Annexe 1 – Librairie OBJ_STAT
(Version 2.0.1 du 17 11 2022)

LYCEE EUGENE LIVET NANTES

PRESENTATION

La librairie **obj_stat** permet d'utiliser les graphes d'état dans l'environnement arduino. Pour l'utiliser, les fichiers *obj_stat.h* et *obj_stat.cpp* doivent se trouver soit dans le **répertoire de travail** ou dans le répertoire *obj_stat* du répertoire librairies arduino (Par défaut : C:\Program Files (x86)\Arduino\libraries\obj_stat).

Inclusion

Pour inclure la librairie dans votre projet arduino, vous devez utiliser la directive suivante :

```
#include "obj_stat.h"
```

si les fichiers *obj_stat.h* et *obj_stat.cpp* sont dans votre répertoire de travail et :

```
#include <obj_stat.h>
```

si les fichiers *obj_stat.h* et *obj_stat.cpp* sont dans le répertoire *obj_stat* du répertoire librairies arduino (Par défaut : C:\Program Files (x86)\Arduino\libraries\obj_stat).

Liste des objets

La librairie *obj_stat* contient les classes suivantes :

- **Event** : permet de gérer les entrées numériques (front montant, front descendant, changement d'état).
- **DigitalOut** : permet de gérer les sorties numériques (on, off, on/off pendant une durée donnée, clignotement on/off suivant un durée donnée).
- **Stat** : permet de gérer le fonctionnement d'un graphe d'état.
- **Bistable** : permet de gérer 2 sorties numériques antagonistes comme des contacteurs inverseurs de moteur ou des distributeurs. Par construction, les deux sorties ne peuvent être actives en même temps.
- **Button** : permet de gérer les entrées numériques provenant d'un bouton. Cette classe reprend les événements de Event (front montant, front descendant, changement d'état) et en ajoute d'autres (clic, double clic, appui long).
- **Timer** : permet de gérer les temporisations.
- **Pulse** : permet de générer un top tous les x ms.

Lorsque l'on programme en graphe d'état, il est important de maitriser le temps de cycle de la fonction *loop*. En conséquence il ne faut jamais utiliser la fonction **delay** qui bloque l'exécution du microcontrôleur. Les temporisations doivent absolument être gérées par des timers, soit en créant des objets de la classe *Timer*, soit en utilisant la méthode *after* de la classe *Stat*.

Lien direct vers les objets (CLIQUER SUR LE NOM DE L'OBJET)

Entrées	Sorties	Autres
Event	DigitalOut	Stat
Button	Bistable	Timer
		Pulse

Lien direct vers les fonctions (CLIQUER SUR LE NOM DE LA FONCTION)

transition, completion, when, after

EVENT

Event permet de gérer les entrées numériques (front montant, front descendant, changement d'état).

Event
log
Event(name) setPin(pin,interne_pull_up) rising() falling() change() stat() getStattime() [V1.7.0] update()

- `bool log`

  Permet d'afficher les logs dans le Moniteur série (*false* par défaut).

- `Event(const char * name = "")`

  Constructeur de l'objet. *name* permet de spécifier le nom donnée à l'objet (vide par défaut).

- `void setPin(uint8_t pin, bool interne_pull_up = false)`

  Spécifie le numéro de l'entrée arduino utilisée (*pin*). La fonction gère elle-même le pinMode. *interne_pull_up* permet d'activer la résistance de pull up interne si nécessaire (*false* par défaut).

- `bool rising()`

  Passe à *true* sur un front montant de l'entrée et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool falling()`

  Passe à *true* sur un front descendant de l'entrée et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool change()`

  Passe à *true* sur un front montant ou sur un front descendant de l'entrée et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool stat()`

  Revoie l'état de l'entrée (*HIGH* ou *LOW*)

- `unsigned long getStattime()`

  Revoie la date du dernier changement d'état de l'entrée.

- `bool update()`

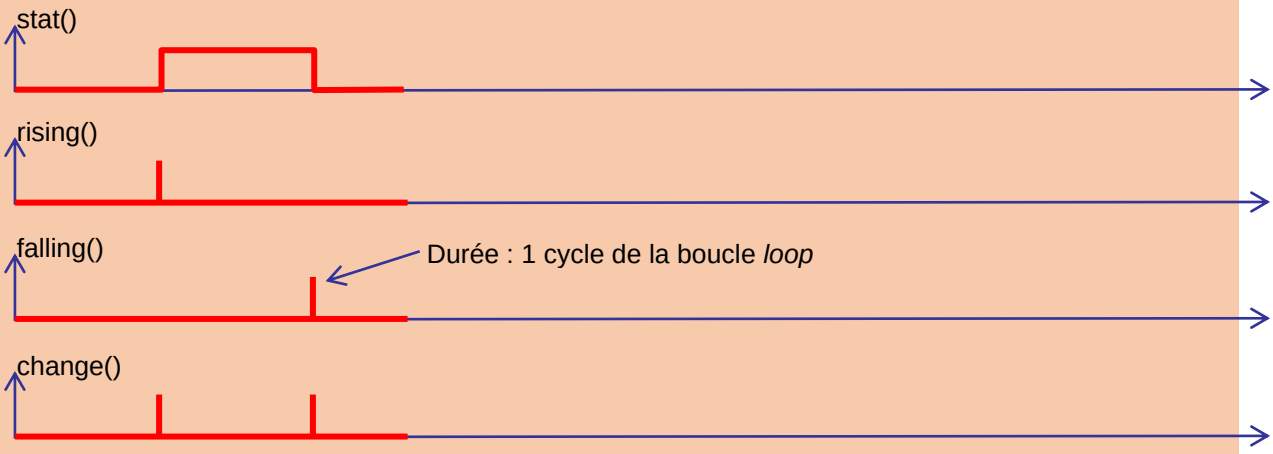
  Méthode à placer OBLIGATOIREMENT à la fin de la fonction *loop* pour actualiser l'état de l'objet. Renvoie *true* si une évolution se produit et *false* sinon.

Exemple de code :

```
//Ce programme affiche front montant ou front descendant dans le moniteur
//série en fonction de l'état du bouton
#include <Arduino.h>
#include "obj_stat.h"

//****************************************
//Hardware
//****************************************
#define EVENEMENT 2          //Câblage d'un bouton poussoir sur la pin 2

//****************************************
//Déclaration des objets
//****************************************
Event evenement("MonEvenement");    //MonEvenement est le nom donnée pour
                                    //l'affichage dans le Moniteur série
```

```
//****************************************
//Setup
//****************************************
void setup() {
  //****************************************
  //Setup hardware
  //****************************************
  evenement.setPin(EVENEMENT);
  //****************************************
  //Port série
  //****************************************
  Serial.begin(9600);
  //****************************************
  //Activation des log
  //****************************************
  evenement.log = true;
}

//****************************************
//Loop
//****************************************
void loop() {

  if (evenement.rising()){
    Serial.println("Front Montant");
  }
  if (evenement.falling()){
    Serial.println("Front Descendant");
  }
  //****************************************
  //Mise à jour
  //****************************************
  evenement.update();
}
```

Chronogrammes :



retour

DIGITALOUT

DigitalOut permet de gérer les sorties numériques (on, off, on/off pendant une durée donnée, clignotement on/off suivant un durée donnée).

DigitalOut
log
DigitalOut(name) setPin(pin,reverse) on(delay_ms) off(delay_ms) set(on_off) toggle() flash(flash_period_ms) change()[V1.5.3] stat() rising() [V1.6.3] falling() [V1.6.3] endOfTimer() [V1.7.0] update()

- `bool log`

  Permet d'afficher les logs dans le Moniteur série (*false* par défaut).

- `DigitalOut(const char * name = ""))`

  Constructeur de l'objet. *name* permet de spécifier le nom donnée à l'objet.

- `void setPin(uint8_t pin, bool reverse = false)`

  Spécifie le numéro de la sortie arduino utilisée (*pin*). La fonction gère elle-même le pinMode. `reverse` permet de renverser la sortie [V1.5.4].

- `void on(unsigned long delay_ms = 0)`

  Passe la sortie à 1 pendant la durée spécifiée. Si *delay_ms* = 0 (par défaut), le *on* est permanent.

- `void off(unsigned long delay_ms = 0)`

  Passe la sortie à 0 pendant la durée spécifiée. Si *delay_ms* = 0 (par défaut), le *off* est permanent.

- `void set(bool on_off)`

  Passe la sortie à 0 ou 1 suivant la valeur de *on_off*.

- `void toogle()`

  Bascule l'état de la sortie.

- `void flash(unsigned long delay_period_ms = 500)`

  Passe la sortie successivement de 0 à 1 tous les x ms (réglé par *delay_period_ms*, 500 ms par défaut).

- `bool change()`

  Passe à *true* sur un changement d'état et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool rising()`

  Passe à *true* sur un front montant de changement d'état et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool falling()`

  Passe à *true* sur un front descendant de changement d'état et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool endOfTimer()`

  Passe à *true* à la fin du timer interne et reste à *true* pendant 1 cycle de la boucle *loop*.

- `bool stat()`

  Renvoie l'état de la sortie (*HIGH* ou *LOW*)

- `void update()`

  Méthode à placer OBLIGATOIREMENT à la fin de la fonction *loop* pour actualiser l'objet.

Exemple de code :

```
//Ce programme gère l'allumage de 2 leds suivant l'état d'un bouton

#include <Arduino.h>
#include "obj_stat.h"          //Si obj_stat.h et obj_stat.cpp sont dans le
                               //répertoire local

//****************************************
//Hardware
//****************************************
#define EVENEMENT 2            //Câblage d'un bouton poussoir sur la pin 2
#define LED1      A0           //Câblage d'une led sur la pin A0
#define LED2      A2           //Câblage d'une led sur la pin A2

//****************************************
//Déclaration des objets
//****************************************
Event evenement("MonEvenement");    //MonEvenement est le nom donnée pour
l'affichage
                                    // dans le Moniteur série
DigitalOut led1("Ma led1");    //Ma led1 est le nom donnée pour l'affichage
                               // dans le Moniteur série
DigitalOut led2("Ma led2");    //Ma led1 est le nom donnée pour l'affichage
                               // dans le Moniteur série
//****************************************
//Setup
//****************************************
void setup() {
  //****************************************
  //Setup hardware
  //****************************************
  evenement.setPin(EVENEMENT);
  led1.setPin(LED1);
  led2.setPin(LED2);
  //****************************************
  //Port série
  //****************************************
  Serial.begin(9600);
  //****************************************
  //Activation des log
  //****************************************
  evenement.log = true;
  led1.log = true;
  led2.log = true;
  //****************************************
  //Initialisation
  //****************************************
  led1.on(2000);               //Allumage de la led1 pour 2 secondes
  led2.on(1000);               //Allumage de la led1 pour 1 secondes
}
//****************************************
//Loop
//****************************************
void loop() {
  if (evenement.rising()) {
    if (led1.stat()==false){
      led1.on();               //Allumage de la led1
      led2.flash(1000);        //Flash tous les 1 secondes de la led2
    }
```

```
    else{
      led1.off();              //Extinction de la led1
      led2.off();              //Extinction de la led2
    }
  }

  //**************************************
  //Mise à jour
  //**************************************
  evenement.update();
  led1.update();
  led2.update();
}
```

retour

STAT

Stat permet de créer des états et de gérer leurs évolutions internes.

Stat
log completion_event
Stat(name,id)
entry_() do_() exit_() after(delay_ms)
stat() interne_stat()
activate() desactivate() resetStattime() getStattime() [V1.7.0]
name() update()

Méthodes

- `Stat(const char * stat_name ="", int stat_id = 0)`

  Constructeur de l'objet. *name* permet de spécifier le nom donné à l'objet et *id* de donner un numéro d'identification.

- `bool log`

  Permet d'afficher les logs dans le Moniteur série (*true* par défaut).

- `bool completion_event`

  Permet de définir l'état comme terminé (*false* par défaut) (cf completion).

- `bool entry_()`

  Renvoie *true* lors de l'entrée dans l'état.

- `bool do_()`

  Renvoie *true* lorsque l'état est actif.

- `bool exit_()`

  Renvoie *true* lors de la sortie de l'état.

Remarque important : dans le code, les méthodes *exit_()* doivent être positionnées en amont des méthodes *entry_()* de manière à respecter la chronologie des évolutions (on sort d'un état avant de rentrer dans le suivant).

- `bool after(unsigned long delay_ms)`

  Renvoie *true* après x ms de l'entrée dans l'état (réglé par *delay_ms*).

- `bool stat()`

  Renvoie *true* lorsque l'état est actif ou en cours d'activation (*entry* ou *do*).

  Remarque important : Attention, *stat()* ne doit **jamais** être utilisé dans une transition immédiatement suivante à l'état, au risque de rendre le système instable. En effet, dans ce cas, l'activité _entry risque de ne jamais être exécutée. Utiliser dans ce cas *intern_stat())*

- `bool interne_stat()`

  Renvoie *true* lorsque l'état est actif (identique à *do_()*).

- `Stat* activate()`

  Permet d'activer l'état. Cette méthode renvoie un pointeur vers l'état.

- `void desactivate()`

  Permet de désactiver l'état.

- `void resetStattime()`

  Permet de remettre à zéro le temps interne de l'état.

- `unsigned long getStattime()`

Revoie la date du dernier changement d'état.

- `String name()`

  Retourne le nom donné à l'état.

- `bool update()`

  Méthode à placer OBLIGATOIREMENT à la fin de la fonction *loop* pour actualiser l'objet. Renvoie *true* si une évolution se produit et *false* sinon.

Fonctions associées

- `bool transition(Stat* &pActive, Stat &from, Stat &to, bool event, bool guard = true)`

Cette fonction permet de spécifier une transition entre deux états. Voici la liste des paramètres :

- `pActive` : pointeur vers l'état actif
- `from` : Etat précédent la transition (de la classe *Stat*)
- `to` : Etat suivant la transition (de la classe *Stat*)
- `event` : événement sous forme d'un booléen. Remarque : il est souhaitable d'utiliser ici les méthodes *rising*, *falling* ou *change* d'un objet de classe *Event* ou les méthodes *click*, *doubleClick* ou *longPress* d'un objet de la classe *Button*.
- `guard` : condition de garde sous forme d'un booléen. Ce paramètre est optionnel et vaut *true* par défaut.

Cette fonction renvoie *true* lorsque la transition est franchie. Ce qui permet d'effectuer une action sur transition si nécessaire.

- `bool completion(Stat* &pActive, Stat &from, Stat &to,bool guard = true)`

Cette fonction permet de spécifier une transition de type *completion* entre deux états. Une transition de ce type est automatiquement franchie lorsque l'état précèdent est déclaré comme terminé (sous réserve de la condition de garde). Pour déclarer un état comme terminé, il faut passer la propriété *completion_event* de l'état précèdent à *true.* Voici la liste des paramètres :

- `pActive` : pointeur vers l'état actif
- `from` : Etat précédent la transition (de la classe *Stat*)
- `to` : Etat suivant la transition (de la classe *Stat*)
- `guard` : condition de garde sous forme d'un booléen. Ce paramètre est optionnel et vaut *true* par défaut.

Cette fonction renvoie *true* lorsque la transition est franchie. Ce qui permet d'effectuer une action sur transition si nécessaire.

- `bool when(Stat* &pActive, Stat &from, Stat &to,bool condition, bool guard = true)`

Cette fonction permet de spécifier une transition de type *when* entre deux états. Voici la liste des paramètres :

- `pActive` : pointeur vers l'état actif
- `from` : Etat précédent la transition (de la classe *Stat*)
- `to` : Etat suivant la transition (de la classe *Stat*)
- `condition` : condition sous forme d'un booléen.
- `guard` : condition de garde sous forme d'un booléen. Ce paramètre est optionnel et vaut *true* par défaut. [V1.6.0]

Cette fonction renvoie *true* lorsque la transition est franchie. Ce qui permet d'effectuer une action sur transition si nécessaire.

- `bool after(Stat* &pActive, Stat &from, Stat &to, unsigned long delay_ms,`

```
      bool guard = true)
```

Cette fonction permet de spécifier une transition de type *after* entre deux états. Voici la liste des paramètres :

- `pActive` : pointeur vers l'état actif
- `from` : Etat précédent la transition (de la classe *Stat*)
- `to` : Etat suivant la transition (de la classe *Stat*)
- `delay_ms` : durée en milliseconde avant le franchissement de la transition.
- `guard` : condition de garde sous forme d'un booléen. Ce paramètre est optionnel et vaut *true* par défaut.

Remarque : Si l'étape est active et le délai est expirée, la transition est franchie dès lors que la condition de garde est vérifiée.

Cette fonction renvoie *true* lorsque la transition est franchie. Ce qui permet d'effectuer une action sur transition si nécessaire.

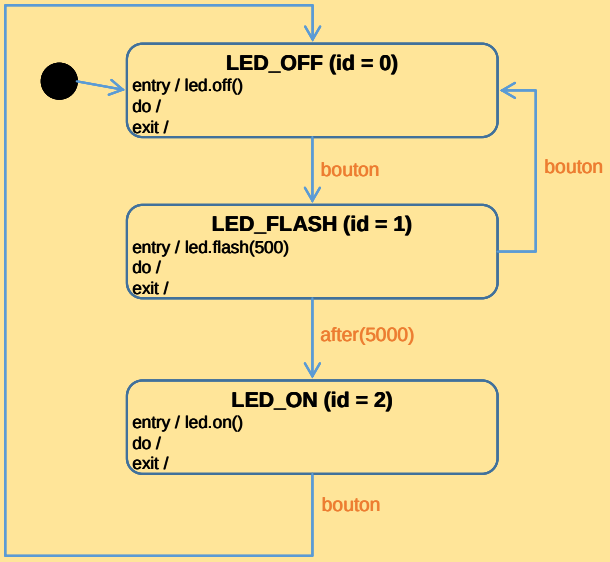


Figure 1 : Graphe d'état exemple 1

Exemple 1, graphe de base

Cet exemple comporte 3 états :

- un état LED_ON où la led est allumée,
- un état LED_OFF où la led est éteinte,
- un état LED_FLASH où la led clignote (500 ms).

L'état LED_OFF est l'état initial. L'appuie sur le bouton poussoir, provoque le passage dans l'état LED_FLASH. 5 secondes plus tard, on passe dans l'état LED_ON. Si entre temps, on appuie sur le bouton, on repasse en LED_OFF. Lorsqu'on est en LED_ON, l'appui sur le bouton permet de repasser en LED_OFF.

La *figure 1* donne le graphe d'état correspondant à ce cahier des charges.

Code associé :

```
#include <Arduino.h>
#include "obj_stat.h"
//****************************************
//Hardware
//****************************************
#define BOUTON 2              //Câblage d'un bouton poussoir sur la pin 2
#define LED    A0             //Câblage d'une led sur la pin A0
//****************************************
//Déclaration des objets
//****************************************
Event bouton ;                      //bouton poussoire
DigitalOut led ;                    //led
Stat LED_OFF("Etat led off",0);      //Etat LED_OFF, id = 0
Stat LED_FLASH("Etat led flash",1); //Etat LED_FLASH, id = 1
Stat LED_ON("Etat led on",2) ;     //Etat LED_ON, id = 2
Stat* pActif ;                  //Pointeur sur l'état actif
//****************************************
//Setup
//****************************************
void setup() {
  //**************************************
  //Setup hardware
```

```
  //**************************************
  bouton.setPin(BOUTON);       //Affectation de la pin à l'objet
  led.setPin(LED);             //Affectation de la pin à l'objet
  //**************************************
  //Port série
  //**************************************
  Serial.begin(9600);
  //**************************************
  //Etat initial
  //**************************************
  pActif = LED_OFF.activate();
}
//**************************************
//Loop
//**************************************
void loop(){
  //Description de l'évolution
  transition(pActif,LED_OFF,LED_FLASH,bouton.rising()); //transition par appui
                                                        // sur le bouton
  after(pActif,LED_FLASH,LED_ON,5000);                  //transition après
                                                        //5000 ms
  transition(pActif,LED_FLASH,LED_OFF,bouton.rising());
  transition(pActif,LED_ON,LED_OFF,bouton.rising());

  //Activités
    //exit (à placer avant les entry)
    //do
    //entry
  if (LED_OFF.entry_()){
    led.off();
  }
  if (LED_FLASH.entry_()){
    led.flash(500);            //changement d'état tout les 500 ms
  }
  if (LED_ON.entry_()){
    led.on();
  }
  //**************************************
  //Mise à jour
  //**************************************
  bouton.update();
  led.update();
  LED_OFF.update();
  LED_FLASH.update();
  LED_ON.update();
}
```

Exemple 2, graphe avec action sur les transitions et condition de garde

Cet exemple comporte 2 états :

- un état LED_ON où la led est allumée,
- un état LED_OFF où la led est éteinte,

L'état LED_OFF est l'état initial. L'appui sur le bouton poussoir, provoque le passage dans l'état LED_ON. Lors du passage de LED_OFF à LED_ON une deuxième led est allumée pendant 2 secondes.

Lorsque l'on est en LED_ON, l'appui sur le bouton permet de repasser en LED_OFF à

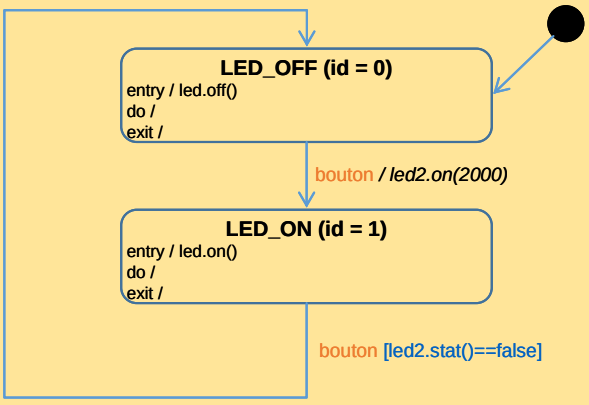


Figure 2 : Graphe d'état exemple 2

condition que la deuxième led soit éteinte (Condition de garde).

En pratique, la première led ne peut donc pas être éteinte tant que la seconde est allumée.

La *figure 2* donne le graphe d'état correspondant à ce cahier des charges.

Code associé :

```
#include <Arduino.h>
#include "obj_stat.h"
//****************************************
//Hardware
//****************************************
#define BOUTON 2                //Câblage d'un bouton poussoir sur la pin 2
#define LED    A0               //Câblage d'une led sur la pin A0
#define LED2   A2               //Câblage d'une led sur la pin A2
//****************************************
//Déclaration des objets
//****************************************
Event bouton ;                  //bouton poussoire
DigitalOut led ;                //led
DigitalOut led2 ;               //led

Stat LED_OFF("Etat led off",0) ;    //Etat LED_OFF, id = 0
Stat LED_ON("Etat led on",1) ;      //Etat LED_ON, id = 2

Stat* pActif ;                  //Pointeur sur l'état actif
//****************************************
//Setup
//****************************************
void setup() {
  //****************************************
  //Setup hardware
  //****************************************
  bouton.setPin(BOUTON);        //Affectation de la pin à l'objet
  led.setPin(LED);              //Affectation de la pin à l'objet
  led2.setPin(LED2);            //Affectation de la pin à l'objet
  //****************************************
  //Port série
  //****************************************
  Serial.begin(9600);
  //****************************************
  //Etat initial
  //****************************************
  pActif = LED_OFF.activate();
}
//****************************************
//Loop
//****************************************
void loop(){
  //Description de l'évolution
  if (transition(pActif,LED_OFF,LED_ON,bouton.rising())){ //transition avec
                                                          //action
    led2.on(2000) ;             //on pendant 2 secondes
  }
  transition(pActif,LED_ON,LED_OFF,bouton.rising(),!led2.stat()); //transition
                                                        //avec condition de garde

  //Activités
    //exit (à placer avant les entry)
    //do
    //entry
  if (LED_OFF.entry_()){
```

```
    led.off();
  }
  if (LED_ON.entry_()){
    led.on();
  }
  //****************************************
  //Mise à jour
  //****************************************
  bouton.update();
  led.update();
  led2.update();
  LED_OFF.update();
  LED_ON.update();
}
```

retour

BISTABLE

Bistable permet de gérer 2 sorties numériques antagonistes comme des contacteurs inverseurs de moteur ou des distributeurs. Par construction, les deux sorties ne peuvent être actives en même temps.

Bistable
log
Bistable(name1,name2) setPin(pin1,pin2) set1(on_off) set2(on_off) stat1() stat2() update()

- `bool log`

  Permet d'afficher les logs dans le Moniteur série (*false* par défaut).

- `Bistable(const char * name1 = "", const char * name2 = "")`

  Constructeur de l'objet. *name1* et *name2* permettent de spécifier le nom donnée aux 2 sorties (vide par défaut).

- `void setPin(uint8_t pin1, uint8_t pin2, bool reverse = false)`

  Spécifie le numéro des sorties arduino utilisées (*pin*). La fonction gère elle-même le pinMode. `reverse` permet de renverser la sortie [V1.5.4].

- `void set1(bool on_off)`

  Passe la sortie 1 à 0 ou 1 suivant la valeur de *on_off*. La sortie 2 est forcée à 0.

- `void set2(bool on_off)`

  Passe la sortie 2 à 0 ou 1 suivant la valeur de *on_off*. La sortie 1 est forcée à 0.

- `bool stat1()`

  Renvoie l'état de la sortie 1 (*HIGH* ou *LOW*)

- `bool stat2()`

  Renvoie l'état de la sortie 2 (*HIGH* ou *LOW*)

- `void update()`

Méthode à placer OBLIGATOIREMENT à la fin de la fonction *loop* pour actualiser l'état de l'objet.

retour

BUTTON

Button permet de gérer les entrées numériques provenant d'un bouton. Cette classe reprend les événements de Event (front montant, front descendant, changement d'état) et en ajoute d'autres (clic, double clic, appui long).

(Cf Event)

- `bool longPressStat()`

  Renvoie *true* pendant toute la durée de l'appui long

- `bool longPressFalling()`

  Renvoie *true* sur le front descendant de l'appui long

Button
`log`

`Button(name)`
`setPin(pin,interne_pull_up)`
`rising()`
`falling()`
`change()`
`click()`
`doubleClick()`
`longPress()`
`longPressStat()``[V1.5.2]`
`longPressFalling()``[V1.5.2]`
`stat()`
`getStattime()` `[V1.7.0]`
`update()`

retour

TIMER

Timer permet de gérer les temporisations.

Timer

`Timer()`
`start()`
`stop()`
`after(delay_ms)`
`stat()`

- `void start()`

  Démarre ou redémarre le timer.

- `void stop()`

  Stop le timer.

- `bool stat()`

  Renvoie l'état du timer.

- `bool after(unsigned long delay_ms)`

  Renvoie *false* si le timer est stoppé ou si la durée n'est pas écoulée. Renvoie *true* si la durée est écoulée.

retour

PULSE

Pulse permet de générer un top tous les x ms.

- `bool every(unsigned long delay_ms)`

  Permet de générer un top tous les x ms réglé par *delay_ms*.

Pulse
`Pulse()` `every(delay_ms)`

retour