Event Timer
Software Manual

By: Jim Schrempp and Bob Glicksman; v1, 8/31/2025

NOTICE: Use of this document is subject to the terms of use described in the document "Terms_of_Use_License_and_Disclaimer" that is included in this release package. This document can be found at:

https://github.com/TeamPracticalProjects/Event_Timer/tree/main/Documents/Terms_of_Use_License_and_Disclaimer.pdf

TABLE OF CONTENTS.

DOCUMENT OVERVIEW.

This document describes the Event Timer software. This software relies on the exceptionally functional *LocalTimeRK* library to perform all of the complex time conversion functions as well as to create, manage, and use schedules to publish events to the Particle[1] Cloud at pre-scheduled times and dates.

The software in this repository was specifically written to publish events to the Particle Cloud that trigger the Annunciator devices of the Maker Nexus Help System[2] to automatically play pre-recorded voice announcements near closing time each day. The purpose of this document is to assist the reader in modifying this software to meet their own, unique, event publication requirements.

The Event Timer, the software herein and the *LocalTimeRK* library are specific to the Particle environment, the Particle Cloud, and the Particle Publish/Subscribe system.

SOFTWARE OVERVIEW.

Software Concepts.

The Event Timer software publishes events to the Particle Cloud based upon pre-determined (in the software) schedules. Each event is subscribed to by external Particle-based devices. Subscribers receive events when they are published and execute an event handler software function that manages the event, according to the event's data. Projects and products that are based upon Particle IoT devices can have their functions automated by subscribing to an event that is published by the Event Timer.

The management, conversion, formatting, and adjusting of time can be very complex. The setting up and execution of time-based schedules, to publish Particle events at specific times, days, and possibly with specific exceptions, is even more complex. Fortunately, the *LocalTimeRK* library is available to handle all of this for us. This library is very complex and it takes time to fully understand it and to learn how to utilize its capabilities. A key purpose of this document is to supplement the *LocallTimeRK* library documentation with an actual project that uses much of this library's capabilities.

The *LocalTimeRK* library provides most of the functionality of the Event Timer software. The software described in this document uses the library to do the following:

[1] Particle.io
[2] https://github.com/TeamPracticalProjects/MN_Help_System

- Set up and maintain the current date/time in a localized manner; i.e. properly adjusted for the local timezone and daylight savings time (if applicable).
- Perform date/time format conversions for human readable display on the Event Timer LCD display panel.
- Create and utilize schedules to determine the date/time of events that are to be published to the Particle Cloud. Very complex schedules can be specified using the *LocalTimeRK* library.
- Manage several schedules in order to publish different events (or at least the same event with different event data) that trigger different actions by the subscribers.

In addition to publishing events to the Particle Cloud, the Event Timer software:

- Displays the current date/time on a 2 line, 16 character LCD display panel.
- Displays the date/time of the next scheduled event, of all schedules that it manages, on a 2 line, 16 character LCD display panel.
- Adjusts for daylight savings time and indicates, via an LED, whether the displayed date/time is daylight savings time or standard time.
- Indicates when an event is being published by flashing an LED.
- Indicates when the Event Timer is powered up, initialized, and connected to the Internet. Internet connection is essential because the Event Timer keeps accurate time by acquiring the current time (in UTC) from the Internet via the Particle *Time.now()* function.

LocalTimeRK Library.

The *LocalTimeRK* library does all of the "heavy lifting" for time management, time conversion, time formatting, schedule setting, and schedule management. This library is part of the Particle environment and can be included in any project using either the Particle Web IDE or the Particle Workbench.

Online documentation for the *LocalTimeRK* library can be found at:

https://rickkas7.github.io/LocalTimeRK/index.html

According to this documentation, the library is applicable for:

- *Managing local time, and daylight saving time, if needed, on devices in a known location*
- *Mostly intended for devices in your own home*
- *Managing scheduling of tasks at a specific local time*
- *Displaying local time*

The main features of this library are:

- *Timezone configuration using a POSIX timezone rule string*
- *Does not require network access to determine timezone and daylight saving transitions*
- *Good for displaying local time, such as on clock-like devices*

- *Good for scheduling operations at a specific local time. For example, every day at 8:00 AM regardless of timezone and DST*
- *Support for locations with DST and without DST (timezone only)*
- *Should work in the southern hemisphere were DST is opposite on the calendar*
- *Should work in any country as long as a compatible POSIX timezone configuration string can be generated*

The *LocalTimeRK* library is open source and can be found at:

https://github.com/rickkas7/LocalTimeRK

Time Conversion and Formatting.

The *LocalTimeRK* automatically uses the Particle *Time.now()* function to obtain the correct current time from the Internet. This Internet time is in UTC format and the library performs all calculations in UTC for performance reasons. However, the library user can communicate with the library in local time, using the library's built-in time conversion functionality.

Conversion from UTC to local time is specified in POSIX standard format. This format is described in detail in the library's on-line documentation. It is easy to read and understand. For example, a local time instance that maintains the local time in the U.S. Pacific timezone that observes US daylight savings time is established in the software via the following one line of code:

```
LocalTime::instance().withConfig(LocalTimePosixTimezone("PST8PDT,M3.2.0/2:00:00,
M11.1.0/2:00:00"));
```

Similarly, conversion of the library's internal time format to a human readable format is easily performed. The following one line of code converts the time that is encapsulated in the "conv" object to month, day, hour, minute, second displayable format:

```
msg = conv.format("%m-%d %I:%M:%S%p");
```

Where msg is a String and conv is an instance of the library's *LocalTimeConvert* class.

Note that the line of code above can also include the year. We did not include year in the displayable message String due to limitations on the line size of the LCD display used in the Event Timer.

Consult the library's on-line documentation for further details:

https://rickkas7.github.io/LocalTimeRK/index.html

Time Schedules.

One fantastic feature of the *LocalTimeRK* library is that it can set up and execute schedules. A summary of this library functionality is extracted from the on-line documentation:

…, it's designed to handle scheduling. For example, say you want to perform an operation at 3:00 AM every day, local time. The class can find the UTC time corresponding to this, taking into account timezones and DST changes. Using Time.now() comparisons (at UTC) is fast and efficient. It's also good when you want to store the desired time in EEPROM, retained memory, or the file system.

The library also handles weird transition scenarios that occur on spring forward (in the north hemisphere) where the hour from 2:00 AM local time to 2:59:59 doesn't exist, and in the fall back where the hour from 1:00:00 to 1:59:59 local time occurs twice.
It also handles other common scheduling scenarios:

- *Tomorrow (same time)*
- *Tomorrow (at a specific local time)*
- *On a specific day of week at a specific time ("every Saturday at 3:00 AM")*
- *On a day of week with an ordinal ("on the 2nd Saturday of the month")*
- *On a specific day of month at a specific time*
- *Also the last day of the month, the second to last day of the month, ...*
- *On the next weekday (Monday - Friday)*
- *On the next weekend day (Saturday - Sunday)*

Schedules are instances of the library's *LocalTimeSchedule* class. There are many helper classes and subclasses in the library that are used to set up and query schedules. The library handles an extremely wide variety of scheduling scenarios that can handle almost any scheduling requirement, including:

- Minute of hour
- Hour of day multiples
- Day of week of the month
- Day of month
- Restricted date ranges
- Exceptions

From the on-line documentation, examples of scheduling include:

- *"At 17:00:00" every day (local time)*
- *"At 17:00:00, Monday - Friday"*
- *"At 09:00:00, Monday - Friday, except for 2022-03-21"*
- *"At 23:59:59 on 2022-03-31"*

Schedules are so flexible that the on-line documentation is lengthy and dense. However, it is well worth the time to read through and understand all that this library can do:

https://rickkas7.github.io/LocalTimeRK/index.html

Schedule objects have a great many methods associated with them. Some important methods are:

- *schedule.isScheduledTime()*: determine if it is time to run the scheduled task based upon the current time.
- *schedule.getNextScheduledTime(conv)*: Modify the "conv" object (instance of the library's *LocalTimeConvert* class) to encapsulate the time for the next event in the schedule. NOTE: the reference for the event time before the "next" event time is passed in via the conv object, so be sure to initialize "conv" to the current time, or whatever time you need to be the refence, before calling this method.

Schedule Manager.

One class in the *LocalTimeRK* library that is not well described in the on-line documentation is the *LocalTimeScheduleManager*. Instances of this class hold multiple named schedules in a C++ vector. The key method associated with this class is:

- *schedule.forEach() {}*: loops through the manager and returns each schedule object in the manager. The code to execute methods on each schedule in the manager is placed inside the curly brackets.

Schedules are added to the manager in a somewhat obscure way. The *LocalTimeScheduleManager* class does not have methods like addSchedule(), removeSchedule() etc. Rather, you add or modify a schedule in the schedule manager using the *manager.getScheduleByName(String name)* method. This method returns a bool indicating whether the named schedule exists in the manager. If the named schedule does not exist in the manager, then it is automatically added to the manager. For example, code to add a schedule to an instance of the *LocalTimeScheduleManager* class called *MNScheduleManager* is:

```
MNScheduleManager.getScheduleByName("13")
    .withTime(LocalTimeHMSRestricted(
    LocalTimeHMS("21:30:00"),
    LocalTimeRestrictedDate(LocalTimeDayOfWeek::MASK_ALL)
));
```

This code adds or updates a schedule named "13" to the *MNScheduleManager* object. This schedule will return true on a call to the *.isScheduledTime()* method at 9:30 pm (21:30) local time every day.

LiquidCrystal Library.

The venerable Arduino *LiquidCrystal* library has long been ported into the Particle environment. This library makes it easy to use character-oriented LCD display devices, of which there exist

many different modules with different numbers of characters per line and different numbers of lines. This library allows a programmer to print to the LCD in the same manner as printing to a Serial port.

The Event Timer uses a 2 line, 15 character per line LCD to display the current time (top line) and the time of the next scheduled event (bottom line). This display is not essential to the functioning of the Event Timer. However, it is helpful to see that the Event Timer is working properly.

Event Publication.

The end product of the Event Timer are publications to the Particle Cloud of events; each event according to a managed schedule(s). *Particle.publish()* statements are used to publish events to the Particle Cloud. Published events contain an event name and event data. Subscribers subscribe to events by the event name and receive the event data for further processing. Coding can be as simple is placing an appropriate *Particle.publish()* statement inside of the code block for a *schedule.isScheduledTime()* statement which itself is inside of a *manager.forEach()* loop block of code.

The software that is in this repository includes two functions that use the schedule name of the returned schedule (in the *manager.forEach()* loop) to generate a data string for the event data. These functions are: *simulateSensor()* and *logToParticle()*. These particular functions are specific to a project and would be removed/replaced in any other application of the Event Timer.

Note that the software in this repository obtains the name of the schedule for which the present time is a scheduled time and passes this name to the *simulateSensor()* function. This software structure allows more complex processing of the event publication to be encapsulated into its own function.

SOFTWARE DETAILS.

The Event Timer software source code can be found in this repository at:

https://github.com/TeamPracticalProjects/Event_Timer/blob/main/Software/ParticleFirmware/Event_Timer_Firmware/src/Event_Timer_Firmware.cpp

This software is specific to a project. The software would be modified in several ways when the Event Timer is used on another project. The purpose of this section is to walk through the software line by line so that the Reader can understand it well enough to make the necessary modifications.

Includes and Defines.

Line 1: #include “application.h” is automatically added by the Particle Workbench. It may be moved below the introductory comments so that it is with the other #include statements.

Line 45: SYSTEM_MODE(AUTOMATIC) is the default system mode and is therefore probably redundant. The automatic system mode tasks the Particle device OS to manage the Photon 2’s WiFi connections automatically, without further involvement from our software.

Lines 47 – 49: includes the libraries.

Lines 54 – 57: define the Photon 2 pins that are used for the Event Timer’s LEDs and buzzer. This specific pinout is for the connections made by the printed circuit board used for the Event Timer. This printed circuit board was originally developed for another project and the board is reused here:

https://github.com/TeamPracticalProjects/MN_ACL/tree/master/Hardware/PC_board

Line 60: instantiates an object (“lcd”) of the *LiquidCrystal* class. The pinouts are those provided on the printed circuit board mentioned above.

Line 63: instantiates the manager object used in this program “*MNScheduleManager*”. It is an instance of the *LocalTimeScheduleManager* class.

Functions.

Lines 65 – 73: the function *logToParticle()* creates a specifically formatted String out of its arguments and publishes a "LoRaHubLogging" event to the Particle Cloud with this String as the event data. This event and its formatted data are specific to the project that the Event Manager was originally created for. It would be removed/replaced for any other project.

Lines 74 – 81. The function *simulatedSensor()* converts its String argument to an integer and uses this as one of the arguments in this function’s call to *logToParticle()*. This function is specific to the project that the Event Manager was originally created for. It would be removed/replaced for any other project.

Setup().

Line 84: Places the version number constant defined in line 51 of the program into a Cloud variable. It is our standard practice to do this so that we can use the Particle Console to read out the version of software that is actually flashed onto our Particle devices.

Lines 86 – 94: Sets up the Photon 2 pins that drive the LEDs and buzzer to outputs and sets all of these pins to LOW for initialization.

Lines 97 - 105: set up the lcd and initially loads " ----------" into the two lines of the display. This is to ensure that the times are not displayed until the Photon 2 connects to the Internet for time initialization; in other words, so that an incorrect time is not displayed.

Lines 108 – 110: wait until the Photon 2 is connected to the Internet so that its time can be properly set.

Line 117: this line of code sets up the internal time conversion to U.S. Pacific time (UTC – 8) and to observe U.S. daylight savings time. It also creates an instance of the *LocalTime* class that is synchronized with Internet time and is the basic "time now" for the program.

Lines 119 – 149: these lines of code create 4 schedules inside of the previously defined *MNScheduleManager* object. The schedule names are "13", "14", "17" and "15" respectively. These names are project specific – they are later used in the creation of the event data Strings for published events from the Event Timer.

Schedule "13" has a scheduled time of 9:30 pm (21:30:00 hrs) every day. Schedule "14" has a scheduled time of 9:45 pm (21:45:00 hrs) every day. Schedule "17" has a scheduled time of 9:55 pm (21:55:00 hrs) every day. Schedule "15" has a scheduled time of 10:00 pm (22:00:00 hrs) every day.

It is worthwhile to deconstruct one of these statements in order to fully explain the syntax. Here is a deconstruction of Schedule "13":

```
MNScheduleManager.getScheduleByName("13")
```

Adds a schedule named "13" to the *MNScheduleManager* object.

```
.withTime(LocalTimeHMSRestricted(
LocalTimeHMS("21:30:00"),
```

Sets a time (21:30:00 or 9:30 pm) for the schedule in local time units (converted from UTC into Pacific time, daylight savings time adjusted).

```
LocalTimeRestrictedDate(LocalTimeDayOfWeek::MASK_ALL)
```

```
));
```

Declares that the set time is to be scheduled every day of the week with no exclusions and no date range (i.e. repeats forever).

Lines 152 – 159: set the READY LED on and beep the buzzer to indicate that setup() is complete and all times and schedules are set.

Loop().

Line 166: creates the object “conv” which is an instance of the *LocalTimeConvert* class. “conv” is hereafter used to encapsulate a time in UTC format and convert it to local time (Pacific time, adjusted for daylight savings time).

Line 167: uses the “conv” object to convert the current Internet time from UTC into local time.

Lines 170 – 174: uses the method *isDST()* of the “conv” object to return true if the current local time inside of “conv” is daylight savings time (false if standard time). It sets the ADMIT LED on for DST and off for standard time.

Lines 176 – 180: uses the *format()* method of the “conv” object to format the current local time into Month – Day _ Hour : Minute : Second format and displays the current time (in this format) on the top line of the LCD.

Lines 182 – 205: this loops through the *MNScheduleManager*, retrieving each managed schedule, in turn, one schedule on each iteration of the loop. When a schedule in hand, two actions are performed:

- The *isScheduledTime()* method returns true if the current time is the next time in the schedule. If true, the schedule name is passed to the *simulateSensor()* function that ultimately results in the publication of an event to the Particle Cloud with the event data determined by the name of the schedule.
- The “conv” object is reset to the current time and the *getNextScheduleTime()* method changes the internal time in “conv” to be the next scheduled time for the current schedule. This time is compared with the current earliest time and replaces the latter if earlier. When all schedules in *MNScheduleManager* have been looped through, “conv” will encapsulate the time of the next scheduled event of all of the schedules in *MNScheduleManager*.

Lines 208 – 211: use methods of the “conv” object to convert its internal time (which is the earliest next scheduled time of all of the managed schedules) to Month – Day _ Hour : Minute : Second format and display it on the bottom line of the LCD.

Line 213: waits a little while (100 ms) before returning to the top of loop. This prevents the LCD display from flickering while not compromising the timekeeping functions of the program.

NOTES ON GENERATING SCHEDULES.

The *LocalTimeRK* library's scheduling capability is extraordinarily powerful and flexible. The downside to this is that it is very complex and takes a lot of time and effort to learn. The on-line documentation contains many examples, but these only scratch the surface of what this library is capable of.

Happily, we have found that Microsoft Copilot AI has an excellent understanding of this library and will generate the code for complex scheduling problems based upon a word description. Below are two examples of Copilot generated code. We have not tested these, but they appear to be correct.

Example: Create two schedules, one daily at 4pm, one only Sunday at 2pm

```
LocalTimeScheduleManager manager;

// Daily at 4pm
manager.getScheduleByName("daily_4pm")
  .withTime(LocalTimeHMSRestricted(
    LocalTimeHMS("16:00:00"),
    LocalTimeRestrictedDate(LocalTimeDayOfWeek::MASK_ALL)
  ));

// Sunday at 2pm
manager.getScheduleByName("sunday_2pm")
  .withTime(LocalTimeHMSRestricted(
    LocalTimeHMS("14:00:00"),
    LocalTimeRestrictedDate(LocalTimeDayOfWeek::MASK_SUNDAY)
  ));
```

Example: Create two schedules, one daily at 4pm except Christmas, one only Sunday at 2pm

```
LocalTimeScheduleManager manager;

// Daily at 4pm, except Christmas (December 25)
manager.getScheduleByName("daily_4pm")
    .withTime(LocalTimeHMSRestricted(
        LocalTimeHMS("16:00:00"),
        LocalTimeRestrictedDate(
            LocalTimeDayOfWeek::MASK_ALL,
            {},
            {"2025-12-25"} // exceptDates: Christmas
        )
    ));

// Sunday at 2pm
manager.getScheduleByName("sunday_2pm")
    .withTime(LocalTimeHMSRestricted(
        LocalTimeHMS("14:00:00"),
        LocalTimeRestrictedDate(LocalTimeDayOfWeek::MASK_SUNDAY)
    ));
```

NOTE ON VOLATILITY AND PERSISTENCE.

The *LocalTimeRK* library supports json specification for many of its functions; setting up schedules is an example. This capability means that the Event Timer software could be generalized to allow schedules to be entered from the Console or from an external app using *Particle.function()* with the json representation of a schedule as the String argument. This approach would obviate our current approach of having to change and re-flash the software anytime there is a change to an Event Timer schedule.

The main downside of this alternative approach is volatility. Anytime the power is lost, even for an instant, the current scheduling information will be lost. An approach to handling this situation is to serialize schedules in the manager and store this as String data in the Photon 2's file system. The file(s) would be read during *startup()* and the schedules restored into the manager.

We considered this alternative and we have established (with the help of Copilot) that it is feasible. However, we rejected this approach because of its added complexity and the fact that our requirements for the Event Timer require only very infrequent changes to schedules. We assessed that json representation of schedule information is more complex that the code

representation of the same schedules in our software. The external app that generates json encoded schedules from some user friendly GUI would be at least as complex as the *LocalTimeRK* library, obviating the benefits of using this library in the first place. There would be additional complexity in serializing schedule data into files and deserializing data from files back into schedule objects. And finally, testing of the software and the app would be much more complex, as we would need to test our new code more thoroughly than we feel is necessary for testing the library code.

By coding our schedules directly into the library methods in software, we automatically have non-volatility via the schedules being stored in flash (program) memory on the Photon 2. Furthermore, we have Copilot as a valuable assistant to generate schedule code from our word specifications.