Arduino and ESP32 Scheduler

Table of Contents

Disclaimer

This SOFTWARE PRODUCT is provided by El Condor "as is" and "with all faults." El Condor makes no representations or warranties of any kind concerning the safety, suitability, lack of viruses, inaccuracies, typographical errors, or other harmful components of this SOFTWARE PRODUCT. There are inherent dangers in the use of any software, and you are solely responsible for determining whether this SOFTWARE PRODUCT is compatible with your equipment and other software installed on your equipment. You are also solely responsible for the protection of your equipment and backup of your data, and El Condor will not be liable for any damages you may suffer in connection with using, modifying, or distributing this SOFTWARE PRODUCT.

You can use this **SOFTWARE PRODUCT freely, if you would you can credit me in program comment:**

El Condor – CONDOR INFORMATIQUE – Turin

Comments, suggestions and criticisms are welcomed: mail to rossati@libero.it

Conventions

Commands syntax, instructions in programming language and examples are with font `COURIER NEW`. The optional parties of syntactic explanation are contained between `[square parentheses]`, alternatives are separated by `|` and the variable parties are in *`italics`*.

1 Introduction

This documentation results on developments of applications using Arduino and ESP32 boards in the Arduino IDE environment.

A characteristic use of this type of card is to control sensors and/or to activate actions by a software which loop indefinitely; below is the skeleton of an Arduino script (called, in the Arduino jargon, sketch):

```
// includes, defines and everywhere accessible variables
void setup() {
  // put your setup code here, to run once:
}
void loop() {
  // put your main code here, to run repeatedly:
}
```

1.1 Background

The reader must have some knowledge on how is organized a program running in the automation cards and the C++ language.

2 Events scheduler

2.1 Event structure

Events are contained in an array, every event contains the information for his handling (the structure is contained in the .h file of the class).

```
typedef void (* fnzPointer) (struct cronStructure);
struct cronStructure {
    uint32_t every;    // every seconds
    uint32_t next;     // next event
    uint32_t duration; // for start and stop events (in seconds)
    int8_t status;     // if 1 bit 1: enabled, bit 2: active, bit 3: active
ending, bit 4: manual, bit 5: at time
    fnzPointer fnzAction;
    const char* name;
};
```

Event's structure

- `every`: is the interval in seconds between two consecutive occurrence of the event, 86400 means a daily event;
- `next`: the time for the next occurrence of the event: when the internal clock become equal or greater of `next`, the associated `action` is called; the initial value of this field means a delay on start or an effective time of day depending by the value of `status` field;
- `duration`: the event duration in seconds; if grater than 0 the function which handle this event is called also when the duration time expires, this permits the control of starting and ending of an action;
- `status`: the event status and type when the value of bits below is 1:
    - bit 1 event enabled
    - bit 2 event active
    - bit 3 event active with duration
    - bit 4 event manual
    - bit 5 event at fixed time
- `action`: the function that handle the event;
- `name`: the event name.

The developer must describe the events, instantiate the class (before the `setup` function) and write the functions for handle the events.

Events can occurs in many way, where the most significant are:

1. at every defined time interval
2. at prefixed time of day
3. on command i.e.. enabled by program

Besides an event can be punctual or with duration.
Other possibilities, like action at the beginning or after a period of time, are all easily achievable as variants of those listed above.

The events must be arranged on array, with a `NULL` event terminator:

```
cronStructure cron[] = {{...},{0,0,0,0,NULL,NULL}};
```

or

```
RTC_DATA_ATTR cronStructure cron[] = {{...},{0,0,0,0,NULL,NULL}};
```

The second form can be used only for ESP32 boards for memorize the events in the memory of ULP co-processor that is preserved during the deep sleep.

The `NULL` event terminator is used for memorize some information (necessary for ESP32 board for to maintain the status of events when awakes after deep sleep):

`duration`	`ag_Clock`	*handleEvents* timer
`next`	`ag_delay`	the delay of the scheduler
`every`	`ag_millis`	the value returned by `millis` function
`status`	It must be `0`, it is set to `1` when *handleEvents* is instantiated	

2.2 How scheduler works

The scheduler works using its own software timer updated by a `millis()` function; this choice is needed because in presence of power reduction techniques the hardware timer is stopped (ATMEL processors) or reinitialized (ESP32 after deep sleep).

The scheduler is invoked in the `loop` function: for every enabled event it controls if the `next` field is equal or lower of the internal clock; in this case the associated function is called whit the event as parameter and the bit 2 `event active` of status is set to 1; at the return of function the `next` field is updated depending on the event and the type of the event:

- punctual events: the `every` field is added to `next` field,
- duration events: at the start the `duration` field is added to the `next` field; at the end the difference from the `every` field and the `duration` field is added to the `next` field.

The `status` field at the return of function is updated depending on the type of the event:

- the bit 1 `event enabled` is set to 0 when the `event manual` (punctual or with duration) is terminated;
- manual commands: the status field is set to `EVENT_ENABLED` if the event is punctual, i.e the `duration` field is `0`,
- punctual events: bit 2 `event active` is set to 0.
- duration events: bit 3 `event active with duration` is st to 1 after the first call; after the second call both bit 2 and 3 are set to 0.

The scheduler returns the time required for the occurrence of the next event, which can be used to delay the processor (or to enter in power reduction).

2.3 Using the code

The class *handleEvent* contains the scheduler. Below there is an example for use the scheduler:

```
...
cronStructure cron[] = {        // the events set
       {...},                   // some events
       {0,0,0,0,NULL,NULL}      // terminator
};
```

```
handleEvents events(cron);      // instantiate the object events
...
events.setTime(day/2);          // possibly set time (at twelve o'clock)
// here the functions for handle events
...
void setup() {
...
// events.begin(cron, pw_Clock); // needed if ESP32 go in deep sleep
...
}
void loop() {
        delay(events.croner(cron));    //  handle Cron Events
}
```

The class `handleEvent` contains the function `croner` which analyzes the array of events and, for events that have reached the time of activation, performs their management function.
The events handler class has a set of constants useful for the status field of the event instance:

`EVENT_ENABLED`	
`EVENT_AT_TIME`	For events occurring at fixed time;
`EVENT_MANUAL`	For events activated by program.
`day`	Is `86400` i.e. the number of seconds per day.

There are two macro for manage events:

`EVENT_ENABLE(`*`event`*`)`
`EVENT_DISABLE(`*`event`*`)`

And also some inline function for test the `status` field:

```
inline bool IS_EVENT_ENABLED(cronStructure a) {return a.status & 1;}
inline bool IS_EVENT_ACTIVE(cronStructure a) {return a.status & 2;}
inline bool IS_EVENT_ENDING(cronStructure a) {return a.status & 4;}
inline bool IS_EVENT_MANUAL(cronStructure a) {return a.status & 8;}
inline bool IS_EVENT_AT_TIME(cronStructure a) {return a.status & 16;}
```

☞ The constant for set the `status` field can be used in `or` (see example below).

```
cronStructure cron[] = {
        {2,0,0,EVENT_ENABLED,(fnzPointer) handle_command,"Serial"},
        {180,10,60,EVENT_ENABLED,(fnzPointer) handle_command,"Pump"},  // starts
10 seconds after program start
        {day*1.5,handleEvents::toSeconds("23:59:30"),60,EVENT_ENABLED|
EVENT_AT_TIME,(fnzPointer) handle_command,"Timed B"},
        {day/12,handleEvents::toSeconds("0:00:30"),60,EVENT_ENABLED|
EVENT_AT_TIME,(fnzPointer) handle_command,"Timed A"},
        {0,0,90,EVENT_MANUAL,(fnzPointer) handle_command,"OnCommand"},   //
start on command end after 90 seconds
        {0,0,0,EVENT_MANUAL,(fnzPointer) handle_command,"pointCmd"},     //
start on command
        {0,0,0,0,NULL,NULL}   // terminator
};
```

2.4 Some functions in class

2.4.1 Time management

`ElapsedTime` **`= clocker(`*`millis`*`)`**

Gives the time elapsed from start of board in seconds if *`millis`* is false otherwise in milliseconds.

The time is stored in an unsigned long variable of 32 bits and maintains also the days elapsed; it can reach 136 years:

`numbers_of_days` `= clocker(false) / day`

```
seconds_from_midnight = clocker(false) % day
```

hhmmssTime = Time()
Returns the actual time in the form `HH:MM:SS+DD` where `DD` are the days after start.

hhmmssTime = Time(seconds)
Returns the *seconds* in the form `HH:MM:SS+DD` where `DD` are the days.

seconds = toSeconds(hh[:mm[:ss]])
Transform the string containing the time in seconds; the time can be partial i.e. seconds and minutes may be missing. In case of wrong values the function returns -1.

☞ The function is declared `static` so it can be invoked without instantiate the class, example:

```
events.setTime(handleEvents::toSeconds("12:30"));
```

2.4.2 Set and modify time

setTime(seconds)
The time should be set before the activation of loop cycle by the `setTime(seconds)` function where *seconds* are the seconds from midnight, for example `setTime(day/2)` set the timer at twelve o'clock; without the setting, the time starts at midnight.
It is possible to change the time mainly for imprecision of the function `millis()`; this may have consequences on the management of events, however, the `setTime` function change the time and rearranges the time of events that aren't at prefixed time, i.e. the events with the `status` bit `EVENT_AT_TIME` equal `0`. However the time correction when the old value and the new are on different days can pose problems; below a table summarizing the possible combinations:

Old time	Correct time	Seconds for correction
today	today	`seconds of today`
today	tomorrow	`seconds of tomorrow + 86400`
tomorrow	today	`seconds of today - 86400`

addTime(seconds)
add seconds to the timer, it is useful when the board sleep doesn't update the clock.

2.4.3 Functions on Events array

listEvents(cronStructure)
Shows the events with the actual status.

searchEvent(event_name)
Returns the position in the events array of *event_name*.

2.5 Events

2.5.1 Order of events

To avoid the overlapping of recurring events, they can be activated in a staggered manner by acting on the `next` field; for example, the following two events are triggered every minute but the second starts delayed by 15 seconds.

```
cronStructure cron[] = {
    {0,0,0,EVENT_ENABLED,(fnzPointer) handle_command,"Serial"},
    {300,00,20,EVENT_ENABLED,(fnzPointer) handle_command,"Pump1"},
    {300,15,30,EVENT_ENABLED,(fnzPointer) handle_command,"Pump2"},
    {0,0,0,0,NULL,NULL}   // terminator
};
```

2.5.2 Event with duration

The events with duration are useful for operating with actions that must be active for a preset time such as the operation of a pump or the start-up of a sensor. At first call of the function the `status` is set to

EVENT_ACTIVE at second call the status is set to EVENT_ACTIVE and also the bit 3 of status is set to 1, this can be tested by the inline function IS_EVENT_ENDING(*event*).

The example below deals with the activation of a device that needs 10 seconds for to be ready.

```
...
{120,50,10,EVENT_ENABLED,(fnzPointer) handle_Condutt,"Siemens"},
...
void handle_Condutt(cronStructure &event) {
  sprintf(workBuffer,"%s %s ",events.Time(),event.name);
  Serial.print(workBuffer);
  if (!IS_EVENT_ENDING(event)) {
    digitalWrite(10,HIGH);     // enabled
    Serial.println("enabled");
  } else {
    int conductibility=(analogRead(4));
    digitalWrite(10,LOW);  // disabled
    ...
  }
}
```

2.5.3 Events at prefixed time

When the EVENT_AT_TIME bit is set the next field indicates the time when the event is activated, the every field indicates the periodicity: for example if contains 86400 (or a day constant) the event occurs daily; he every field can have a multiple or a fraction of day, for example in the fragment below the event Timed A is scheduled every two hours starting at 0:00:30 and the event Timed B is scheduled for a day and half starting at 23:59:30.

```
…
{day*1.5,handleEvents::toSeconds("23:59:30"),60,EVENT_ENABLED|EVENT_AT_TIME,(fnzPointer)
handle_command,"Timed B"},
{day/12,handleEvents::toSeconds("0:00:30"),60,EVENT_ENABLED|EVENT_AT_TIME,(fnzPointer)
handle_command,"Timed A"},
...
```

2.5.4 Repetitive events on limited intervals

The repetitive events which must occurs at prefixed time for a limited time, for example a pump working for one minute every three minutes in the period from twelve o'clock and ending after one hour, can be handled using two events like in the fragment below:

```
...
cronStructure cron[] = {
    {180,10,60,0,(fnzPointer) handle_pumps,"Pump"},
    {day,43200,3600,EVENT_ENABLED|EVENT_AT_TIME,(fnzPointer) handle_StartPump,""},
    ...
    {0,0,0,0,NULL,NULL}   // terminator
};
handleEvents events(cron);
...
void handle_StartPump(cronStructure &event) {
  if (!IS_EVENT_ENDING(event)) {
    EVENT_ENABLE(cron[events.searchEvent("Pump")]);
  } else {
    EVENT_DISABLE(cron[events.searchEvent("Pump")]);
  }
}
void handle_pumps(cronStructure &event) {
  sprintf(workBuffer, "%s %s %s",
events.Time(),event.name,IS_EVENT_ENDING(event)?"ended":"started");
  Serial.println(workBuffer);
```

```
  delay(150);
}
```

It is also possible do this with only one event:

```
...
cronStructure cron[] = {
   {180,43200,60,EVENT_ENABLED|EVENT_AT_TIME,(fnzPointer) handle_StartPump,"Pump"},
    ...
   {0,0,0,0,NULL,NULL}   // terminator
};
handleEvents events(cron);
...
void handle_StartPump(cronStructure &event) {
  static uint32_t long startTime = event.next;
  sprintf(workBuffer, "%s %s %s",
events.Time(),event.name,IS_EVENT_ENDING(event)?"ended":"started");
  Serial.println(workBuffer);
  delay(150);
  if (IS_EVENT_ENDING(event)) {
    if ((events.clocker(false) % day) > day/2+3600) {
            startTime = startTime + day;
            event.next = startTime;
    }
  }
}
```

2.5.5 Start and stop manual events with duration

To start a manual event is done simply by setting his `status` field by `EVENT_ENABLE(`*`event`*`)`; to stop an event active we must force the end by setting the `next` field to the current time: if we set the event by `EVENT_DISABLE(`*`event`*`)` the second call of the management function is not performed.

☞ In any case the scheduler disables the event at the end.

For an immediate execution of the action, the parent event must precede the event controlled.

```
...
cronStructure cron[] = {
   {0,0,30,EVENT_MANUAL,(fnzPointer) handle_command,"OnCommand"},
...
   {0,0,0,0,NULL,NULL}   // terminator
};
handleEvents events(cron);
...
   iEvent = events.searchEvent((char *) "OnCommand");
    if (!IS_EVENT_ENABLED(cron[iEvent])) {
       EVENT_ENABLE(cron[iEvent]);
        Serial.println("OnCommand enabled");
    } else {
        cron[iEvent].next = events.clocker(false);      // force close
        Serial.println("OnCommand disabled");
   }
...
```

2.5.6 Start and stop repetitive events

The repetitive events can be stopped setting the `status` to disabled; the restarting is done by setting the `status` to enabled and setting the `next` event equal or greater of the actual time, see example below.

```
...
iEvent = events.searchEvent((char *) "Temp");
eStatus = cron[iEvent].status;
if (!IS_EVENT_ENABLED(cron[iEvent])) {
    EVENT_ENABLE(cron[iEvent]);
    cron[iEvent].next = events.clocker(false); // force first capture
    Serial.println("Capture enabled");
```

```
} else {
    EVENT_DISABLE(cron[iEvent]);
    Serial.println("Capture disabled");
}
...
```

2.5.7 Other events type

An event which must occurs on starting can be declared normally, and the event must be disabled in the function which handle this, see the example:

```
...
{0,5,0,EVENT_ENABLED,(fnzPointer) handle_cmd,"Scan"}, // 5 seconds after start
...
void handle_cmd(cronStructure &event) {
        EVENT_DISABLE(event);
        Serial.Println("Arduino started");
}
```

3 Appendices

3.1 SimCron a demo sketch

SimCron is a sketch which shows the use of handleEvents with a set of significant events:

- delayed starting event,
- recurrent events,
- timed event whit different frequency,
- manual commands.

The sketch can be managed via serial interface substantially for investigate the status, activate manuals event and change the timer.

3.2 Technical notes

3.2.1 Boards tested

- Arduino UNO
- Arduino MEGA
- ESP32

3.2.2 Internal timer

The scheduler has an internal `timer` (`ag_Clock`) that can be set to time of day by the `setTime` function; both timer and function manage seconds.

When `setTime` is invoked with `t` seconds `ag_Clock` is changed according to the following algorithm:

```
void handleEvents::setTime(uint32_t t) {  // sets the time of day
  uint32_t oldTime = clocker(false);
  *ag_Clock = oldTime - (oldTime % day) + t - millis()/1000; // new time
  for (int8_t i=0;ev[i].name != NULL;i++) {   // adjust events time
      if (!IS_EVENT_AT_TIME(ev[i])) ev[i].next = clocker(false) + ev[i].next -
oldTime;
      else {
        while (ev[i].next < clocker(false)) ev[i].next += ev[i].every;
      }
  }
}

void handleEvents::setTime(uint32_t t) {  // sets the time of day
    uint32_t oldTime = clocker(false);
    *ag_Clock = *ag_Clock - (*ag_Clock % day) + t - millis()/1000; // new time
    // adjust events time
    for (int8_t i=0;ev[i].name != NULL;i++) {
      if (!IS_EVENT_AT_TIME(ev[i])) ev[i].next = clocker(false) + ev[i].next -
     oldTime;
    }
}
```