Montage

7

La machine à états

Au sommaire :

- la déclaration et l'initialisation de plusieurs variables ;
- la programmation de plusieurs broches comme sortie (`OUTPUT`) ;
- la programmation d'un port comme entrée (`INPUT`) ;
- le sketch complet ;
- l'analyse du schéma ;
- la réalisation du circuit ;
- le sketch élargi (circuit interactif pour feux de circulation) ;
- un exercice complémentaire.

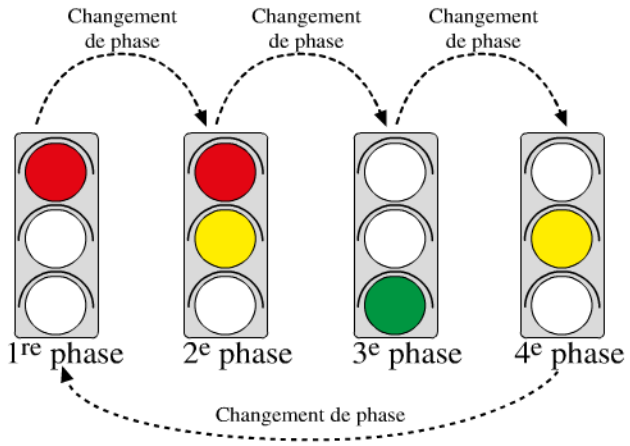
Des feux de circulation

La programmation de feux de circulation est une tâche classique. La plupart du temps, on y croise pour la première fois une *state machine* – son nom complet est *Finite State Machine* ou FSM. Il s'agit d'une machine supportant des états certes différents, mais finis. Voici ce qui caractérise ce modèle :

- l'état ;
- la transition d'état ;
- l'action.

Voyons d'abord les différentes phases de signalisation possibles.

Figure 7-1 ▶
États de signalisation avec changement de phase

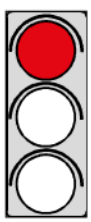
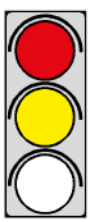
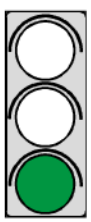
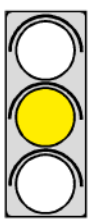


Ici, on prévoit 4 phases de signalisation (de 1 à 4), le cycle reprenant ensuite au début. Pour plus de simplicité, nous nous en tiendrons à des feux pour un sens de circulation. La signification des différentes couleurs est bien connue de tous :

- rouge : interdiction de passer ;
- orange : attendre le signal suivant ;
- vert : autorisation de passer.

Chaque phase a une durée d'allumage définie. L'usager de la route doit avoir en effet le temps de comprendre les différentes phases pour réagir en conséquence. Voici les durées d'allumage qui seront utilisées dans notre exemple (voir tableau 7-1), qui n'ont rien à voir avec la réalité. Bien évidemment, il est possible de régler le temps selon vos goûts.

Tableau 7-1 ▶
Phases avec durée d'allumage

1re phase	2e phase	3e phase	4e phase
			
Durée : 10 s	Durée : 2 s	Durée : 10 s	Durée : 3 s

Une fois le code transmis, les feux de circulation doivent exécuter les 4 phases énoncées précédemment, puis recommencer au début.

Composants nécessaires

	1 LED rouge
	1 LED orange
	1 LED verte
	3 résistances de 330 Ω
	Plusieurs cavaliers flexibles de couleurs et de longueurs diverses

Code du sketch

Voici le code du sketch pour commander les feux de circulation :

```
#define DELAY1 10000      //Pause 1, 10 secondes
#define DELAY2 2000       //Pause 2, 2 secondes
#define DELAY3 3000       //Pause 3, 3 secondes
int ledPinRed = 7;        //Broche 7 commande la LED rouge
int ledPinOrange = 6;     //Broche 6 commande la LED orange
int ledPinGreen = 5;      //Broche 5 commande la LED verte
void setup(){
  pinMode(ledPinRed, OUTPUT);      //Broche comme sortie
  pinMode(ledPinOrange, OUTPUT);   //Broche comme sortie
  pinMode(ledPinGreen, OUTPUT);    //Broche comme sortie
}

void loop(){
  digitalWrite(ledPinRed, HIGH);     //Allumage LED rouge
  delay(DELAY1);                     //Attendre 10 secondes
  digitalWrite(ledPinOrange, HIGH);  //Allumage LED orange
  delay(DELAY2);                     //Attendre 2 secondes
  digitalWrite(ledPinRed, LOW);      //Extinction LED rouge
  digitalWrite(ledPinOrange, LOW);   //Extinction LED orange
  digitalWrite(ledPinGreen, HIGH);   //Allumage LED verte
  delay(DELAY1);                     //Attendre 10 secondes
  digitalWrite(ledPinGreen, LOW);    //Extinction LED verte
```

```
  digitalWrite(ledPinOrange, HIGH);  //Allumage LED orange
  delay(DELAY3);                     //Attendre 3 secondes
  digitalWrite(ledPinOrange, LOW);   //Extinction LED orange
}
```

Revue de code

Voici les variables qui sont techniquement nécessaires à notre expérimentation.

Tableau 7-2 ▶
Variables nécessaires et leur objet

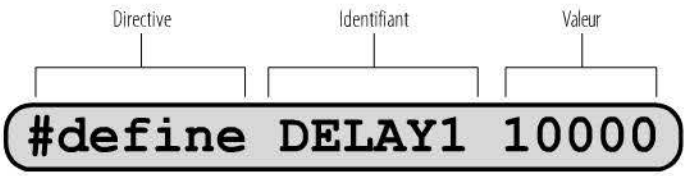
Variable	Objet
ledPinRed	Commande la LED rouge.
ledPinOrange	Commande la LED orange.
ledPinGreen	Commande la LED verte.

J'ai déjà une question ! Le code du sketch commence par trois lignes dont le contenu m'est complètement inconnu. Que signifient `#define` et le reste de la ligne après ?

Encore une fois, vous allez plus vite que la musique ! En fait, l'instruction `#define` n'est pas une véritable instruction, mais une directive de prétraitement. Rappelez-vous la directive de prétraitement `#include`, reconnaissable au point-virgule manquant en fin de ligne qui caractérise normalement la fin d'une instruction.

Quand le compilateur entreprend de traduire le code source, une partie spéciale de celui-ci appelée préprocesseur suit les directives de prétraitement, qui sont toujours introduites par le signe dièse `#`. Vous croiserez encore d'autres directives de ce type au cours de ce livre. La directive `#define` permet d'utiliser des noms symboliques et des constantes. La syntaxe pour l'utiliser est la suivante (voir figure 7-2).

Figure 7-2 ▶
La directive `#define`



Cette ligne agit comme suit : partout où le compilateur trouve l'identifiant `DELAY1` dans le code du sketch, il le remplace par la valeur `10000`. Vous pouvez vous servir de la directive `#define` partout où vous souhaitez utiliser des constantes dans le code. J'ai déjà soulevé ce problème auparavant : pas de *magic numbers* !

Pourquoi n'avez-vous pas utilisé `#define` partout où des broches ont été définies ? Ce sont pourtant également des constantes qui ne varient plus au cours du sketch.

Vous avez raison ! J'aurais pu le faire partout et certains sketches Arduino, que vous trouverez sur Internet, emploient ce mode d'écriture. Au lieu de :

```
int ledPinRed = 7;
```

on peut alors écrire :

```
#define ledPinRed 7
```

Le sketch agit comme avant et cela ne change rien que vous utilisiez la première ou la seconde variante – mais il est important de vous en tenir à celle choisie et de ne pas en changer au gré de votre humeur. Pour ma part, j'emploie dans mon sketch la déclaration et l'initialisation de variable quand il s'agit de broches, et la directive `#define` pour les constantes. Revenons maintenant à notre sketch et voyons comment il fonctionne.

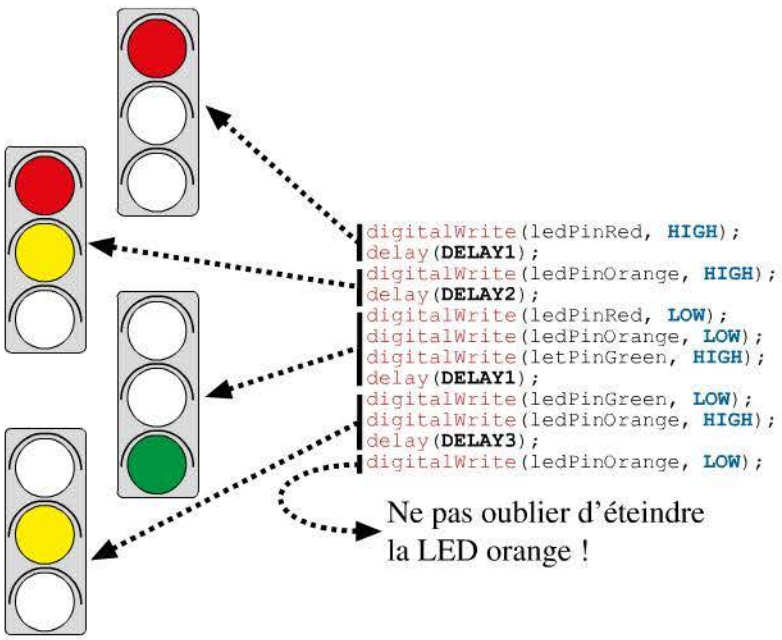
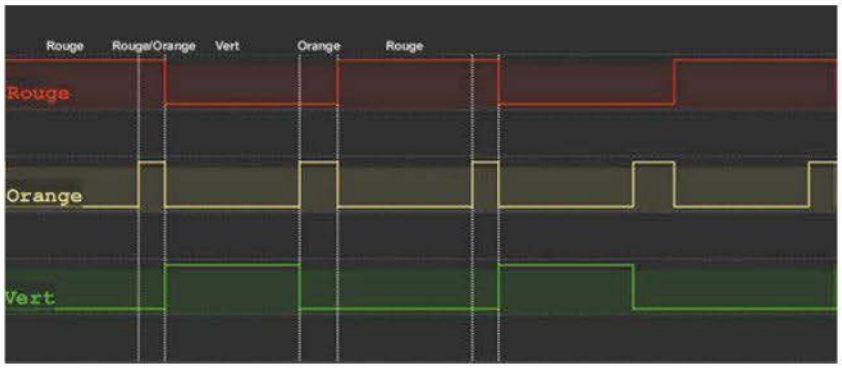


Figure 7-3
Commande des différentes phases de signalisation

Pensez non seulement à allumer les diverses LED, mais aussi à les éteindre lors des différents changements de phase. Au passage de la phase 1 à la phase 2, seule une LED orange vient s'ajouter à la LED rouge. Mais au passage de la phase 2 à la phase 3, veillez à ce que les

LED rouge et orange s'éteignent avant que la LED verte ne s'allume. Jetez un coup d'œil au chronogramme pour voir comment les LED sont allumées à tour de rôle pendant les différentes phases.

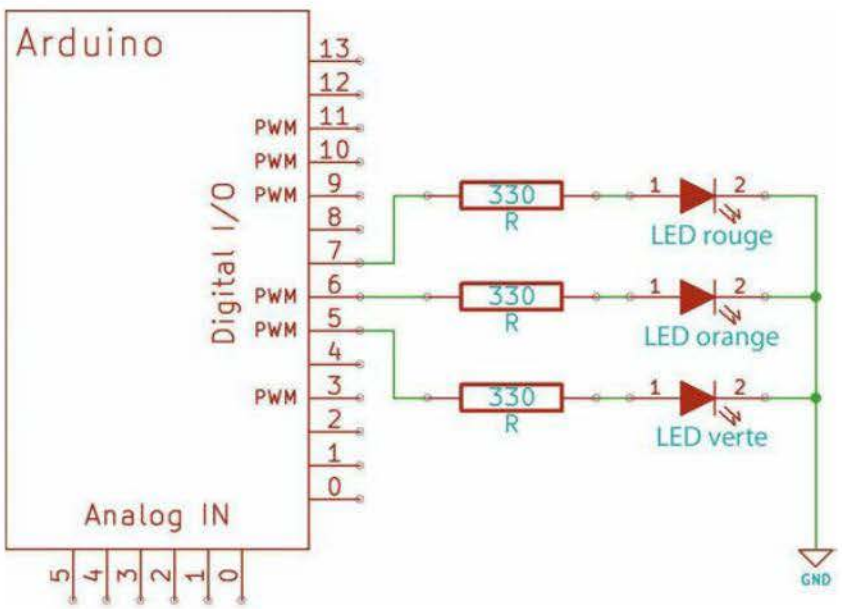
Figure 7-4 ▶
Chronogramme des feux de circulation



Schéma

Le schéma montre les trois LED de couleur avec leurs résistances de 330 ohms. Les besoins en composants supplémentaires sont légèrement moindres, mais les choses vont bientôt changer.

Figure 7-5 ▶
Carte Arduino gérant nos feux de circulation



Réalisation du circuit

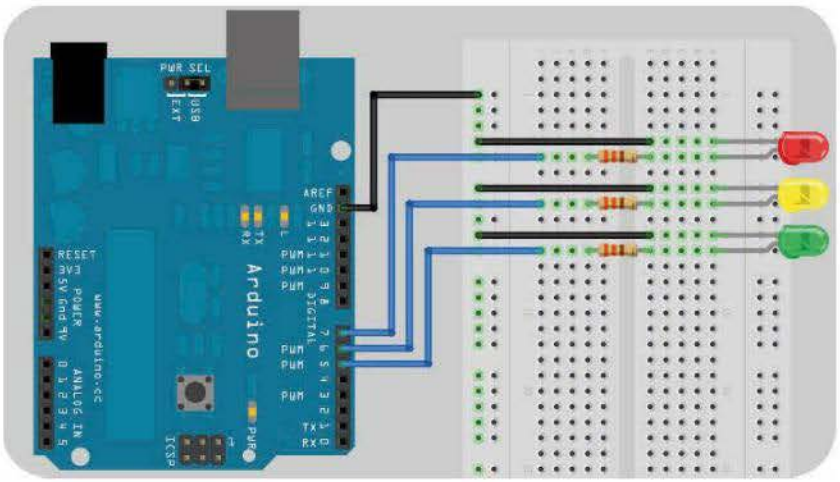


Figure 7-6
Réalisation du circuit pour feux de circulation avec Fritzing

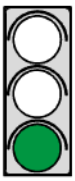
Sketch élargi (circuit interactif pour feux de circulation)

Ce sketch étant relativement simple dans sa programmation et sa construction, modifions un peu les choses.

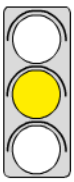
Imaginons maintenant des feux pour piétons installés sur un tronçon droit d'une route nationale. Il n'est pas utile que les phases pour les automobilistes changent sans cesse si aucun piéton ne veut traverser la voie de circulation. Comment les feux doivent-ils fonctionner avec leurs phases ? Quel matériel supplémentaire faut-il et comment faire pour étendre la logique ? Voici ce qu'il faut prendre en compte.

- Si aucun piéton ne se présente pour traverser la route, le feu reste vert pour les automobilistes et rouge pour les piétons.
- Si un piéton appuie sur le bouton gérant les feux pour traverser en toute sécurité, le feu passe à l'orange puis au rouge pour les automobilistes. Le feu passe ensuite au vert pour les piétons. Après un temps prédéfini, le feu repasse au rouge pour les piétons, et le feu rouge passe à l'orange puis au vert pour les automobilistes.

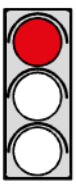
1re phase

Automobiliste	Piéton	Explications
		Ces deux signaux lumineux restent allumés jusqu'à ce qu'un piéton s'approche et appuie sur le bouton gérant les feux. C'est alors seulement que les changements de phase se déclenchent et font en sorte que le feu passe au rouge pour les automobilistes et au vert pour les piétons.

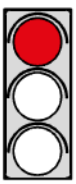
2e phase

Automobiliste	Piéton	Explications
		Le changement de phase est déclenché par l'appui sur le bouton gérant les feux. Le feu passe à l'orange pour les automobilistes, ce qui signifie qu'il va passer au rouge sous peu. Durée : 3 s

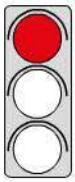
3e phase

Automobiliste	Piéton	Explications
		Pour des raisons de sécurité, le feu est d'abord rouge pour les automobilistes et pour les piétons. Cela permet aux automobilistes de libérer le cas échéant le passage piétons. Durée : 7 s

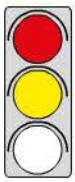
4e phase

Automobiliste	Piéton	Explications
		Le feu passe au vert pour le piéton après un court moment. Durée : 10 s

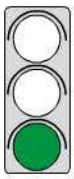
5e phase

Automobiliste	Piéton	Explications
		Le feu repasse au rouge pour les piétons. Durée : 1 s

6e phase

Automobiliste	Piéton	Explications
		Le feu passe du rouge à l'orange pour les automobilistes, ce qui les avertit que le feu va bientôt passer au vert. Durée : 2 s

7e phase

Automobiliste	Piéton	Explications
		Le feu repasse au vert pour les automobilistes et au rouge pour les piétons. Cette dernière phase est semblable à la première. Durée : jusqu'au prochain appui sur le bouton

Composants supplémentaires

Ce sketch élargi nécessite les composants supplémentaires suivants.

	1 LED rouge
	1 LED verte
	2 résistances de 330 Ω

1 résistance de 10 kΩ

1 bouton-poussoir

Le code élargi est alors le suivant :

```
#define DELAY0 10000          //Pause 0, 10 secondes
#define DELAY1 1000           //Pause 1, 1 seconde
#define DELAY2 2000           //Pause 2, 2 secondes
#define DELAY3 3000           //Pause 3, 3 secondes
int ledPinRedDrive = 7;       //Broche 7 commande la LED rouge
                              //(feux pour automobilistes)
int ledPinOrangeDrive = 6;    //Broche 6 commande la LED orange
                              //(feux pour automobilistes)
int ledPinGreenDrive = 5;     //Broche 5 commande la LED verte
                              //(feux pour automobilistes)
int ledPinRedWalk = 3;        //Broche 3 commande la LED rouge
                              //(feux pour piétons)
int ledPinGreenWalk = 2;      //Broche 2 commande la LED verte
                              //(feux pour piétons)
int buttonPinLight = 8;       //Bouton gérant les feux reliés
                              //à la broche 8
int buttonLightValue = LOW; //Variable pour l'état
                              //du bouton gérant les feux
void setup(){
  pinMode(ledPinRedDrive, OUTPUT);       //Broche comme sortie
  pinMode(ledPinOrangeDrive, OUTPUT);    //Broche comme sortie
  pinMode(ledPinGreenDrive, OUTPUT);     //Broche comme sortie
  pinMode(ledPinRedWalk, OUTPUT);        //Broche comme sortie
  pinMode(ledPinGreenWalk, OUTPUT);      //Broche comme sortie
  pinMode(buttonPinLight, INPUT);        //Broche comme entrée
  digitalWrite(ledPinGreenDrive, HIGH); //Valeurs de départ
                                        //(vert pour automobilistes)
  digitalWrite(ledPinRedWalk, HIGH);    //Valeurs de départ
                                        //(rouge pour piétons)
}
void loop(){
     //Lire l'état du bouton gérant les feux dans la variable
     buttonLightValue = digitalRead (buttonPinLight);
     //Si bouton appuyé, fonction appelée
     if(buttonLightValue == HIGH)
          lightChange();
}
void lightChange(){
  digitalWrite (ledPinGreenDrive, LOW);
```

```
  digitalWrite(ledPinOrangeDrive, HIGH); delay(DELAY3);
  digitalWrite(ledPinOrangeDrive, LOW);
  digitalWrite(ledPinRedDrive, HIGH); delay(DELAY1);
  digitalWrite(ledPinRedWalk, LOW);
  digitalWrite(ledPinGreenWalk, HIGH); delay(DELAY0);
  digitalWrite(ledPinGreenWalk, LOW);
  digitalWrite(ledPinRedWalk, HIGH); delay(DELAY1);
  digitalWrite(ledPinOrangeDrive, HIGH); delay(DELAY2);
  digitalWrite(ledPinRedDrive, LOW);
  digitalWrite(ledPinOrangeDrive, LOW);
  digitalWrite(ledPinGreenDrive, HIGH);
}
```

Le nombre de ports nécessaires est passé à 6, mais cela ne signifie pas pour autant que les choses sont devenues plus difficiles. Vous devez seulement soigner un peu plus le câblage et l'affectation des broches. Commençons par les variables à spécifier tout au début du programme.

D'un point de vue logiciel, voici les variables nécessaires à notre programmation expérimentale.

Tableau 7-3
Variables nécessaires et leur objet

Variable	Objet
`ledPinRedDrive`	Commande la LED rouge (feux pour automobilistes)
`ledPinOrangeDrive`	Commande la LED orange (feux pour automobilistes)
`ledPinGreenDrive`	Commande la LED verte (feux pour automobilistes)
`ledPinRedWalk`	Commande la LED rouge (feux pour piétons)
`ledPinGreenWalk`	Commande la LED verte (feux pour piétons)
`buttonPinLight`	Broche de raccordement du bouton-poussoir des feux pour piétons
`buttonLightValue`	Enregistre la valeur de l'état du bouton-poussoir

Les différentes broches sont programmées comme entrées ou sorties, et la valeur de démarrage `LOW` est attribuée à la variable `buttonLightValue` au sein de la fonction `setup`. Parce qu'aucun changement de phase ne survient tant qu'on n'appuie pas sur le bouton, le circuit doit présenter un état de départ bien défini. Aussi les feux pour automobilistes et piétons sont-ils initialisés par les deux lignes :

```
digitalWrite(ledPinGreenDrive, HIGH);
digitalWrite(ledPinRedWalk, HIGH);
```

Au sein de la fonction `loop`, l'état du bouton-poussoir est constamment interrogé par la fonction `digitalRead` et le résultat est affecté à la variable `buttonLightValue`. L'évaluation intervient immédiatement dans le test de la structure de contrôle `if` :

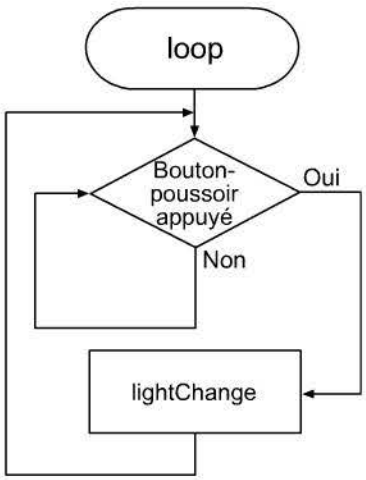
```
if(buttonLightValue == HIGH)
  lightChange();
```

En cas de niveau `HIGH`, on passe directement à la fonction `lightChange`, qui déclenche alors les changements de phase.

> Et que se passe-t-il si j'appuie une deuxième fois sur le bouton-poussoir ? Le déroulement s'en trouve-t-il d'une manière ou d'une autre perturbé ?

Cette question vient à point nommé. Récapitulons le déroulement du sketch. L'organigramme suivant devrait répondre à votre question.

Figure 7-7 ▶
Appel de la fonction `lightChange`



Comme vous pouvez le voir, l'état du bouton-poussoir est constamment interrogé et évalué en début de traitement dans la fonction `loop`. Ce sont les seules étapes de traitement dans cette fonction. Elle n'a donc rien d'autre à faire que d'observer l'état du bouton-poussoir et de bifurquer dans la fonction `lightChange` quand le niveau passe de `LOW` à `HIGH`. Une fois la fonction appelée, les divers changements de phase sont initiés et les phases sont maintenues par différents appels de la fonction `delay`.

Nous venons alors tout juste quitter la fonction `loop`. Un nouvel appui sur le bouton-poussoir ne serait donc pas enregistré par la logique, car la fonction `digitalRead` n'est plus constamment appelée. Il ne le serait qu'après avoir quitté la fonction `lightChange`.

Figure 7-8
Appel et retour

Avant de passer au schéma, voici encore un chronogramme montrant les différentes durées d'allumage l'une par rapport à l'autre. La situation de départ nous montre que le feu est vert pour les automobilistes et rouge pour les piétons. Un piéton ayant l'intention de traverser la route à un endroit supposé plus sûr appuie sur le bouton gérant les feux, ce qui initie les changements de phase.

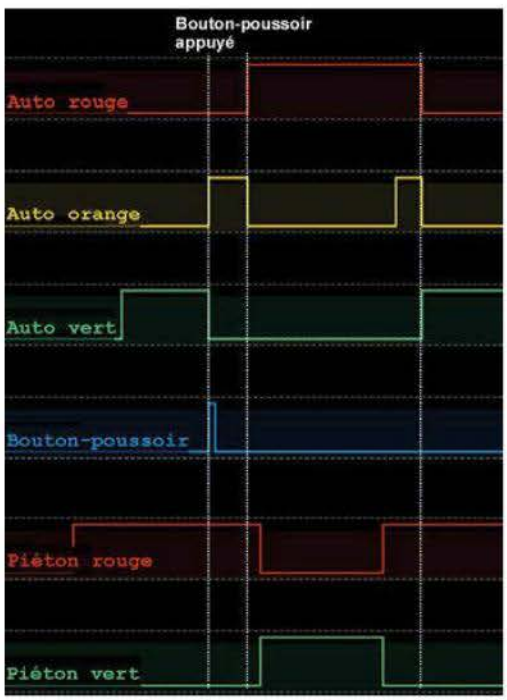
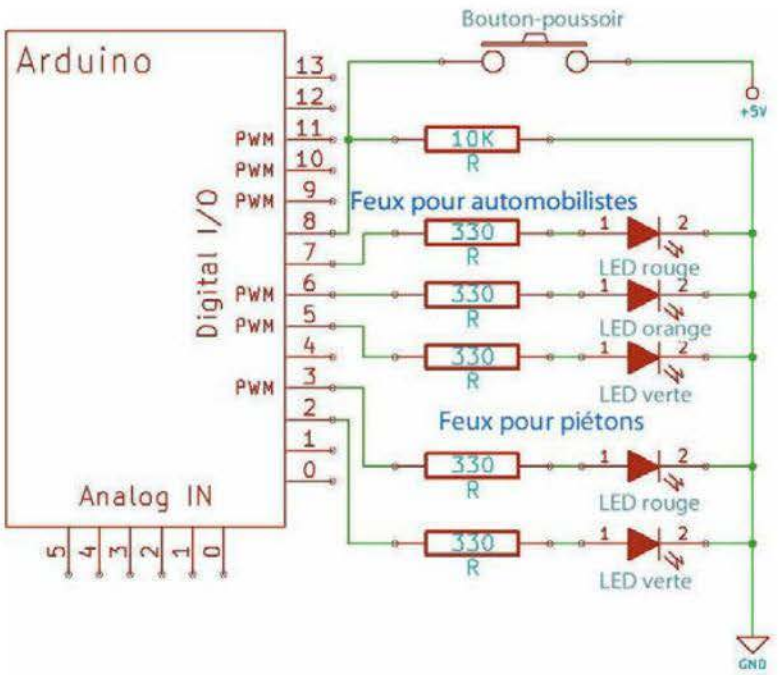


Figure 7-9
Chronogramme des feux interactifs

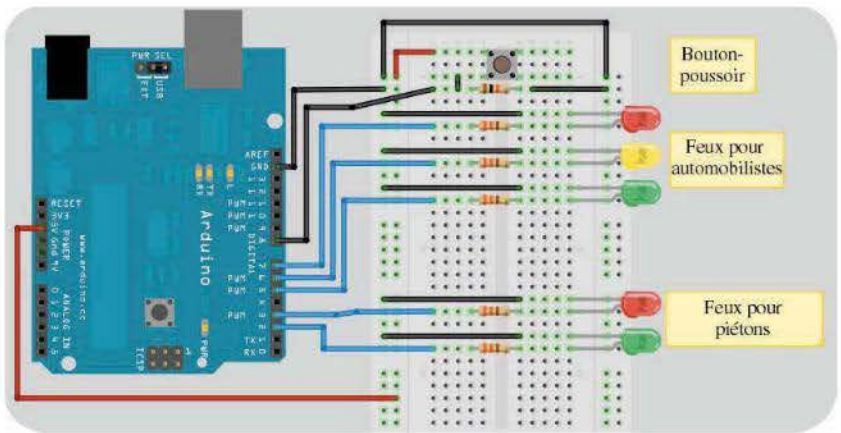
Dans le schéma relatif au dernier sketch, vous pouvez voir les extensions à ajouter pour faire fonctionner le circuit.

Figure 7-10 ▶
Circuit interactif avec feux pour automobilistes et piétons



La construction est alors la suivante sur la plaque d'essais.

Figure 7-11 ▶
Réalisation du circuit interactif pour feux avec Fritzing



Autre sketch élargi

Je vais maintenant modifier encore un peu le sketch gérant les feux de circulation, afin que vous fassiez travailler davantage votre matière grise.

Dans la programmation du circuit pour feux de circulation, j'ai toujours oublié d'éteindre l'une ou l'autre LED lors d'un changement

de phase avant d'allumer la suivante lors du premier essai, ce qui m'a gêné. J'ai donc imaginé de configurer plus simplement l'allumage et l'extinction des LED. Certes, cela demande un peu de préparation mais peut s'avérer utile pour vos montages à venir. Mais avant de commencer, je dois d'abord vous parler un peu des bits et des octets.

Le circuit ne change pas. Ordinateur et carte Arduino stockent toutes les données au niveau le plus bas de la mémoire sous forme de bits et d'octets (8 bits). J'ai déjà abordé ce thème dans le montage n° 6 sur l'extension de port numérique. En voici les grandes lignes (voir figure 7-12).

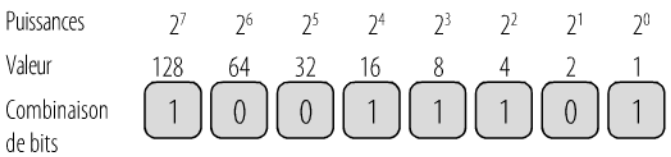


Figure 7-12
Combinaison binaire pour le nombre entier 157

Le nombre qui s'écrit en binaire 10011101 s'écrit en décimal :

$$1 \times 2^0 + 0 \times 2^1 + 1 \times 2^2 + 1 \times 2^3 + 1 \times 2^4 + 0 \times 2^5 + 0 \times 2^6 + 1 \times 2^7 = 157_{10}$$

Il suffit maintenant que certains bits de cet octet servent à commander les différentes LED de nos feux de circulation pour pouvoir allumer ou éteindre toutes les LED au moyen d'une seule valeur exprimée en décimal.

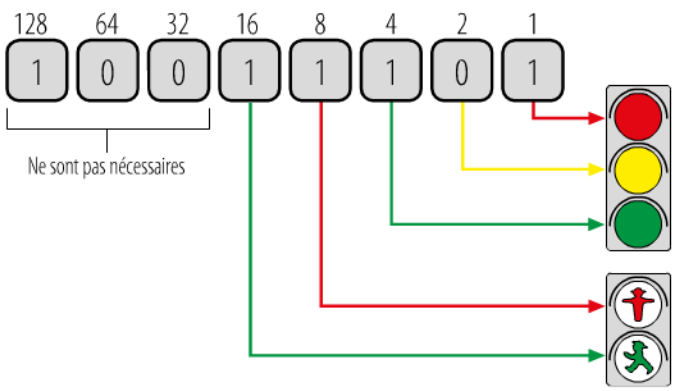


Figure 7-13
Quel bit pour quelle LED ?

On voit que 5 bits de cet octet suffisent à commander les feux. Mais comment fait-on au juste ? J'ai reporté dans le tableau 7-4 les nombres décimaux correspondants, que j'ai déterminés à partir des différentes phases.

Tableau 7-4 ▶
Nombres décimaux pour commander les LED

	Piéton		Automobiliste			
LED	Verte	Rouge	Verte	Orange	Rouge	Nombre décimal
Poids	2^3=8	2^3=8	2^2=4	2^1=2	2^0=1	
Phase 1	0	1	1	0	0	12
Phase 2	0	1	0	1	0	10
Phase 3	0	1	0	0	1	9
Phase 4	1	0	0	0	1	17
Phase 5	0	1	0	0	1	9
Phase 6	0	1	0	1	1	11

Reste à trouver, à partir des nombres décimaux correspondants, quel bit commande une LED particulière. C'est possible avec l'opérateur ET bit à bit `&`. Le tableau 7-5 montre que le résultat n'est `1` que si les deux opérandes ont `1` pour valeur.

Tableau 7-5 ▶
Opération ET bit à bit

Opérande 1	Opérande 2	Opération ET
0	0	0
0	1	0
1	0	0
1	1	1

Voici un exemple : vérifions que la LED rouge des feux pour piétons s'allume pendant la phase 1 de notre commande pour feux de circulation.

Tableau 7-6 ▶
Contrôle de correspondance du bit

	Piéton		Automobiliste			
LED	Verte	Rouge	Verte	Orange	Rouge	Nombre décimal
Poids	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
Phase 1	0	**1**	1	0	0	12
Opérande	0	**1**	0	0	0	8
Résultat	0	**1**	0	0	0	8

Le deuxième opérande avec la valeur décimale `8` sert en quelque sorte de filtre. Il vérifie, seulement dans la position du bit de poids 2^3, qu'un `1` se trouve bien dans le premier opérande. C'est le cas dans notre exemple et le résultat est `8`. Le tableau 7-7 donne les nombres décimaux pour lesquels des opérations ET bit à bit doivent être effectuées avec les valeurs provenant des différentes phases pour déterminer l'état voulu de la LED.

LED	Valeur du 2e opérande
LED rouge (automobiliste)	1
LED orange (automobiliste)	2
LED verte (automobiliste)	4
LED rouge (piéton)	8
LED verte (piéton)	16

Tableau 7-7
Valeurs pour déterminer les bits à 1 ou à 0

Nous nous servons de l'opérateur conditionnel ? pour la vérification. Il s'agit d'une forme d'évaluation spéciale d'une expression. La syntaxe générale est la suivante (voir figure 7-4).

```
Condition?Instruction1:Instruction2
```

Figure 7-14
Opérateur conditionnel ?

Quand l'exécution du programme arrive à cette ligne, la condition est d'abord évaluée. Si le résultat est vrai, `Instruction1` est exécutée, sinon c'est `Instruction2` qui l'est. Pour commander toutes les LED avec cette structure, les lignes de code suivantes doivent être écrites, le nombre décimal pour commander les LED étant mémorisé dans la variable `lightValue`.

```
digitalWrite(ledPinRedDrive, (lightValue&1)==1?HIGH:LOW);
digitalWrite(ledPinOrangeDrive, (lightValue&2)==2?HIGH:LOW);
digitalWrite(ledPinGreenDrive, (lightValue&4)==4?HIGH:LOW);
digitalWrite(ledPinRedWalk, (lightValue&8)==8?HIGH:LOW);
digitalWrite(ledPinGreenWalk, (lightValue&16)==16?HIGH:LOW);
```

Ces 5 lignes de code permettent de commander l'état (allumé ou éteint) des 5 LED.

> Il y a quelque chose que je ne comprends pas. Comment obtient-on les différentes durées d'allumage des diverses phases de signalisation ? Je ne vois nulle part l'instruction `delay` qui sert à définir les pauses.

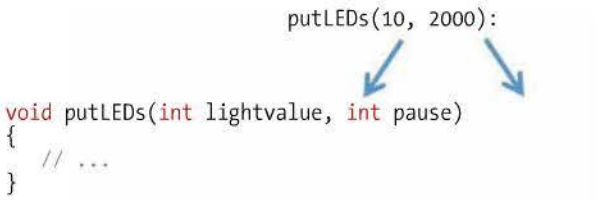
Bonne remarque Ardus, aussi ces lignes de code sont-elles insérées dans une fonction à part et complétées par `lightValue` et une deuxième valeur pour la fonction `delay`. Cela donne :

```
void putLEDs(int lightvalue, int pause){
  digitalWrite(ledPinRedDrive, (lightValue&1)= =1?HIGH:LOW);
  digitalWrite(ledPinOrangeDrive, (lightValue&2)==2?HIGH:LOW);
  digitalWrite(ledPinGreenDrive, (lightValue&4)==4?HIGH:LOW);
  digitalWrite(ledPinRedWalk, (lightValue&8)==8?HIGH:LOW);
  digitalWrite(ledPinGreenWalk, (lightValue&16)==16?HIGH:LOW);
  delay(pause);
}
```

Pour commander les différentes phases de signalisation, il ne vous reste plus qu'à appeler cette fonction avec les valeurs correspondantes qui figurent dans le tableau 7-4. Les appels sont alors les suivants :

```
void lightChange(){
      putLEDs(10, 2000);
      putLEDs(9, 1000);
      putLEDs(17, 10000);
      putLEDs(9, 1000);
      putLEDs(11, 2000);
      putLEDs(12, 0);
}
```

On voit que la fonction `putLEDs` est appelée dans la fonction `lightChange`. Regardons maintenant les choses de plus près à l'aide d'un exemple. La fonction présentant plusieurs paramètres, il est certainement utile de savoir dans quel ordre ils sont transmis lors de l'appel.



L'ordre de transmission des arguments `10` et `2000` aux paramètres de la fonction `putLEDs` est exactement celui dans lequel vous les avez écrits entre parenthèses. Les paramètres de la fonction sont définis par les variables locales `lightValue` et `pause`, dans lesquelles les valeurs transmises sont copiées.

Attention !

Respectez impérativement l'ordre des arguments lors de l'appel de la fonction. Faute de quoi, le sketch ne plantera pas dans ce cas, mais le circuit ne réagira pas comme prévu. Il faut donc que :

- le nombre des arguments doit coïncider avec celui des paramètres ;
- les types de données des arguments transmis doivent correspondre à ceux des paramètres ;
- l'ordre doit être respecté lors de l'appel.

Vous avez employé encore une fois l'expression « variable locale ». Je n'ai pas encore bien saisi la différence entre variable locale et variable globale.

Pas de problème ! La différence est toute simple. Les variables globales sont déclarées et initialisées en début de sketch et sont visibles partout, même à l'intérieur des fonctions, pendant le fonctionnement. La ligne de code suivante montre une variable globale de notre sketch :

```
int ledPinRedDrive = 7;  //Broche 7 commande la LED rouge (feux pour
                         //automobilistes)
// …
```

Celle-ci est utilisée plus tard dans la fonction `setup`. Elle y est donc visible et vous pouvez y accéder.

```
void setup( ){
  pinMode(ledPinRedDrive, OUTPUT);  //Broche comme sortie
}
```

Les variables locales sont déclarées ou initialisées dans des fonctions ou, par exemple, dans une boule `for`. Elles ont une durée de vie limitée et ne sont visibles que dans la fonction ou le bloc d'exécution. « Durée de vie » signifie qu'une zone spéciale est mise à la disposition des variables locales lorsque la fonction est appelée dans la mémoire. Une fois la fonction quittée, ces variables ne sont plus utiles et la mémoire est libérée. Une variable locale n'est jamais visible hormis dans la fonction où elle a été déclarée et elle ne peut pas non plus être utilisée depuis l'extérieur.

Bon, j'ai compris. Mais qu'en est-il des valeurs définies avec `#define` au début du sketch ? Quel comportement ont-elles ?

Vous pouvez aussi les considérer comme des définitions globales qui sont visibles et accessibles partout dans le sketch. Maintenant que vous connaissez la directive `#define`, je peux vous dire que des constantes telles que `HIGH`, `LOW`, `INPUT` ou `OUTPUT` – et de nombreuses autres encore – ont été également définies par ces directives.

Pour aller plus loin

Dans le répertoire suivant :

`arduino-1.x.y\hardware\arduino\cores\arduino`

vous trouverez, entre autres, un fichier nommé `Arduino.h`. Ce fichier de l'EDI d'Arduino contient beaucoup de définitions importantes, notamment celles dont je viens de parler. En voici un court extrait :

```
#define HIGH 0x1
#define LOW  0x0

#define INPUT 0x0
#define OUTPUT 0x1

#define true 0x1
#define false 0x0

#define PI 3.14159265358979323846264338327795
#define HALF_PI 1.57079632679489661923132169163975
#define TWO_PI 6.28318530717958647692528676655900
#define DEG_TO_RAD 0.017453292519943295769236907684886
#define RAD_TO_DEG 57.295779513082320876798154814105

#define SERIAL  0x0
#define DISPLAY 0x1

#define LSBFIRST 0
#define MSBFIRST 1
```

Cela ne vous rappelle rien ? Vous en saurez bientôt plus sur ce qu'est un fichier d'en-tête dans le montage n° 9. Contentez-vous pour l'instant de savoir qu'il est intégré par le compilateur dans le projet et que toutes les définitions qu'il contient sont globales et disponibles dans le sketch.

Problèmes courants

Si les LED ne s'allument pas les unes après les autres, débranchez le port USB de la carte pour plus de sécurité et vérifiez ce qui suit.

- Vos fiches de raccordement sur la plaque d'essais correspondent-elles vraiment au circuit ?
- Est-ce qu'il y a un court-circuit éventuel ?
- Les différentes LED sont-elles correctement branchées ? La polarité est-elle correcte ?
- Les résistances ont-elles bien les bonnes valeurs ?
- Le code du sketch est-il correct ?
- Le bouton-poussoir est-il correctement câblé ? Vérifiez encore une fois les contacts en question avec un testeur de continuité.

Qu'avez-vous appris ?

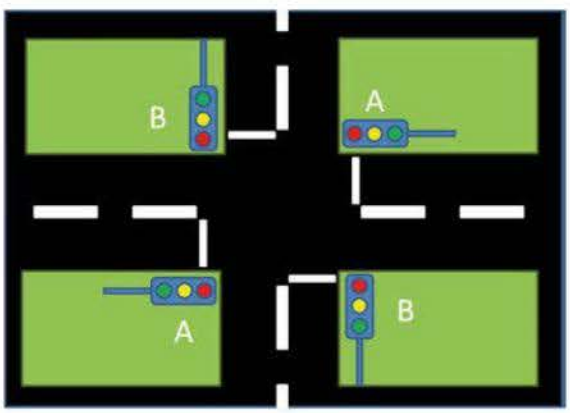
- Vous avez découvert comment évaluer le niveau d'une entrée numérique à l'aide de la fonction `digitalRead`.
- Vous avez réalisé un circuit pour feux de circulation à la fois simple, car initiant automatiquement les différents changements de phase indépendamment des influences externes, mais aussi interactif car réagissant avec un capteur (ici, un bouton-poussoir)

à des impulsions de l'extérieur et déclenchant alors seulement les changements de phase.

- Utiliser la directive de prétraitement `#define` ne devrait plus vous poser de problèmes. Elle est employée la plupart du temps là où des constantes sont définies. Le compilateur remplace partout dans le code le nom de l'identifiant par l'expression correspondante.
- L'opérateur conditionnel `?` peut être employé pour retourner différentes valeurs en fonction de l'évaluation d'une expression. Le mode d'écriture est vraiment compact et n'est pas toujours immédiatement compréhensible.
- Vous avez appris à transmettre plusieurs valeurs à une fonction, et vous savez en détail à quoi il faut faire attention.
- Vous connaissez la différence entre variable locale et variable globale, et vous savez ce que visibilité et durée de vie signifient dans ce cas.

Exercice complémentaire

Réalisez un circuit pour feux de circulation à un croisement. Le dessin suivant peut vous servir de base.



Les paires de feux de circulation A et B doivent être commandées en même temps. Cette fois-ci, il n'y aura pas de feux pour piétons. Veillez à ce que quand un sens passe au rouge, l'autre ne passe pas tout de suite au vert. Un délai de sécurité doit être prévu pour que les automobilistes déjà engagés puissent encore franchir le croisement quand le feu passe du vert au rouge.

Cadeau !

Je vous propose ici une carte à construire facilement, qui vous servira à réaliser le circuit pour feux de circulation avec feux pour piétons.

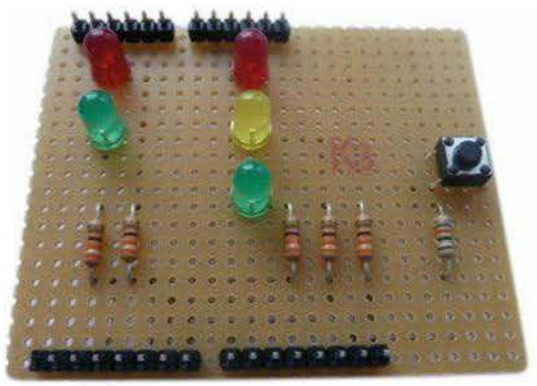


Figure 7-15 ▶
Circuit pour feux de circulation avec feux pour piétons