UML/SysML Diagramme d'état
Programmation Arduino

Document d'accompagnement

LYCEE EUGENE LIVET NANTES

Historiquement, pour décrire le fonctionnement des systèmes séquentiels et programmer ceux-ci, les solutions classiques consistaient à utiliser des langages comme le Ladder, le Grafcet ou les outils propriétaires des fabricants d'automates.

Avec l'évolution technologique, l'apparition de cartes programmables bon marché (comme les cartes Arduino, Wemos, ESP32 …) et le développement des objets connectés programmables, ces outils se sont révélés inadaptés. Aussi dans la plupart des nouveaux programmes, l'enseignement du Grafcet a été remplacé par un outil plus performant pour décrire le fonctionnement d'un système séquentiel : le diagramme d'état de la norme SysML.

Pour mémoire, cet outil n'est pas nouveau car il vient de la norme UML où il est utilisé pour décrire et coder des programmes informatiques complexes. Des bibliothèques existent dans plusieurs langages pour implanter des graphes d'états dans un programme informatique (par exemple Qt State Machine en C++ ou en Python). En ce qui concerne la programmation des objets connectés, des petits automates et des solutions domotiques basés sur une architecture Arduino, il est pertinent d'utiliser les diagrammes d'état lorsque le caractère séquentiel du projet est fort. Malheureusement, il n'est pas immédiat de traduire un diagramme d'état directement dans le langage Arduino. Quelques solutions commerciales existent, notamment Matlab qui permet de construire un diagramme d'état (State Machine) et de le compiler vers une cible Arduino, mais ces solutions sont souvent incompatibles avec l'environnement Arduino et ses nombreuses librairies développées pour créer des objets de toute sorte.

Aussi la solution proposée par le présent projet vise à développer une librairie permettant d'implanter facilement des diagrammes d'états dans une carte Arduino sans recourir à aucun logiciel externe. Cette librairie nommée **obj_stat** propose une collection de classes permettant d'implémenter un ou plusieurs diagrammes d'états directement dans l'IDE 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).

Dans cet article, nous expliquerons les points fondamentaux nécessaires à la programmation d'un diagramme d'état puis nous aborderons au travers de plusieurs exemples l'utilisation de la librairie.

1 CONCEPTS FONDAMENTAUX

Dans cette partie, nous allons aborder les concepts fondamentaux permettant de programmer un arduino en vue de piloter un système complexe. L'objectif de cette partie est de comprendre les concepts inclus dans la librairie *obj_stat.*

Remarque : bien que la librairie soit développée intégralement en langage objet, il n'est pas nécessaire de connaitre le langage objet pour l'utiliser.

1.1 Notion de temps réel

En informatique, on parle d'un **système temps réel** lorsque ce système est capable de contrôler (ou piloter) un procédé physique à une vitesse adaptée à l'évolution du procédé contrôlé [1]. Les systèmes informatiques temps réel se différencient des autres systèmes informatiques par la prise en compte de contraintes temporelles dont le respect est aussi important que l'exactitude du résultat, autrement dit le système ne doit pas simplement délivrer des résultats exacts, il doit les délivrer dans des délais imposés. On distingue le *temps réel strict* ou *dur* (de l'anglais hard real-time) et le temps réel souple ou mou (en anglais soft real-time) suivant l'importance accordée aux contraintes temporelles [1]:

Le temps réel strict, ne tolère aucun dépassement de ces contraintes, ce qui est souvent le cas lorsque de tels dépassements peuvent conduire à des situations critiques, voire catastrophiques : pilote automatique d'avion, système de surveillance de centrale nucléaire, etc. On peut considérer qu'un système temps réel strict doit respecter des limites temporelles données même dans la pire des situations d'exécution possibles.

Le temps réel souple, s'accommode de dépassements des contraintes temporelles dans certaines limites au-delà desquelles le système devient inutilisable ou pénible : visioconférence, jeux en réseau, etc. Un système temps réel

souple doit respecter ses limites pour une moyenne de ses exécutions. On tolère un dépassement exceptionnel, qui pourra être compensé à court terme.

Un programme Arduino comporte trois parties :

- la partie déclaration des variables,
- la partie initialisation et configuration des entrées/sorties : la fonction **setup ()**, qui est exécutée une fois à la mise sous tension de l'arduino ou après une réinitialisation,
- la partie principale qui s'exécute en boucle : la fonction **loop ().**

Si l'on veut piloter un système industriel, il est donc fondamental que notre programme respecte les contraintes d'un système temps réel. Il faut donc que le temps de cycle de la fonction **loop ()** soit sous contrôle.

Analysons la figure 1, donnant le code permettant de faire clignoter une led toutes les secondes.

```
bool led ;               //Déclaration de la variable led

void setup() {
  pinMode(2, OUTPUT);   //La pin 2 est définie comme une sortie
  led = false ;         //Initialisation de la variable led
  digitalWrite(2,led); //On éteint la led sur la sortie 2
  }

void loop() {

  //Clignotement d'une led (période 1 seconde)
  led = !led;           //On bascule l'état de la variable led
  digitalWrite(2,led); //On allume/éteint la led sur la sortie 2
  delay(500);           //On attend 500 ms
  }
```

Figure 1 : Clignotement d'une led toutes les secondes (exemple 1)

On remarque que la commande *delay* bloque la fonction **loop ()** pendant une durée de 500 ms à chaque cycle. Pendant ces 500 ms, notre arduino est en pause et donc est aveugle à tout ce qui pourrait se passer pendant ce laps de temps. Ce morceau de code, s'il devait être intégré dans un programme plus vaste est donc incompatible avec la notion de temps réel. **Plus précisément, l'instruction delay qui met l'arduino en pause ne peut être utilisée dans un programme temps réel**.

Heureusement, la solution pour contourner le problème est relativement simple. Il suffit de mémoriser à quelle moment a eu lieu le dernier changement d'état (à l'aide de l'instruction *millis())* et de vérifier si 500 ms se sont écoulées depuis celui-ci. Voici le code correspondant :

```
bool led ;                //Déclaration de la variable led
unsigned long stat_time ; //La variable stat_time est utilisée pour mémoriser la date du
dernier changement d'état de la led

void setup() {
  pinMode(2, OUTPUT);    //La pin 2 est définie comme une sortie
  led = false ;          //Initialisation de la variable led
  digitalWrite(2,led);   //On éteint la led sur la sortie 2
  stat_time = millis() ; //Mémorisation de la date du changement d'état
  }

void loop() {
  //Clignotement d'une led (période 1 seconde)
  if ((stat_time-millis())>500) {
    led = !led;          //On bascule l'état de la variable led
    digitalWrite(2,led); //On allume/éteint la led sur la sortie 2
    stat_time = millis() ; //Mémorisation de la date du changement d'état
  }
}
```

Figure 2 : Clignotement d'une led toutes les secondes compatible temps-réel (exemple 2)

On constate :

- que l'instruction *delay* a été remplacée par un test « `if ((stat_time-millis())>500)` »,
- que le temps de cycle de la fonction **loop()** tend vers 0,
- que ce code peut être intégré dans un programme plus vaste sans impacter de manière significative le temps de cycle.

L'idée générale qui se cache derrière cette manière de programmer est le concept de **timer**. Un timer peut être implémenté de manière matérielle (hardware) ou logicielle (software). La figure 2, illustre la manière logicielle.

Pour faciliter, l'utilisation des **timers**, la bibliothèque développée propose une classe **Ctimer** qui permet une approche plus explicite de l'exemple proposé figure 2 avec en particulier le test « `if (timer_led.after(500))` » qui est vrai au bout de 500 ms :

```
#include "obj_stat.h"//Inclusion de la bibliothèque
                     // placée dans le répertoire de travail

bool led ;           //Déclaration de la variable led
Ctimer timer_led ;   //Déclaration d'un objet de classe Ctimer pour gérer la led
void setup() {
  pinMode(2, OUTPUT);    //La pin 2 est définie comme une sortie
  led = false ;          //Initialisation de la variable led
  digitalWrite(2,led);   //On éteint la led sur la sortie 2
  timer_led.start() ;    //On démarre le timer
  }

void loop() {
  //Clignotement d'une led (période 1 seconde)
  if (timer_led.after(500)) { //Vrai après 500 ms depuis le timer_led.start()
    led = !led;               //On bascule l'état de la variable led
    digitalWrite(2,led);      //On allume/éteint la led sur la sortie 2
    timer_led.start() ;       //On redémarre le timer
  }
}
```

Figure 3 : Clignotement d'une led, utilisation d'un timer Ctimer (exemple 3)

L'utilisation de la classe **Cpulse** conçu plus spécifiquement pour ce type de problème permet d'obtenir un code plus compact :

```
#include "obj_stat.h"      //Inclusion de la bibliothèque placée dans le répertoire de
travail

bool led ;                //Déclaration de la variable led
Cpulse pulse_led ;        //Déclaration d'un objet de classe Cpulse pour gérer la led
void setup() {
  pinMode(2, OUTPUT);    //La pin 2 est définie comme une sortie
  led = false ;          //Initialisation de la variable led
  digitalWrite(2,led);   //On éteint la led sur la sortie 2
}

void loop() {
  //Clignotement d'une led (période 1 seconde)
  if (pulse_led.every(500)) { //Vrai toute les 500 ms sur un cycle
    led = !led;               //On bascule l'état de la variable led
    digitalWrite(2,led);      //On allume/éteint la led sur la sortie 2
  }
}
```

Figure 4 : Clignotement d'une led, utilisation d'un timer spécifique Cpulse (exemple 4)

Toutes les 500 ms on bascule l'état de la led (commande « `pulse_led.every(500))` »).

Pour plus d'information sur l'utilisation de **Ctimer** et **Cpulse**, on pourra se reporter à la documentation accompagnant la librairie **obj_stat**.

1.2 Notion de multitâche

On dit qu'un système informatique est multitâche s'il permet d'exécuter, de façon **apparemment** simultanée, plusieurs programmes informatiques. Pour faire du "vrai" multitâche, avec plusieurs tâches exécutées simultanément, il faut avoir plusieurs processeurs pour exécuter plusieurs tâches simultanément, indépendamment les unes des autres, ce qui est impossible avec une carte arduino ou même avec un PC actuel.

La définition de « multitâche » ne précise pas que les tâches doivent être exécutées en parallèle simultanément, mais **apparemment** simultanément. Cela change tout car au lieu d'attribuer une tâche par processeur, on alterne plusieurs tâches sur un même processeur. Chaque tâche est exécutée en séquence, l'une après l'autre suivant un ordre et une logique bien définie. Il y a deux techniques possibles de séquencement : le multitâche coopératif et le multitâche préemptif [2]. Nous n'aborderons pas le multitâche préemptif, trop complexe à mettre en œuvre sur un arduino.

```
#include "obj_stat.h"      //Inclusion de la bibliothèque placée dans le répertoire de
travail

//Déclarations des Tache 1
bool led ;                 //Déclaration de la variable led
Cpulse pulse_led ;         //Déclaration d'un objet de classe Cpulse pour gérer la led

//Déclarations des Tache 2
//rien dans ce cas

//Setup Tache 1
void setup_tache_1() {
  pinMode(2, OUTPUT);     //La pin 2 est définie comme une sortie (led)
  led = false ;           //Initialisation de la variable led
  digitalWrite(2,led);    //On éteint la led sur la sortie 2
}
//Setup Tache 2
void setup_tache_2() {
  pinMode(3, INPUT );     //La pin 3 est définie comme une entrée (bouton poussoir)
  pinMode(4, OUTPUT );    //La pin 2 est définie comme une sortie (led)
  digitalWrite(4,LOW);    //On éteint la led sur la sortie 4
}

//Loop Tache 1
void loop_tache_1() {
  //Clignotement d'une led (période 1 seconde)
  if (pulse_led.every(500)) { //Vrai tous 500 ms sur un cycle
    led = !led;               //On bascule l'état de la variable led
    digitalWrite(2,led);      //On allume/éteint la led sur la sortie 2
  }
}

//Loop Tache 2
void loop_tache_2() {
  if (digitalRead(3)) {        //Si le bouton poussoir est enfoncé
    digitalWrite(4,HIGH);      //On allume la led sur la sortie 4
  }
  else {
    digitalWrite(4,LOW);      //On éteint la led sur la sortie 4
  }
}

//Setup
void setup() {
  setup_tache_1();
  setup_tache_2();
}
//Loop
void loop() {
  loop_tache_1();
  loop_tache_2();
```

Figure 5 : Multitâche coopératif (exemple 5)

Avec le multitâche coopératif, chaque tâche donne la main aux autres tâches dès qu'elle a fini son traitement ou qu'elle passe en attente d'une ressource externe. C'est la forme la plus simple de multitâche. Evidemment, pour que cela fonctionne, il faut que le code contenu dans chaque tâche soit non bloquant [2] ce qui implique de respecter ce qui a été énoncé dans le paragraphe précédent : notion de temps réel.

La figure 5 est un exemple de programme multitâche coopératif composé de 2 tâches. La tâche 1 fait clignoter une led à une période de 1000 ms et la tâche 2 allume une autre led lorsqu'un bouton poussoir est enfoncé. Cette dernière reste allumée tant que le bouton poussoir n'a pas été relâché.

On remarque, qu'il est facile de séparer le code des deux tâches et que la fonction **loop()**, ne contient plus que l'appel des deux fonctions (non bloquantes) **loop_tache_1()** et **loop_tache_2()**. Cette manière de programmer permet donc de clarifier un programme en le découpant en tâches. Cependant, on imagine bien que si les tâches ont des interactions entre elles, le code va vite devenir complexe à développer et à comprendre.

Pour nos systèmes complexes à piloter, l'idée est donc d'utiliser un outil de formalisation plus puissant qui s'appuie sur le multitâche coopératif mais qui encapsule cette notion. Une des solutions est d'utiliser le diagramme d'état.

2 CODAGE D'UN DIAGRAMME D'ETAT

Remarque préliminaire : Pour aborder le diagramme d'état d'un point de vue programmation, nous supposerons les normes SysML / UML sur le diagramme d'état connues [3], [4]. La norme SysML [3] présente de manière très succincte le fonctionnement des diagrammes d'états (State Machine) car cette norme est issue de la norme UML [4] qui présente en détail ceux-ci. Aussi, pour aborder sereinement les diagrammes d'états, il est important de se reporter à la norme UML [4] et de ne pas se contenter de la norme SysML qui manque parfois de précisions. SysML est définie explicitement dans la norme comme une extension d'UML.

2.1 Approche basique du diagramme d'état

Il est tout à fait possible de coder un diagramme d'état en C/C++ sans recourir à un logiciel tiers ou à une librairie. Pour cela on peut utiliser la structure *switch/case* comme dans l'exemple ci-dessous. Dans cet exemple, nous avons un diagramme d'un système comportant deux états **Marche** et **Arrêt** et 2 boutons poussoirs ***bp_on*** et ***bp_off***. La figure 6 présente le diagramme et la figure 7 le code correspondant.

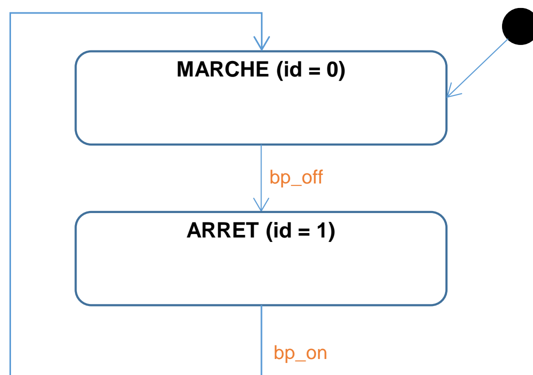


Figure 6 : Diagramme simple à 2 états

L'approche présentée ici est l'approche classique pour coder des machines à états. Il y a autant d'items dans le *switch* qu'il y a d'états et chaque item se termine par un test permettant d'évaluer la transition vers l'état suivant (cf figure 7). On peut, en allant un peu plus loin et en suivant la même logique intégrer les *entry*, *do* et *exit* caractéristiques du diagramme d'état. On remarquera que le code présenté ne gère pas correctement les événements associés à l'appui sur les boutons poussoirs. En effet, si les 2 boutons sont enfoncés, le comportement devient instable ce qui ne devrait pas être le cas. Nous verrons un peu plus loin comment régler ce problème.

On comprend bien que cette approche, bien que correcte, va rapidement conduire à un code complexe dès lors que nous voulons intégrer toutes les spécificités liées aux diagrammes d'états. La librairie proposée permet une gestion automatique du fonctionnement externe (gestion des transitions) et interne (gestion des *entry*, *do* et *exit*) et des événements du diagramme d'état de manière à ce que le codage se réduise à la partie descriptive du

diagramme d'état. La figure 8 présente le code associé au diagramme proposé figure 6 en utilisant la classe **CStat** de la librairie.

```
//Déclarations
bool bp_on ;                  //variable associée au bouton poussoir on
bool bp_off ;                 //variable associée au bouton poussoir off
enum Etat { MARCHE, ARRET }; //Déclaration des états possibles
Etat etatCourant ;            //variable associée à l'état courant

//Setup
void setup() {
  pinMode(3, INPUT );     //La pin 3 est définie comme une entrée (bouton poussoir)
  pinMode(4, INPUT );     //La pin 4 est définie comme une entrée (bouton poussoir)
  etatCourant = MARCHE ; //Etat initial
}
//Loop
void loop() {
  //Mise à jour des entrées
  bp_on = digitalRead(3) ;
  bp_off = digitalRead(4) ;
  //Gestion de l'évolution des états
  switch(etatCourant) {
    case MARCHE :
      // Action état Marche
      //...

      //Transition vers l'état ARRET
      if (bp_off) {
        etatCourant = ARRET ;
      }
      break;

    case ARRET :
      // Action état Arrêt
      //...

      //Transition vers l'état MARCHE
      if (bp_on) {
        etatCourant = MARCHE ;
      }
      break;
  }
```

Figure 7 : Codage d'un diagramme simple à 2 états (exemple 6)

```
#include "obj_stat.h"     //Inclusion de la bibliothèque placée dans le répertoire de travail

//Déclarations
bool bp_on ;              //variable associée au bouton poussoir on
bool bp_off ;             //variable associée au bouton poussoir off
Cstat Marche ;            //variable de type CStat associée à l'état Marche
Cstat Arret ;             //variable de type CStat associée à l'état Arret

Cstat * p_etatCourant ;  //pointeur associé à l'état courant

//Setup
void setup() {
  pinMode(3, INPUT );     //La pin 3 est définie comme une entrée (bouton poussoir)
  pinMode(4, INPUT );     //La pin 4 est définie comme une entrée (bouton poussoir)
  p_etatCourant = Marche.activate() ; //Etat initial
}
//Loop
void loop() {
  //Mise à jour des entrées
  bp_on = digitalRead(3) ;
  bp_off = digitalRead(4) ;
  //Gestion de l'évolution des états
  transition(p_etatCourant,Marche,Arret,bp_off);
  transition(p_etatCourant,Arret,Marche,bp_on);
  //Gestion automatique de l'évolution du diagramme
  Marche.update();
  Arret.update();
`
```

Figure 8 : Codage d'un diagramme simple à 2 états avec la librairie (exemple 7)

Dans ce code, les états Marche et Arrêt (ligne 6 et 7, « `Cstat Marche ;` ») sont déclarés explicitement comme états de la classe **Cstat**, l'activation de l'état initial se fait par la commande « `Marche.activate()` » dans le **setup()**. Les transitions entre les états sont gérées par la fonction **transition()** qui prend comme argument :

- le pointeur vers l'état actif,
- l'état précédent,
- l'état suivant,
- l'événement,
- éventuellement, la condition de garde si il y en a une.

Toute la gestion du diagramme est réalisée par les 2 instructions « `Marche.update();` » et « `Arret.update();` ». Le code de la fonction **loop()**, seulement descriptif est beaucoup plus clair et compact que le code de la figure 7.

2.2 Gestion des activités internes à chaque état

Le fonctionnement interne (gestion des *entry*, *do* et *exit*) à chaque état est géré automatiquement par la classe **Cstat**. Pour mémoire en SysML, l'activité *entry* est exécutée une seule fois quand on rentre dans l'état, l'activité *do* est exécutée en continu tant que l'état est actif et l'activité *exit* est exécutée une fois à la sortie de l'état.

Dans l'exemple de la figure 6, si on souhaite associer du code à l'activité *entry* de l'état **Marche**, il suffit d'ajouter dans la fonction **loop()** le bloc de code suivant :

```
//Code à placer dans la fonction loop()
if (Marche.entry_()) {
  //Mettre ici le code à exécuter à l'entrée dans l'état marche
  digitalWrite(5,true);      //On allume la led sur la sortie 5 par exemple
}
```

La méthode `entry_()` renvoie `true` lorsqu'on rentre dans l'état **Marche**. Les méthodes `do_()` et `exit_()` fonctionnent de la même manière.

Remarque importante : On peut placer les blocs *entry*, *do* et *exit* où on veut dans la fonction **loop()**, en revanche il est important de les grouper par type (tous les blocs *entry* doivent être groupés ensembles, tous les blocs *exit* doivent être groupés ensembles…) et surtout les blocs *exit* doivent se trouver avant les blocs *entry* dans le code. En effet, lors du franchissement d'une transition, il faut que le code associé à l'état qui est désactivé (*Etat_precedent.exit_()*) soit exécuté avant le code associé à l'état qui est activé (*Etat_suivant.entry_()*). Le non-respect de cette règle peut conduire à une inversion de chronologie provoquant inévitablement des effets indésirables au franchissement des transitions.

2.3 Notion d'événement

Un événement est quelque chose qui se produit pendant l'exécution d'un système et qui mérite d'être modélisé. Les diagrammes d'états-transitions permettent justement de spécifier les réactions d'une partie du système à des événements discrets. Un événement se produit à un instant précis et est dépourvu de durée (pour le modèle théorique). Quand un événement est reçu, une transition peut être déclenchée et faire basculer le système dans un nouvel état [6].

La principale force du diagramme d'état relativement aux outils plus anciens comme le Grafcet ou les réseaux de Pétri réside dans cette notion d'évènement. En effet, dans un diagramme d'état, le franchissement d'une transition est OBLIGATOIREMENT associé à un événement. Un événement correspond par exemple au front montant d'une variable binaire associé à un bouton poussoir, la réception d'un message, l'échéance d'un timer… Cela peut paraitre anecdotique, mais cela permet de régler la plupart des instabilités qui peuvent apparaitre dans un système à états. En effet, deux évènements ne peuvent par nature se produire simultanément s'ils ne sont pas corrélés.

Comme évoqué au paragraphe 2.1, le code proposé figure 7 et 8, n'est pas conforme à l'obligation d'associer un événement à une transition. En effet, le code présenté ne gère pas correctement les événements associés à l'appui sur les boutons poussoirs. Si les 2 boutons restent enfoncés, le comportement devient instable ce qui ne devrait pas être le cas d'un point de vue du modèle. Pour régler le problème, il faut mettre comme argument de la fonction *transition()* non pas l'état du bouton poussoir mais l'action sur celui-ci (front montant), ce qui implique décrire le bout de code permettant de détecter les fronts. Ici encore, on peut le faire directement en C/C++ mais la bibliothèque proposée permet de gérer cela facilement via la classe **Cevent**.

Le code suivant traduit le comportement correct du diagramme d'état proposé figure 6, même dans le cas ou un des boutons poussoir reste enfoncé. Pour générer un nouvel événement, il faudra obligatoirement relâcher le bouton avant de le réenfoncer.

```
#include "obj_stat.h"     //Inclusion de la bibliothèque placée dans le répertoire de travail

//Déclarations
Cevent bp_on ;            //variable associée au bouton poussoir on
Cevent bp_off ;           //variable associée au bouton poussoir off
Cstat Marche ;            //variable de type CStat associée à l'état Marche
Cstat Arret ;             //variable de type CStat associée à l'état Arret

Cstat * p_etatCourant ;   //pointeur associé à l'état courant

//Setup
void setup() {
  bp_on.setPin(3);      //La pin 3 est définie comme une entrée (bouton poussoir)
  bp_off.setPin(4);     //La pin 4 est définie comme une entrée (bouton poussoir)
  p_etatCourant = Marche.activate() ; //Etat initial
}
//Loop
void loop() {
  //Mise à jour des entrées
  bp_on.update();
  bp_off.update();
  //Gestion de l'évolution des états
  transition(p_etatCourant,Marche,Arret,bp_off.rising());
  transition(p_etatCourant,Arret,Marche,bp_on.rising());
  //Gestion automatique de l'évolution du diagramme
  Marche.update();
  Arret.update();
}
```

Figure 9 : Codage d'un diagramme simple à 2 états conforme (exemple 8)

Si on compare le code figure 8 et 9, on constate peu de différences. Les variables *bp_on* et *bp_off* sont maintenant de classe **Cevent** (**bool** sur figure 8), l'association des pins se fait par `bp_on.setPin(3)` au lieu de `pinMode(3, INPUT)` et la mise à jour des entrées par `bp_on.update()` au lieu de `bp_on = digitalRead(3)`.

La différence fondamentale est dans l'appel des fonctions transitions où on passe maintenant `bp_on.rising()` (front montant de bp_on) comme argument ce qui n'était évidemment pas possible avec un booléen.

Les événements possibles gérés par la classe **Cevent** sont :

- `rising()` : front montant,
- `falling()` : front descendant,
- `change()` : front montant ou descendant,

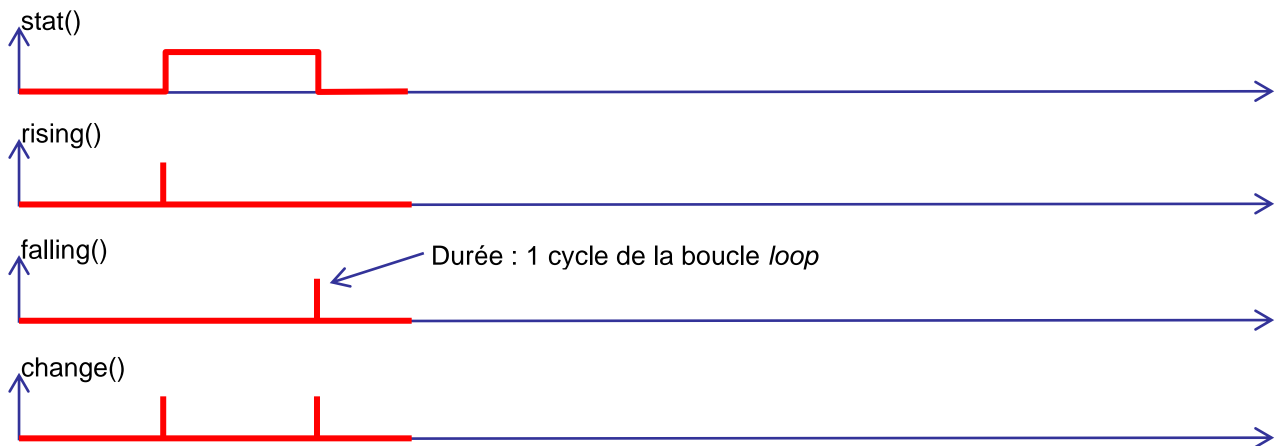


Figure 10 : Chronogramme d'un évènement de classe Cevent

La classe **Cbutton** est construite sur la base de **Cevent** mais est spécifiquement prévue pour les boutons poussoirs et intègre les événements supplémentaires suivants :

- `click()` : front montant suivi d'un front descendant rapprochés (durée paramétrable),
- `doubleClick()` : deux clics à la suite,
- `longPress()` : appui long (durée paramétrable),

La documentation associée à la librairie, détaille les méthodes des différentes classes **Cstat**, **Cevent**, **Cbutton**.

2.4 Les différentes transitions

En SysML/UML, 4 types de transitions sont possibles :

- les transitions sur un évènement externe (c'est le cas classique),
- les transitions sur un évènement de durée (after),
- les transitions sur un évènement associé à une condition booléen (when),
- les transitions sur un évènement interne (fin d'activité).

2.4.1 transition sur un évènement externe

Pour utiliser ce type de transition, il suffit d'appeler la fonction **transition()** avec les arguments suivants (cf exemple figure 8 et 9) :

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

Voici la liste des paramètres :

- `pActive` : pointeur vers l'état actif
- `from` : Etat précédent la transition (de la classe *Cstat*)
- `to` : Etat suivant la transition (de la classe *Cstat*)
- `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 *Cevent* ou les méthodes *click*, *doubleClick* ou *longPress* d'un objet de la classe *Cbutton*.
- `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.

2.4.2 transition sur un évènement de durée

```
bool after(Cstat* &pActive, Cstat &from,Cstat &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 *Cstat*)
- `to` : Etat suivant la transition (de la classe *Cstat*)
- `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.

Remarques :

- Si l'étape est active et le délai est expiré, la transition est franchie dès lors que la condition de garde est vérifiée.
- La documentation de la bibliothèque présente un exemple complet utilisant les fonctions **transition()** et **after()**.

2.4.3 transition sur un évènement associé à une condition booléen (à partir de V1.5.0 du 05/05/2020)

```
bool when(Cstat* &pActive, Cstat &from,Cstat &to, bool condition)
```

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 *Cstat*)
- `to` : Etat suivant la transition (de la classe *Cstat*)
- `condition` : condition sous forme d'un booléen.

2.4.4 transition sur un évènement interne (completion transition) (à partir de V1.5.1 à venir)

En SysML/UML il est possible de faire apparaitre une transition sans évènement associé. En réalité, la franchissement de la transition est conditionné par la fin d'activité associée à l'état (méthode `do_()` de la classe **Cstat**). Ce type de transition s'écrit grâce à la fonction **completion()** :

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

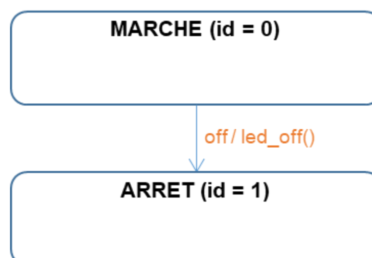
Voici la liste des paramètres :

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

Pour déclencher, le franchissement de cette transition, il suffit de passer la propriété `completion_event` à *true* dans la méthode `do_()` de l'état précèdent la transition (cf exemple 10 proposé figure 12).

2.5 Action ou activité sur transition

Toutes les fonctions de transitions renvoient *true* lorsque la transition est franchie. Cela permet d'effectuer une action sur transition si nécessaire. Ainsi par exemple, la transition entre 2 états **Marche** et **Arrêt**, associée à l'événement *off*, qui aurait l'action *led_off()* à effectuer lors du franchissement de la transition qui s'écrit en SysML par :



Se traduit par le code suivant :

```
if (transition(p_etatCourant,Marche,Arret,off.rising()) {
  led_off() ;
}
```

3 CAS PARTICULIERS

3.1 Pointeur vers l'état actif et état composite

Comme nous avons pu le voir dans les exemples précédents les fonctions de transitions (transition, after, when et completion) prennent comme premier argument un pointeur vers l'état actif. Ce pointeur permet d'accéder directement à l'état actif. Dans les exemples simples, cela ne présente pas beaucoup d'intérêt, en revanche lorsque nous voulons modéliser un état composite, le pointeur permet lors de la désactivation de l'état composite de désactiver le sous état. La figure 11 présente un exemple simple d'état composite.

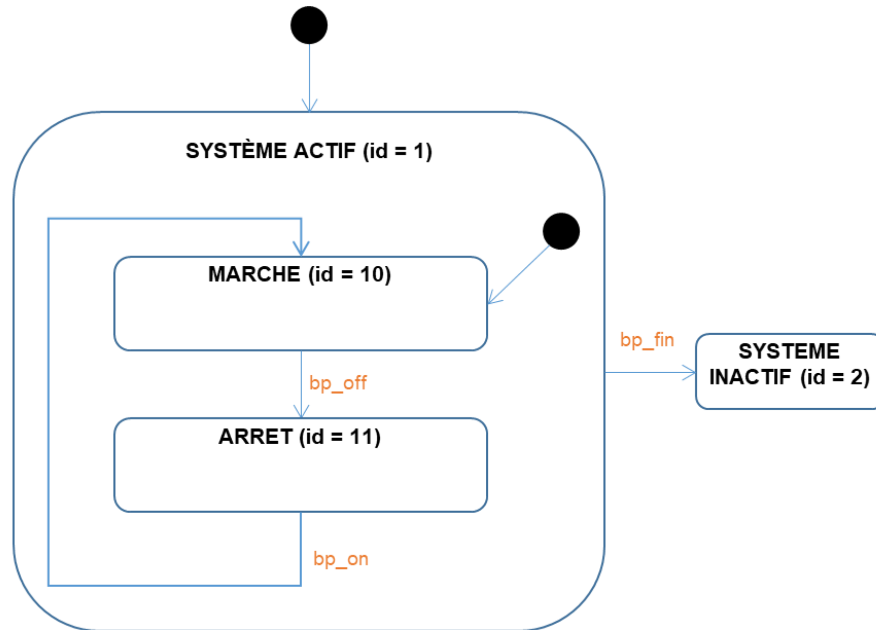


Figure 11 : Etat composite

Voici le code complet correspondant à la figure 11 :

```
#include "obj_stat.h"     //Inclusion de la bibliothèque placée dans le répertoire de travail

//Déclarations
Cevent bp_on ;            //variable associée au bouton poussoir on
Cevent bp_off ;           //variable associée au bouton poussoir off
Cevent bp_fin ;           //variable associée au bouton poussoir fin
Cstat Systeme_actif ;     //variable de type CStat associée à l'état Système Actif
Cstat Systeme_inactif ;   //variable de type CStat associée à l'état Système Inactif
Cstat Marche ;            //variable de type CStat associée au sous état Marche de l'état Système
Actif
Cstat Arret ;             //variable de type CStat associée au sous état Arret de l'état Système Actif

Cstat * p_etatCourant ;   //pointeur associé à l'état courant
Cstat * p_etatCourant_1 ;//pointeur associé au sous état courant de l'état 1 Système Actif

//Setup
void setup() {
  Serial.begin(115200) ; //Activation port série pour visualisation
  bp_on.setPin(3) ;      //La pin 3 est définie comme une entrée (bouton poussoir)
  bp_off.setPin(4) ;     //La pin 4 est définie comme une entrée (bouton poussoir)
  bp_fin.setPin(5) ;     //La pin 5 est définie comme une entrée (bouton poussoir)
  p_etatCourant = Systeme_actif.activate() ; //Etat initial
}
//Loop
void loop() {
  //Mise à jour des entrées
  bp_on.update() ;
  bp_off.update() ;
  bp_fin.update() ;
  //Gestion de l'évolution des états
  transition(p_etatCourant,Systeme_actif,Systeme_inactif,bp_fin.rising());
  transition(p_etatCourant_1,Marche,Arret,bp_off.rising());
  transition(p_etatCourant_1,Arret,Marche,bp_on.rising());
  //Gestion de l'état composite
  if (Systeme_actif.exit_()) {
    p_etatCourant_1->desactivate() ; //Désactivation de l'état actif
  }
  if (Systeme_actif.entry_()) {
    p_etatCourant_1 = Marche.activate() ; //Sous état initial
  }
  //Gestion automatique de l'évolution du diagramme
  Systeme_actif.update();
  Systeme_inactif.update();
  Marche.update();
  Arret.update();
}
```

Figure 12 : Codage d'un diagramme avec état composite (exemple 9)

Pour gérer l'état composite 1, il nous faut un deuxième pointeur (`p_etatCourant_1` pour gérer ce qui se passe dans l'état). Lors de l'activation de l'état 1 (**Système actif**), pour activer le sous état **Marche**, il nous suffit d'écrire :

```
if (Systeme_actif.entry_()) {
  p_etatCourant_1 = Marche.activate() ; //Sous état initial
}
```

De la même manière, lorsque l'état 1 se désactive, il nous faut désactiver le sous état **Marche** ou le sous état **Arrêt.** Voici la commande :

```
if (Systeme_actif.exit_()) {
  p_etatCourant_1->desactivate() ; //Désactivation de l'état actif
}
```

On remarque que `p_etatCourant_1` nous donne un accès direct à l'état actif.

3.2 Gestion des pseudo-états

3.2.1 Etats initiaux et finaux

Les pseudo-états permettent, en particulier, de décrire les états initiaux et finaux d'un système ou d'un état composite. D'une manière générale, dans les cas simples, il n'est pas nécessaire de les créer explicitement dans le programme Arduino. En effet, dans l'exemple donnée figure 6, on active directement l'état **Marche** dans la fonction **setup()** par la commande *Marche.activate().* De la même manière, dans l'exemple donnée figure 11, l'activation du sous-état **Marche** est réalisée par la méthode **entry_()** de l'état composite **Système_actif**.

Dans certains cas, il est nécessaire de créer explicitement le pseudo-état, en particulier les pseudos états de fin de régions orthogonales. Ce point est détaillé au paragraphe 3.3. Dans ce cas, il suffit de créer le pseudo-état comme s'il s'agissait d'un état normal en veillant bien à ne pas attribuer d'activité `do_()`.

3.2.2 Points de jonction et de décision

Les points de jonction sont un artefact graphique (un pseudo-état en l'occurrence) qui permet de partager des segments de transition, l'objectif étant d'aboutir à une notation plus compacte ou plus lisible des chemins alternatifs [6]. Il n'est pas nécessaire de créer le pseudo-état de jonction explicitement, en effet un pseudo-état n'ayant pas vocation à rester actif, la structure *if, then, else* associée aux différentes transitions (*transition(), after(), when(), completion()*) permettent de gérer les différentes situations possible. Il en est de même pour les points de décision. Un exemple est proposé sur le portail GTI.

3.3 Régions orthogonales

Un état orthogonal est un état composite comportant plus d'une région, chaque région représentant un flot d'exécution. Graphiquement, dans un état orthogonal, les différentes régions sont séparées par un trait horizontal en pointillé allant du bord gauche au bord droit de l'état composite. Toutes les régions concurrentes d'un état composite orthogonal doivent atteindre leur état final pour que l'état composite soit considéré comme terminé [6].

Pour le codage Arduino, il est nécessaire de créer explicitement les pseudo-états de fin de chaque région, de manière à placer chaque région terminée en position d'attente. Pour illustrer le propos, nous allons prendre l'exemple de la figure 12 (code figure 13). Dans l'extrait de code qui suit, les pseudo-états sont nommés **FinRegion1** et **FinRegion2**.

Grace à la méthode `do_()` de l'état composite **Systeme_actif**, nous allons pouvoir vérifier si les états **FinRegion1** et **FinRegion2** ont été atteints :

```
if (Systeme_actif.do_()) {
  if (FinRegion1.stat()&&FinRegion2.stat()) {
    Systeme_actif.completion_event = true ;
  }
}
```

Si oui, on déclare l'état composite **Systeme_actif** comme terminé grâce à l'instruction : `Systeme_actif.completion_event = true` , ce qui va provoquer le franchissement de la transition `completion(p_etatCourant,Systeme_actif,Systeme_inactif);`.

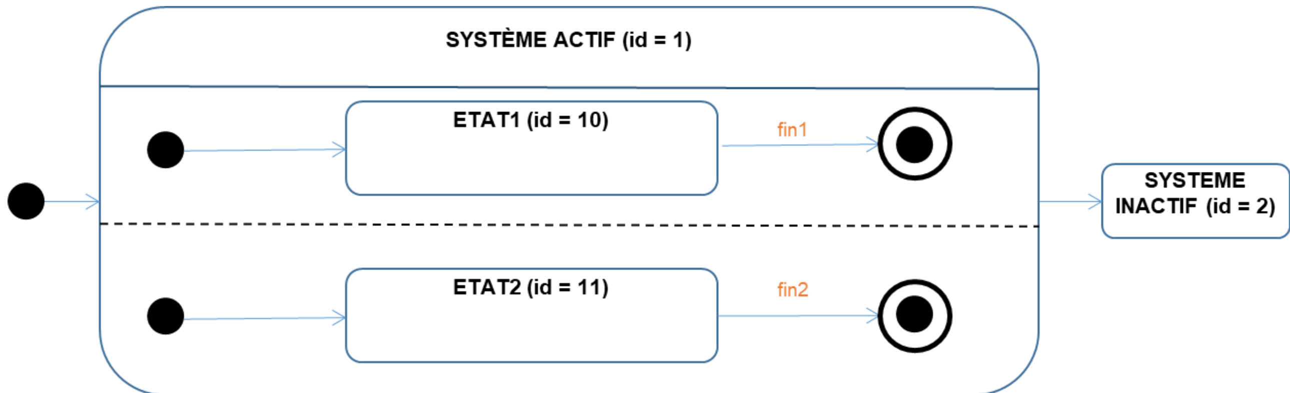


Figure 13 : Régions orthogonales (exemple 10)

```
#include "obj_stat.h"     //Inclusion de la bibliothèque placée dans le répertoire de travail

//Déclarations
Cevent fin1 ;            //variable associée à l'entrée fin1
Cevent fin2 ;            //variable associée à l'entrée fin2
Cstat Systeme_actif ;    //variable de type CStat associée à l'état Système Actif
Cstat Systeme_inactif ;  //variable de type CStat associée à l'état Système Inactif
Cstat Etat1 ;            //variable de type CStat associée au sous état Etat1 de l'état Système Actif
(Région 1)
Cstat Etat2 ;            //variable de type CStat associée au sous état Etat2 de l'état Système Actif
(Région 2)
Cstat FinRegion1 ;       //variable de type CStat associée au pseudo-état de fin de la Région 1
Cstat FinRegion2 ;       //variable de type CStat associée au pseudo-état de fin de la Région 2

Cstat * p_etatCourant ;        //pointeur associé à l'état courant
Cstat * p_etatCourant_1_R1 ;  //pointeur associé au sous état courant de l'état 1 Système Actif
(Région 1)
Cstat * p_etatCourant_1_R2 ;  //pointeur associé au sous état courant de l'état 1 Système Actif
(Région 2)

//Setup
void setup() {
  Serial.begin(115200) ; //Activation port série pour visualisation
  fin1.setPin(3) ;       //La pin 3 est définie comme une entrée (bouton poussoir par exemple)
  fin2.setPin(4) ;       //La pin 4 est définie comme une entrée  (bouton poussoir par exemple)
  p_etatCourant = Systeme_actif.activate() ; //Etat initial
}
//Loop
void loop() {
  //Mise à jour des entrées
  fin1.update() ;
  fin2.update() ;
  //Gestion de l'évolution des états
  completion(p_etatCourant,Systeme_actif,Systeme_inactif);
  transition(p_etatCourant_1_R1,Etat1,FinRegion1,fin1.rising());
  transition(p_etatCourant_1_R2,Etat2,FinRegion2,fin2.rising());
  //Gestion de l'état composite
  if (Systeme_actif.exit_()) {
    p_etatCourant_1_R1->desactivate() ; //Désactivation de l'état actif région 1
    p_etatCourant_1_R2->desactivate() ; //Désactivation de l'état actif région 2
  }
  if (Systeme_actif.do_()) {
    if (FinRegion1.stat()&&FinRegion2.stat()) { //Si tous les états finaux sont atteints,
      Systeme_actif.completion_event = true ;   //on déclare l'état composite comme terminé
    }
  }
  if (Systeme_actif.entry_()) {
    p_etatCourant_1_R1 = Etat1.activate() ; //Sous état initial région 1
    p_etatCourant_1_R2 = Etat2.activate() ; //Sous état initial région 2
  }
  //Gestion automatique de l'évolution du diagramme
  Systeme_actif.update();
  Systeme_inactif.update();
  Etat1.update();
  Etat2.update();
  FinRegion1.update();
  FinRegion2.update();
}
```

Figure 14 : Codes régions orthogonales (exemple 10)

3.4 Pseudo-état historique simple et pseudo-état historique profond

Un état historique, également qualifié d'*état historique plat*, est un pseudo-état qui mémorise le dernier sous-état actif d'un état composite. Graphiquement, il est représenté par un cercle contenant un *H*.

Une transition ayant pour cible l'état historique est équivalente à une transition qui a pour cible le dernier état visité de l'état englobant. Un état historique peut avoir une transition sortante non étiquetée indiquant l'état à exécuter si la région n'a pas encore été visitée.

Il est également possible de définir un *état historique profond* représenté graphiquement par un cercle contenant un *H**. Cet état historique profond permet d'atteindre le dernier état visité dans la région, quel que soit son niveau d'imbrication, alors que le l'état historique plat limite l'accès aux états de son niveau d'imbrication. [6]

Pour gérer les pseudo-états historiques, cela est relativement simple en utilisant le pointeur vers l'état actif. En effet, lorsque l'on sort d'un état composite, le pointeur vers le sous-état actif pointe toujours vers le dernier sous-état actif, bien que celui-ci ait été désactivé. Lorsque l'on réactive l'état composite, il suffit de réactiver le sous état à l'aide de la commande `pointeur_sur_sous_etat->activate()` placé dans l'activité `entry_()` de l'état composite. Pour savoir, si la région a déjà été visitée, on peut initialiser le pointeur `pointeur_sur_sous_etat` à 0 en début de fonctionnement, ce qui permet par un simple test de savoir s'il faut initialiser la région à l'aide du pseudo-état initial ou à partir du pseudo-état historique :

```
if (Mon_etat_composite.entry_()) {
  if (pointeur_sur_sous_etat == 0) {
    Sous_etat_initial.activate() ; //La région n'a pas encore été visitée, on active l'état
                                   // inital
  }
  else {
    pointeur_sur_sous_etat->activate() ; //La région a déjà été visitée, on active l'état
                                         // mémorisé
  }
}
```

3.5 Nommage des objets et affichage des log

De manière à présenter dans ce document un code minimal, les objets de classe **Cstat, Cevent, Cbutton** n'ont pas été renseignés totalement, ce qui ne pose aucun problème de fonctionnement mais ne permet pas de suivre le fonctionnement dans le moniteur série.

La déclaration des informations textuelles utilisées par le moniteur série se fait au niveau de la déclaration de l'objet. Si on reprend le diagramme donné figure 6 et que l'on corrige le bloc de déclaration donnée figure 9 cela nous donne :

```
//Déclarations (Cf exemple 11)
Cevent bp_on("bouton on") ;    //variable associée au bouton poussoir on
Cevent bp_off("bouton off") ;  //variable associée au bouton poussoir off
Cstat Marche("Marche",0) ;     //variable de type CStat associée à l'état Marche
Cstat Arret("Arret",1) ;       //variable de type CStat associée à l'état Arret
```

Pour afficher le log dans le moniteur série il faut ajouter dans la fonction **setup()** :

```
//Activation des log et du moniteur série
bp_on.log = true ;
bp_off.log = true ;
Marche.log = true ;
Arret.log = true ;
Serial.begin(115200) ; //Moniteur série
```

La possibilité d'afficher ou de masquer les log de chaque objet facilite le débugage en choisissant de visualiser seulement ce qui est nécessaire.

L'initialisations de l'Arduino donne au niveau du moniteur série (cf exemple 11):

```
Cstat : Activation de l'etat : Marche (Id:0) a 0 ms.
```

L'appui sur le bouton pb_off provoque l'enchaînement suivant dans le moniteur :

```
Cevent : bouton off rising at 9221 ms.
Cstat : Desactivation de l'etat : Marche (Id:0) a 9221 ms.
```

```
Cstat : Activation de l'etat : Arret (Id:1) a 9224 ms.
```

On peut donc ainsi suivre l'évolution de notre diagramme d'état.

4 LIBRAIRIE *OBJ_STAT*

La librairie proposée comporte d'autres classes, non présentées ici, facilitant la création de diagrammes d'états et le pilotage de systèmes complexes. Voici la liste complète :

- **Cevent** : permet de gérer les entrées numériques (front montant, front descendant, changement d'état).
- **CdigitalOut** : permet de gérer les sorties numériques (on, off, on/off pendant une durée donnée, clignotement on/off suivant une durée donnée).
- **Cstat** : permet de gérer le fonctionnement d'un graphe d'état.
- **Cbistable** : 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.
- **Cbutton** : permet de gérer les entrées numériques provenant d'un bouton. Cette classe reprend les événements de Cevent (front montant, front descendant, changement d'état) et en ajoute d'autres (clic, double clic, appui long).
- **Ctimer** : permet de gérer les temporisations.
- **Cpulse** : permet de générer un top tous les x ms.

Pour les détails, on se reportera à la documentation accompagnant la librairie.

5 CODAGE D'UN DIAGRAMME D'ACTIVITE

Pour coder un diagramme d'activité en C/C++, la structure *switch/case* proposée de base par le langage est amplement suffisante. Un exemple de code est proposé figure 16 pour le diagramme d'activité donné figure 15 correspondant à l'activité **do** d'un état **Système Actif**.

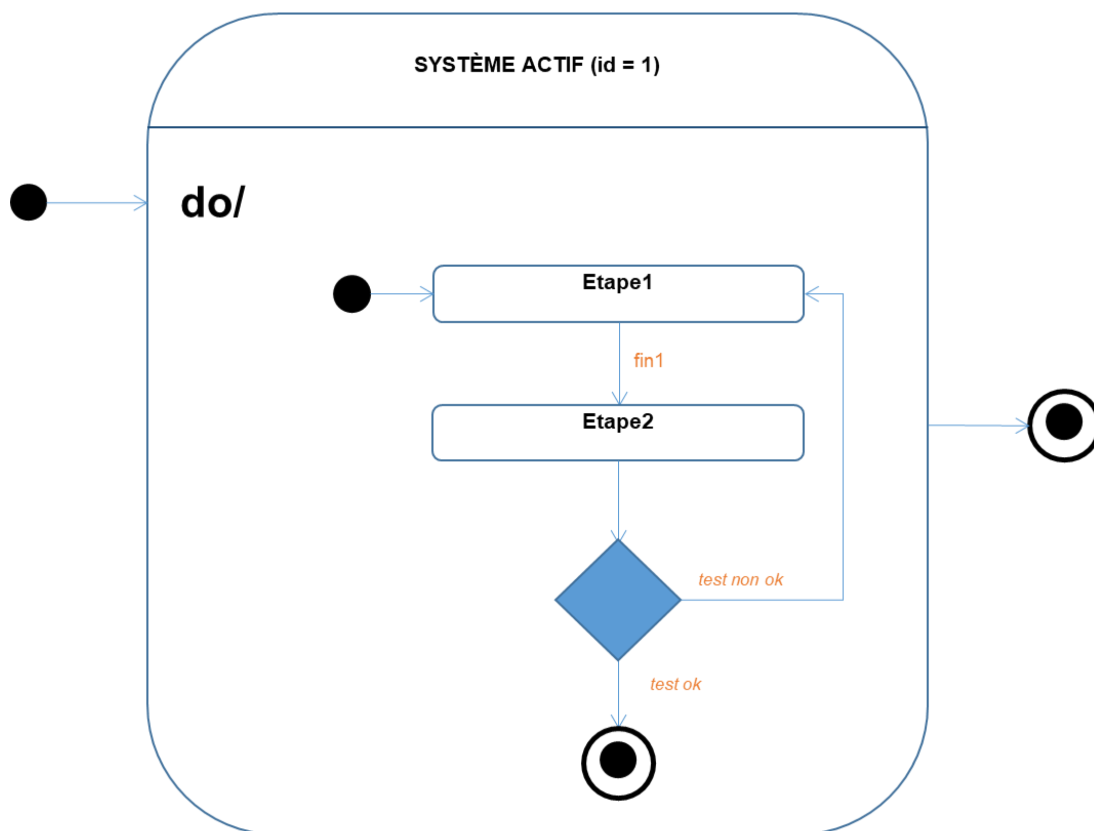


Figure 15 : Diagramme d'activité associé à l'état Système Actif

```
#include "obj_stat.h"     //Inclusion de la bibliothèque placée dans le répertoire de travail

//Déclarations
Cevent fin1("bouton fin1") ;              //variable associée au bouton poussoir
Cevent bp_test("bouton test") ;           //variable associée au bouton poussoir
Cstat Systeme_Actif("Systeme_actif",1) ; //variable de type CStat associée à l'état Systeme_Actif
Cstat Fin("psudo-état fin",2) ;           //variable de type CStat associée au pseudo-état Fin

Cstat * p_etatCourant ;                   //pointeur associé à l'état courant

unsigned int Activite ;                   //Numéro de l'activité en cours
bool test ;                               //Variable booléenne du diagramme d'activité
//Setup
void setup() {
  Systeme_Actif.log = true ;//Activation des log
  Fin.log = true ;          //Activation des log
  Serial.begin(115200) ;    //Moniteur série
  fin1.setPin(3) ;          //La pin 3 est définie comme une entrée (bouton poussoir)
  bp_test.setPin(4) ;       //La pin 4 est définie comme une entrée (bouton poussoir)
  p_etatCourant = Systeme_Actif.activate() ; //Etat initial
}
//Loop
void loop() {
  //Mise à jour des entrées
  fin1.update() ;
  bp_test.update() ;

  //Gestion de l'évolution des états
  completion(p_etatCourant,Systeme_Actif,Fin);

  //Gestion des activités
  if (Systeme_Actif.do_()) {
    switch (Activite) {
      case 0:
        Serial.println("Systeme_Actif : Activité 0 vers 1") ;
        Activite++ ; //On passe à l'étape suivante sans condition (entrée)
      break ;
      case 1: //Activité 1
        //Ecrire ici le code associé à l'étape 1
        //...
        //...
        if (fin1.rising()) { //On passe à l'étape suivante seulement si on a un front montant de fin1
          Serial.println("Systeme_Actif : Activité 1 vers 2") ;
          Activite++ ;
        }
      break ;
      case 2: //Activité 2
        //Ecrire ici le code associé à l'étape 2
        //...
        test = bp_test.stat() ; //Pour l'exemple, la variable test est associée à l'état d'un bouton
                                //poussoir
        //...
        if (test) {    //Si le test est vérifié
          Serial.println("Systeme_Actif : Activité 2 vers 3") ;
          Activite++ ;   //on passe à l'étape suivante
        }
        else {
          Serial.println("Systeme_Actif : Retour Activité 1") ;
          Activite = 1 ; //Sinon on revient à l'étape 1
        }
      break;
      case 3: //Activité 3
        //L'étape 3 est l'étape de fin d'activité
        //On peut donc déclarer l'état comme terminé
        Systeme_Actif.completion_event = true ;
      break;
    }
  }
  if (Systeme_Actif.entry_()) {
    Activite = 0 ; //Initialisation du diagramme d'activité à l'étape 0 (entrée)
  }
  //Gestion automatique de l'évolution du diagramme
  Systeme_Actif.update();
  Fin.update();
}
```

Figure 16 : Codage d'un diagramme d'activité associé à l'état Système Actif (exemple 12)

Dans ce code, on utilise la variable `Activite` pour connaitre le numéro de l'activité active. Le code suivant permet d'initialiser `Activite` à l'activation de l'état **Système_Actif**.

```
if (Systeme_Actif.entry_()) {
  Etape = 0 ; //Initialisation du diagramme d'activité à l'étape 0 (entrée)
}
```

Le code correspondant à chaque activité du diagramme se trouve dans les différents **`case / break.`**

Voici ce que l'on peut observer dans le moniteur série :

A la mise sous tension :

```
Cstat : Activation de l'etat : Systeme_actif (Id:1) a 0 ms.
Systeme_Actif : Activité 0 vers 1
```

Si on appui sur le bouton fin1 :

```
Systeme_Actif : Activité 1 vers 2
Systeme_Actif : Retour Activité 1
```

Si on appui sur le bouton fin1 avec le bouton pb_test enfoncé :

```
Systeme_Actif : Activité 1 vers 2
Systeme_Actif : Activité 2 vers 3
Cstat : Désactivation de l'etat : Systeme_actif (Id:1) a 10000 ms.
Cstat : Activation de l'etat : Fin (Id:2) a 10001 ms.
```

6 FAQ

Quelles sont les cartes compatibles pour faire des diagrammes d'états ?

Toutes les cartes compatibles Arduino, y compris les cartes ESP32 ou ESP8266 fonctionnent avec la bibliothèque proposée.

Quelles sont les cartes préconisées pour faire des diagrammes d'états ?

Dès que le système devient un peu complexe, les besoins en mémoire vive augmentent. Il est donc préférable de choisir une carte Arduino Méga qui a 4 fois plus de mémoire qu'une Uno. Si on veut vraiment de la performance, il faut choisir une Arduino DUE ou une carte ESP qui ont des processeurs beaucoup plus rapide. En revanche, ces dernières sont en 3,3V.

Pourquoi privilégier les cartes compatibles Arduino plutôt que les cartes compatibles Micro-Python ?

Récemment sont apparues des cartes fonctionnant en MicroPython plutôt qu'en C/C++. MicroPython est un portage de python sur microcontrôleur. Python est un langage beaucoup plus moderne que C/C++, il est donc tentant d'utiliser MircoPython pour programmer nos systèmes. Malheureusement, à part pour les petits systèmes très simples, MicroPython se révèle très vite inadapté pour la programmation de nos systèmes complexes :

- MicroPython n'est pas Python, ce qui fait qu'on ne peut pas profiter de tout l'écosystème Python qui fait la force de Python,
- MicroPython n'est pas compatible avec les librairies Arduino, on ne peut donc pas profiter de tout l'écosystème Arduino, il faut donc tout redévelopper,
- Un programme MicroPython est interprété par le microcontrôleur et non compilé. L'exécution du programme est donc très lente,
- Le débugage est fastidieux car on ne dispose pas à ce jour d'IDE vraiment performante en MicroPython,
- MicroPython demande d'avoir de solides connaissances en C/C++ pour comprendre les subtilités de langages (MicroPython introduit des notions issues du C/C++ non présentent dans Python pour tenir compte des spécificités des microcontrôleurs).

Peut-on faire du diagramme d'état en Python ?

Oui, il existe une bibliothèque Python proposé par Qt qui permet de faire du diagramme d'état (QStatMachine) [8]. Pour utiliser cette bibliothèque, il faut une machine supportant Python comme un PC ou une carte Raspberry PI.

Mon système est lent, comment peut-on améliorer les choses ?

Si le système comporte beaucoup d'état et d'entrées/sorties, celui-ci devient lent. Il est possible d'optimiser le code en analysant plus finement à chaque cycle les évolutions. Cela sort du cadre de ce document mais l'idée est d'utiliser le retour booléen des méthodes **update()** qui précise si au cours du cycle l'objet a évolué. On peut aussi utiliser des bibliothèques spécifiques améliorant la rapidité de lecture des entrées. Cela n'a pas été fait car chaque carte à sa spécificité et nous avons voulu garder un code portable sur tout type de carte. Enfin, la solution radicale consiste à prendre une carte rapide comme une Arduino DUE.

Je veux piloter mon système à distance via le web (application android, page web…), quelle est la bonne architecture matérielle ?

Pour un système complet, la bonne architecture est de coupler une ou plusieurs cartes Arduino à un Raspberry PI qui se charge de la supervision globale. La communication entre les cartes peut se faire par liaison série, I2C ou SPI. La (ou les) cartes Arduino se charge du pilotage bas niveau (diagramme d'états, asservissement) et la Raspberry du haut niveau (Base de données, affichage, serveur Web).

Une autre alternative est de remplacer la Raspberry par une carte ESP32 ou ESP8266 pour créer un serveur Web à moindre coût.

7 REFERENCES

[1] Système temps réel : https://fr.wikipedia.org/wiki/Système_temps_réel

[2] Multitâche Arduino : https://www.carnetdumaker.net/articles/faire-plusieurs-choses-la-fois-avec-une-carte-arduino/

[3] Norme SysML ISO/IEC 19514:2017(E), StateMachines : https://www.omg.org/spec/SysML/ISO/19514/PDF

[4] Norme UML 2.5.1, StateMachines : https://www.omg.org/spec/UML/2.5.1/PDF

[5] Programmer un diagramme d'état :https://www.electro-info.ovh/comment-programmer-un-diagramme-d-etat-en-C-Cpp

[6] Uml-sysml.org : https://www.uml-sysml.org/diagrammes-uml-et-sysml/diagramme-uml/diagramme-detat/, https://laurent-audibert.developpez.com/Cours-UML/?page=diagramme-etats-transitions#L5-3-1

[7] MicroPython : https://micropython.org/

[8] QStatMachine : https://doc-snapshots.qt.io/qtforpython-dev/PySide2/QtCore/QStateMachine.html