Module `ntp`

Functions

```
def const(v)
```

Classes

```
class Ntp
```

Class variables

```
var EPOCH_1900
```

```
var EPOCH_1970
```

```
var EPOCH_2000
```

```
var MONTH_APR
```

```
var MONTH_AUG
```

```
var MONTH_DEC
```

```
var MONTH_FEB
```

```
var MONTH_JAN
```

```
var MONTH_JUL
```

```
var MONTH_JUN
```

```
var MONTH_MAR
```

```
var MONTH_MAY
```

var MONTH_NOV

var MONTH_OCT

var MONTH_SEP

var WEEKDAY_FRI

var WEEKDAY_MON

var WEEKDAY_SAT

var WEEKDAY_SUN

var WEEKDAY_THU

var WEEKDAY_TUE

var WEEKDAY_WED

var WEEK_FIFTH

var WEEK_FIRST

var WEEK_FOURTH

var WEEK_LAST

var WEEK_SECOND

var WEEK_THIRD

Static methods

def day_from_week_and_weekday(year, month, week, weekday)

Calculate the day based on year, month, week and weekday. If the selected week is outside the boundaries of the month, the last weekday of the month will be returned. Otherwise, if the weekday is within the boundaries of the month but is outside the boundaries of the week, raise an exception. This behaviour is desired when you want to select the last weekday of the month, like the last Sunday of October or the last Sunday of March. Example: day_from_week_and_weekday(2021,

Ntp.MONTH_MAR, Ntp.WEEK_LAST, Ntp.WEEKDAY_SUN)
day_from_week_and_weekday(2021, Ntp.MONTH_OCT, Ntp.WEEK_LAST, Ntp.WEEKDAY_SUN)

Args

`year` : `int`
number greater than 1

`month` : `int`
number in range 1(Jan) - 12(Dec)

`week` : `int`
number in range 1-6

`weekday` : `int`
number in range 0(Mon)-6(Sun)

Returns

`int`
the calculated day. If the day is outside the boundaries of the month, returns the last weekday in the month. If the weekday is outside the boundaries of the given week, raise an exception

`def days_in_month(year, month)`

Calculate how many days are in a given year and month

Args

`year` : `int`
number greater than 1

`month` : `int`
number in range 1(Jan) - 12(Dec)

Returns

`int`
the number of days in the given month

`def device_epoch()`

Get the device's epoch. Most of the micropython ports use the epoch of 2000, but some like the Unix port does use a different epoch. Functions like time.gmtime() and RTC.datetime() will use the device's epoch.

Returns

`int`

Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000

`def drift_calculate(new_time=None)`

Calculate the drift of the RTC. Compare the time from the RTC with the time from the NTP server and calculates the drift in ppm units and the absolute drift time in micro seconds. To bypass the NTP server, you can pass an optional parameter with the new time. This is useful when your device has an accurate RTC on board, which can be used instead of the costly NTP queries. To be able to calculate the drift, the RTC has to be synchronized first. More accurate results can be achieved if the time between last RTC synchronization and calling this function is increased. Practical tests shows that the minimum time from the last RTC synchronization has to be at least 20 min. To get more stable and reliable data, periods of more than 2 hours are suggested. The longer, the better. Once the drift is calculated, the device can go offline and periodically call drift_compensate() to keep the RTC accurate. To calculate the drift in absolute micro seconds call drift_us(). Example: drift_compensate(drift_us()). The calculated drift is stored and can be retrieved later with drift_ppm().

Args

`new_time` : `tuple`

None or 2-tuple(time, timestamp). If None, the RTC will be synchronized from the NTP server. If 2-tuple is passed, the RTC will be compensated with the given value. The 2-tuple format is (time, timestamp), where: * time = the micro second time in UTC relative to the device's epoch * timestamp = micro second timestamp in CPU ticks at the moment the time was sampled. Example: from time import ticks_us timestamp = ticks_us()

Returns

`tuple`

2-tuple(ppm, us) ppm is a float and represents the calculated drift in ppm units; us is integer and contains the absolute drift in micro seconds. Both parameters can have negative and positive values. The sign shows in which direction the RTC is drifting. Positive values represent an RTC that is speeding, while negative values represent RTC that is lagging

def drift_compensate(compensate_us: int)

Compensate the RTC by adding the compensate_us parameter to it. The value can be positive or negative, depending on how you wish to compensate the RTC.

Args

compensate_us : int

the microseconds that will be added to the RTC

def drift_last_calculate(epoch: int = None, utc: bool = False)

Get the last time the drift was calculated.

Args

utc : bool

the returned time will be according to UTC time

epoch : int, None

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

Returns

int

the last drift calculation time in micro seconds by taking into account epoch and utc

def drift_last_compensate(epoch: int = None, utc: bool = False)

Get the last time the RTC was compensated based on the drift calculation.

Args

utc : bool

the returned time will be according to UTC time

epoch : int, None

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

Returns

`int`

RTC last compensate time in micro seconds by taking into account epoch and utc

`def drift_ppm()`

Get the calculated or manually set drift in ppm units.

Returns

`float`

positive or negative number containing the drift value in ppm units

`def drift_us(ppm_drift: float = None)`

Calculate the drift in absolute micro seconds.

Args

`ppm_drift` : `float, None`

if None, use the previously calculated or manually set ppm. If you pass a value other than None, the drift is calculated according to this value

Returns

`int`

number containing the calculated drift in micro seconds. Positive values represent a speeding, while negative values represent a lagging RTC

`def dst(dt=None)`

Calculate if DST is currently in effect and return the bias in seconds.

Args

`dt` : `tuple, None`

If a None - current datetime will be read using the callback. If an 8-tuple(year, month, day, weekday, hour, minute, second, subsecond), it's value will be used to calculate the DST

Returns

`int`

Calculated DST bias in seconds

`def epoch_delta(from_epoch: int, to_epoch: int)`

Calculates the delta between two epochs. 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.

Args

`from_epoch` : `int, None`

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

`to_epoch` : `int, None`

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

Returns

`int`

The delta between the two epochs in seconds. Positive or negative number

`def get_dst_end()`

Get the end point of DST.

Returns

`tuple`

4-tuple(month, week, weekday, hour) or None if DST is disabled

`def get_dst_start()`

Get the start point of DST.

Returns

`tuple`

4-tuple(month, week, weekday, hour) or None if DST is disabled

`def get_dst_time_bias()`

Get Daylight Saving Time bias expressed in minutes.

Returns

`int`

minutes of the DST bias. Valid values are 30, 60, 90 and 120

`def get_epoch()`

Get the default epoch

Returns

`int`

One of (Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000)

`def get_hosts()`

Get a tuple of NTP servers.

Returns

`tuple`

NTP servers

`def get_ntp_timeout()`

Get the timeout of the network requests to the NTP servers.

Returns

`int`

Timeout of the request in seconds

def get_timezone()

Get the timezone as a tuple.

Returns

`tuple`

The timezone as a 2-tuple(hour, minute)

def ntp_time(epoch: int = None)

Retrieves the current UTC time from the first responsive NTP server in the provided server list with microsecond precision. This method attempts to connect to NTP servers sequentially, based on the server list order, until a response is received or the list is exhausted.

In case of a server timeout, defined by the class-level timeout setting (`set_ntp_timeout()`), the method proceeds to the next server in the list. The default timeout is 1 sec. If no servers respond within the timeout period, the method raises an exception.

The time received from the server is adjusted for network delay and converted to the specified epoch time format.

Args

epoch : `int` , optional

The epoch year from which the time will be calculated. If None, the epoch set by the user is used. Possible values are Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, or None for the user-defined default.

Returns

`tuple`

A 2-tuple(ntp_time, timestamp) containing: 1. The adjusted current time from the NTP server in microseconds since the specified epoch. 2. The timestamp in microseconds at the moment the request was sent to the server. This value can be used to compensate for time differences between when the response was received and when the returned time is used.

Raises

`RuntimeError`

If unable to connect to any NTP server from the provided list.

Note

The function assumes that the list of NTP servers (`cls._hosts`), NTP message template (`cls.__ntp_msg`), and timeout setting (`cls._ntp_timeout_s`) are properly initialized in the class.

`def rtc_last_sync(epoch: int = None, utc: bool = False)`

Get the last time the RTC was synchronized.

Args

`epoch` : `int, None`

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

`utc` : `bool`

the returned time will be according to UTC time

Returns

`int`

RTC last sync time in micro seconds by taking into account epoch and utc

`def rtc_sync(new_time=None)`

Synchronize the RTC with the time from the NTP server. To bypass the NTP server, you can pass an optional parameter with the new time. This is useful when your device has an accurate RTC on board, which can be used instead of the costly NTP queries.

Args

`new_time` : `tuple, None`

If None - the RTC will be synchronized from the NTP server.

If 2-tuple - the RTC will be synchronized with the given value. The 2-tuple format is (time, timestamp), where: * time = the micro second time in UTC since the device's epoch * timestamp = micro second timestamp at the moment the time was sampled

`def set_datetime_callback(callback)`

Set a callback function for reading and writing an RTC chip. Separation of the low level functions for accessing the RTC allows the library te be chip-agnostic. With this strategy you can manipulate the internal RTC, any external or even multiple RTC chips if you wish.

Args

`callback` : `function`

A callable object. With no arguments, this callable returns an 8-tuple with the current date and time. With 1 argument (being an 8-tuple) it sets the date and time of the RTC. The format of the 8-tuple is (year, month, day, weekday, hour, minute, second, subsecond)

!!! NOTE !!! Monday is index 0

`def set_drift_ppm(ppm: float)`

Manually set the drift in ppm units. If you know in advance the actual drift you can set it with this function. The ppm can be calculated in advance and stored in a Non-Volatile Storage as calibration data. That way the drift_calculate() as well as the initial long wait period can be skipped.

Args

`ppm` : `float, int`

positive or negative number containing the drift value in ppm units. Positive values represent a speeding, while negative values represent a lagging RTC

`def set_dst(start: tuple = None, end: tuple = None, bias: int = 0)`

A convenient function that set DST data in one pass. Parameters 'start' and 'end' are of type 4-tuple(month, week, weekday, hour) where: * month is in (Ntp.MONTH_JAN … Ntp.MONTH_DEC) * week is in (Ntp.WEEK_FIRST … Ntp.WEEK_LAST) * weekday is in (Ntp.WEEKDAY_MON … Ntp.WEEKDAY_SUN) * hour is in (0 … 23) To disable DST set 'start' or 'end' to None or 'bias' to 0. Clearing one of them will clear the others. To quickly disable DST just call the function without arguments - set_dst()

Args

`start` : `tuple`

4-tuple(month, week, weekday, hour) start of DST. Set to None to disable DST

end (tuple) :4-tuple(month, week, weekday, hour) end of DST. Set to None to disable DST

`bias` : `int`

Daylight Saving Time bias expressed in minutes. Set to 0 to disable DST

`def set_dst_bias(bias: int)`

Set Daylight Saving Time bias expressed in minutes. To disable DST set the bias to 0. By disabling the DST, the dst_start and dst_end will be set None

Args

bias : int
: minutes of the DST bias. Correct values are 0, 30, 60, 90 and 120. Setting to 0 effectively disables DST

def set_dst_end(month: int, week: int, weekday: int, hour: int)

Set the end date and time of the DST

Args

month : int
: number in (Ntp.MONTH_JAN … Ntp.MONTH_DEC)

week : int
: integer in (Ntp.WEEK_FIRST … Ntp.WEEK_LAST). Sometimes there are months that stretch into 6 weeks. Ex. 05.2021

weekday : int
: integer in (Ntp.WEEKDAY_MON … Ntp.WEEKDAY_SUN)

hour : int
: integer in range 0 - 23

def set_dst_start(month: int, week: int, weekday: int, hour: int)

Set the start date and time of the DST

Args

month : int
: number in (Ntp.MONTH_JAN … Ntp.MONTH_DEC)

week : int
: integer in (Ntp.WEEK_FIRST … Ntp.WEEK_LAST). Sometimes there are months that stretch into 6 weeks. Ex. 05.2021

weekday : int

integer in (Ntp.WEEKDAY_MON … Ntp.WEEKDAY_SUN)

hour : `int`

integer in range 0 - 23

def set_dst_time_bias(bias: int)

TO BE DEPRECATED. The function is renamed to set_dst_bias(). This name will be deprecated soon

def set_epoch(epoch: int = None)

Set the default epoch. All functions that return a timestamp value, calculate the result relative to an epoch. If you do not pass an epoch parameter to those functions, the default epoch will be used.

!!! NOTE: If you want to use an epoch other than the device's epoch, it is recommended to set the default epoch before you start using the class.

Args

epoch : `int, None`

If None - the device's epoch will be used. If int in (Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000) - a default epoch according to which the time will be calculated.

def set_hosts(value: tuple)

Set a tuple with NTP servers.

Args

value : `tuple`

NTP servers. Can contain hostnames or IP addresses

def set_logger_callback(callback=<built-in function print>)

Set a callback function for the logger, it's parameter is a callback function - func(message: str) The default logger is print(). To set it call the setter without any parameters. To disable logging, set the callback to "None".

Args

callback : `function`

A callable object. Default value = print; None = disabled logger; Any other value raises exception

def set_ntp_timeout(timeout_s: int = 1)

Set a timeout of the network requests to the NTP servers. Default is 1 sec.

Args

timeout_s : int

Timeout of the network request in seconds

def set_timezone(hour: int, minute: int = 0)

Validates if the provided hour and minute values represent a valid timezone offset. The valid hour range is -12 to +14 for a zero minute offset, and specific values for 30 and 45 minute offsets. Raises ValueError if the inputs are not integers, and raises Exception for invalid timezone combinations.

Args

hour : int

Timezone hour offset.

minute : int , optional

Timezone minute offset. Default is 0.

def time(utc: bool = False)

Get a tuple with the date and time in UTC or local timezone + DST.

Args

utc : bool

the returned time will be according to UTC time

Returns

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)

def time_ms(epoch: int = None, utc: bool = False)

Return the current time in milliseconds according to the selected epoch, timezone and Daylight Saving Time. To skip the timezone and DST calculation set utc to True.

Args

utc : bool

the returned time will be according to UTC time

epoch : int, None

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

Returns

int

the time in milliseconds since the selected epoch

def time_s(epoch: int = None, utc: bool = False)

Return the current time in seconds according to the selected epoch, timezone and Daylight Saving Time. To skip the timezone and DST calculation set utc to True.

Args

utc : bool

the returned time will be according to UTC time

epoch : int, None

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

Returns

int

the time in seconds since the selected epoch

def time_us(epoch: int = None, utc: bool = False)

Return the current time in microseconds according to the selected epoch, timezone and Daylight Saving Time. To skip the timezone and DST calculation set utc to True.

Args

utc : bool

the returned time will be according to UTC time

epoch : int, None

an epoch according to which the time will be calculated. If None, the user selected epoch will be used. Possible values: Ntp.EPOCH_1900, Ntp.EPOCH_1970, Ntp.EPOCH_2000, None

Returns

int

the time in microseconds since the selected epoch

def weekday(year: int, month: int, day: int)

Find Weekday using Zeller's Algorithm, from the year, month and day.

Args

year : int

number greater than 1

month : int

number in range 1(Jan) - 12(Dec)

day : int

number in range 1-31

Returns

int

0(Mon) 1(Tue) 2(Wed) 3(Thu) 4(Fri) 5(Sat) to 6(Sun)

def weekday_in_month(year: int, month: int, ordinal_weekday: int, weekday: int)

Calculate and return the day of the month for the Nth ordinal occurrence of the specified weekday within a given month and year. If there are fewer occurrences of the specified weekday in the month, the function returns the day of the last occurrence of the specified weekday. For instance, if you are looking for the second Tuesday of a month, "second" is the ordinal representing the occurrence of the weekday "Tuesday," and you would use 2 as the value for the ordinal_weekday parameter in the function.

Example

weekday_in_month(2021, Ntp.MONTH_MAR, Ntp.WEEK_SECOND, Ntp.WEEKDAY_SUN)
weekday_in_month(2021, Ntp.MONTH_OCT, Ntp.WEEK_LAST, Ntp.WEEKDAY_SUN)

Args

`year` : `int`

The year for which the calculation is to be made, must be an integer greater than 1.

`month` : `int`

The month for which the calculation is to be made, must be an integer in the range 1(Jan) - 12(Dec).

`ordinal_weekday` : `int`

Represents the ordinal occurance of the weekday in the specified month, must be an integer in the range 1-6.

`weekday` : `int`

Represents the specific weekday, must be an integer in the range 0(Mon)-6(Sun).

Returns

`int`

The day of the month of the Nth ordinal occurrence of the specified weekday. If the ordinal specified is greater than the total occurrences of the weekday in that month, it returns the day of the last occurrence of the specified weekday.

Raises

`ValueError`

If any of the parameters are of incorrect type or out of the valid range.

`def weeks_in_month(year, month)`

Split the month into tuples of weeks. The definition of a week is from Mon to Sun. If a month starts on a day different from Monday, the first week will be: day 1 to the day of the first Sunday. If a month ends on a day different from the Sunday, the last week will be: the last Monday till the end of the month. A month can have up to 6 weeks in it. For example if we run this function for May 2021, the result will be: [(1, 2), (3, 9), (10, 16), (17, 23), (24, 30), (31, 31)]. You can clearly see that the first week consists of just two days: Sat and Sun; the last week consists of just a single day: Mon

Args

`year` : `int`

number greater than 1

`month` : `int`

number in range 1(Jan) - 12(Dec)

Returns

`list`

2-tuples of weeks. Each tuple contains the first and the last day of the current week. Example result for May 2021: [(1, 2), (3, 9), (10, 16), (17, 23), (24, 30), (31, 31)]

Index

Functions

- const

Classes

- **Ntp**
  - EPOCH_1900
  - EPOCH_1970
  - EPOCH_2000
  - MONTH_APR
  - MONTH_AUG
  - MONTH_DEC
  - MONTH_FEB
  - MONTH_JAN
  - MONTH_JUL
  - MONTH_JUN
  - MONTH_MAR
  - MONTH_MAY
  - MONTH_NOV
  - MONTH_OCT
  - MONTH_SEP
  - WEEKDAY_FRI
  - WEEKDAY_MON
  - WEEKDAY_SAT
  - WEEKDAY_SUN
  - WEEKDAY_THU
  - WEEKDAY_TUE
  - WEEKDAY_WED
  - WEEK_FIFTH
  - WEEK_FIRST
  - WEEK_FOURTH
  - WEEK_LAST
  - WEEK_SECOND
  - WEEK_THIRD

- day_from_week_and_weekday
- days_in_month
- device_epoch
- drift_calculate
- drift_compensate
- drift_last_calculate
- drift_last_compensate
- drift_ppm
- drift_us
- dst
- epoch_delta
- get_dst_end
- get_dst_start
- get_dst_time_bias
- get_epoch
- get_hosts
- get_ntp_timeout
- get_timezone
- ntp_time
- rtc_last_sync
- rtc_sync
- set_datetime_callback
- set_drift_ppm
- set_dst
- set_dst_bias
- set_dst_end
- set_dst_start
- set_dst_time_bias
- set_epoch
- set_hosts
- set_logger_callback
- set_ntp_timeout
- set_timezone
- time
- time_ms
- time_s
- time_us
- weekday
- weekday_in_month

- `weeks_in_month`
