tellcore-py Documentation

Release 1.1.3

Erik Johansson

Sep 01, 2017

Contents

tellcore-py is a Python wrapper for Telldus' home automation library Telldus Core.

Contents:

CHAPTER 1

tellcore.telldus (module)

This module provides a high level Python interface to Telldus' C API.

Since most functions are Python-ified wrappers around the C API, please also refer to the Telldus Core documentation for further information.

The classes in this module all use the *library.Library* class under the hood, and thus also has e.g. automatic memory manangement, exceptions (*library.TelldusError*) and transparent string conversion with full Python 3 support.

Some example programs are included in the documentation to help understand how to use the different classes:

- *tellcore_tool (example)*
- *Event handling (example)*

TelldusCore

class tellcore.telldus.**TelldusCore**(*library_path=None*, *callback_dispatcher=None*)

The main class for tellcore-py.

Has methods for adding devices and for enumerating controllers, devices and sensors. Also handles callbacks; both registration and making sure the callbacks are processed in the main thread instead of the callback thread.

__init__(*library_path=None*, *callback_dispatcher=None*)

Create a new TelldusCore instance.

Only one instance should be used per program.

Parameters

- **library_path** (*str*) – Passed to the *library.Library* constructor.
- **callback_dispatcher** (*str*) – An instance implementing the *library.BaseCallbackDispatcher* interface (e.g. *QueuedCallbackDispatcher* or *AsyncioCallbackDispatcher*) A callback dispatcher must be provided if callbacks are to be used.

callback_dispatcher
The callback dispatcher used. Set when constructing the instance and should not be changed.

add_device(*name*, *protocol*, *model=None*, ***parameters*)
Add a new device.

Returns a *Device* or *DeviceGroup* instance.

add_group(*name*, *devices*)
Add a new device group.

Returns a *DeviceGroup* instance.

connect_controller(*vid*, *pid*, *serial*)
Connect a controller.

controllers()
Return all known controllers.

Requires Telldus core library version >= 2.1.2.

Returns list of *Controller* instances.

devices()
Return all known devices.

Returns list of *Device* or *DeviceGroup* instances.

disconnect_controller(*vid*, *pid*, *serial*)
Disconnect a controller.

register_controller_event(*callback*)
Register a new controller event callback handler.

See *Event handling (example)* for more information.

Returns the callback id

register_device_change_event(*callback*)
Register a new device change event callback handler.

See *Event handling (example)* for more information.

Returns the callback id

register_device_event(*callback*)
Register a new device event callback handler.

See *Event handling (example)* for more information.

Returns the callback id

register_raw_device_event(*callback*)
Register a new raw device event callback handler.

See *Event handling (example)* for more information.

Returns the callback id

register_sensor_event(*callback*)
Register a new sensor event callback handler.

See *Event handling (example)* for more information.

Returns the callback id

send_raw_command(*command*, *reserved=0*)
Send a raw command.

sensors()
Return all known sensors.

Returns list of *Sensor* instances.

unregister_callback(*cid*)
Unregister a callback handler.

Parameters **id** (*int*) – the callback id as returned from one of the register_*_event methods.

DeviceFactory

tellcore.telldus.**DeviceFactory**(*id*, *lib=None*)
Create the correct device instance based on device type and return it.

Returns a *Device* or *DeviceGroup* instance.

Device

class tellcore.telldus.**Device**(*id*, *lib=None*)
A device that can be controlled by Telldus Core.

Can be instantiated directly if the id is known, but using *DeviceFactory()* is recommended. Otherwise returned from *TelldusCore.add_device()* or *TelldusCore.devices()*.

name
The name of the device (read/write).

protocol
The protocol used for the device (read/write).

model
The device's model (read/write).

type
The device type (read only). One of the device type constants from *tellcore.constants*.

bell()
Send "bell" command to the device.

dim(*level*)
Dim the device.

Parameters **level** (*int*) – The level to dim to in the range [0, 255].

down()
Send "down" command to the device.

execute()
Send "execute" command to the device.

get_parameter(*name*)
Get a parameter.

last_sent_command(*methods_supported*)
Get the last sent (or seen) command.

last_sent_value()
Get the last sent (or seen) value.

learn()
Send "learn" command to the device.

methods(*methods_supported*)
Query the device for supported methods.

parameters()
Get dict with all set parameters.

remove()
Remove the device from Telldus Core.

set_parameter(*name*, *value*)
Set a parameter.

stop()
Send "stop" command to the device.

turn_off()
Turn off the device.

turn_on()
Turn on the device.

up()
Send "up" command to the device.

DeviceGroup

class tellcore.telldus.**DeviceGroup**(*id*, *lib=None*)
Extends *Device* with methods for managing a group

E.g. when a group is turned on, all devices in that group are turned on.

add_to_group(*devices*)
Add device(s) to the group.

devices_in_group()
Fetch list of devices in group.

remove_from_group(*devices*)
Remove device(s) from the group.

Sensor

class tellcore.telldus.**Sensor**(*protocol*, *model*, *id*, *datatypes*, *lib=None*)
Represents a sensor.

Returned from *TelldusCore.sensors()*

has_value(*datatype*)
Return True if the sensor supports the given data type.

sensor.has_value(TELLSTICK_TEMPERATURE) is identical to calling sensor.has_temperature().

value(*datatype*)
Return the *SensorValue* for the given data type.

sensor.value(TELLSTICK_TEMPERATURE) is identical to calling sensor.temperature().

SensorValue

class tellcore.telldus.**SensorValue**(*datatype*, *value*, *timestamp*)
Represents a single sensor value.

Returned from *Sensor.value()*.

datatype
One of the sensor value type constants from *tellcore.constants*.

value
The sensor value.

timestamp
The time the sensor value was registered (in seconds since epoch).

Controller

class tellcore.telldus.**Controller**(*id*, *type*, *lib=None*)
Represents a Telldus controller.

Returned from *TelldusCore.controllers()*

id

type
One of the controller type constants from *tellcore.constants*.

QueuedCallbackDispatcher

class tellcore.telldus.**QueuedCallbackDispatcher**
The default callback dispatcher used by *TelldusCore*.

Queues callbacks that arrive from Telldus Core. Then calls them in the main thread (or more precise: the thread calling *process_callback()*) instead of the callback thread used by Telldus Core. This way the application using *TelldusCore* don't have to do any thread synchronization. Only make sure *process_pending_callbacks()* is called regularly.

process_callback(*block=True*)
Dispatch a single callback in the current thread.

Parameters **block** (*boolean*) – If True, blocks waiting for a callback to come.

Returns True if a callback was processed; otherwise False.

process_pending_callbacks()
Dispatch all pending callbacks in the current thread.

AsyncioCallbackDispatcher

class tellcore.telldus.**AsyncioCallbackDispatcher**(*loop*)

Dispatcher for use with the event loop available in Python 3.4+.

Callbacks will be dispatched on the thread running the event loop. The loop argument should be a BaseEventLoop instance, e.g. the one returned from asyncio.get_event_loop().

CHAPTER 2

tellcore.constants (module)

Contains all constants used in the API.

All the TELLSTICK_* defines documented in the Telldus Core documentation are available here.

The recommended usage is to do:: import tellcore.constants as const

The constants will then be available as const.TELLSTICK_TURNON, etc. See below for the full list of all available constants.

```
# Device methods
TELLSTICK_TURNON = 1
TELLSTICK_TURNOFF = 2
TELLSTICK_BELL = 4
TELLSTICK_TOGGLE = 8
TELLSTICK_DIM = 16
TELLSTICK_LEARN = 32
TELLSTICK_EXECUTE = 64
TELLSTICK_UP = 128
TELLSTICK_DOWN = 256
TELLSTICK_STOP = 512

# Sensor value types
TELLSTICK_TEMPERATURE = 1
TELLSTICK_HUMIDITY = 2
TELLSTICK_RAINRATE = 4
TELLSTICK_RAINTOTAL = 8
TELLSTICK_WINDDIRECTION = 16
TELLSTICK_WINDAVERAGE = 32
TELLSTICK_WINDGUST = 64

# Error codes
TELLSTICK_SUCCESS = 0
TELLSTICK_ERROR_NOT_FOUND = -1
TELLSTICK_ERROR_PERMISSION_DENIED = -2
TELLSTICK_ERROR_DEVICE_NOT_FOUND = -3
TELLSTICK_ERROR_METHOD_NOT_SUPPORTED = -4
```

```
TELLSTICK_ERROR_COMMUNICATION = -5
TELLSTICK_ERROR_CONNECTING_SERVICE = -6
TELLSTICK_ERROR_UNKNOWN_RESPONSE = -7
TELLSTICK_ERROR_SYNTAX = -8
TELLSTICK_ERROR_BROKEN_PIPE = -9
TELLSTICK_ERROR_COMMUNICATING_SERVICE = -10
TELLSTICK_ERROR_CONFIG_SYNTAX = -11
TELLSTICK_ERROR_UNKNOWN = -99

# Device types
TELLSTICK_TYPE_DEVICE = 1
TELLSTICK_TYPE_GROUP = 2
TELLSTICK_TYPE_SCENE = 3

# Controller types
TELLSTICK_CONTROLLER_TELLSTICK = 1
TELLSTICK_CONTROLLER_TELLSTICK_DUO = 2
TELLSTICK_CONTROLLER_TELLSTICK_NET = 3

# Device changes
TELLSTICK_DEVICE_ADDED = 1
TELLSTICK_DEVICE_CHANGED = 2
TELLSTICK_DEVICE_REMOVED = 3
TELLSTICK_DEVICE_STATE_CHANGED = 4

# Change types
TELLSTICK_CHANGE_NAME = 1
TELLSTICK_CHANGE_PROTOCOL = 2
TELLSTICK_CHANGE_MODEL = 3
TELLSTICK_CHANGE_METHOD = 4
TELLSTICK_CHANGE_AVAILABLE = 5
TELLSTICK_CHANGE_FIRMWARE = 6
```

CHAPTER 3

tellcore.library (module)

The classes in this module are not meant to be used directly. They are mostly support classes for the higher level API described in *tellcore.telldus (module)*.

Library

class tellcore.library.**Library**(*name=None*, *callback_dispatcher=None*)

Wrapper around the Telldus Core C API.

With the exception of tdInit, tdClose and tdReleaseString, all functions in the C API (see Telldus Core documentation) can be called. The parameters are the same as in the C API documentation. The return value are mostly the same as for the C API, except for functions with multiple out parameters.

In addition, this class:

- automatically frees memory for strings returned from the C API,
- converts errors returned from functions into (*TelldusError*) exceptions,
- transparently converts between Python strings and C style strings.

__init__(*name=None*, *callback_dispatcher=None*)

Load and initialize the Telldus core library.

The underlaying library is only initialized the first time this object is created. Subsequent instances uses the same underlaying library instance.

Parameters

- **name** (*str*) – If None than the platform specific name of the Telldus library is used, but it can be e.g. an absolute path.
- **callback_dispatcher** – If callbacks are to be used, this parameter must refer to an instance of a class inheriting from *BaseCallbackDispatcher*.

__del__()

Close and unload the Telldus core library.

Any callback set up is removed.

The underlaying library is only closed and unloaded if this is the last instance sharing the same underlaying library instance.

tdController()
Get the next controller while iterating.

Returns a dict with the keys: id, type, name, available.

tdSensor()
Get the next sensor while iterating.

Returns a dict with the keys: protocol, model, id, datatypes.

tdSensorValue(*protocol*, *model*, *sid*, *datatype*)
Get the sensor value for a given sensor.

Returns a dict with the keys: value, timestamp.

TelldusError

exception tellcore.library.**TelldusError**(*error*, *lib=None*)
Error returned from Telldus Core API.

Automatically raised when a function in the C API returns an error.

Attributes: error: The error code constant (one of TELLSTICK_ERROR_* from *tellcore.constants*).

__str__()
Return the human readable error string.

BaseCallbackDispatcher

class tellcore.library.**BaseCallbackDispatcher**
Base callback dispatcher class.

Inherit from this class and override the *on_callback()* method to change how callbacks are dispatched.

on_callback(*callback*, **args*)
Called from the callback thread when an event is received.

Parameters

- **callback** (*callable*) – The callback function to call.
- **args** – The arguments to pass to the callback.

DirectCallbackDispatcher

class tellcore.library.**DirectCallbackDispatcher**
Dispatches callbacks directly.

Since the callback is dispatched directly, the callback is called in the callback thread.

CHAPTER 4

tellcore_tool (example)

The example below is an implementation of tdtool similar to the one included in the official Telldus Core distribution.

```
#!/usr/bin/env python
# Copyright (c) 2014 Erik Johansson <erik@ejohansson.se>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation; either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
# USA

import argparse
import sys
import time

import tellcore.telldus as td
import tellcore.constants as const

if sys.version_info < (3, 0):
    import tellcore.library as lib
    lib.Library.DECODE_STRINGS = False

def print_devices(devices):
    print("Number of devices: {}\n".format(len(devices)))
    print("{:<5s} {:<25s} {:<10s} {:<10s} {:<20s} {}".format(
            "ID", "NAME", "STATE", "PROTOCOL", "MODEL", "PARAMETERS"))
```

```
    for device in devices:
        cmd = device.last_sent_command(
            const.TELLSTICK_TURNON
            | const.TELLSTICK_TURNOFF
            | const.TELLSTICK_DIM)
        if cmd == const.TELLSTICK_TURNON:
            cmd_str = "ON"
        elif cmd == const.TELLSTICK_TURNOFF:
            cmd_str = "OFF"
        elif cmd == const.TELLSTICK_DIM:
            cmd_str = "DIMMED:{}".format(device.last_sent_value())
        else:
            cmd_str = "UNKNOWN:{}".format(cmd)
        params_str = ""
        for name, value in device.parameters().items():
            params_str += " {}:{}".format(name, value)
        print("{:<5d} {:<25s} {:<10s} {:<10s} {:<20s}{}".format(
                device.id, device.name, cmd_str,
                device.protocol, device.model, params_str))

def print_sensors(sensors):
    print("Number of sensors: {}\n".format(len(sensors)))
    print("{:<15s} {:<15s} {:<5s} {:<8s} {:<8s} {:<18s} {:<20s} {}".format(
            "PROTOCOL", "MODEL", "ID", "TEMP", "HUMIDITY", "RAIN", "WIND",
            "LAST UPDATED"))

    def format_value(sensor, datatype, formatter):
        if not sensor.has_value(datatype):
            return ("", None)
        value = sensor.value(datatype)
        return (formatter(value.value), value.datetime)

    for sensor in sensors:
        values = []
        values.append(format_value(sensor, const.TELLSTICK_TEMPERATURE,
                                   lambda x: "{} C".format(x)))
        values.append(format_value(sensor, const.TELLSTICK_HUMIDITY,
                                   lambda x: "{} %".format(x)))
        values.append(format_value(sensor, const.TELLSTICK_RAINRATE,
                                   lambda x: x + " mm/h "))
        values.append(format_value(sensor, const.TELLSTICK_RAINTOTAL,
                                   lambda x: x + " mm"))
        values.append(format_value(sensor, const.TELLSTICK_WINDDIRECTION,
                                   lambda x: ["N", "NNE", "NE", "ENE",
                                              "E", "ESE", "SE", "SSE",
                                              "S", "SSW", "SW", "WSW",
                                              "W", "WNW", "NW", "NNW"]
                                   [int(float(x) / 22.5)] + " "))
        values.append(format_value(sensor, const.TELLSTICK_WINDAVERAGE,
                                   lambda x: x + " m/s "))
        values.append(format_value(sensor, const.TELLSTICK_WINDGUST,
                                   lambda x: "({} m/s)".format(x)))

        # Get first valid timestamp
        timestamp = [v[1] for v in values if v[1] is not None][0]

        s = [v[0] for v in values]
        values_str = "{:<8s} {:<8s} ".format(s[0], s[1])
```

```
        values_str += "{:<18s} ".format(s[2] + s[3])
        values_str += "{:<20s} ".format(s[4] + s[5] + s[6])

        print("{:<15s} {:<15s} {:<5d} {}{}".format(
            sensor.protocol, sensor.model, sensor.id, values_str,
            timestamp))

def find_device(device, devices):
    for d in devices:
        if str(d.id) == device or d.name == device:
            return d
    print("Device '{}' not found".format(device))
    return None

parser = argparse.ArgumentParser(
    description='Telldus administration tool',
    epilog='DEVICE can be either device id or name')
group = parser.add_mutually_exclusive_group(required=True)

group.add_argument(
    '-l', '--list', action='store_true',
    help='List all configured devices and discovered sensors')
group.add_argument(
    '--list-devices', action='store_true',
    help='List all configured devices')
group.add_argument(
    '--list-sensors', action='store_true',
    help='List all discovered sensors')
group.add_argument(
    '--on', metavar='DEVICE', help='Turn on device')
group.add_argument(
    '--off', metavar='DEVICE', help='Turn off device')
group.add_argument(
    '--learn', metavar='DEVICE', help='Send learn command to device')
group.add_argument(
    '--remove', metavar='DEVICE', help='Remove device')

args = vars(parser.parse_args())

core = td.TelldusCore()

if args['list']:
    print_devices(core.devices())
    print("")
    print_sensors(core.sensors())
elif args['list_devices']:
    print_devices(core.devices())
elif args['list_sensors']:
    print_sensors(core.sensors())
elif args['on'] is not None:
    device = find_device(args['on'], core.devices())
    if device is not None:
        device.turn_on()
elif args['off'] is not None:
    device = find_device(args['off'], core.devices())
    if device is not None:
        device.turn_off()
elif args['learn'] is not None:
```

```
    device = find_device(args['learn'], core.devices())
    if device is not None:
        device.learn()
elif args['remove'] is not None:
    device = find_device(args['remove'], core.devices())
    if device is not None:
        device.remove()
```

CHAPTER 5

Event handling (example)

The example below illustrates how event callbacks are registered and processed using the default *telldus.QueuedCallbackDispatcher* dispatcher.

For more information regarding the arguments to the callback functions, please refere to the official Telldus Core documentation (see the callback typedefs). Please note that the context mentioned there is not included in tellcore-py.

```
#!/usr/bin/env python
# Copyright (c) 2012-2014 Erik Johansson <erik@ejohansson.se>
#
# This program is free software; you can redistribute it and/or
# modify it under the terms of the GNU General Public License as
# published by the Free Software Foundation; either version 3 of the
# License, or (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307
# USA

import argparse
import sys

import tellcore.telldus as td
import tellcore.constants as const

METHODS = {const.TELLSTICK_TURNON: 'turn on',
           const.TELLSTICK_TURNOFF: 'turn off',
           const.TELLSTICK_BELL: 'bell',
           const.TELLSTICK_TOGGLE: 'toggle',
           const.TELLSTICK_DIM: 'dim',
```

```
            const.TELLSTICK_LEARN: 'learn',
            const.TELLSTICK_EXECUTE: 'execute',
            const.TELLSTICK_UP: 'up',
            const.TELLSTICK_DOWN: 'down',
            const.TELLSTICK_STOP: 'stop'}

EVENTS = {const.TELLSTICK_DEVICE_ADDED: "added",
          const.TELLSTICK_DEVICE_REMOVED: "removed",
          const.TELLSTICK_DEVICE_CHANGED: "changed",
          const.TELLSTICK_DEVICE_STATE_CHANGED: "state changed"}

CHANGES = {const.TELLSTICK_CHANGE_NAME: "name",
           const.TELLSTICK_CHANGE_PROTOCOL: "protocol",
           const.TELLSTICK_CHANGE_MODEL: "model",
           const.TELLSTICK_CHANGE_METHOD: "method",
           const.TELLSTICK_CHANGE_AVAILABLE: "available",
           const.TELLSTICK_CHANGE_FIRMWARE: "firmware"}

TYPES = {const.TELLSTICK_CONTROLLER_TELLSTICK: 'tellstick',
         const.TELLSTICK_CONTROLLER_TELLSTICK_DUO: "tellstick duo",
         const.TELLSTICK_CONTROLLER_TELLSTICK_NET: "tellstick net"}


def device_event(id_, method, data, cid):
    method_string = METHODS.get(method, "UNKNOWN METHOD {0}".format(method))
    string = "[DEVICE] {0} -> {1}".format(id_, method_string)
    if method == const.TELLSTICK_DIM:
        string += " [{0}]".format(data)
    print(string)


def device_change_event(id_, event, type_, cid):
    event_string = EVENTS.get(event, "UNKNOWN EVENT {0}".format(event))
    string = "[DEVICE_CHANGE] {0} {1}".format(event_string, id_)
    if event == const.TELLSTICK_DEVICE_CHANGED:
        type_string = CHANGES.get(type_, "UNKNOWN CHANGE {0}".format(type_))
        string += " [{0}]".format(type_string)
    print(string)


def raw_event(data, controller_id, cid):
    string = "[RAW] {0} <- {1}".format(controller_id, data)
    print(string)


def sensor_event(protocol, model, id_, dataType, value, timestamp, cid):
    string = "[SENSOR] {0} [{1}/{2}] ({3}) @ {4} <- {5}".format(
        id_, protocol, model, dataType, timestamp, value)
    print(string)


def controller_event(id_, event, type_, new_value, cid):
    event_string = EVENTS.get(event, "UNKNOWN EVENT {0}".format(event))
    string = "[CONTROLLER] {0} {1}".format(event_string, id_)
    if event == const.TELLSTICK_DEVICE_ADDED:
        type_string = TYPES.get(type_, "UNKNOWN TYPE {0}".format(type_))
        string += " {0}".format(type_string)
    elif (event == const.TELLSTICK_DEVICE_CHANGED
```

```
          or event == const.TELLSTICK_DEVICE_STATE_CHANGED):
        type_string = CHANGES.get(type_, "UNKNOWN CHANGE {0}".format(type_))
        string += " [{0}] -> {1}".format(type_string, new_value)
    print(string)


parser = argparse.ArgumentParser(description='Listen for Telldus events.')

parser.add_argument(
    '--all', action='store_true', help='Trace all events')
parser.add_argument(
    '--device', action='store_true', help='Trace device events')
parser.add_argument(
    '--change', action='store_true', help='Trace device change events')
parser.add_argument(
    '--raw', action='store_true', help='Trace raw events')
parser.add_argument(
    '--sensor', action='store_true', help='Trace sensor events')
parser.add_argument(
    '--controller', action='store_true', help='Trace controller events')

args = vars(parser.parse_args())

try:
    import asyncio
    loop = asyncio.get_event_loop()
    dispatcher = td.AsyncioCallbackDispatcher(loop)
except ImportError:
    loop = None
    dispatcher = td.QueuedCallbackDispatcher()

core = td.TelldusCore(callback_dispatcher=dispatcher)
callbacks = []

for arg in args:
    if not (args[arg] or args['all']):
        continue
    try:
        if arg == 'device':
            callbacks.append(core.register_device_event(device_event))
        elif arg == 'change':
            callbacks.append(
                core.register_device_change_event(device_change_event))
        elif arg == 'raw':
            callbacks.append(core.register_raw_device_event(raw_event))
        elif arg == 'sensor':
            callbacks.append(core.register_sensor_event(sensor_event))
        elif arg == 'controller':
            callbacks.append(core.register_controller_event(controller_event))
        else:
            assert arg == 'all'
    except AttributeError:
        if not args['all']:
            raise

if len(callbacks) == 0:
    print("Must enable at least one event")
    parser.print_usage()
```

```
    sys.exit(1)

try:
    if loop:
        loop.run_forever()
    else:
        import time
        while True:
            core.callback_dispatcher.process_pending_callbacks()
            time.sleep(0.5)
except KeyboardInterrupt:
    pass
```

CHAPTER 6

Changelog

1.1.3 (2016-11-22)

- Added datetime attribute to `SensorValue`.
- Fixed strange problem in `Library` where the class itself could sometimes be None in `__del__`.

1.1.2 (2015-06-24)

- Added option to `Library` to make it possible to select if strings should be decoded or not.
- Made tellcore_tool not decode strings (i.e. convert to unicode) when running under Python 2.x to avoid unicode errors when printing non ascii characters.

1.1.1 (2015-05-01)

- Fixed a bug that made tellcore_tool not work with Python 3.x.

1.1.0 (2014-11-19)

- The callback dispatcher is no longer global, but tied to a `Library` instance. Applications wishing to use callbacks must now pass an explicit dispatcher instance to the `TelldusCore` constructor.

1.0.4 (2014-11-05)

- Made `last_sent_value` return an int instead of string.

1.0.3 (2014-02-02)

- Work around crash in Telldus Core (< v2.1.2) when re-initalizing the library after `tdClose`.

1.0.2 (2014-01-28)

- Packaging fixes.

1.0.1 (2014-01-26)

- Added `AsyncioCallbackDispatcher` class for integrating callbacks with the new event loop available in Python 3.4 (asyncio).
- Include tools from bin/ when installing.

1.0.0 (2014-01-09)

- Added high level support for device groups in the form of the new class `DeviceGroup`.
- More complete documentation.
- Removed the methods process_callback and process_pending_callbacks from `TelldusCore`. Instead, callback_dispatcher is now a public attribute of `TelldusCore` and the default callback dispatcher `QueuedCallbackDispatcher` implements the two methods instead.

0.9.0 (2014-01-03)

- Telldus functions that used to return bool (`tdSetName`, `tdSetProtocol`, `tdSetModel`, `tdSetDeviceParameter` and `tdRemoveDevice`) now raise an exception instead of returning False.
- Support for rain- and windsensors.
- Include data type in `SensorValue`.

0.8.0 (2013-08-11)

- Improved callback handling to simplify integration with different event loops. Parameter conversion is now done in the library code and the adaptation to different event loops is done by a simple callback dispatch class. The default dispatcher (when using `TelldusCore`) is still done using a queue.
- New documentation for parts of the package. Can be read online at https://tellcore-py.readthedocs.org/.
- Fix problem with strings and python 3 (issue #2).

0.1.0 (2013-06-26)

- First release.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

D

E

G

H

I

L

M

N

O

P

Q

R

S

T

U

V