```
/**
 *  Everspring Motion Detector v1.0.4
 *    (Model: HSP02)
 *
 *  Author:
 *    Kevin LaFramboise (krlaframboise)
 *
 *  URL to documentation:
 *
 *
 *  Changelog:
 *
 *    1.0.4 (12/25/2017)
 *        - Implemented ST new color scheme.
 *
 *    1.0.3 (09/10/2017)
 *        - Removed old style fingerprint to eliminate conflicts with other generic sensors.
 *
 *    1.0.2 (04/23/2017)
 *        - SmartThings broke parse method response handling so switched to sendhubaction.
 *
 *    1.0.1 (04/20/2017)
 *      - Added workaround for ST Health Check bug.
 *
 *    1.0 (04/19/2017)
 *      - Initial Release
 *
 *  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

metadata {
```

```
	definition (name:"Everspring Motion Detector", namespace:"krlaframboise", author:
"Kevin LaFramboise") {
		capability "Sensor"
		capability "Battery"
		capability "Motion Sensor"
		capability "Contact Sensor"
		capability "Tamper Alert"
		capability "Refresh"
		capability "Configuration"

		attribute "lastCheckin", "string"

		fingerprint mfr:"0060", prod:"0001", model:"0003"

		// fingerprint deviceId:"0x2001",
inClusters:"0x20,0x30,0x70,0x71,0x72,0x80,0x84,0x85,0x86"
	}

	preferences {
		input "retriggerInterval", "number",
			title: "Re-Trigger Interval (Seconds)\n(When the device detects motion, it
waits for at least 1 minute of inactivity before sending the inactive event.  The default re-trigger
wait time is 180 (3 minutes).",
			range: "5..3600",
			defaultValue: retriggerIntervalSetting,
			displayDuringSetup: true,
			required: false
	defaultValue: 13
		input "motionSensitivity", "number",
			title: "Motion Sensitivity: (1-10)\n(1: Least Sensitive, 10: Most Sensitive)",
			range: "1..10",
			defaultValue: motionSensitivitySetting,
			displayDuringSetup: true,
			required: false
		input "luxThreshold", "number",
			title: "Create Contact Open event when motion is detected and the
percentage of lux level of ambient illumination is below this percentage. (0-100)",
			required: false,
			displayDuringSetup: true,
			range: "0..100",
			defaultValue: 12
		input "checkinInterval", "enum",
			title: "Checkin Interval:",
			defaultValue: checkinIntervalSetting,
```

```
                required: false,
                displayDuringSetup: true,
                options: checkinIntervalOptions.collect { it.name }
            input "reportBatteryEvery", "enum",
                title: "Battery Reporting Interval:",
                defaultValue: batteryReportingIntervalSetting,
                required: false,
                displayDuringSetup: true,
                options: checkinIntervalOptions.collect { it.name }
            input "debugOutput", "bool",
                title: "Enable debug logging?",
                defaultValue: debugOutputSetting,
                required: false,
                displayDuringSetup: true
    }
}

def updated() {
    if (!isDuplicateCommand(state.lastUpdated, 3000)) {
        state.lastUpdated = new Date().time
        logTrace "updated()"
        refresh()
    }
}

def configure() {
    logTrace "configure()"
    def cmds = []

    if (state.pendingConfig == null) {
        logTrace "Waiting 1 second because this is the first time being configured"
        // Give inclusion time to finish.
        sendEvent(name: "contact", value: "closed", displayed: false)
        cmds << "delay 1000"
        cmds << associationSetCmd(2)
    }

    cmds += initializeCheckin()

    cmds += updateConfigVal(basicSetLevelParam, basicSetLevelSetting, false)

    cmds += updateConfigVal(onOffDurationParam, onOffDurationSetting, false)
```

```
	cmds += updateConfigVal(enableDisableGroupsParam, enableDisableGroupsSetting,
false)

	cmds += updateConfigVal(retriggerIntervalParam, retriggerIntervalSetting,
state.pendingConfig)

	cmds += updateConfigVal(motionSensitivityParam, motionSensitivitySetting,
state.pendingConfig)

	cmds += updateConfigVal(luxThresholdParam, luxThresholdSetting ?: 1,
state.pendingConfig) // When set to 0, use 1 because device doesn't support 0.

	if (cmds) {
		return delayBetween(cmds, 500)
	}
	else {
		return []
	}
}

private updateConfigVal(param, val, refreshAll) {
	def result = []
	def configVal = state["configVal${param.num}"]

	if (refreshAll || (configVal != val)) {
		result << configSetCmd(param, val)
		result << configGetCmd(param)
	}
	return result
}

private initializeCheckin() {
	def result = []
	if (state.pendingConfig || state.checkinIntervalMinutes != checkinIntervalSettingMinutes)
{

		state.checkinIntervalMinutes = checkinIntervalSettingMinutes

		result << wakeUpIntervalSetCmd(checkinIntervalSettingMinutes * 60)

		// Set the Health Check interval so that it can be skipped once plus 2 minutes.
		def checkInterval = ((checkinIntervalSettingMinutes * 2 * 60) + (5 * 60))
```

```
		sendEvent(name: "checkInterval", value: checkInterval, displayed: false, data:
[protocol: "zwave", hubHardwareId: device.hub.hardwareID])
	}
	return result
}


def refresh() {
	if (device.currentValue("tamper") == "detected") {
		logDebug "Clearing Tamper Event"
		sendEvent(getTamperEventMap("clear"))
	}
	else {
		logDebug "The settings will be sent to the device the next time it wakes up.  To
apply the changes immediately, take off the back cover of the device wait about 5 seconds and
then put it back on."
		state.pendingConfig = true
	}
}

private configSetCmd(param, val) {
	return zwave.configurationV1.configurationSet(parameterNumber: param.num, size:
param.size, scaledConfigurationValue: val).format()
}

private configGetCmd(param) {
	return zwave.configurationV1.configurationGet(parameterNumber: param.num).format()
}

private wakeUpIntervalSetCmd(val) {
	return zwave.wakeUpV2.wakeUpIntervalSet(seconds:val,
nodeid:zwaveHubNodeId).format()
}

private wakeUpNoMoreInfoCmd() {
	return zwave.wakeUpV2.wakeUpNoMoreInformation().format()
}

private sensorBinaryGetCmd() {
	return zwave.sensorBinaryV1.sensorBinaryGet().format()
}

private batteryGetCmd() {
	return zwave.batteryV1.batteryGet().format()
```

```
}

private basicGetCmd() {
	return zwave.basicV1.basicGet().format()
}

private associationSetCmd(group) {
	return zwave.associationV2.associationSet(groupingIdentifier:group,
nodeId:zwaveHubNodeId).format()
}

// Configuration Parameters
private getConfigParams() {
	return [
		basicSetLevelParam,
		enableDisableGroupsParam,
		motionSensitivityParam,
		retriggerIntervalParam,
		luxThresholdParam,
		onOffDurationParam
	]
}

private getBasicSetLevelParam() {
	// 0x00/0xFF or 1-99 (level) default 0xFF
	return createConfigParamMap(1, "Basic Set Level", 1)
}

private getEnableDisableGroupsParam() {
	// 0-3 (0 both disabled, 1: both enabled, 2: group1 only enable, 3: group2 only enabled)
	return createConfigParamMap(2, "Enable/Disable Groups", 1)
}

private getMotionSensitivityParam() {
	// 1-10 default 6
	return createConfigParamMap(3, "Motion Sensitivity", 1)
}

private getRetriggerIntervalParam() {
	// 5-3600 Seconds default 180
	return createConfigParamMap(4, "Re-trigger Interval", 1)
}

private getLuxThresholdParam() {
```

```
	// 1-100 % default 10
	return createConfigParamMap(5, "Lux Threshold", 1)
}

private getOnOffDurationParam() {
	// 5-3600 seconds default 15
	return createConfigParamMap(6, "On/Off Delay", 1)
}

private createConfigParamMap(num, name, size) {
	return [num: num, name: name, size: size]
}

private getCommandClassVersions() {
	[
		0x20: 1,  // Basic
		0x30: 1,  // Sensor Binary
		0x70: 1,  // Configuration
		0x71: 3,  // Notification v4
		0x72: 1,  // ManufacturerSpecific
		0x80: 1,  // Battery
		0x84: 2,  // WakeUp
		0x85: 2,  // Association
		0x86: 1,  // Version (2)
	]
}

def parse(String description) {
	def result = []

	sendEvent(name: "lastCheckin", value: convertToLocalTimeString(new Date()),
displayed: false, isStateChange: true)

	def cmd = zwave.parse(description, commandClassVersions)
	if (cmd) {
		result += zwaveEvent(cmd)
	}
	else {
		logDebug "Unknown Description: $desc"
	}
	return result
}

def zwaveEvent(hubitat.zwave.commands.wakeupv2.WakeUpNotification cmd)
```

```
{
	logTrace "WakeUpNotification"
	def result = []

	if (state.pendingConfig != false) {
		result += configure()
	}
	else if (canReportBattery()) {
		result << batteryGetCmd()
	}

	if (result) {
		result << "delay 2000"
	}
	result << wakeUpNoMoreInfoCmd()
	return sendResponse(result)
}

private sendResponse(cmds) {
	def actions = []
	cmds?.each { cmd ->
		actions << new hubitat.device.HubAction(cmd)
	}
   sendHubCommand(hubitat.device.HubMultiAction)
	return []
}

private canReportBattery() {
	def reportEveryMS = (batteryReportingIntervalSettingMinutes * 60 * 1000)

	return (!state.lastBatteryReport || ((new Date().time) - state.lastBatteryReport >
reportEveryMS))
}

   def zwaveEvent(hubitat.zwave.commands.sensorbinaryv1.SensorBinaryReport cmd) {

    log.info "The Sensor Cmd Value is ${cmd}"
	def val = (cmd.sensorValue == 0xFF) ? "active" : "inactive"
	logDebug "Motion ${val.capitalize()}"



	def result = []
```

```
	result << createEvent(name: "motion", value: "${val}",displayed: true, isStateChange:
true)
	return result

}

def zwaveEvent(hubitat.zwave.commands.basicv1.BasicSet cmd) {
	log.info "The BasicSet Cmd Value is ${cmd}"
   def result = []
   logTrace "BasicSet: ${cmd}"
	// Only handle open event if the lux threshold setting > 0
	if (cmd.value || luxThresholdSetting) {
	   log.info "The BasicSet After If Cmd Value is ${cmd}"
		def val = cmd.value ? "open" : "closed"
		def desc = cmd.value ? "Motion detected and lux is below
${luxThresholdSetting}%" : null

		if (desc) {
			logDebug "$desc"
		}

		result << createEvent(name: "contact",
			value: val,
			isStateChange: true,
			displayed: (desc != null),
			descriptionText: desc)
	}
	return result
}

def zwaveEvent(hubitat.zwave.commands.configurationv1.ConfigurationReport cmd) {
	// logTrace "ConfigurationReport: ${cmd}"
   info.log "The config report value is: ${cmd}"
	def val = (cmd.scaledConfigurationValue == -1 ? 255 : cmd.scaledConfigurationValue)
	def result = []

	def configParam = configParams.find { param ->
		param.num == cmd.parameterNumber
	}

	if (configParam) {
		logDebug "${configParam.name} = ${val}"
		state["configVal${cmd.parameterNumber}"] = val
	}
```

```
	else {
		logDebug "Parameter ${cmd.parameterNumber} = ${val}"
	}
	state.pendingConfig = false
	return result
}

def zwaveEvent(hubitat.zwave.commands.batteryv1.BatteryReport cmd) {
	logTrace "BatteryReport: $cmd"
	def result = []
	result << createEvent(getBatteryEventMap(cmd.batteryLevel))
	return result
}



def zwaveEvent(hubitat.zwave.commands.notificationv3.NotificationReport cmd) {
	// logTrace "NotificationReport: $cmd"
	def result = []
	def msg
	if (cmd.v1AlarmLevel == 1) {
		switch (cmd.v1AlarmType) {
			case 1:
				msg = "Low Battery Alarm"
				result << createEvent(getBatteryEventMap(0xFF))
				break
			case 2:
				msg = "Power Applied"
				break
			case 3:
				msg = "Tamper Detected"
				result << createEvent(getTamperEventMap("detected"))
				break
			default:
				msg = ""
		}
	}
	if (!msg) {
		msg = "Unknown Alarm: ${cmd}"
	}
	logDebug "$msg"
	return result
}
```

```
def getBatteryEventMap(val) {
	state.lastBatteryReport = new Date().time
	logDebug "Battery ${val}%"

	val = (val == 0xFF ? 1 : val)
	if (val > 100) {
		val = 100
	}

	def isNew = (device.currentValue("battery") != val || val == 1)

	return [
		name: "battery",
		value: val,
		unit: "%",
		display: isNew,
		isStateChange: isNew
	]
}

def getTamperEventMap(val) {
	[
		name: "tamper",
		value: val,
		isStateChange: true,
		displayed: (val == "detected"),
		descriptionText: "Tamper is $val"
	]
}

def zwaveEvent(hubitat.zwave.Command cmd) {
	logDebug "Unknown Command: $cmd"
	return []
}


// Settings
private getCheckinIntervalSettingMinutes() {
	return convertOptionSettingToInt(checkinIntervalOptions, checkinIntervalSetting) ?: 60
}
private getCheckinIntervalSetting() {
	return settings?.checkinInterval ?: findDefaultOptionName(checkinIntervalOptions)
}
private getBatteryReportingIntervalSettingMinutes() {
```

```
	return convertOptionSettingToInt(checkinIntervalOptions,
batteryReportingIntervalSetting) ?: 720
}
private getBatteryReportingIntervalSetting() {
	return settings?.reportBatteryEvery ?: findDefaultOptionName(checkinIntervalOptions)
}
private getDebugOutputSetting() {
	return (settings?.debugOutput		!= false)
}
private getRetriggerIntervalSetting() {
	return safeToInt(settings?.retriggerInterval, 13)
}
private getMotionSensitivitySetting() {
	return safeToInt(settings?.motionSensitivity, 7)
}
private getLuxThresholdSetting() {
	return safeToInt(settings?.luxThreshold, 12)
}
private getBasicSetLevelSetting() {
	return safeToInt(settings?.BasicLevel, 90)
}
private getOnOffDurationSetting() {
	return safeToInt(settings?.OnOffDurationSetting, 7)
}
private getEnableDisableGroupsSetting() {
	return safeToInt(settings?.EnableDisableGroups, 1)
}

private getCheckinIntervalOptions() {
	[
		[name: "10 Minutes", value: 10],
		[name: "15 Minutes", value: 15],
		[name: "30 Minutes", value: 30],
		[name: "1 Hour", value: 60],
		[name: "2 Hours", value: 120],
		[name: "3 Hours", value: 180],
		[name: formatDefaultOptionName("6 Hours"), value: 360],
		[name: "9 Hours", value: 540],
		[name: "12 Hours", value: 720],
		[name: "18 Hours", value: 1080],
		[name: "24 Hours", value: 1440]
	]
}
```

```
private convertOptionSettingToInt(options, settingVal) {
	return safeToInt(options?.find { "${settingVal}" == it.name }?.value, 0)
}

private formatDefaultOptionName(val) {
	return "${val}${defaultOptionSuffix}"
}

private findDefaultOptionName(options) {
	def option = options?.find { it.name?.contains("${defaultOptionSuffix}") }
	return option?.name ?: ""
}

private getDefaultOptionSuffix() {
	return "   (Default)"
}

private safeToInt(val, defaultVal=-1) {
	return "${val}"?.isInteger() ? "${val}".toInteger() : defaultVal
}

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

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

private logDebug(msg) {
	if (debugOutputSetting) {
		log.debug "$msg"
	}
}

private logTrace(msg) {
	// log.trace "$msg"
}
```