFrequency != null) {
                setStatusLedBlinkFrequency(blinkFrequency)
        }

        if (colorName) {
                executeSetStatusLedColor(colorName, ledNumber)
        }
}


def setStatusLedBlinkFrequency(frequency) {
        logDebug "setStatusLedBlinkFrequency($frequency)..."

        Map param = statusLedBlinkFrequencyParam

        sendCommands([
                configSetCmd(param, safeToIntRange(frequency, 0, 0, 255)),
```

```
                configGetCmd(param)
        ])
}


def setStatusLedOneColor(colorName) {
        executeSetStatusLedColor(colorName, 1)
}
def setStatusLedTwoColor(colorName) {
        executeSetStatusLedColor(colorName, 2)
}
def setStatusLedThreeColor(colorName) {
        executeSetStatusLedColor(colorName, 3)
}
def setStatusLedFourColor(colorName) {
        executeSetStatusLedColor(colorName, 4)
}
def setStatusLedFiveColor(colorName) {
        executeSetStatusLedColor(colorName, 5)
}
def setStatusLedSixColor(colorName) {
        executeSetStatusLedColor(colorName, 6)
}
def setStatusLedSevenColor(colorName) {
        executeSetStatusLedColor(colorName, 7)
}

def setStatusLedColor(colorName) {
        executeSetStatusLedColor(colorName, null)
}

void executeSetStatusLedColor(colorName, Integer led=null) {
        logDebug "setStatusLedColor($colorName, $led)..."

        colorName = ("${colorName}".isInteger() ? statusLedColorOptions.get(safeToInt(colorName)) :
colorName.toString().trim())

        List<String> cmds = []

        boolean blinking = isBlinkingColorName(colorName)
        String color = formatColorName(colorName, false)

        if (led) {
                cmds += getChangeLedColorCmds(getStatusLedColorParam(led), statusLedColorOptions, color)
        }
        else {
                sendStatusLedColorEvents(formatColorName(colorName, blinking))

                statusLedColorParams.each {
                        cmds += getChangeLedColorCmds(it, statusLedColorOptions, color)
                }
        }

        Map blinkParam = statusLedBlinkBitmaskParam
        int newBlinkValue = calculateStatusLedBlinkBitmask(blinking, led)

        if (getParamStoredValue(blinkParam.num) != newBlinkValue) {
                state.pendingStatusLedBlinkBitmaskReport = true

                cmds << configSetCmd(blinkParam, newBlinkValue)
                cmds << configGetCmd(blinkParam)
        }
        sendCommands(cmds, state.ledCmdDelay)
}


def setNormalLedColor(color) {
        logDebug "setNormalLedColor($color)..."

        sendCommands(getChangeLedColorCmds(normalLedColorParam, normalLedColorOptions, color))
}
```

```
List<String> getChangeLedColorCmds(Map param, Map options, color) {
	List<String> cmds = []

	Integer colorId = ("${color}".isInteger() ? safeToInt(color) : findIntKeyByValue(options, color.
toLowerCase().trim()))
	if (colorId != null) {
		cmds += [
			configSetCmd(param, colorId),
			configGetCmd(param)
		]
	}
	else {
		log.warn "${color} is not a ${param.name}"
	}
	return cmds
}


def on() {
	logDebug "on()..."
	if (isWX300InSwitchMode()) {
		sendCommands(getSwitchCmds(0xFF))
	} else {
		sendCommands(getSetLevelCmds(device.currentValue("level")))
	}
}


def off() {
	logDebug "off()..."
	if (isWX300InSwitchMode()) {
		sendCommands(getSwitchCmds(0x00))
	} else {
		sendCommands(getSetLevelCmds(0x00))
	}
}


def setLevel(level, rate=null) {
	logDebug "setLevel($level, $rate)..."

	if (isWX300InSwitchMode()) {
		sendCommands(getSwitchCmds(level ? 0xFF : 0x00))
	} else {
		sendCommands(getSetLevelCmds(level, rate))
	}
}

List<String> getSwitchCmds(int value) {
	return [
		switchBinarySetCmd(value)
	]
}

List<String> getSetLevelCmds(level, rate=null) {
	int durationVal = safeToIntRange(rate, remoteDimmerRampRateParam.value, 0, 30)
	return [
		switchMultilevelSetCmd(safeToPercentInt(level, 99), durationVal),
		switchMultilevelGetCmd()
	]
}


def refresh() {
	logDebug "refresh()..."

	List<String> cmds = [
		switchMultilevelGetCmd(),
		versionGetCmd(),
		lifelineAssociationGetCmd()
```

```
        ]

        allConfigParams.each {
                cmds << configGetCmd(it)
        }

        sendCommands(cmds, 500)
}


List<String> sendCommands(List<String> cmds, Integer delay=null) {
        if (cmds) {
                delay = ((delay == null) ? defaultCmdDelay : delay)

                def actions = []
                cmds.each {
                        actions << new physicalgraph.device.HubAction(it)
                }
                sendHubCommand(actions, delay)
        }
        return []
}


String lifelineAssociationGetCmd() {
        return secureCmd(zwave.associationV2.associationGet(groupingIdentifier: 1))
}

String lifelineAssociationSetCmd() {
        return secureCmd(zwave.associationV2.associationSet(groupingIdentifier: 1, nodeId: [zwaveHubNodeId]))
}

String versionGetCmd() {
        return secureCmd(zwave.versionV1.versionGet())
}

String switchBinarySetCmd(Integer value) {
        // the WX300 doesn't support this switch binary set even though it sends those reports when the
device turns on/off.
        // return secureCmd(zwave.switchBinaryV1.switchBinarySet(switchValue: value))
        return secureCmd(zwave.basicV1.basicSet(value: value))
}

String switchMultilevelSetCmd(Integer value, Integer duration) {
        return secureCmd(zwave.switchMultilevelV3.switchMultilevelSet(dimmingDuration: duration, value:
value))
}

String switchMultilevelGetCmd() {
        return secureCmd(zwave.switchMultilevelV3.switchMultilevelGet())
}

String configSetCmd(Map param, Integer value) {
        return secureCmd(zwave.configurationV1.configurationSet(parameterNumber: param.num, size: param.
size, scaledConfigurationValue: value))
}

String configGetCmd(Map param) {
        return secureCmd(zwave.configurationV1.configurationGet(parameterNumber: param.num))
}

String secureCmd(cmd) {
        try {
                if (zwaveInfo?.zw?.contains("s") || ("0x98" in device?.rawDescription?.split(" "))) {
                        return zwave.securityV1.securityMessageEncapsulation().encapsulate(cmd).format()
                }
                else {
                        return cmd.format()
                }
        }
        catch (ex) {
```

```
                return cmd.format()
        }
}


def parse(String description) {
        def cmd = zwave.parse(description, commandClassVersions)
        if (cmd) {
                if (cmd.commandClassId == 38 && cmd.commandId == 3) {
                        handleSwitchMultilevelReport(cmd, description)
                }
                else {
                        zwaveEvent(cmd)
                }

        }
        else {
                log.warn "Unable to parse: $description"
        }

        updateLastCheckIn()
        return []
}

void updateLastCheckIn() {
        if (!isDuplicateCommand(state.lastCheckInTime, 60000)) {
                state.lastCheckInTime = new Date().time

                sendEvent(name: "lastCheckIn", value: convertToLocalTimeString(new Date()), displayed: false)
        }
}

String convertToLocalTimeString(dt) {
        try {
                def timeZoneId = location?.timeZone?.ID
                if (timeZoneId) {
                        return dt.format("MM/dd/yyyy hh:mm:ss a", TimeZone.getTimeZone(timeZoneId))
                }
                else {
                        return "$dt"
                }
        }
        catch (ex) {
                return "$dt"
        }
}


void zwaveEvent(physicalgraph.zwave.commands.securityv1.SecurityMessageEncapsulation cmd) {
        def encapsulatedCmd = cmd.encapsulatedCommand(commandClassVersions)
        if (encapsulatedCmd) {
                zwaveEvent(encapsulatedCmd)
        }
        else {
                log.warn "Unable to extract encapsulated cmd from $cmd"
        }
}


void zwaveEvent(physicalgraph.zwave.commands.configurationv1.ConfigurationReport cmd) {
        logTrace "${cmd}"

        Map param = allConfigParams.find { it.num == cmd.parameterNumber }
        if (param) {
                int val = cmd.scaledConfigurationValue

                setParamStoredValue(param.num, val)

                switch (param.num) {
                        case ledModeParam.num:
                                sendEventIfNew("ledMode", ledModeOptions.get(val))
```

```
                                findChild(normalModeChildDNI)?.sendLedModeEvents(ledModeOptions.get(val))
                                break

                        case normalLedColorParam.num:
                                sendEventIfNew("normalLedColor", normalLedColorOptions.get(val))
                                findChild(normalModeChildDNI)?.sendColorEvents(normalLedColorOptions.get
(val))
                                break

                        case statusLedBlinkFrequencyParam.num:
                                sendEventIfNew("statusLedBlinkFrequency", val)
                                findChild(statusBlinkChildDNI)?.sendBlinkFrequencyEvents(val)
                                break

                        case statusLedBlinkBitmaskParam.num:
                                handleStatusLedBlinkBitmaskReport(val)
                                break

                        case { it in statusLedColorParamNums }:
                                if (!state.pendingStatusLedBlinkBitmaskReport) {
                                        handleStatusLedColorReport(param, val)
                                }
                                break

                        default:
                                logDebug "${param.name}(#${param.num}) = ${val}"
                }
        }
        else {
                logDebug "Parameter #${cmd.parameterNumber} = ${cmd.scaledConfigurationValue}"
        }
}

void handleStatusLedBlinkBitmaskReport(int value) {
        state.pendingStatusLedBlinkBitmaskReport = false

        Map ledBlinkValues = getStatusLedBlinkConfigValues(value)

        statusLeds.each { ledNum, ledName ->
                String colorName = formatColorName(getStatusLedStoredColor(ledNum), ledBlinkValues.get
(ledNum))

                sendStatusLedColorEvents(colorName, ledNum)
        }
}

void handleStatusLedColorReport(Map param, int value) {
        int led = ((statusLedColorParamNums.findIndexOf { it == param.num }) + 1)

        Map ledBlinkValues = getStatusLedBlinkConfigValues(getParamStoredValue(statusLedBlinkBitmaskParam.
num, 0))

        String colorName = formatColorName(statusLedColorOptions.get(value), ledBlinkValues.get(led))

        sendStatusLedColorEvents(colorName, led)
}


void sendStatusLedColorEvents(String color, Integer led=null) {
        if (led) {
                sendEventIfNew("statusLed${statusLeds.get(led)}Color", color)

                findChild(getStatusLedChildDNI(led))?.sendColorEvents(color)
        }
        else {
                sendEventIfNew("statusLedColor", color)

                findChild(statusColorChildDNI)?.sendColorEvents(color)
        }
}
```

```
void zwaveEvent(physicalgraph.zwave.commands.associationv2.AssociationReport cmd) {
	if (cmd.groupingIdentifier == 1) {
		logDebug "Lifeline Association: ${cmd.nodeId}"
		state.lifelineAssoc = (cmd.nodeId == [zwaveHubNodeId]) ? true : false
	}
}


void zwaveEvent(physicalgraph.zwave.commands.versionv1.VersionReport cmd) {
	String subVersion = String.format("%02d", cmd.applicationSubVersion)
	String fullVersion = "${cmd.applicationVersion}.${subVersion}"

	sendEventIfNew("firmwareVersion", fullVersion.toBigDecimal())
}


void zwaveEvent(physicalgraph.zwave.commands.basicv1.BasicReport cmd) {
	logTrace "${cmd}"
	if (cmd.value && (device.currentValue("switch") == "off")) {
		sendEventIfNew("switch", "on", "physical")
	}
}

void zwaveEvent(physicalgraph.zwave.commands.switchbinaryv1.SwitchBinaryReport cmd) {
	logTrace "${cmd}"
	sendEventIfNew("switch", (cmd.value ? "on" : "off"))
}

void handleSwitchMultilevelReport(physicalgraph.zwave.commands.switchmultilevelv3.SwitchMultilevelReport
cmd, String description) {
	logTrace "${cmd}: ${description}"

	int targetValue = parseTargetValue(description, cmd.value)
	sendSwitchEvents(targetValue, "digital")
}

int parseTargetValue(String description, int value) {
	try {
		List<String> payload = description.substring(description.indexOf("payload:")).split(" ")
		return Integer.parseInt(payload[2], 16)
	}
	catch (e) {
		log.warn "unable to parse targetValue from '${description}'"
		return value
	}
}

void sendSwitchEvents(int value, String type) {
	String switchVal = value ? "on" : "off"

	sendEventIfNew("switch", switchVal, type)

	if (value) {
		sendEventIfNew("level", value, type, "%")
	}
}


void zwaveEvent(physicalgraph.zwave.commands.centralscenev1.CentralSceneNotification cmd){
	if (state.lastSequenceNumber != cmd.sequenceNumber) {
		state.lastSequenceNumber = cmd.sequenceNumber

		String paddle = (cmd.sceneNumber == 1) ? "up" : "down"
		String btnVal
		switch (cmd.keyAttributes){
			case 0:
				btnVal = paddle
				break
			case 1:
				logDebug "${paddle}_released is not supported by SmartThings"
```

```
                                break
                        case 2:
                                btnVal = paddle + "_hold"
                                break
                        case { it >= 3 && it <= 7}:
                                btnVal = paddle + "_${cmd.keyAttributes - 1}x"
                                break
                        default:
                                logDebug "keyAttributes ${cmd.keyAttributes} not supported"
                }

                if (btnVal) {
                        sendButtonEvent(btnVal)
                }
        }
}

void sendButtonEvent(String value) {
        String desc = "${device.displayName} ${value}"
        logDebug(desc)

        sendEvent(name: "button", value: value, data:[buttonNumber: 1], isStateChange: true,
descriptionText: desc)
}


void zwaveEvent(physicalgraph.zwave.Command cmd) {
        logDebug "Unhandled zwaveEvent: $cmd"
}


String formatColorName(String colorName, int blinkBit) {
        return formatColorName(colorName, (blinkBit ? true : false))
}

String formatColorName(String colorName, boolean blinking) {
        colorName = "${colorName}".toLowerCase().replace("blinking", "").trim()
        if ((colorName != "off") && blinking) {
                colorName = "blinking".concat(colorName.capitalize())
        }
        return colorName
}


boolean isBlinkingColorName(colorName) {
        return ("${colorName}".toLowerCase().indexOf("blinking") > -1)
}


int calculateStatusLedBlinkBitmask(boolean blinking, Integer led=null) {
        int blinkBit = (blinking ? 1 : 0)
        int ledBlinkBitmaskValue = getParamStoredValue(statusLedBlinkBitmaskParam.num, 0)
        Map ledBlinkValues =  getStatusLedBlinkConfigValues(ledBlinkBitmaskValue)

        if (led) {
                ledBlinkValues.put(led, blinkBit)
        }
        else {
                statusLeds.each { ledNum, ledName ->
                        ledBlinkValues.put(ledNum, blinkBit)
                }
        }
        return Integer.parseInt(ledBlinkValues.collect { it.value }.reverse().join(""), 2)
}

Map getStatusLedBlinkConfigValues(int configVal) {
        Map data = [:]

        char[] blinkBits = Integer.toBinaryString(configVal).padLeft(8, "0").reverse().toCharArray()

        statusLeds.each {
```

```
                data.put(it.key, safeToInt(blinkBits[it.key - 1]))
        }
        return data
}


String getStatusLedStoredColor(int led) {
        int paramNum = statusLedColorParamNums[led - 1]
        return statusLedColorOptions.get(getParamStoredValue(paramNum, 0))
}

Integer getParamStoredValue(Integer paramNum, Integer defaultVal=null) {
        return safeToInt(state["configVal${paramNum}"] , defaultVal)
}

void setParamStoredValue(Integer paramNum, Integer value) {
        state["configVal${paramNum}"] = value
}


List<Map> getAllConfigParams() {
        List<Map> params = [
                ledModeParam,
                normalLedColorParam,
                statusLedBlinkBitmaskParam
        ]

        params += settingsConfigParams

        params += statusLedColorParams

        return params
}

List<Map> getSettingsConfigParams() {
        return [
                normalBottomLedBehaviorParam,
                paddleLoadOrientationParam,
                lowestDimmingThresholdParam,
                centralSceneEnabledParam,
                remoteDimmerRampRateParam,
                localDimmerRampRateParam,
                statusLedBlinkFrequencyParam,
                wireModeParam
        ]
}

List<Map> getStatusLedColorParams() {
        List<Map> params = []
        statusLeds.each {
                params << getStatusLedColorParam(it.key)
        }
        return params
}


Map getNormalBottomLedBehaviorParam() {
        return getParam(3, "Normal Bottom LED Behavior", 1, 1, normalBottomLedBehaviorOptions)
}

Map getPaddleLoadOrientationParam() {
        return getParam(4, "Paddle's Load Orientation", 1, 0, paddleLoadOrientationOptions)
}

Map getLowestDimmingThresholdParam() {
        return getParam(5, "Lowest Dimming Threshold (FIRMWARE >= 5.14)", 1, 1,
lowestDimmingThresholdOptions)
}

Map getCentralSceneEnabledParam() {
        return getParam(6, "Enable/Disable Central Scene (FIRMWARE >= 5.12)", 1, 0,
```

```
centralSceneEnabledOptions)
}

Map getRemoteDimmerRampRateParam() {
        return getParam(11, "Dimmer Ramp Rate for Remote Control", 1, 3, dimmerRampRateOptions)
}

Map getLocalDimmerRampRateParam() {
        return getParam(12, "Dimmer Ramp Rate for Local Control", 1, 3, dimmerRampRateOptions)
}

Map getLedModeParam() {
        return getParam(13, "LED Mode", 1, 0, ledModeOptions)
}

Map getNormalLedColorParam() {
        return getParam(14, "Normal LED Color", 1, 0, normalLedColorOptions)
}

Map getStatusLedColorParam(led) {
        return getParam(statusLedColorParamNums[(led - 1)], "Status LED ${led} Color", 1, 0,
statusLedColorOptions)
}

Map getStatusLedBlinkFrequencyParam() {
        return getParam(30, "Status LED Blink Frequency (all LEDs)", 1, 3, statusBlinkFrequencyOptions)
}

Map getStatusLedBlinkBitmaskParam() {
        return getParam(31, "Status LED Blink Bitmask", 1, 0, [:])
}

Map getWireModeParam() {
        return getParam(32, "Wire Mode (WX300 Only)", 1, null, wireModeOptions)
}

Map getParam(Integer num, String name, Integer size, Integer defaultVal, Map options) {
        Integer val = safeToInt((settings ? settings["configParam${num}"] : null), defaultVal)

        return [num: num, name: name, size: size, value: val, options: options]
}


void sendEventIfNew(String name, value, String type=null, unit="") {
        String desc = "${device.displayName}: ${name} is ${value}${unit}"
        if (device.currentValue(name) != value) {

                logDebug(desc)

                Map evt = [name: name, value: value, descriptionText: desc]

                if (type) {
                        evt.type = type
                }
                if (unit) {
                        evt.unit = unit
                }
                sendEvent(evt)
        }
}


Integer findIntKeyByValue(Map items, value) {
        return safeToInt(items.find { it.value == value }?.key, null)
}


int safeToPercentInt(val, int defaultVal=0) {
        return safeToIntRange(val, defaultVal, 0, 99)
}
```

```
Integer safeToIntRange(val, Integer defaultVal, Integer lowVal, Integer highVal) {
	Integer intVal = safeToInt(val, defaultVal)
	if (intVal > highVal) {
		return highVal
	}
	else if (intVal < lowVal) {
		return lowVal
	}
	else {
		return intVal
	}
}

Integer safeToInt(val, Integer defaultVal=0) {
	if ("${val}"?.isInteger()) {
		return "${val}".toInteger()
	}
	else if ("${val}".isDouble()) {
		return "${val}".toDouble()?.round()
	}
	else {
		return  defaultVal
	}
}

boolean isWX300InSwitchMode() {
	def zwMap = getZwaveInfo()
	return (zwMap && (zwMap.model == "4037"))
}

boolean isDuplicateCommand(lastExecuted, allowedMil) {
	!lastExecuted ? false : (lastExecuted + allowedMil > new Date().time)
}

void logDebug(String msg) {
	if (state.debugLoggingEnabled) {
		log.debug(msg)
	}
}



// *** WORKAROUND CODE ***

void createWorkaroundPreferences() {
	input "ledCmdDelay", "paragraph",
		title:"<br><br>LED Control Delay",
		description: "I used a 500ms delay between each command related to LED control which causes
them to turn on slowly, but they reliably show the correct state afterwards when controlling them together.
Depending on the strength of your mesh and the type of security the device was joined with, it might be
reliable with a much shorter delay or it might not be reliable with the default so you'll need to increase
it."
	createEnumInput("ledCmdDelay", "LED Control Delay", 500, ledCmdDelayOptions)

	input "workarounds", "paragraph",
		title:"<br><br>Child Device Workarounds",
		description: "The Automations app doesn't support custom capabilities yet so the settings
below create child devices that can be used with Automations to control specific functionality.  They also
serve as a backup method of controlling the device if SmartThings makes a breaking change to Custom
Capabilities."

	input "normalModeText", "paragraph",
		title:"<br><br>LED Mode",
		description: "The 'Normal Mode' child device is a Color Switch that requires the custom
handler 'HomeSeer Normal Mode Child'.<br>SWITCH: Sets LED Mode to 'Normal' when ON and 'Status' when OFF.
<br>COLOR PICKER: Changes the Normal LED Color to closest match."
	createEnumInput("normalModeChildEnabled", "Create 'Normal Mode' Child Device?", 0,
createChildOptions)

	input "ledBlinkText", "paragraph",
		title:"<br><br>Status Mode Blink Frequency",
```

```
                description: "The 'Status Mode Blink Frequency' child device is a Dimmer Switch and it
requires the custom handler 'HomeSeer Status LED Blink Frequency Child'.<br>SWITCH: Toggles the Status LED
Blink Frequency between 'Off' and the last value.<br>DIMMER: Changes Status LED Blink Frequency to the
selected value.   (1=100ms, 2=200ms, ..., 100=10s)."
        createEnumInput("statusBlinkChildEnabled", "Create 'Status Mode Blink Frequency' Child Device?", 0,
createChildOptions)

        input "ledColorText", "paragraph",
                title:"<br><br>Status Mode Color",
                description: "The 'Status Mode Color' child device requires the custom handler 'HomeSeer
Status LED Blinking Color Child' and supports Switch, Color Control, and Color Temperature.<br>COLOR PICKER:
Changes the Status LED Color of ALL LEDs to closest match.<br>SWITCH: Toggles the Status LED Color of ALL
LEDs between 'Off' and this child device's last selected color.<br>COLOR TEMPERATURE: Enables Blinking for
All LEDs when temperature is greater than 4500 and Disables Blinking when it's below that."
        createEnumInput("statusColorChildEnabled", "Create 'Status Mode Color' Child Device to Control All
LEDs?", 0, createChildOptions)

        input "individualLedText", "paragraph",
                title: "<br><br>Status Mode Color for Individual LEDs",
                description: "They work the same as the 'Status Mode Color' child except they control a
single LED."

        statusLeds.each { ledNum, ledName ->
                createEnumInput("statusLed${ledName}ChildEnabled", "Create LED ${ledName} Child Device?", 0,
createChildOptions)
        }
}


void initializeChildDevices() {
        def modeChild = initializeChildDevice("normalModeChildEnabled", normalModeChildDTH,
normalModeChildDNI, "Normal Mode")
        modeChild?.sendLedModeEvents(device.currentValue("ledMode"))
        modeChild?.sendColorEvents(device.currentValue("normalLedColor"))

        def blinkChild = initializeChildDevice("statusBlinkChildEnabled", statusBlinkChildDTH,
statusBlinkChildDNI, "Status Mode Blink Frequency")
        blinkChild?.sendBlinkEvents(device.currentValue("statusLedBlinkFrequency"))

        def colorChild = initializeChildDevice("statusColorChildEnabled", statusLedChildDTH,
statusColorChildDNI, "Status Mode Color")
        colorChild?.sendColorEvents(device.currentValue("statusLedColor"))

        statusLeds.each { ledNum, ledName ->
                def child = initializeChildDevice("statusLed${ledName}ChildEnabled", statusLedChildDTH,
getStatusLedChildDNI(ledNum), "Status LED ${ledName}", [ledNum: ledNum, ledName: ledName])

                if (child) {
                        child.sendColorEvents(device.currentValue("statusLed${ledName}Color"))
                }
        }
}

private initializeChildDevice(String settingName, String dthName, String dniSuffix, String nameSuffix, Map
data=[:]) {
        boolean childEnabled = (settings && (safeToInt(settings["${settingName}"]) != 0))

        def child = childDevices?.find { it.deviceNetworkId?.endsWith(dniSuffix) }

        if (childEnabled && !child) {
                return createChildDevice("HomeSeer", dthName, dniSuffix, nameSuffix, data)
        }
        else if (!childEnabled && child) {
                removeChildDevice(child)
                return null
        }
        else {
                return null
        }
}
```

```
private createChildDevice(String namespace, String dthName, String dniSuffix, String nameSuffix, Map data) {
	try {
		logDebug "Creating ${nameSuffix} Child Device"
		return addChildDevice(
			namespace,
			dthName,
			"${device.deviceNetworkId}:${dniSuffix}",
			null,
			[
				completedSetup: true,
				label: "${device.displayName}-${nameSuffix}",
				isComponent: false,
				data: data
			]
		)
	}
	catch (e) {
		log.warn "Unable to create child device for ${nameSuffix}.  Make sure you've installed the
custom DTH '${dthName}'"
		return null
	}
}

void removeChildDevice(child) {
	try {
		log.warn "Removing ${child.displayName} "
		deleteChildDevice(child.deviceNetworkId)
	}
	catch (e) {
		log.warn "Unable to remove ${child?.displayName}"
	}
}


String getStatusLedChildDNI(int led) {
	return statusLedChildDNIPrefix.concat(statusLeds.get(led))
}

private findChild(String dni) {
	return childDevices?.find { it.deviceNetworkId?.endsWith(dni) }
}


void logTrace(String msg) {
	 // log.trace(msg)
}
```