Montage

3

Clignotement avec gestion des intervalles

Au sommaire :

- la déclaration et l'initialisation de plusieurs variables ;
- la programmation de plusieurs broches aussi bien comme entrée (`INPUT`) que comme sortie (`OUTPUT`) ;
- l'instruction `digitalRead()` ;
- l'instruction `millis()` ;
- l'utilisation de la structure de contrôle `if-else` ;
- le sketch complet ;
- l'analyse du schéma ;
- la réalisation du circuit ;
- un exercice complémentaire.

Appuyez sur le bouton-poussoir et il réagit

Dans notre premier exemple, nous avons vu comment interrompre l'exécution d'un sketch avec la fonction de retardement `delay`. La LED reliée à la broche de sortie numérique 13 clignotait à intervalles réguliers. Un tel circuit ou une telle programmation présente cependant un inconvénient que nous entendons déceler et éliminer. Pour cela, il faut approfondir un peu le circuit clignotant. Que se passerait-il si vous branchiez en plus un bouton-poussoir sur une entrée numérique pour interroger continuellement son état ?

Une LED est censée s'allumer si vous appuyez sur le bouton-poussoir. Peut-être voyez-vous déjà où je veux en venir ? Tant que

l'exécution du sketch est prisonnière de la fonction `delay`, le traitement du code est interrompu et l'entrée numérique ne peut être interrogée. Vous appuyez donc sur le bouton-poussoir et rien ne se passe.

Composants nécessaires

	1 LED rouge
	1 LED jaune
	1 bouton-poussoir
	1 résistance de 10 kΩ
	2 résistances de 330 Ω
	Plusieurs cavaliers flexibles de couleurs et de longueurs différentes

Code du sketch

Le code du sketch suivant ne fonctionne pas comme nous l'aurions voulu.

```
//Le code suivant ne fonctionne pas comme prévu
int ledPinBlink = 13;          //LED clignotante rouge en broche 13
int ledPinButton = 10;         //LED de bouton-poussoir jaune en
                               //broche 10
int buttonPin = 8;             //Bouton-poussoir en broche 8
int buttonState;               //Variable pour enregistrement
                               //de l'état du bouton-poussoir
void setup(){
  pinMode(ledPinBlink, OUTPUT); //Broche LED clignotante comme sortie
  pinMode(ledPinButton, OUTPUT);//Broche LED de bouton-poussoir comme
                                 //sortie
  pinMode(buttonPin, INPUT);    //Broche bouton-poussoir comme entrée
}

void loop(){
  //Faire clignoter la LED clignotante
```

```
  digitalWrite(ledPinBlink, HIGH);  //LED rouge au niveau HIGH (5 V)
  delay(1000);                      //Attendre une seconde
  digitalWrite(ledPinBlink, LOW);   //LED rouge au niveau LOW (0 V)
  delay(1000);                      //Attendre une seconde
  //Interrogation de l'état du bouton-poussoir
  buttonState = digitalRead(buttonPin);
  if(buttonState == HIGH)
    digitalWrite(ledPinButton, HIGH); //LED jaune au niveau HIGH (5 V)
  else
    digitalWrite(ledPinButton, LOW);  //LED jaune au niveau LOW (0 V)
}
```

Il y a quelque chose que je ne comprends pas. L'exécution repasse bien à un moment donné sur la ligne d'interrogation du bouton-poussoir dans la boucle sans fin. L'état est alors bien interrogé correctement.

Vous avez tout compris : l'expression « à un moment donné » sied ici à merveille ! Vous voulez cependant que le traitement du code réagisse à tout moment et pas seulement à un moment donné quand l'exécution reparaît. Les fonctions `delay` entravent bien quasiment la poursuite du sketch. C'est compris ? Je vous montre le comportement sur un chronogramme, où figurent les trois signaux pertinents, ceux de la LED clignotante (broche 13), du bouton-poussoir (broche 8) et de la LED du bouton-poussoir (broche 10).

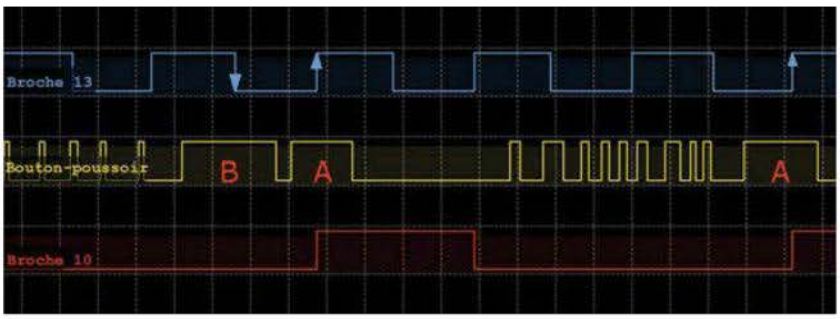


Figure 3-1
Chronogramme des signaux sur les broches 13, 8 et 10

Regardez le signal jaune qui représente l'état du bouton-poussoir. J'ai beau appuyer plusieurs fois sur ce dernier, le signal rouge sur la broche 10 ne réagit pas au début. Si je maintiens cependant le bouton-poussoir enfoncé pendant un temps plus long (à l'endroit signalé par un A), le signal de la broche 10 finit par passer lui aussi au niveau `HIGH`. Mais pourquoi ne s'est-il rien passé à l'endroit signalé par un B ? Je maintiens pourtant bien aussi le bouton-poussoir enfoncé pendant un temps plus long.

C'est très simple ! Vous avez deux activations de `delay` et la deuxième sert à faire durer le niveau `LOW`. Une fois ce délai écoulé, l'état du bouton-poussoir est interrogé très brièvement, c'est-à-dire précisément au moment où le niveau passe de `LOW` à `HIGH`. Le niveau sur la broche 10 réagit donc toujours au flanc montant (A) et ne réagit pas au flanc descendant (B). C'est simple, non ? Toujours est-il que nous devons ici renoncer à `delay` et choisir une autre solution, comme le montre l'exemple suivant. Ne vous en faites pas pour les lignes de code car nous allons le développer progressivement :

```
int ledPinBlink = 13;   //LED clignotante rouge en broche 13
int ledPinButton = 10;  //LED de bouton-poussoir jaune en broche 10
int buttonPin = 8;      //Bouton-poussoir en broche 8
int buttonState;        //Variable pour enregistrement de l'état du
                        //bouton-poussoir
int interval = 2000;    //Intervalle de temps (2 secondes)
unsigned long prev;     //Variable de temps
int ledState = LOW;     //Variable d'état pour la LED clignotante

void setup(){
  pinMode(ledPinBlink, OUTPUT);  //Broche LED clignotante comme sortie
  pinMode(ledPinButton, OUTPUT); //Broche LED de bouton-poussoir comme
                                 //sortie
  pinMode (buttonPin, INPUT);    //Broche bouton-poussoir comme entrée
  prev = millis();               //Mémoriser le compteur de temps actuel
}

void loop(){
  //Faire clignoter la LED clignotante via la gestion des intervalles
  if((millis() - prev) > interval){
   prev = millis();
   ledState = !ledState;                //Bascule état de la LED
   digitalWrite(ledPinBlink, ledState); //Bascule la LED rouge
  }
  //Interrogation de l'état du bouton-poussoir
  buttonState = digitalRead (buttonPin);
  if(buttonState == HIGH)
    digitalWrite(ledPinButton, HIGH); //LED jaune au niveau HIGH (5 V)
  else
    digitalWrite(ledPinButton, LOW); //LED jaune au niveau LOW (0 V)
}
```

Revue de code

Vous voyez ici que les variables à déclarer et à initialiser au début sont de plus en plus nombreuses. Faisons maintenant un peu le tour.

Variable	Objet
`ledPinBlink`	Contient le numéro de broche pour la LED sur la sortie numérique broche 13.
`ledPinButton`	Contient le numéro de broche pour la LED sur la sortie numérique broche 10.
`buttonPin`	Contient le numéro de broche pour le bouton-poussoir sur l'entrée numérique broche 8.
`buttonState`	Sert à enregistrer l'état du bouton-poussoir pour une exploitation ultérieure.
`interval`	Contient la valeur pour la gestion des intervalles.
`prev`	Enregistre la valeur actuelle de la fonction `millis`.
`ledState`	Mémorise l'état de la LED du bouton-poussoir.

◀ Tableau 3-1
Variables nécessaires et leur objet

Je commencerai par la gestion des intervalles car c'est le plus important. Le diagramme de la figure 3-2 nous montre une évolution chronologique avec certaines valeurs de temps marquantes. Je dois vous expliquer auparavant certaines choses dans le code source. Il s'agit d'une part de la nouvelle fonction `millis`, qui fournit en retour le temps écoulé depuis le début du sketch actuel en millisecondes. Il faut ici tenir compte de quelque chose d'important. Le type de la donnée de retour est `unsigned long`, donc un type de nombre entier de 32 bits non signé dont le domaine de valeurs s'étend de 0 à 4 294 967 295 ($2^{32} - 1$). Ce domaine de valeurs est aussi vaste parce qu'il doit être en mesure de traiter des valeurs correspondant à une période prolongée (49,71 jours maximum) sans débordement (dépassement de capacité).

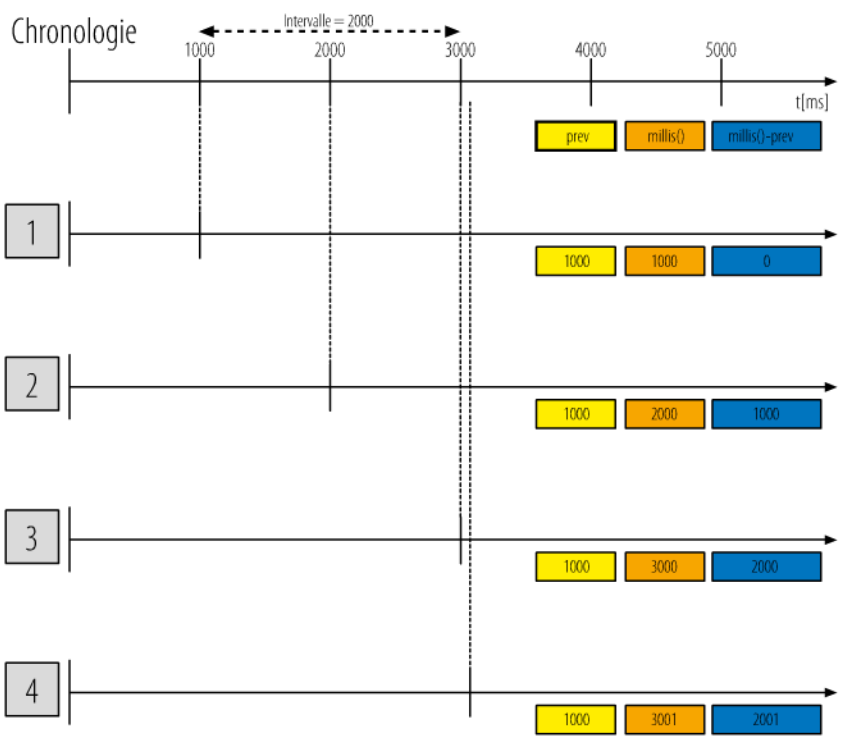
Pour aller plus loin

Un débordement signifie pour des variables que le maximum des valeurs que leur type de données peut traiter a été dépassé et va maintenant recommencer à 0. Pour le type de donnée `byte`, qui présente une donnée de 8 bits et peut, par conséquent, stocker $2^8 = 256$ états (de 0 à 255), un débordement se produit au moment de l'action 255 + 1. Le type de donnée `byte` n'est plus en mesure de traiter la valeur 256.

Trois autres variables ont été ajoutées par mes soins, dont les rôles sont les suivants :

- `interval` (enregistre en ms le temps applicable à l'intervalle de clignotement) ;
- `prev` (enregistre en ms le temps actuellement écoulé. `prev` vient de *previous* qui signifie précédent) ;
- `ledState` (la LED clignotante est commandée en fonction de l'état `HIGH` ou `LOW`).

Figure 3-2 ▶
Évolution chronologique de la gestion des intervalles



Analysons maintenant le diagramme, dans lequel j'ai pris au hasard des moments marquants pour plus de clarté. Le temps ne s'écoule évidemment pas réellement pendant ces étapes.

Tableau 3-2 ▶
Contenu des variables dans l'évolution chronologique

Moment	Explication
1	Le temps actuel en millisecondes (1 000 dans le cas présent) est chargé dans la variable `prev`. La chose se produit une seule fois dans la fonction `setup`. La différence `millis() - prev` donne la valeur 0 comme résultat. Cette valeur n'est pas supérieure à la valeur d'intervalle 2 000. La condition n'est pas remplie et le bloc `if` n'est pas exécuté.
2	1 000 ms plus tard, la différence `millis() - prev` est à nouveau calculée et il est vérifié que le résultat n'est pas supérieur à la valeur d'intervalle 2 000. 1 000 n'étant pas supérieur à 2 000, la condition n'est toujours pas remplie.
3	1000 ms se sont encore écoulées, la différence `millis() - prev` est à nouveau calculée et il est vérifié que le résultat n'est pas supérieur à la valeur d'intervalle 2 000. 2 000 n'étant pas supérieur à 2 000, la condition n'est toujours pas remplie.
4	Après une durée de fonctionnement de 3 001 ms, la différence donne cependant une valeur supérieure à la valeur d'intervalle 2 000. La condition est remplie et le bloc `if` mis à exécution. L'ancienne valeur `prev` est remplacée par le temps actuel provenant de la fonction `millis`. L'état de la LED clignotante peut être inversé. Le jeu reprend au début sur la base de la nouvelle valeur temps dans la variable `prev`.

Pendant tout le déroulement, aucun arrêt n'a été inséré à aucun endroit sous la forme d'une pause dans le code source, si bien que l'interrogation de la broche numérique 8 pour gérer la LED du bouton-poussoir n'a été absolument pas gênée.

Un pression sur le bouton-poussoir est presque simultanément évaluée et affichée. La seule nouvelle instruction, qui s'avère être une fonction délivrant une valeur en retour, se nomme `millis`.

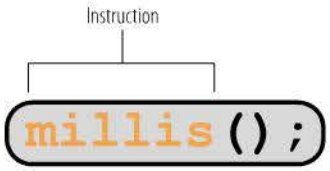


Figure 3-3
L'instruction `millis`

On voit qu'il n'accepte aucun argument et présente par conséquent une paire de parenthèses vides. Sa valeur de retour possède le type de donnée `unsigned long`.

Une seule ligne me chagrine encore. Que signifie `ledState = !ledState` et qu'entend-on par bascule ?

Vous allez plus vite que moi dites donc ! Mais qu'à cela ne tienne puisque vous en parlez. La variable `ledState` stocke le niveau qui commande la LED rouge ou plutôt qui est en charge du clignotement (`HIGH` pour allumée et `LOW` pour éteinte). La ligne suivante :

```
digitalWrite(ledPinBlink, ledState);
```

permet de commander la LED. Le clignotement est précisément obtenu par un va-et-vient entre les états `HIGH` et `LOW`. Ce va-et-vient est également appelé bascule. Je vais reformuler la ligne de manière à la rendre peut-être plus claire.

```
if(ledState == LOW)
  ledState = HIGH;
else
  ledState = LOW;
```

La première ligne demande si le contenu de la variable `ledState` est égal à `LOW`. Si oui, il est mis sur `HIGH` ; sinon, il est mis sur `LOW`. Il s'agit également d'une bascule d'état. La variante à une ligne suivante, que j'ai déjà utilisée, est beaucoup plus courte.

```
ledState = !ledState; //Bascule état de la LED
```

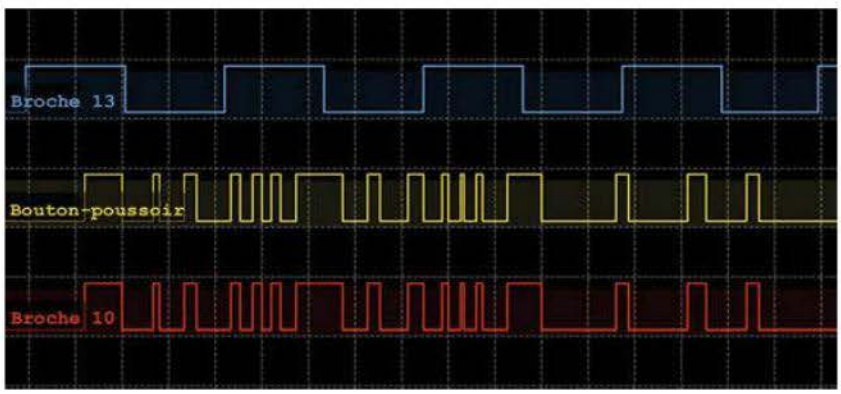
J'utilise ici l'opérateur logique `not`, représenté par le point d'exclamation. Il est souvent utilisé pour des variables booléennes qui ne peuvent accepter que les valeurs logiques `true` ou `false`. Le résultat de l'opérateur `not` est la valeur logique opposée à celle de l'opérande. Ceci est également valable pour les deux niveaux `HIGH` et `LOW`.

Le port 8 est finalement interrogé tout à fait normalement et sans retardement du bouton-poussoir.

```
buttonState = digitalRead(buttonPin);
if(buttonState == HIGH)
  digitalWrite(ledPinButton, HIGH);
else
  digitalWrite(ledPinButton, LOW);
```

Je vous montre à nouveau le comportement sur un chronogramme, dans lequel les trois signaux en question – à savoir la LED clignotante (broche 13), le bouton-poussoir (broche 8) et la LED de bouton-poussoir – sont représentés l'un en dessous de l'autre de la même manière que plus haut :

Figure 3-4 ▶
Chronogramme des signaux sur les broches 13, 8 et 10



On voit que le signal bleu représente la LED clignotante sur la broche 13. Si maintenant j'actionne à intervalles réguliers le bouton-poussoir – représenté par le signal jaune – sur la broche 8, le signal rouge de la LED du bouton-poussoir réagit aussitôt. Aucun retard et aucune interruption ne sont à observer. Le comportement du circuit est exactement celui que nous voulions.

Schéma

La lecture du schéma ne devrait plus vous poser de problème maintenant. Seule une autre LED, censée réagir quand le bouton-poussoir est enfoncé, a été ajoutée.

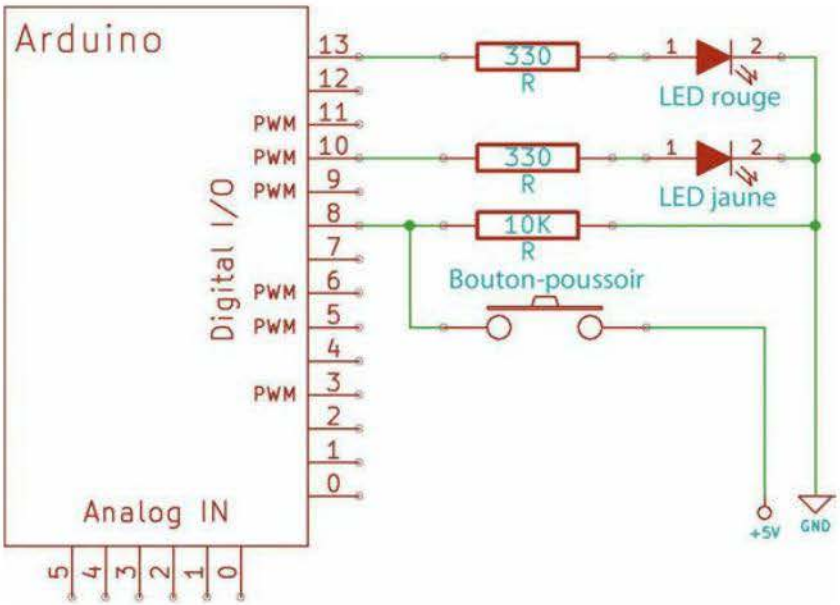


Figure 3-5
Carte Arduino avec un bouton-poussoir et deux LED

Réalisation du circuit

La plaque d'essais est maintenant un peu plus remplie.

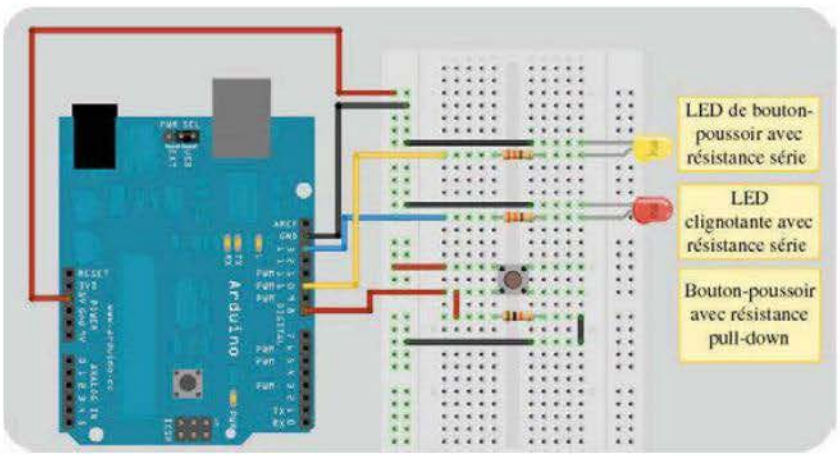


Figure 3-6
Construction du circuit avec Fritzing

Pour aller plus loin

Comme vous avez pu le voir dans cette réalisation et aussi dans la précédente, j'utilise des cavaliers flexibles de couleurs différentes. Quand vous composez des circuits sur votre plaque d'essais, je vous conseille d'utiliser également des couleurs différentes. J'ai choisi par exemple le rouge pour la tension d'alimentation et le noir pour la masse. Les autres lignes de signal peuvent être bleues, jaunes ou même rouges.

Il n'y pas de règle précise en la matière, mais vous devriez constituer votre propre système de couleurs afin de conserver une bonne vue d'ensemble. Cela peut être également utile aux personnes extérieures de trouver une maquette conçue proprement.

Problèmes courants

Si la LED ne s'allume pas quand le bouton-poussoir est enfoncé ou si la LED reste allumée, vérifiez ce qui suit.

- Vos fiches de raccordement sur la plaque d'essais correspondent-elles vraiment au schéma ?
- Les LED ont-elles été mises dans le bon sens ? Pensez à la polarité !
- Les boutons–poussoir peuvent être à 2 ou 4 connexions. S'il s'agit d'un modèle à 4 connexions, ont-elles été correctement branchées ? Faites, le cas échéant, un essai de continuité avec un multimètre et vérifiez ainsi l'adéquation du bouton-poussoir et des pattes correspondantes.
- Les deux résistances ont-elles bien les bonnes valeurs ? Ont-elles été éventuellement interverties ?
- Le code du sketch est-il correct ?

Qu'avez-vous appris ?

- Vous savez utiliser plusieurs variables à des fins les plus diverses (déclaration pour broche d'entrée ou de sortie et enregistrement des informations d'état).
- L'instruction `delay` interrompt le déroulement du sketch et instaure une pause, de telle sorte que toutes les instructions subséquentes ne soient pas exécutées tant que le temps d'attente n'est pas écoulé.
- Vous avez appris, à travers la gestion des intervalles avec la fonction `millis`, un moyen permettant de maintenir malgré tout

l'exécution continue du sketch de la boucle sans fin `loop`, de telle sorte que d'autres instructions de la boucle `loop` soient exécutées et qu'une utilisation d'autres capteurs, tels que le bouton-poussoir raccordé, soit possible.

- Vous avez appris à lire divers chronogrammes, qui représentent très bien graphiquement les différents états de niveau dans la courbe d'évolution temporelle.

Exercice complémentaire

Créez simplement un sketch qui allume la LED en cas de pression sur le bouton-poussoir et qui l'éteint en cas de nouvelle pression, et ainsi de suite. Un cas épineux qui fera l'objet du montage suivant. Il se peut que vous rencontriez un problème que nous résoudrons plus tard. Le mot-clé est rebond.