micropython-ntp

Description

A robust MicroPython **Time library** for manipulating the **RTC** and and syncing it from a list of **NTP** servers.

Features:

1. Sync the RTC from a NTP host
2. Multiple NTP hosts
3. Microsecond precision
4. RTC chip-agnostic
5. Calculate and compensate RTC drift
6. Timezones
7. Epochs
8. Day Light Saving Time
9. Get time in sec, ms and us
10. Custom Logger with callback function

!!!At this point all the implemented features are robustly tested and they seem stable enough for production, BUT I do not recommended to use it in a production environment until the API stabilization phase is finished and some unit tests are developed!!!

Quick Guide

Before using the library there are a few thing that need to be done.

The first and the most important one is setting a callback for manipulating the RTC. The second and of the same importance is setting a list of NTP hosts/IPs.

Next things to configure are the Timezone and Daylight Saving Time but they are not mandatory if do not need them.

Other things that can be configured are:

- Network timeout
- Default epoch - if you need to get the time in other epoch other than the default for the device. Setting a default epoch allows you the convenience of not passing the epoch parameter each time you want to read the time. Each micropython port is compiled with a default epoch. For most of the ports but not all, it is 2000. For example the Unix port uses an epoch of 1970
- The drift value of the RTC if it is know in advance.
- The library logging output can be directed trough a callback. If not needed it can be disabled entirely

RTC access callback

The first thing to do when using the library is to set a callback function for accessing the RTC chip. The idea behind this strategy is that the library can manipulate multiple RTC chips(internal, external or combination of both) and is chip agnostic. Providing this function is your responsibility. It's declaration is:

```
def func(datetime: tuple = None) -> tuple
```

With no arguments, this method acts as a getter and returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time. The 8-tuple has the following format:

```
(year, month, day, weekday, hour, minute, second, subsecond)

# year is the year including the century part
# month is in (Ntp.MONTH_JAN ... Ntp.MONTH_DEC)
# day is in (1 ... 31)
# weekday is in (Ntp.WEEKDAY_MON ... Ntp.WEEKDAY_SUN)
# hour is in (0 ... 23)
# minute is in (0 ... 59)
# second is in (0 ... 59)
# subsecond is in (0 ... 999999)
```

The `RTC` class in the `machine` module, provides a drop-in alternative for a callback:

```
from machine import RTC
from ntp import Ntp

_rtc = RTC()
Ntp.set_datetime_callback(_rtc.datetime)
```

RTC sync

To be able to synchronize the RTC with NTP servers you have to set a list of hosts:

```
Ntp.set_hosts(('0.pool.ntp.org', '1.pool.ntp.org', '2.pool.ntp.org'))
```

You can pass a valid hostname or an IP. A basic validation is run when saving each host/ip. If the value is neither a valid hostname or IP address, it is skipped WITHOUT an error being thrown. It is your responsibility to pass the correct values.

After setting a list of NTP hosts, you can synchronize the RTC:

```
Ntp.rtc_sync()
```

This function will loop trough all the hostnames in the list and will try to read the time from each one. The first with a valid response will be used to sync the RTC. The RTC is always synchronized in UTC.

A network timeout in seconds can be set to prevent hanging

```
Ntp.set_ntp_timeout(timeout_s: int = 1)
```

Reading the time

There are two types of functions that read the time:

- ones that return the time as a timestamp
- ones that return the time in a date time tuple

The set of functions that return a timestamp is:

```
Ntp.time_s(epoch: int = None, utc: bool = False) -> int
Ntp.time_ms(epoch: int = None, utc: bool = False) -> int
Ntp.time_us(epoch: int = None, utc: bool = False) -> int
```

The suffix of each function shows the timestamp representation:

- _**s** - seconds
- _**ms** - milliseconds
- _**us** - microseconds

If you want to get the time relative to an epoch, you can pass one of the following constants:

```
Ntp.EPOCH_1900
Ntp.EPOCH_1970
Ntp.EPOCH_2000
```

If epoch parameter is `None`, the default epoch will be used. Otherwise the parameter will have a higher precedence.

If `utc = True` the returned timestamp will be in UTC format which excludes the Daylight Saving Time and the Timezone offsets.

To get the date and time in tuple format:

```
Ntp.time(utc: bool = False) -> tuple

# 9-tuple(year, month, day, hour, minute, second, weekday, yearday, us)
# year is the year including the century part
# month is in (Ntp.MONTH_JAN ... Ntp.MONTH_DEC)
# day is in (1 ... 31)
# hour is in (0 ... 23)
# minutes is in (0 ... 59)
# seconds is in (0 ... 59)
# weekday is in (Ntp.WEEKDAY_MON ... Ntp.WEEKDAY_SUN)
# yearday is in (1 ... 366)
# us is in (0 ... 999999)
```

!!! Both types of function read the time from the RTC!!!

To read the time directly from the NTP:

```
Ntp.ntp_time(epoch: int = None) -> tuple
# 2-tuple(ntp_time, timestamp)
# * ntp_time contains the accurate time(UTC) from the NTP server
#    in nanoseconds since the selected epoch.
# * timestamp contains timestamp in microseconds taken at the time the
#     request to the server was sent. This timestamp can be used later to
#   compensate for the time difference between the request was sent
#    and the later moment the time is used. The timestamp is the output
#    of time.ticks_us()
```

Get the accurate time from the first valid NTP server in the list with microsecond precision. When the server does not respond within the timeout period, the next server in the list is used. The default timeout is 1 sec. The timeout can be changed with `set_ntp_timeout()`. When none of the servers respond, throw an Exception. The epoch parameter serves the same purpose as with the other time functions.

Epochs

In micropython every port has it's own epoch configured during the compilation. Most of the ports use the epoch of `2000-01-01 00:00:00 UTC`, but some like the Unix port use a different. All of the micropython's build in time functions work according to this epoch. In this library I refer to this compiled in epoch as a **device's epoch**. It is not possible to change it during run-time.

Why is this important? There are multiple occasions where you need the time in different epochs. One example is that NTP uses an epoch of 1900. Another example is if you want to store a timestamp in a database. Some of the databases use an epoch of 1970, others use an epoch of 1900. The list can go on and on.

All time functions that return a timestamp supports an `epoch` parameter as described in above section.

Passing an epoch parameter every time is cumbersome, that is why there is a convenience functions that allows you to set a default epoch that all time functions will use.

```
# Set a default epoch
Ntp.set_epoch(epoch: int = None):
```

If `None` - the device's epoch will be used. Setting the epoch is not mandatory, the device's epoch will be used as default.

To get epoch of the device:

```
Ntp.device_epoch()
```

The `'time()` function does not have an epoch parameter, because it returns a structured tuple.

A helper function that is available for calculating the delta between two epochs:

```
epoch_delta(from_epoch: int, to_epoch: int) -> int
```

If you want to convert a timestamp from an earlier epoch to a latter, you will have to subtract the seconds between the two epochs. If you want to convert a timestamp from a latter epoch to an earlier, you will have to add the seconds between the two epochs. The function takes that into account and returns a positive or negative value.

RTC drift

All RTC are prone to drifting over time. This is due to manufacturing tolerances of the crystal oscillator, PCB, passive components, system aging, temperature excursions, etc. Every chip manufacturer states in the datasheet the clock accuracy of their chip. The unit of measure is **ppm**(parts per million). By knowing the frequency and ppm of the crystal, you can calculate how much the RTC will deviate from the real time. For example if you have a 40MHz clock which is stated +-10ppm.

```
# 1 part is equal 1 tick

frequency = 40_000_000
ppm  = 10 # 10 ticks for every 1_000_000 ticks
ticks_drift_per_sec = (frequency / 1_000_000) * ppm = 400

# The duration of one tick in seconds
tick_time = 1 / frequency = 0.000000025

# Calculate how many seconds will be the drift
# of the RTC every second
drift_every_sec = tick_time * ticks_drift_per_sec = 0.000_01
```

From the calculation above we know that the RTC can drift +-10us every second. If we know the exact drift, we can calculate the exact deviation from the real time. Unfortunately the exact ppm of every oscillator in unknown and has to be determined per chip manually.

To calculate the drift, the library uses a simpler approach. Every time the RTC is synchronized from NTP, the response is stored in a class variable. When you want to calculate the drift by

calling `Ntp.drift_calculate()`, the function reads the current time from NTP and compares it with the stored from the last RTC sync. By knowing the RTC microsecond ticks and the real delta between the NTP queries, calculating the ppm is a trivial task. The longer the period between `Ntp.rtc_sync()` and `Ntp.drift_calculate()` the more accurate result you will get. Empirically I found that in order to get results that vaguely come close to the real, calculating the drift shall be called at least 15 minutes after syncing the RTC.

To calculate the drift:

```
Ntp.drift_calculate(new_time = None) -> float
```

The return value is a float where positive values represent a RTC that is speeding, negative values represent RTC that is lagging, and zero means the RTC hasn't drifted at all.

To get the current drift of the RTC in microseconds:

```
Ntp.drift_us(ppm_drift: float = None)
```

This function does not read the time from the NTP server(no internet connection is required), instead it uses the previously calculated ppm.

To manually set the drift:

```
Ntp.set_drift_ppm(ppm: float)
```

The `ppm` parameter can be positive or negative. Positive values represent a RTC that is speeding, negative values represent RTC that is lagging. This is useful if you have in advance the ppm of the current chip, for example if you have previously calculated and stored the ppm.

The function `Ntp.rtc_sync()` is a pretty costly operation since it requires a network access. For an embedded IoT device this is unfeasible. Instead, you can compensate for the drift at regular and much shorter intervals by:

```
Ntp.drift_compensate(Ntp.drift_us())
```

A NTP sync can be performed at much longer intervals, like a day or week, depending on your device stability. If your device uses a TXCO(Temperature Compensated Crystal Oscillator), the period between NTP syncs can be much longer.

Here is a list of all the functions that are managing the drift:

```
Ntp.drift_calculate(cls)
Ntp.drift_last_compensate()
Ntp.drift_last_calculate()
Ntp.drift_ppm()
Ntp.set_drift_ppm(ppm: float)
Ntp.drift_us(ppm_drift: float = None)
Ntp.drift_compensate(compensate_us: int)
```

Timezones

The library has support for timezones. When setting the timezone ensures basic validity check.

```
Ntp.set_timezone(hour: int, minute: int = 0)
```

!!! NOTE: When syncing or drift compensating the RTC, the time will be set in UTC

Functions that support the `utc` argument can be instructed to return the time with the Timezone and DST calculated or the UTC time:

```
Ntp.time(utc: bool = False) -> tuple
Ntp.time_s(epoch: int = None, utc: bool = False) -> int
Ntp.time_ms(epoch: int = None, utc: bool = False) -> int
Ntp.time_us(epoch: int = None, utc: bool = False) -> int

Ntp.rtc_last_sync(epoch: int = None, utc: bool = False) -> int
Ntp.drift_last_compensate(epoch: int = None, utc: bool = False) -> int
Ntp.drift_last_calculate(epoch: int = None, utc: bool = False) -> int
```

Daylight Saving Time

The library supports calculating the time according to the Daylight Saving Time. To start using the DST functionality you have to set three things first:

- DST start date and time
- DST end date and time
- DST bias

These parameters can be set with just one function `set_dst(start: tuple, end: tuple, bias: int)` for convenience or you can set each parameter separately with a dedicated function. Example:

```
# Set DST data in one pass
# start (tuple): 4-tuple(month, week, weekday, hour) start of DST
# end (tuple) :4-tuple(month, week, weekday, hour) end of DST
# bias (int): Daylight Saving Time bias expressed in minutes
Ntp.set_dst(cls, start: tuple = None, end: tuple = None, bias: int = 0)

# Set the start date and time of the DST
# month (int): number in range 1(Jan) - 12(Dec)
# week (int): integer in range 1 - 6. Sometimes there are months when they can
spread over a 6 weeks ex. 05.2021
# weekday (int): integer in range 0(Mon) - 6(Sun)
# hour (int): integer in range 0 - 23
Ntp.set_dst_start(month: int, week: int, weekday: int, hour: int)

# Set the end date and time of the DST
# month (int): number in range 1(Jan) - 12(Dec)
# week (int): number in range 1 - 6. Sometimes there are months when they can
spread over 6 weeks.
# weekday (int): number in range 0(Mon) - 6(Sun)
# hour (int): number in range 0 - 23
Ntp.set_dst_end(cls, month: int, week: int, weekday: int, hour: int)

# Set Daylight Saving Time bias expressed in minutes.
# bias (int): minutes of the DST bias. Correct values are 30, 60, 90 and 120
Ntp.set_dst_time_bias(cls, bias: int)
```

You can disable DST functionality by setting any of the start or end date time to `None`

```
# Default values are `None` which disables the DST
Ntp.set_dst()
```

To calculate if DST is currently in effect:

```
Ntp.dst() -> int
```

Returns the bias in seconds. A value of `0` means no DST is in effect or it is disabled.

To get a boolean value:

```
bool(Ntp.dst())
```

Logger

The library support setting a custom logger. If you want to redirect the error messages to another destination, set your logger

```
Ntp.set_logger(callback = print)
```

The default logger is `print()` and to set it just call the method without any parameters. To disable logging, set the callback to "None"

Example

```
from machine import RTC
from ntp import Ntp
import time

def ntp_log_callback(msg: str):
    print(msg)

_rtc = RTC()

# Initializing
Ntp.set_datetime_callback(_rtc.datetime)
Ntp.set_logger_callback(ntp_log_callback)

# Set a list of valid hostnames/IPs
Ntp.set_hosts(('0.pool.ntp.org', '1.pool.ntp.org', '2.pool.ntp.org'))
# Network timeout set to 1 second
Ntp.set_ntp_timeout(1)
# Set timezone to 2 hours and 0 minutes
Ntp.set_timezone(2, 0)
# If you know the RTC drift in advance, set it manually to -4.6ppm
Ntp.set_drift_ppm(-4.6)
# Set epoch to 1970. All time calculations will be according to this epoch
Ntp.set_epoch(Ntp.EPOCH_1970)
# Set the DST start and end date time and the bias in one go
Ntp.set_dst((Ntp.MONTH_MAR, Ntp.WEEK_LAST, Ntp.WEEKDAY_SUN, 3),
            (Ntp.MONTH_OCT, Ntp.WEEK_LAST, Ntp.WEEKDAY_SUN, 4),
            60)


# Syncing the RTC with the time from the NTP servers
Ntp.rtc_sync()

# Let the RTC drift for 1 minute
time.sleep(60)

# Calculate the RTC drift
Ntp.drift_calculate()

# Let the RTC drift for 1 minute
time.sleep(60)

# Compensate the RTC drift
Ntp.drift_compensate(Ntp.drift_us())

# Get the last timestamp the RTC was synchronized
Ntp.rtc_last_sync()
```

```
# Get the last timestamp the RTC was compensated
Ntp.drift_last_compensate()

# Get the last timestamp the RTC drift was calculated
Ntp.drift_last_calculate()

# Get the calculated drift in ppm
Ntp.drift_ppm()

# Get the calculated drift in us
Ntp.drift_us()
```

Dependencies

- Module sockets
- Module struct
- Module time
- Module re
-

Download

You can download the project from GitHub:

```
git clone https://github.com/ekondayan/micropython-ntp.git micropython-ntp
```

License