TellStick ZNet Documentation

Release v1.0.11-115-g7169e47

Telldus Technologies

May 22, 2017

Contents

TellStick ZNet allows developers to build own plugins and scripts run the device to extend the functionality with features not yet supported by Telldus.

It is also possible to alter the behaviour on how TellStick ZNet should interpret signals and messages from devices.

CHAPTER 1

Intro

TellStick ZNet offers two ways of integrating custom scripts. They can be written in either Python or Lua. The difference is outlined below.

Python

Python plugins are only available for TellStick ZNet Pro. Python plugins cannot be run on TellStick ZNet Lite. Python plugins offers the most flexible solution since full access to the service is exposed. This also makes it fragile since Python plugins can affect the service negative.

Lua

Lua code is available on both TellStick ZNet Pro and TellStick ZNet Lite. Lua code runs in a sandbox and has only limited access to the system.

To create a Lua script you need to access the local web server in TellStick ZNet. Browse to: http://{[}ipaddress{]}/lua to access the editor.

Lua codes works by signals from the server triggers the execution.

CHAPTER 2

Lua

class `lua.`**`List`**
This object is available in Lua code as the object `list` and is a helper class for working with Python lists.

`len`(*object*) → integer
Return the number of items of a sequence or collection.

static `new`(**args*)
Create a new Python list for use with Python code.

Example: local pythonList = list.new(1, 2, 3, 4)

static `slice`(*collection*, *start=None*, *end=None*, *step=None*)
Retrieve the start, stop and step indices from the slice object *list*. Treats indices greater than length as errors.

This can be used for slicing python lists (e.g. l[0:10:2]).

`lua.`**`sleep`**(*ms*)
Delay for a specified amount of time.

ms The number of milliseconds to sleep.

Example: Real wind

A thermometer measures the actual temperature but it is not the same as the perceived temperature. To get perceived temperature you must also take the wind into account. If TellStick ZNet has an anemometer this can be used to calculate the perceived temperature.

The script below calculates this and gives the anemometer a thermometer value.

Source of the algorithm: http://www.smhi.se/kunskapsbanken/meteorologi/vindens-kyleffekt-1.259

```
-- EDIT THESE

local windSensor = 287
```

```
local tempSensor = 297

-- DO NOT EDIT BELOW THIS LINE

local tempValue = deviceManager:device(tempSensor).sensorValue(1, 0)
local windValue = deviceManager:device(windSensor).sensorValue(64, 0)

function calculate()
  if tempValue == nil or windValue == nil then
    return
  end
  local w = math.pow(windValue, 0.16)
  local v = 13.12 + 0.6215*tempValue - 13.956*w + 0.48669*tempValue*w
  v = math.floor(v * 10 + 0.5) / 10
  local windDevice = deviceManager:device(windSensor)
  windDevice:setSensorValue(1, v, 0)
end

function onSensorValueUpdated(device, valueType, value, scale)
  if device:id() == windSensor and valueType == 64 and scale == 0 then
    windValue = value
    calculate()
  elseif device:id() == tempSensor and valueType == 1 and scale == 0 then
    tempValue = value
    calculate()
  end
end
```

Example: Zipato RFID

Telldus does not support the RFID reader from Zipato.

http://www.zipato.com/default.aspx?id=24&pid=88&page=1&grupe=0,2_15,3_37,0

It can be used any way with some Lua code.

```
-- Change these
local zipatoNodeId = 892

local tags = {}
-- Add tags below

-- Example code from a tag
-- tags[1] = {device=881, code={143, 188, 119, 84, 42, 0, 1, 4, 0, 0}};
-- Code for entering 1-2-3-4 on the keyboard
-- tags[2] = {device=813, code={49, 50, 51, 52, 0, 0, 0, 0, 0, 0}};

-- Do not change below

COMMAND_CLASS_USER_CODE = 0x63
USER_CODE_SET = 0x01
USER_CODE_REPORT = 0x03
COMMAND_CLASS_ALARM = 0x71
ALARM_REPORT = 0x05

local zipatoNode = deviceManager:device(zipatoNodeId):zwaveNode()
```

```
function compareTags(tag1, tag2)
	for index, item in python.enumerate(tag2) do
		if item ~= tag1[index+1] then
			return false
		end
	end
	return true
end

function configureTag(index)
	local data = list.new(index, 1)
	for key,code in pairs(tags[index]['code']) do
		data.append(code)
	end
	zipatoNode:sendMsg(COMMAND_CLASS_USER_CODE, USER_CODE_SET, data)
	print("A new tag was configured in the Zipato.")
	print("This will be sent the next time the reader is awake")
end

function checkNewTag(code)
	-- New tag received. Check if it should be configured?
	for key,tag in pairs(tags) do
		if compareTags(tag['code'], code) then
			configureTag(key)
			return
		end
	end
	-- Not yet configured. Must be configured first.
	print("New unknown tag received. Add this to the codes if this should be␣
↪recognized")
	print("Tag data is %s", code)
end

function handleAlarm(data)
	if list.len(data) < 8 then
		return
	end

	local event = data[5]
	local tag = data[7]
	local device = deviceManager:device(tags[tag]['device'])
	if device == nil then
		print("Device not found")
	end
	if event == 5 then
		print("Away, tag %s", tag)
		zipatoNode:sendMsg(0x20, 0x01, list.new(0xFF))
		device:command("turnoff", nil, "RFID")
	elseif event == 6 then
		print("Home, tag %s", tag)
		device:command("turnon", nil, "RFID")
	end
end

function onZwaveMessageReceived(device, flags, cmdClass, cmd, data)
	if device:id() ~= zipatoNodeId then
		return
```

```
	end
	if cmdClass == COMMAND_CLASS_ALARM and cmd == ALARM_REPORT then
		handleAlarm(data)
		return
	end
	if cmdClass ~= COMMAND_CLASS_USER_CODE or cmd ~= USER_CODE_REPORT then
		return
	end
	local identifier = data[0]
	local status = data[1]
	if identifier == 0 and status == 0 then
		checkNewTag(list.slice(data,2))
		return
	end
end

-- This command clears all configured codes in the reader
-- zipatoNode:sendMsg(COMMAND_CLASS_USER_CODE, USER_CODE_SET, list.new(0, 0))
```

CHAPTER 3

Python plugins

Python plugins offers the most flexible way of extending the functionality of TellStick. To get started a development environment should first be setup on a computer running Linux or macOS. Windows is not supported at the moment.

Installation

Check out and follow the instructions on getting the server software running on a computer here: https://github.com/telldus/tellstick-server

After installation the tellstick server is installed without any plugins. For development the lua-plugin is a recommended plugin to install. Install it with:

```
./tellstick.sh install lua
```

Telldus own plugins are open source and can be used as a base for new plugins. These can be found here: https://github.com/telldus/tellstick-server-plugins

This guide will describe the example plugin found here: https://github.com/telldus/tellstick-server-plugins/tree/master/templates/device

The plugin adds one dummy device to the system.

During the development it is recommended to install it within the server software. This way the software will restart itself whenever a file has changed. To install it use the tellstick command `install`:

```
./tellstick.sh install [path-to-plugin]
```

Replace *[path-to-plugin]* with the path to the plugin root folder.

Anatomy of a plugin

TellStick plugins are packaged as python eggs combined in a zip file. The eggs are signed with a pgp signature.

The metadata for a plugin is described in the file setup.py. This is a standard setuptools file with a couple custom configurations added.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

try:
	from setuptools import setup
except ImportError:
	from distutils.core import setup

setup(
	name='Dummy device',
	version='1.0',
	author='Alice',
	author_email='alice@wonderland.lit',
	category='appliances',
	color='#2c3e50',
	description='Dummy template for implementings device plugins',
	icon='dummy.png',
	long_description="""
		This plugin is used as a template when creating plugins that support␣
↪new device types.
	""",
	packages=['dummy'],
	package_dir = {'':'src'},
	entry_points={ \
		'telldus.startup': ['c = dummy:Dummy [cREQ]']
	},
	extras_require = dict(cREQ = 'Base>=0.1\nTelldus>=0.1'),
)
```

Most of the fields can be found in the setuptools documentation.

author The name of the developer of the plugin. This name must match the pgp signin certificate.

author_email The email of the developer of the plugin. This must match the pgp singning certificate.

category This must be one of:

- security
- weather
- climate
- energy
- appliances
- multimedia
- notifications

color A color used in plugin selection user interface in the format #000000.

compatible_platforms Reserved for future use.

description A short description of the plugins. This should only be one line.

entry_points TellStick plugins can be loaded by one of two entry points.

telldus.startup This plugin will auto load on startup. Use this when it is important that the plugin is always loaded.

telldus.plugins This plugin will be loaded on-demand. This speeds up loading times and keep the memory footprint to a minimum.

icon Filename of icon in size 96x96.

long_description A long description describing the plugin. Markdown can be used.

name The name of the plugin.

packages A list of python packages included in the plugin. This should match the folder structure of the files. Please see setuptools documentation for more information.

required_features Reserved for future use.

version The version of the plugin.

Building a deployable plugin

Once development is finished it's time to package the code into a deployable package. Before this command a working pgp code signing key must be setup on the computer. The name and email must match the metadata `author` and `author_email` specified in setup.py.

Setting up a key

You can safely skip this step if you already have a pgp-key setup on your computer.

```
gpg --gen-key
```

This will take you through a few questions that will configure your keys.

```
Please select what kind of key you want: (1) RSA and RSA (default)
What keysize do you want? 4096
Key is valid for? 0
Is this correct? y
Real name: Enter the same name as in setup.py
Email address: Enter the same email as in setup.py
Comment:
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
Enter passphrase: Enter a secure passphrase here (upper & lower case, digits, symbols)
```

Build the plugin

To build the package use the `build-plugin` command to tellstick.sh

```
./tellstick.sh build-plugin [path-to-plugin]
```

Replace *[path-to-plugin]* with the path to the plugin root folder. During building the plugin will be signed using your pgp key and if a passphrase has been setup you will be asked for your password.

This will build a .zip file ready to be uploaded to a TellStick.

CHAPTER 4

Local API

TellStick ZNet has a local REST interface to integrate into third party applications not running on the TellStick ZNet itself

A list of all available functions can be browsed on the device itself. Browse to: http://{[}ipaddress{]}/api to list the functions.

Authentication

Before making any REST calls to TellStick ZNet the application must request a token that the user has authenticated.

Step 1 - Request a request token

Request a request token by performing a PUT call to the endpoint /api/token. You need to supply the application name as a parameter "app"

```
$ curl -i -d app="Example app" -X PUT http://0.0.0.0/api/token
HTTP/1.1 200 OK
Date: Fri, 15 Jan 2016 13:33:54 GMT
Content-Length: 148
Content-Type: text/html;charset=utf-8
Server: CherryPy/3.8.0

{
  "authUrl": "http://0.0.0.0/api/authorize?token=0996b21ee3f74d2b99568d8207a8add9",
  "token": "0996b21ee3f74d2b99568d8207a8add9"
}
```

Step 2 - Authenticate the app

Redirect the user to the url returned in step 1 to let him/her authenticate the app.

Step 3 - Exchange the request token for an access token

When the user has authenticated the request token in step 2 the application needs to exchange this for an access token. The access token can be used in the API requests. To exchange the request token for an access token perform a GET call to the same endpoint in step 1. Supply the request token as the parameter "token".

```
$ curl -i -X GET http://0.0.0.0/api/token?token=0996b21ee3f74d2b99568d8207a8add9
HTTP/1.1 200 OK
Date: Fri, 15 Jan 2016 13:39:22 GMT
Content-Length: 230
Content-Type: text/html;charset=utf-8
Server: CherryPy/3.8.0

{
  "allowRenew": true,
  "expires": 1452951562,
  "token":
↪"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDUyOTUxNTYyfQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.HeqoFM6-K5IuQa08Zr9HM9V2TKGRI9VxXlgdsutP7sg"
}
```

If the returned data contains allowRenew=true then the token was authorized to renew its expiration itself without letting the user authorize the app again. The application must renew the token before it expires or else the application must start the autorization again from step 1.

If allowRenew is not set to true it is not possible for the app to renew the token and it will always expire after the time set in the parameter "expires".

Step 4 - Making a request

To make a requst to a TellStick ZNet API endpoint the token in step 3 must be supplied as a bearer token in the header. This is an example requesting a list of devices:

```
$ curl -i -X GET http://0.0.0.0/api/devices/list?supportedMethods=3 -H
↪"Authorization: Bearer␣
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDUyOTUxNTYyfQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.HeqoFM6-K5IuQa08Zr9HM9V2TKGRI9VxXlgdsutP7sg"
HTTP/1.1 200 OK
Date: Tue, 19 Jan 2016 10:21:29 GMT
Content-Type: Content-Type: application/json; charset=utf-8
Server: CherryPy/3.7.0

{
  "device": [
    {
      "id": 1,
      "methods": 3,
      "name": "Example device",
      "state": 2,
      "statevalue": "",
      "type": "device"
    }
  ]
}
```

Refreshing a token

If the user allowed the application to renew the token in steg 2 it can be renewed by the calling application. The token must be refreshed before it expires. If the token has expired the authentication must be restarted from step 1 again.

```
$ curl -i -X GET http://0.0.0.0/api/refreshToken -H "Authorization: Bearer␣
↪eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDUyOTUxNTYyfQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.HeqoFM6-K5IuQa08Zr9HM9V2TKGRI9VxXlgdsutP7sg"
HTTP/1.1 200 OK
Date: Tue, 19 Jan 2016 10:21:29 GMT
Content-Type: Content-Type: application/json; charset=utf-8
Server: CherryPy/3.7.0

{
  "expires": 1455295348,
  "token":
↪"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCIsImF1ZCI6IkV4YW1wbGUgYXBwIiwiZXhwIjoxNDU1Mjk1MzQ4fQ.
↪eyJyZW5ldyI6dHJ1ZSwidHRsIjo4NjQwMH0.M4il4_2SqJwlCjmuXlU5DS6h-gX7493Tnk9oBJXbgPw"
}
```

The new token returned must be used from now on and the old be discarded.

CHAPTER 5

API reference

Module: base

Classes in the base module are only accessible from Python applications.

class base.**Application**(*run=True*)

This is the main application object in the server. There can only be once instance of this object. The default constructor returns the instance of this object.

registerScheduledTask(*fn*, *seconds=0*, *minutes=0*, *hours=0*, *days=0*, *runAtOnce=False*, *strictInterval=False*, *args=None*, *kwargs=None*)

Register a semi regular scheduled task to run at a predefined interval. All calls will be made by the main thread.

fn The function to be called.

seconds The interval in seconds. Optional.

minutes The interval in minutes. Optional.

hours The interval in hours. Optional.

days The interval in days. Optional.

runAtOnce If the function should be called right away or wait one interval?

strictInterval Set this to True if the interval should be strict. That means if the interval is set to 60 seconds and it was run ater 65 seconds the next run will be in 55 seconds.

args Any args to be supplied to the function. Supplied as *args.

kwargs Any keyworded args to be supplied to the function. Supplied as **kwargs.

Note: The interval in which this task is run is not exact and can be delayed one minute depending on the server load.

Note: Calls to this method are threadsafe.

queue (*fn*, **args*, ***kwargs*)
Queue a function to be executed later. All tasks in this queue will be run by the main thread. This is a thread safe function and can safely be used to syncronize with the main thread

registerShutdown (*fn*)
Register shutdown method. The method fn will be called the the server shuts down. Use this to clean up resources on shutdown.

static signal (*msg*, **args*, ***kwargs*)
Send a global signal to registered slots. It is not recommended to call this method directly but instead use the signal decorator

class base.**mainthread** (*f*)

@**mainthread**

This decorator forces a method to be run in the main thread regardless of which thread calls the method.

class base.**ISignalObserver**
Bases: base.Plugin.IInterface

Implement this IInterface to recieve signals using the decorator @slot()

static SignalManager.**slot** (*message=''*)

@**slot**

This is a decorator for receiveing signals. The class must implement ISignalObserver

Args:

message This is the signal name to receive

Module: scheduler

class scheduler.base.**Scheduler**

addMaintenanceJob (*nextRunTime*, *timeoutCallback*, *recurrence=0*)
nextRunTime - GMT timestamp, timeoutCallback - the method to run, recurrence - when to repeat it, in seconds Returns: An id for the newly added job (for removal and whatnot) Note, if the next nextRunTime needs to be calculated, it's better to do that in the callback-method, and add a new job from there, instead of using "recurrence"

calculateJobs (*jobs*)
Calculate nextRunTime for all jobs in the supplied list, order it and assign it to self.jobs

calculateNextRunTime (*job*)
Calculates nextRunTime for a job, depending on time, weekday and timezone.

calculateRunTimeForDay (*runDate*, *job*)
Calculates and returns a timestamp for when this job should be run next. Takes timezone into consideration.

checkNewlyLoadedJob (*job*)
Checks if any of the jobs (local or initially loaded) should be running right now

fetchLocalJobs()
Fetch local jobs from settings

receiveJobsFromServer(*msg*)
Receive list of jobs from server, saves to settings and calculate nextRunTimes

receiveOneJobFromServer(*msg*)
Receive one job from server, add or edit, save to settings and calculate nextRunTime

runJob(**args*, ***kwargs*)
None

Note: Calls to this method are threadsafe.

---

runMaintenanceJob(**args*, ***kwargs*)
None

Note: Calls to this method are threadsafe.

---

successfulJobRun(*jobId*, *state*, *stateValue*)
Called when job run was considered successful (acked by Z-Wave or sent away from 433), repeats should still be run

Module: telldus

class telldus.**DeviceManager**
The devicemanager holds and manages all the devices in the server

addDevice(**args*, ***kwargs*)
Call this function to register a new device to the device manager.

Note: The `localId()` function in the device must return a unique id for the transport type returned by `typeString()`

Note: Calls to this method are threadsafe.

---

device(*deviceId*)
Retrieves a device.

Returns: the device specified by *deviceId* or None of no device was found

finishedLoading(**args*, ***kwargs*)
Finished loading all devices of this type. If there are any unconfirmed, these should be deleted

---

Note: Calls to this method are threadsafe.

---

removeDevice(**args*, ***kwargs*)
Removes a device.

> **Warning:** This function may only be called by the module supplying the device since removing of a device may be transport specific.

> **Note:** Calls to this method are threadsafe.

retrieveDevices(*deviceType=None*)
Retrieve a list of devices.

Args:

deviceType If this parameter is set only devices with this type is returned

Returns: Returns a list of devices

class telldus.**IDeviceChange**
Bases: base.Plugin.IInterface

Implement this IInterface to recieve notifications on device changes

deviceAdded(*device*)
This method is called when a device is added

deviceConfirmed(*device*)
This method is called when a device is confirmed on the network, not only loaded from storage (not applicable to all device types)

deviceRemoved(*deviceId*)
This method is called when a device is removed

sensorValueUpdated(*device*, *valueType*, *value*, *scale*)
This method is called when a new sensor value is received from a sensor

stateChanged(*device*, *state*, *statevalue*)
Called when the state of a device changed

class telldus.**Device**
A base class for a device. Any plugin adding devices must subclass this class.

BAROMETRIC_PRESSURE = 2048
Sensor type flag for barometric pressure

BELL = 4
Device flag for devices supporting the bell method.

DEW_POINT = 1024
Sensor type flag for dew point

DIM = 16
Device flag for devices supporting the dim method.

DOWN = 256
Device flag for devices supporting the down method.

EXECUTE = 64
Device flag for devices supporting the execute method.

HUMIDITY = 2
Sensor type flag for humidity

LEARN = 32
Device flag for devices supporting the learn method.

LUMINANCE = 512
Sensor type flag for luminance

RAINRATE = 4
Sensor type flag for rain rate

RAINTOTAL = 8
Sensor type flag for rain total

RGBW = 1024
Device flag for devices supporting the rgbw method.

STOP = 512
Device flag for devices supporting the stop method.

TEMPERATURE = 1
Sensor type flag for temperature

THERMOSTAT = 2048
Device flag for devices supporting thermostat methods.

TOGGLE = 8
Device flag for devices supporting the toggle method.

TURNOFF = 2
Device flag for devices supporting the off method.

TURNON = 1
Device flag for devices supporting the on method.

UNKNOWN = 0
Sensor type flag for an unknown type

UP = 128
Device flag for devices supporting the up method.

UV = 128
Sensor type flag for uv

WATT = 256
Sensor type flag for watt

WINDAVERAGE = 32
Sensor type flag for wind average

WINDDIRECTION = 16
Sensor type flag for wind direction

WINDGUST = 64
Sensor type flag for wind gust

_command(*action*, *value*, *success*, *failure*, ***kwargs*)
Reimplement this method to execute an action to this device.

battery()
Returns the current battery value

command(*action*, *value=None*, *origin=None*, *success=None*, *failure=None*, *callbackArgs=[]*, *ignore=None*)
This method executes a method with the device. This method must not be subclassed. Please subclass `_command()` instead.

param action description

return return description

Here below is the results of the `Device.methods()` docstring.

isDevice()
Return True if this is a device.

isSensor()
Return True if this is a sensor.

localId()
This method must be reimplemented in the subclass. Return a unique id for this device type.

static methodStrToInt(*method*)
Convenience method to convert method string to constants.

Example: "turnon" => Device.TURNON

methods()
Return the methods this supports. This is an or-ed in of device method flags.

Example: return Device.TURNON | Device.TURNOFF

sensorValue(*valueType*, *scale*)
Returns a sensor value of a the specified valueType and scale. Returns None is no such value exists

sensorValues()
Returns a list of all sensor values this device has received.

state()
Returns a tuple of the device state and state value

Example: state, stateValue = device.state()

typeString()
Must be reimplemented by subclass. Return the type (transport) of this device. All devices from a plugin must have the same type.

class telldus.**Sensor**
Bases: telldus.Device.Device

A convenience class for sensors.

Index

Symbols

A

B

C

D

E

F

H

I

L

M

N

Q

R

S

T

U

W