

K12 :: О ESP32 и не только
2,5K подписчиков

Подписаться



Работа с датой и временем и SNTP-синхронизация на ESP32 и ESP8266

26 декабря 2022 · 880 прочитали

Добрый день, уважаемый читатель! В прошлой статье [я рассказывал про подключение к сети WiFi](#), что нужно сделать сразу после этого? Ну, не строго обязательно, конечно, но строго желательно?

Подключаться к MQTT-брокеру? Нет, рановато - защищенное подключение потребует проверки TLS-сертификата, а для этого потребуется проверить срок его действия.

Да вы, наверное, уже догадались из названия статьи: первое, что потребуется сделать сразу после подключения - это получить актуальные дату и время с любого публичного [NTP-сервера](#). Это позволяет не тратиться на автономные I2C-часики, а всегда иметь "под рукой" достаточно точные отметки времени. Впрочем, даже когда в вашей системе установлена микросхема часов реального времени, никогда не помешает синхронизировать её с более точными данными из сети Интернет.



Дата и время используются в микроконтроллерах повсеместно - для журналирования, для установки защищенных интернет соединений, для работы по расписаниям, да много для чего ещё.

не могу.

Почти всё, что описано в данной статье, применимо не только к ESP32, но и к ESP8266 + Arduino, там также можно использовать стандартную библиотеку <time.h> (с небольшими изменениями, но не суть). Поэтому в этот раз я решил написать "универсальную" статью.

Для начала обсудим методы работы с датой и временем с использованием стандартной библиотеки <time.h>, а уже потом я расскажу, как получить актуальное время из сети интернет. В том числе и для Arduino в лице esp8266. Ну в заключение приведу нескольких несложных "самодельных" функций для контроля интервалов суток, которыми я частенько пользуюсь.

ESP32 использует два аппаратных таймера для сохранения системного времени. Системное время можно сохранить с помощью одного или обоих аппаратных таймеров в зависимости от назначения устройства и требований к точности системного времени:

- **Таймер RTC** (по умолчанию): этот таймер позволяет сохранять время в различных спящих режимах, а также может сохранять отсчет времени при любых сбросах (за исключением сбросов при включении питания, которые сбрасывают таймер RTC). Отклонение частоты зависит от [источников часов таймера RTC](#) и влияет на точность только в спящих режимах, в этом случае время будет измеряться с разрешением 6,6667 мкс.
- **Таймер высокого разрешения**: этот таймер недоступен в спящих режимах и не будет сохраняться после сброса, но имеет большую точность. Таймер использует источник тактового сигнала APB_CLK (обычно 80 МГц), который имеет девиацию частоты менее ± 10 ppm. Время будет измеряться с разрешением 1 мкс.

Стандартная библиотека time.h

Как я уже написал, для работы с датой и временем используется стандартная библиотека time.h, которая была унаследована миром микроконтроллеров из UNIX и других POSIX-совместимых операционных систем, поэтому используемый на большинстве микроконтроллеров способ кодирования даты и времени называется **UNIX-время** или **POSIX-время** (англ. Unix time).

Дата и время в этом стандарте представлены в виде обычного 64-битного числа, в котором хранится количество секунд, прошедших с начала условного отсчёта. Моментом начала отсчёта считается полночь (по UTC) с 31 декабря 1969 года на **1 января 1970**, время с этого момента называют "эрой UNIX" (англ. Unix Epoch). Способ хранения времени в виде количества секунд очень удобно использовать при сравнении даты и времени с точностью до секунды, а также для хранения: дата и время занимают в памяти относительно мало места и при необходимости их можно легко преобразовать в любой удобочитаемый формат. Недостатками такого способа хранения времени являются невозможность хранения времени с точностью меньше секунды, а также потери в производительности при очень частом обращении к "вычисляемым" элементам даты, вроде номера месяца и т. п. Но в большинстве случаев всё-таки эффективнее хранить время в виде одной величины, а не набора полей.

1 день	86400 секунд
1 неделя	604800 секунд
1 месяц (30.44 дней)	2629743 секунд
1 год (365.24 дней)	31556926 секунд

Представление стандартных временных интервалов в секундах

Пересчитать "человеческие" дату и время в unix time и наоборот можно с помощью [online конверторов](#).

Получение системного времени

Итак, с форматом определились, как же его получить? А очень просто - для этого используйте функцию, которая так и называется - `time()`:

```
C

time_t time( time_t *destTime );
__time32_t _time32( __time32_t *destTime );
__time64_t _time64( __time64_t *destTime );
```

Позволю себе процитировать справочную систему:

Функция `time()` возвращает число секунд, истекших после полуночи (00:00:00) 1 января 1970 г. (UTC) по системным часам. Возвращаемое значение сохраняется в расположении, предоставленном `destTime`. Этот параметр может иметь значение `NULL`, в этом случае возвращаемое значение не сохраняется.

`time()` является оболочкой для `_time64()`, а `time_t` по умолчанию равнозначно `uint64_t`. Если необходимо, чтобы компилятор принудительно интерпретировал `time_t` как старое 32-разрядное значение `time_t`, можно определить макрос `_USE_32BIT_TIME_T`, но это не приветствуется, так как приложение может завершиться сбоем после 18 января 2038 г.

Теперь давайте посмотрим на практике, как это сделать на ESP32 или ESP8266:

```
// Получаем время, способ 1
time_t now1 = time(NULL);

// Получаем время, способ 2
time_t now2;
time(&now2);
```

Как использовать - решайте сами в зависимости от ситуации. Чаще удобнее использовать первый способ, передав `NULL` в функцию. Но когда требуется периодически "обновлять" одну и ту же переменную, разумнее использовать второй способ.

Получение более точного системного времени

Если вам нужно получить время с разрешением в одну микросекунду, используйте функцию `gettimeofday()`. Данная функция принимает в качестве аргумента структуру, с двумя полями:

- в поле `tv_sec` хранится целое число секунд, это и есть `time_t`
- в поле `tv_usec` хранится дополнительное число микросекунд.

У функции есть также второй аргумент, который должен быть равен `NULL`.

Функции - счетчики циклов работы процессора

На Arduino-платформе есть популярная функция `millis()`, которая возвращает количество **миллисекунд**, прошедших с момента запуска контроллера. По сути это просто счетчик, который каждую миллисекунду добавляет к своему значению единичку.

На ESP32 есть несколько функций с аналогичным назначением:

- `int64_t esp_timer_get_time()` - возвращает количество **микросекунд** с момента запуска базового таймера (чуть-чуть позже момента запуска контроллера). Можно использовать её вместо `millis()`, но нужно поделить полученное значение на 1000UL. В отличие от рассмотренной выше функции `gettimeofday()`, значение, возвращаемое функцией `esp_timer_get_time()` будет сброшено в 0 при выходе из режима глубокого сна и к нему не применяется корректировка часового пояса.
- `uint32_t esp_cpu_get_cycle_count()` (или `cpu_hal_get_cycle_count()`) - вернет количество выполненных циклов для текущего ядра процессора. Эта функция имеет наименьшие накладные расходы, чем все остальные. Она отлично подходит для измерения очень короткого времени выполнения с высокой точностью. Циклы ЦП подсчитываются для каждого ядра отдельно, поэтому используйте этот метод только из обработчика прерываний или задачи, закрепленной за одним ядром.
- `TickType_t xTaskGetTickCount()` - количество тиков операционной системы с момента вызова `vTaskStartScheduler`, то есть с момента запуска FreeRTOS. Её удобно использовать внутри задач для подсчета относительно коротких интервалов времени (длиннее одного тика ОС). Для работы из прерывания существует специальная версия `xTaskGetTickCountFromISR()`.

Например, вики шоу 24 часа

Войти

```
// получить количество выполненных циклов ЦП
uint32_t cycles1 = esp_cpu_get_cycle_count();
uint32_t cycles2 = cpu_hal_get_cycle_count();
```

Примечание: `cpu_hal_get_cycle_count()` используется в том числе в библиотеке "esp_log.h" для вывода меток времени.

Формирование временных интервалов

Целочисленный формат времени `time_t` очень удобен для вычисления и отчёта интервалов времени с точностью до секунды. Просто вычитите одно значения `time_t` из другого `time_t`, и вы получите искомый интервал в секундах. Для интервалов до месяца это очень удобно - целочисленное деление поможет вам легко и очень быстро (в процессорных тактах) определить количество дней, часов, минут и секунд между двумя отметками времени:



```

uint16_t s = value % 3600 % 60;

return malloc_stringf("%.2d:%.2d:%.2d", h, m, s);
}

char * malloc_timespan_dhms(time_t value)
{
    uint16_t d = value / 86400;
    uint16_t h = value % 86400 / 3600;
    uint16_t m = value % 86400 % 3600 / 60;
    uint16_t s = value % 86400 % 3600 % 60;

    return malloc_stringf("%d:%.2d:%.2d:%.2d", d, h, m, s);
}

```

В данном случае аргумент `value` это не отметка времени, а разница между двумя отметками, то есть $(time_t\ now - time_t\ prev)$. Функция

`malloc_timespan_hms(time_t value)` возвращает интервал времени в виде строки в часах, минутах и секундах, а похожая функция `malloc_timespan_dhms(time_t value)` возвращает строку с интервалов в днях, часах, минутах и секундах. Эти функции я описывал в одной из предыдущих статей.

Точно также можно посчитать и недели, а вот с месяцами уже начинаются сложности, так как месяц имеет переменную длину. И тут без перекодировки данных в день месяца и год уже не обойтись.

Интерпретация `time_t` в набор полей - часы, минуты, день, месяц и т.д.

Целочисленный формат времени очень удобен для контупера, но не подходит для использования человеком. Да и для расписаний зачастую нужно знать конкретные месяц и год, а не количество секунд в них. Для конвертации в "раздельный" формат даты и времени предназначена функции `localtime(const time_t *sourceTime)` и `localtime_r(const time_t *sourceTime, struct tm *targetTime)`. Эти функции принимают в качестве аргумента отметку времени в UNIX-формате `time_t`, а возвращают в виде такой структуры:

```

struct tm
{
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
#ifdef __TM_GMTOFF
    long __TM_GMTOFF;
#endif
#ifdef __TM_ZONE
    const char *__TM_ZONE;
#endif
};

```

где:

- `tm_sec` - секунды после минуты (0 – 59);
- `tm_min` - минуты после часа (0 – 59);
- `tm_hour` - часы с полуночи (от 0 до 23);

- **tm_year** - год (текущий год минус 1900);
- **tm_wday** - день недели (0 – 6; Воскресенье = 0);
- **tm_yday** - день года (0 – 365; 1 января = 0);
- **tm_isdst** - положительное значение, если летнее время действует; 0, если летнее время не действует; отрицательное значение, если состояние летнего времени неизвестно;

Как видите, полученные данные требуют "доработки напильником": как минимум прибавить 1 к значению месяца и дня года, 1900 к значению года. Да и обозначение дней недели не соответствует российским, но это везде так.

Важное замечание! Данные функции корректируют входное значение с учетом часового пояса, если он был до этого установлен (как установить часовой пояс я расскажу чуть ниже). Поэтому и называются localtime() - локальное (то есть местное) время.

Отличие этих функций только в том, что первая принимает только один аргумент - *указатель* на **time_t**, а возвращает указатель на описанную выше структуру **struct tm** в куче; а вторая имеет уже два аргумента, где можно передать указатель на заранее выделенный буфер под выходные данные **struct tm**. Лично я предпочитаю второй способ (дабы не возиться потом с освобождением выделенной памяти):

```
// Конвертация целочисленного UNIX-формата в структуру с отдельными полями
struct tm tm_now;
localtime_r(&now1, &tm_now);
ESP_LOGI("time", "hour = %02d, min = %02d, week day = %02d, month day = %d, year day = %d, month = %d, year = %d, is dst = %d",
tm_now.tm_hour, tm_now.tm_min, tm_now.tm_sec,
tm_now.tm_wday, tm_now.tm_mday, tm_now.tm_yday,
tm_now.tm_mon, tm_now.tm_year,
tm_now.tm_isdst);
```

На этом скриншоте закралась ошибка... Вначале хотел поправить, а потом решил оставить в качестве небольшого регбуса-кроссворда ;-)

Пример результата декодирования представлен на скриншоте ниже:

```
I (264) heap_init: At 3FFE0440 len 00003AE0 (14 KiB): D/IRAM
I (271) heap_init: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (277) heap_init: At 4008B1A4 len 00014E5C (83 KiB): IRAM
I (285) spi_flash: detected chip: generic
I (288) spi_flash: flash io: dio
I (293) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
I (20298) time: now1 = 10
I (30298) time: now2 = 20
I (30298) time: hour = 00, min = 00, sec = 10, week day = 4, month day = 1, year day = 0, month = 0, year = 70, is dst = 0
```

Да, да, у нас пока что первые секунды нового 1970 года....

Ну а дальше можно делать с полученными данными что необходимо.

Установка часового пояса

Если вы планируете подключить к своему микроконтроллеру аппаратные ЧРВ (часы реального времени), то вам, вероятно, не потребуется синхронизация с SNTP, и следовательно, нет особого смысла возиться с установкой часового пояса.

Но если вы планируете получать точное время из этих ваших интырнетов, то придется заранее определиться с часовым поясом в системе, так как SNTP (сервера точного времени) всегда возвращают время "по Гринвичу".

установить для переменной среды TZ правильное значение часового пояса. Формат описания часового пояса такой же, как описано в [документации GNU libc](#). Для Москвы это будет "MSK-3". [Список часовых поясов вы можете найти здесь](#).

2. Вызовите `tzset()`, чтобы обновить данные времени выполнения библиотеки C для нового часового пояса.

```
// Установка часового пояса
setenv("TZ", "MSK-3", 1);
tzset();
```

Для ESP8266 это можно сделать точно так же, а также [дополнительно имеется функция setTZ\(\)](#), которая выполняет то же самое, но копирует часовой пояс в отдельную область памяти (кучи). Кроме того, на ESP8266 установить часовой пояс можно непосредственно перед запуском синхронизации времени, о чем будет рассказано чуть ниже.

Примечание: для ESP8266 и Arduino разработчиками заботливо предусмотрен файл `TZ.h`, в котором определены всевозможные часовые пояса. Почему такой файл отсутствует в ISP-IDF - я не знаю... Но вы вполне можете скопировать нужную зону оттуда.

Преобразование даты и времени в строку

Однако чаще всего нужно просто вывести дату или время (или и дату и время вместе) в виде строки - например в логах, в каких-либо уведомлениях и т.д. Про формирование строковых представлений для относительно небольших интервалов я уже рассказывал чуть выше, теперь давайте разберемся, как формировать строковые представления для даты и времени в произвольном формате. Для этого воспользуемся функцией

```
strftime(char *strDest, size_t bufSize, const char *format, const struct tm *timeptr);
```

где:

- `strDest` - указатель на буфер, в который будет помещены результаты форматирования
- `bufSize` - размер буфера
- `format` - строка управления форматом, [список поддерживаемых символов вы можете найти в справочной системе](#) (он очень обширный, я не буду приводить его здесь)
- `timeptr` - указатель на структуру `struct tm`, которую мы рассматривали выше

Пример использования буфера со статическим буфером фиксированного размера:

```
// Форматирование строк с датой и временем
char strftime_buf[64];
strftime(strftime_buf, sizeof(strftime_buf), "%Y-%m-%d %H:%M:%S", &tm_now);
```

В данном случае время будет выведено в формате 1900-01-01 03:00:10

Для ESP8266 и ESP32, поскольку они имеют встроенный модуль WiFi, имеется возможность получить точное время с серверов SNTP через интернет. Конечно, подключить к интернету можно и обычный Arduino, но тогда придется найти библиотеку SNTP самостоятельно, а для ESPxx они уже имеются в встроенных framework-ах. Для начала давайте разберемся, как это сделать на ESP32.

Запуск синхронизации времени на ESP32

За SNTP-синхронизацию времени на ESP32 отвечает библиотека "esp_sntp.h".

Библиотека поддерживает два режима синхронизации:

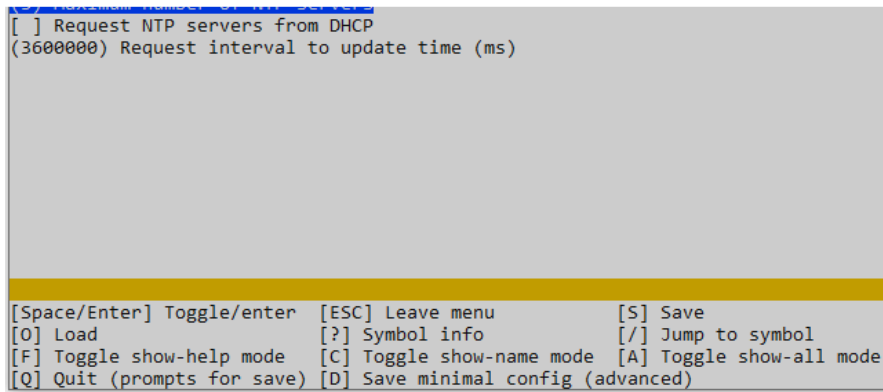
- **SNTP_SYNC_MODE_IMMED** (по умолчанию) - обновляет системное время сразу после получения ответа от сервера SNTP. То есть было 01/01/1900, после обновления сразу стало 23/12/2022. В большинстве случаев это самый удобный способ.
- **SNTP_SYNC_MODE_SMOOTH** - плавное обновление времени за счет постепенного уменьшения ошибки времени с помощью функции `adjtime()`. Если разница между временем ответа SNTP и системным временем превышает 35 минут, немедленно обновите системное время с помощью `settimeofday()`. Это бывает необходимо для приложений, чувствительных к системному времени, для которых резкий переход через года может привести к краху.

Давайте рассмотрим самый простой способ - `SNTP_SYNC_MODE_IMMED`.

Процедура запуска синхронизации времени в этом случае выглядит следующим образом:

1. Установить корректный часовой пояс (time zone), так как сервера NTP всегда возвращают время в UTC. Это мы обсудили чуть выше.
2. Установить режим работы с помощью `sntp_setoperatingmode()`. Функция это не описана в справочной системе, но насколько я понимаю, она позволяет выбрать режим работы - получение времени по нашему запросу `SNTP_OPMODE_POLL` или когда сервер сам соизволит разослать широковещательное сообщение `SNTP_OPMODE_LISTENONLY`. Очевидно, для первого раза нужно использовать `SNTP_OPMODE_POLL`, а затем можно перейти на `SNTP_OPMODE_LISTENONLY`, если есть особое желание, но я так никогда не делал.
3. Установить адрес SNTP-сервера с помощью `sntp_setservername()`. Библиотека поддерживает установку до 16 серверов, но по умолчанию настроен только один. Я обычно использую пять. Это не значит, что синхронизация выполняется сразу по всем пяти серверам, это значит, если первый не доступен - будет выполнен запрос к следующему, и так до конца списка. Настроить количество используемых серверов времени можно через утилиту SDK config (командой `pio run -t menuconfig`) в разделе **Component config** → **LWIP** → **SNTP**.

После первой успешной синхронизации времени запускается специально выделенный программный таймер, по истечении времени которого происходит повторная синхронизация времени. Таким образом всегда поддерживается актуальное время на устройстве. Период автоматической синхронизации также настраивается через `menuconfig`:



Настройки SNTP в SDK config

Впрочем, период повторной синхронизации времени можно и изменить "вручную", с помощью функции `sntp_set_sync_interval(uint32_t interval_ms)`. Это может быть полезно, чтобы задать разные интервалы в случае успешной и не успешной синхронизации.

Пример запуска синхронизации с двумя серверами:

```
// Установка часового пояса
setenv("TZ", "MSK-3", 1);
tzset();

// Запускаем синхронизацию времени с SNTP
sntp_setoperatingmode(SNTP_OPMODE_POLL);
sntp_setservername(0, "pool.ntp.org");
sntp_setservername(1, "time.nist.gov");
sntp_init();
```

Однако тут есть небольшая проблемка - как понять, случилась таки синхронизация времени или ещё нет? Процесс получения времени асинхронный и может занимать довольно длительный промежуток времени. Можно поступить двумя способами:

- создать цикл и тупо ждать, пока системное время не превысит 1000000000 (это 09/08/2001 г.)

```
// Ждем, пока локальное время синхронизируется
Serial.print("Waiting for NTP time sync: ");
i = 0;
time_t now = time(nullptr);
while (now < 1000000000) {
    now = time(nullptr);
    i++;
    if (i > 60) {
        // Если в течение этого времени не удалось подключиться - выходим с false
        // Бесконечно ждать подключения опасно - если подключение было разорвано во время работы
        // нужно всё равно "обслуживать" реле и датчики, иначе может случиться беда
        Serial.println("");
        Serial.println("Time sync failed!");
        return false;
    };
    Serial.print(".");
    delay(500);
}
```

Пример цикла ожидания синхронизации времени для Arduino-платформы

- настроить функцию обратного вызова, при вызове которой можно, например, оповестить рабочих задачам, что время синхронизировано и можно устанавливать TLS-соединения.

```

    localtime_r(&tv->tv_sec, &timeinfo);
    if (timeinfo.tm_year < (1970 - 1900)) {
        rlog_e(logTAG, "Time synchronization failed!");
    } else {
        // Post time event
        eventLoopPost(RE_TIME_EVENTS, RE_TIME_SNTP_SYNC_OK, nullptr, 0, portMAX_DELAY);
        // Log
        strftime(strftime_buf, sizeof(strftime_buf), "%d.%m.%Y %H:%M:%S", &timeinfo);
        rlog_i(logTAG, "Time synchronization completed, current time: %s", strftime_buf);
    }
}

```

Пример функции обратного вызова при синхронизации времени

Во втором случае процедура запуска SNTP-службы будет выглядеть чуть-чуть по другому:

```

// Установка часового пояса
setenv("TZ", "MSK-3", 1);
tzset();

// Запускаем синхронизацию времени с SNTP
sntp_setoperatingmode(SNTP_OPMODE_POLL);
sntp_set_time_sync_notification_cb(sntp_notification);
sntp_setservername(0, "pool.ntp.org");
sntp_setservername(1, "time.nist.gov");
sntp_init();

```

Запуск синхронизации времени на ESP8266

Теперь давайте рассмотрим, как можно сделать то же самое на ESP8266 или ESP32 на Arduino-платформе. По большому счету, процедура запуска синхронизации времени для ESP8266 почти "один в один" повторяет реализацию на ESP32 - те же `setenv()`, `tzset()`, `setServer()`, `sntp_init()` и т.д.

Но, как и во многих других случаях, Arduino предоставляет более простые функции, дабы не заморачиваться со всей этой кутерьмой. В итоге, для запуска синхронизации времени вполне достаточно вызывать всего одну функцию - `configTime()`, в которой необходимо указать минимально необходимые аргументы: часовой пояс и сервер (или несколько серверов) синхронизации. Существуют две перезагружаемых версии `configTime()`, которые отличаются только способом определения часовой зоны:

- `void configTime(int timezone_sec, int daylightOffset_sec, const char* server1, const char* server2, const char* server3)` - в этой версии сдвиг времени указывается в секундах
- `void configTime(const char* tz, const char* server1, const char* server2, const char* server3)` - в этой версии часовая зона указывается в виде строки, список часовых зон определен в `TZ.h`:

```

#define TZ_Europe_Kaliningrad PSTR("EET-2")
#define TZ_Europe_Kiev PSTR("EET-2EEST,M3.5.0/3,M10.5.0/4")
#define TZ_Europe_Kirov PSTR("<+03>-3")
#define TZ_Europe_Lisbon PSTR("WET0WEST,M3.5.0/1,M10.5.0")
#define TZ_Europe_Ljubljana PSTR("CET-1CEST,M3.5.0,M10.5.0/3")
#define TZ_Europe_London PSTR("GMT0BST,M3.5.0/1,M10.5.0")
#define TZ_Europe_Luxembourg PSTR("CET-1CEST,M3.5.0,M10.5.0/3")
#define TZ_Europe_Madrid PSTR("CET-1CEST,M3.5.0,M10.5.0/3")
#define TZ_Europe_Malta PSTR("CET-1CEST,M3.5.0,M10.5.0/3")
#define TZ_Europe_Mariehamn PSTR("EET-2EEST,M3.5.0/3,M10.5.0/4")
#define TZ_Europe_Minsk PSTR("<+03>-3")
#define TZ_Europe_Monaco PSTR("CET-1CEST,M3.5.0,M10.5.0/3")
#define TZ_Europe_Moscow PSTR("MSK-3")

```

отсутствующие аргументы:

```
// Для работы TLS-соединения нужны корректные дата и время, получаем их с NTP серверов
configTime(3 * 3600, 0, "pool.ntp.org", "time.nist.gov");
// Ждем, пока локальное время синхронизируется
Serial.print("Waiting for NTP time sync: ");
i = 0;
time_t now = time(nullptr);
while (now < 1000000000) {
    now = time(nullptr);
    i++;
    if (i > 60) {
        // Если в течение этого времени не удалось подключиться - выходим с false
        // Бескорнечно ждать подключения опасно - если подключение было разорвано во время работы
        // нужно всё равно "обслуживать" реле и датчики, иначе может случиться беда
        Serial.println("");
        Serial.println("Time sync failed!");
        return false;
    }
    Serial.print(".");
    delay(500);
}
```

Пример запроса времени с ожиданием результата

Указанный выше пример синхронизации времени вы можете посмотреть [здесь](#):

arduino/arduino_eps8266_pub_gpio_state_ssl at master · kotyara12/arduino
github.com

Формирование суточных расписаний с точностью до минуты

Для большинства сценариев бытовой (да и не только) автоматизации нет необходимости учитывать секунды. Вполне достаточно, если какой-либо сценарий начнет свою работу, скажем в 22:00 и закончит в 08:30 следующего дня. Секунды можно легко отбросить, и тем самым сильно упростить себе жизнь при хранении интервалов в памяти. Но при этом хотелось бы, чтобы сценарий начал свое выполнение в 22:00:00 и закончил в 08:29:59, а не с 22:00:18 по 08:30:55.

Самое простое, что можно сделать в данном случае - запустить программный таймер ([про них я уже рассказывал на данном канале](#)). Единственная проблемка, которую нужно будет решить - как привязаться к началу каждой минуты, про это подробнее я расскажу чуть ниже.

Хранение суточных интервалов

Для хранения суточных интервалов я использую обычное 32-х разрядное число `uint32_t`. Часы, не мудрствуя лукаво, умножаем на 10, минуты записываем как есть. Получаем начало интервала, для нашего примера это будет $2200 = 22 \cdot 10 + 00$. Далее точно таким же способом формируем конец интервала, получаем 0830.

Затем опять же "сдвигаем" начало интервала на 4 разряда вправо, просто умножив его на 10000 и складываем с вычисленным концом интервала, получаем 22000830. Это и есть "зашифрованный" суточный интервал. Его вполне можно хранить в `uint32_t`, и легко сохранить в flash-памяти в качестве параметров.

- `uint16_t time_begin = timespan / 10000;`
- `uint16_t time_end = timespan % 10000;`

Сравнение интервала с текущим временем

Осталось сравнить эти данные с пересчитанным в такой же формат текущим временем:

```
bool checkTimespan(struct tm* timeinfo, timespan_t timespan)
{
    if (timespan > 0) {
        int16_t t0 = timeinfo->tm_hour * 100 + timeinfo->tm_min;
        uint16_t t1 = timespan / 10000;
        uint16_t t2 = timespan % 10000;

        // t1 < t2 :: ((t0 >= t1) && (t0 < t2))
        // t0=0559 t1=0600 t2=2300 : (0559 >= 0600) && (0559 < 2300) = 0 && 1 = 0
        // t0=0600 t1=0600 t2=2300 : (0600 >= 0600) && (0600 < 2300) = 1 && 1 = 1
        // t0=0601 t1=0600 t2=2300 : (0601 >= 0600) && (0601 < 2300) = 1 && 1 = 1
        // t0=2259 t1=0600 t2=2300 : (2259 >= 0600) && (2259 < 2300) = 1 && 1 = 1
        // t0=2300 t1=0600 t2=2300 : (2300 >= 0600) && (2300 < 2300) = 1 && 0 = 0
        // t0=2301 t1=0600 t2=2300 : (2301 >= 0600) && (2301 < 2300) = 1 && 0 = 0

        // t1 > t2 :: !((t0 >= t2) && (t1 > t0))
        // t0=2259 t1=2300 t2=0600 : (2259 >= 0600) && (2300 > 2259) = 1 && 1 = 1 !=> 0
        // t0=2300 t1=2300 t2=0600 : (2300 >= 0600) && (2300 > 2300) = 1 && 0 = 0 !=> 1
        // t0=2301 t1=2300 t2=0600 : (2301 >= 0600) && (2300 > 2301) = 1 && 0 = 0 !=> 1
        // t0=0559 t1=2300 t2=0600 : (0559 >= 0600) && (2300 > 0559) = 0 && 1 = 0 !=> 1
        // t0=0600 t1=2300 t2=0600 : (0600 >= 0600) && (2300 > 0600) = 1 && 1 = 1 !=> 0
        // t0=0601 t1=2300 t2=0600 : (0601 >= 0600) && (2300 > 0601) = 1 && 1 = 1 !=> 0

        return (t1 < t2) ? ((t0 >= t1) && (t0 < t2)) : !((t0 >= t2) && (t1 > t0));
    };
    return false;
}
```

Функция `checkTimespan()` учитывает интервалы не только в пределах одних суток (когда t1 меньше t2), но и интервалы с переходом через полночь (когда t1 больше t2). Но вот сформировать такой интервал более одних суток с помощью неё уже не получится, это как в истории с классическим будильником со стрелками.

Привязка программного таймера к началу минуты

Как я уже упомянул, на данный момент времени генерацией отметок времени и обработкой расписаний занимается специально выделенный программный таймер.

Таймер генерирует следующие события в системную очередь событий:

- `RE_TIME_EVERY_MINUTE` - данное событие генерируется в начале каждой минуты (00 секунд)
- `RE_TIME_START_OF_HOUR` - событие, обозначающее начало каждого часа
- `RE_TIME_START_OF_DAY` - событие, обозначающее начало каждого дня
- `RE_TIME_START_OF_WEEK` - событие, обозначающее начало очередной недели
- `RE_TIME_START_OF_MONTH` - событие, обозначающее начало месяца
- `RE_TIME_START_OF_YEAR` - событие, обозначающее начало года

- **RE_TIME_TIMESPAN_OFF** - событие, генерируемое при окончании суточного расписания, добавленного пользователем
- **RE_TIME_SILENT_MODE_ON** - событие, обозначающее начало тихого режима (погасить все светодиоды, подсветку экранов, отключить звуки)
- **RE_TIME_SILENT_MODE_OFF** - событие, обозначающее окончание тихого режима (разрешить светодиоды, подсветку экранов, звуки)

Получив одно из указанных событий, прикладная задача может корректировать свои алгоритмы, не заботясь о контроле времени.

Осталось понять, каким образом сделать так, чтобы событие

RE_TIME_EVERY_MINUTE генерировалось не когда-нибудь в течение минуты, а именно в 00 секунд. Для этого я использовал не периодический таймер, а одnorазовый. Затем в конце функции-обработчика таймера вычисляю количество микросекунд до начала следующей минуты, и запускаю его заново:

```
// Calculate the timeout until the beginning of the next minute
gettimeofday(&now_time, nullptr);
localtime_r(&now_time.tv_sec, &now_tm);
uint32_t timeout_us = ((int)60 - (int)now_tm.tm_sec) * 1000000 - now_time.tv_usec;
RE_OK_CHECK(esp_timer_start_once(_schedulerTimerMain, timeout_us), return);
```

Как видите, всё достаточно просто.

Список ссылок

1. [ESP-IDF :: System Time](#)
2. [ESP8266 / Arduino :: time.h](#)
3. [Пример на GitHub](#)

На этом пока всё, до встречи [на сайте](#) и [на dzen-канале](#)!

👍 Понравилась статья? Поддержите канал лайком или комментарием! Каналы на Дзене "живут" только за счет ваших лайков.

✦ Подпишитесь на канал и вы всегда будете в курсе новых статей.

◆ [Полный архив статей вы найдете здесь](#)

Вы можете поддержать канал прямым добровольным пожертвованием

Укажите сумму

₽

Сумма может быть изменена по вашему желанию

Поддержать проект

Создано пользователем с помощью Yandex Forms [Пожаловаться](#)

Благодарю за вашу поддержку! 🙏

публикации

публикации

👍 59

💬 13

➦

🔒

Комментарии 13

Сначала популярные ▾

Написать комментарий

😊

📷

Войти, чтобы комментировать

K12 :: О ESP32 и не только 1 г

Что-то эта статья далась мне с большим трудом...
Больше недели на небольшой текст это слишком 🙄

Ответить 👍 7 🗨

Alexandr Fedorov 9 м

Спасибо. Хорошая статья про внутренние таймеры. Я даже разместил ссылку на своем телеграмм канале. Что бы читатели смогли сравнить все способы реализации. Предлагаю рассмотреть и работу со стандартным Фреймворком Arduino (Arduino IDE) <https://www.theelectronics.co.in/2022/04/how-to-use-internal-rtc-of-esp32.html> Это будет работать и в спящем режиме ESP32. Ну и конечно можно использовать с встроенным ядром Free RTOS

Ответить 👍 1 🗨

Показать 1 ответ

Alexandr Fedorov 1 г

Можно не проверять сертификат или штамп времени сертификата
WiFiClientSecure sslclient;
PubSubClient mqttClient(mqttBroker, 11595, mqttCallback, sslclient);
//Настраиваем Async webserver
еще

Ответить 👍 1 🗨

✎ изменено

Показать 9 ответов

takoedelo · Подписаться

1,1K подписчиков

⋮

UEFI и почему я его так не люблю.

UEFI означает унифицированный расширяемый интерфейс прошивки. Это стандарт для прошивки, который играет роль, которую BIOS (базовая система ввода-вывода...

2 года назад

👍 83

💬 50


➦

🔒

Электронные схемы · Подписаться

63,1K подписчиков

⋮



08:21


Программатор XGecu T48.Как проверить микросхемы серии к155,к561,к555 и другие цифровые
2 недели назад

52

5

Денис Теричев Samodelkin · [Подписаться](#)

147 подписчиков



00:34

Самодельный контроллер для управления вентиляцией на Esp8266
1 неделю назад

2